Sphinx 0.9.9 reference manual

Table of Contents

1. Introduction

1.1. About

Sphinx is a full-text search engine, distributed under GPL version 2. Commercial licensing (eg. for embedded use) is also available upon request.

Generally, it's a standalone search engine, meant to provide fast, size-efficient and relevant full-text search functions to other applications. Sphinx was specially designed to integrate well with SQL databases and scripting languages.

Currently built-in data source drivers support fetching data either via direct connection to MySQL, or PostgreSQL, or from a pipe in a custom XML format. Adding new drivers (eg. to natively support some other DBMSes) is designed to be as easy as possible.

Search API is natively ported to PHP, Python, Perl, Ruby, Java, and also available as a pluggable MySQL storage engine. API is very lightweight so porting it to new language is known to take a few hours.

As for the name, Sphinx is an acronym which is officially decoded as SQL Phrase Index. Yes, I know about CMU's Sphinx project.

1.2. Sphinx features

- high indexing speed (upto 10 MB/sec on modern CPUs);
- high search speed (avg query is under 0.1 sec on 2-4 GB text collections);
- high scalability (upto 100 GB of text, upto 100 M documents on a single CPU);
- provides good relevance ranking through combination of phrase proximity ranking and statistical (BM25)

ranking;
- provides distributed searching capabilities;
- provides document exceprts generation;
- provides searching from within MySQL through pluggable storage engine;
- supports boolean, phrase, and word proximity queries;
- supports multiple full-text fields per document (upto 32 by default);
- supports multiple additional attributes per document (ie. groups, timestamps, etc);
- supports stopwords;
- supports both single-byte encodings and UTF-8;
- supports English stemming, Russian stemming, and Soundex for morphology;
- supports MySQL natively (MyISAM and InnoDB tables are both supported);
- supports PostgreSQL natively.

1.3. Where to get Sphinx

Sphinx is available through its official Web site at http://www.sphinxsearch.com/.

Currently, Sphinx distribution tarball includes the following software:

- `indexer`: an utility which creates fulltext indexes;
- `search`: a simple command-line (CLI) test utility which searches through fulltext indexes;
- `searchd`: a daemon which enables external software (eg. Web applications) to search through fulltext indexes;
- `sphinxapi`: a set of searchd client API libraries for popular Web scripting languages (PHP, Python, Perl, Ruby).
- `spelldump`: a simple command-line tool to extract the items from an `ispell` or `MySpell` (as bundled with OpenOffice) format dictionary to help customize your index, for use with wordforms.
- `indextool`: an utility to dump miscellaneous debug information about the index, added in version 0.9.9-rc2.

1.4. License

1.5. Author and contributors

Author

Sphinx initial author and current primary developer is:

- Andrew Aksyonoff, <shodan(at)shodan.ru>

Contributors

People who contributed to Sphinx and their contributions (in no particular order) are:

- Robert "coredev" Bengtsson (Sweden), initial version of PostgreSQL data source;

- Len Kranendonk, Perl API
- Dmytro Shteflyuk, Ruby API

Many other people have contributed ideas, bug reports, fixes, etc. Thank you!

1.6. History

Sphinx development was started back in 2001, because I didn't manage to find an acceptable search solution (for a database driven Web site) which would meet my requirements. Actually, each and every important aspect was a problem:

- search quality (ie. good relevance)
  - statistical ranking methods performed rather bad, especially on large collections of small documents (forums, blogs, etc)
- search speed
  - especially if searching for phrases which contain stopwords, as in "to be or not to be"
- moderate disk and CPU requirements when indexing
  - important in shared hosting enivronment, not to mention the indexing speed.

Despite the amount of time passed and numerous improvements made in the other solutions, there's still no solution which I personally would be eager to migrate to.

Considering that and a lot of positive feedback received from Sphinx users during last years, the obvious decision is to continue developing Sphinx (and, eventually, to take over the world).

2. Installation

2.1. Supported systems

Most modern UNIX systems with a C++ compiler should be able to compile and run Sphinx without any modifications.

Currently known systems Sphinx has been successfully running on are:

- Linux 2.4.x, 2.6.x (various distributions)
- Windows 2000, XP
- FreeBSD 4.x, 5.x, 6.x
- NetBSD 1.6, 3.0
- Solaris 9, 11
- Mac OS X

CPU architectures known to work include X86, X86-64, SPARC64.

I hope Sphinx will work on other Unix platforms as well. If the platform you run Sphinx on is not in this list, please do report it.

At the moment, Windows version of Sphinx is not intended to be used in production, but rather for testing and debugging only. Two most prominent issues are missing concurrent queries support (client queries are stacked on TCP connection level instead), and missing index data rotation support. There are succesful production installations which workaround these issues. However, running high-volume search service under Windows is still not recommended.

2.2. Required tools

On UNIX, you will need the following tools to build and install Sphinx:

- a working C++ compiler. GNU gcc is known to work.
- a good make program. GNU make is known to work.

On Windows, you will need Microsoft Visual C/C++ Studio .NET 2003 or 2005. Other compilers/environments will probably work as well, but for the time being, you will have to build makefile (or other environment specific

project files) manually.

2.3. Installing Sphinx on Linux

1. Extract everything from the distribution tarball (haven't you already?) and go to the `sphinx` subdirectory:

   $ tar xzvf sphinx-0.9.8.tar.gz
   $ cd sphinx

2. Run the configuration program:

   $ ./configure

   There's a number of options to configure. The complete listing may be obtained by using `--help` switch. The most important ones are:

   - `--prefix`, which specifies where to install Sphinx; such as `--prefix=/usr/local/sphinx` (all of the examples use this prefix)
   - `--with-mysql`, which specifies where to look for MySQL include and library files, if auto-detection fails;
   - `--with-pgsql`, which specifies where to look for PostgreSQL include and library files.

3. Build the binaries:

   $ make

4. Install the binaries in the directory of your choice: (defaults to `/usr/local/bin/` on *nix systems, but is overridden with `configure --prefix`)

   $ make install

2.4. Installing Sphinx on Windows

Installing Sphinx on a Windows server is often easier than installing on a Linux environment; unless you are preparing code patches, you can use the pre-compiled binary files from the Downloads area on the website.

1. Extract everything from the .zip file you have downloaded - `sphinx-0.9.8-win32.zip` (or `sphinx-0.9.8-win32-pgsql.zip` if you need PostgresSQL support as well.) You can use Windows Explorer in Windows XP and up to extract the files, or a freeware package like 7Zip to open the archive.

   For the remainder of this guide, we will assume that the folders are unzipped into `C:\Sphinx`, such that `searchd.exe` can be found in `C:\Sphinx\bin\searchd.exe`. If you decide to use any different location for the folders or configuration file, please change it accordingly.

2. Install the `searchd` system as a Windows service:

   C:\Sphinx> C:\Sphinx\searchd --install --config C:\Sphinx\sphinx.conf --servicename SphinxSearch

3. The `searchd` service will now be listed in the Services panel within the Management Console, available from Administrative Tools. It will not have been started, as you will need to configure it and build your indexes with `indexer` before starting the service. A guide to do this can be found under Quick tour.

2.5. Known installation issues

If `configure` fails to locate MySQL headers and/or libraries, try checking for and installing `mysql-devel` package. On some systems, it is not installed by default.

If `make` fails with a message which look like

```
/bin/sh: g++: command not found
```

```
make[1]: *** [libsphinx_a-sphinx.o] Error 127
```

try checking for and installing `gcc-c++` package.

If you are getting compile-time errors which look like

```
sphinx.cpp:67: error: invalid application of `sizeof' to
    incomplete type `Private::SizeError<false>'
```

this means that some compile-time type size check failed. The most probable reason is that off_t type is less than 64-bit on your system. As a quick hack, you can edit sphinx.h and replace off_t with DWORD in a typedef for SphOffset_t, but note that this will prohibit you from using full-text indexes larger than 2 GB. Even if the hack helps, please report such issues, providing the exact error message and compiler/OS details, so I could properly fix them in next releases.

If you keep getting any other error, or the suggestions above do not seem to help you, please don't hesitate to contact me.

2.6. Quick Sphinx usage tour

All the example commands below assume that you installed Sphinx in `/usr/local/sphinx`, so `searchd` can be found in `/usr/local/sphinx/bin/searchd`.

To use Sphinx, you will need to:

1. Create a configuration file.

   Default configuration file name is `sphinx.conf`. All Sphinx programs look for this file in current working directory by default.

   Sample configuration file, `sphinx.conf.dist`, which has all the options documented, is created by `configure`. Copy and edit that sample file to make your own configuration: (assuming Sphinx is installed into `/usr/local/sphinx/`)

   $ cd /usr/local/sphinx/etc
   $ cp sphinx.conf.dist sphinx.conf
   $ vi sphinx.conf

   Sample configuration file is setup to index `documents` table from MySQL database `test`; so there's `example.sql` sample data file to populate that table with a few documents for testing purposes:

   $ mysql -u test < /usr/local/sphinx/etc/example.sql

2. Run the indexer to create full-text index from your data:

   $ cd /usr/local/sphinx/etc
   $ /usr/local/sphinx/bin/indexer

3. Query your newly created index!

To query the index from command line, use `search` utility:

$ cd /usr/local/sphinx/etc
$ /usr/local/sphinx/bin/search test

To query the index from your PHP scripts, you need to:

1. Run the search daemon which your script will talk to:

   $ cd /usr/local/sphinx/etc
   $ /usr/local/sphinx/bin/searchd

2. Run the attached PHP API test script (to ensure that the daemon was succesfully started and is ready to serve the queries):

```
$ cd sphinx/api
$ php test.php test
```

3. Include the API (it's located in `api/sphinxapi.php`) into your own scripts and use it.

Happy searching!

3. Indexing

3.1. Data sources

The data to be indexed can generally come from very different sources: SQL databases, plain text files, HTML files, mailboxes, and so on. From Sphinx point of view, the data it indexes is a set of structured *documents*, each of which has the same set of *fields*. This is biased towards SQL, where each row correspond to a document, and each column to a field.

Depending on what source Sphinx should get the data from, different code is required to fetch the data and prepare it for indexing. This code is called *data source driver* (or simply *driver* or *data source* for brevity).

At the time of this writing, there are drivers for MySQL and PostgreSQL databases, which can connect to the database using its native C/C++ API, run queries and fetch the data. There's also a driver called xmlpipe, which runs a specified command and reads the data from its `stdout`. See Section 3.8, "xmlpipe data source" section for the format description.

There can be as many sources per index as necessary. They will be sequentially processed in the very same order which was specifed in index definition. All the documents coming from those sources will be merged as if they were coming from a single source.

3.2. Attributes

Attributes are additional values associated with each document that can be used to perform additional filtering and sorting during search.

It is often desired to additionally process full-text search results based not only on matching document ID and its rank, but on a number of other per-document values as well. For instance, one might need to sort news search results by date and then relevance, or search through products within specified price range, or limit blog search to posts made by selected users, or group results by month. To do that efficiently, Sphinx allows to attach a number of additional *attributes* to each document, and store their values in the full-text index. It's then possible to use stored values to filter, sort, or group full-text matches.

Attributes, unlike the fields, are not full-text indexed. They are stored in the index, but it is not possible to search them as full-text, and attempting to do so results in an error.

For example, it is impossible to use the extended matching mode expression `@column 1` to match documents where column is 1, if column is an attribute, and this is still true even if the numeric digits are normally indexed.

Attributes can be used for filtering, though, to restrict returned rows, as well as sorting or result grouping; it is entirely possible to sort results purely based on attributes, and ignore the search relevance tools. Additionally, attributes are returned from the search daemon, while the indexed text is not.

A good example for attributes would be a forum posts table. Assume that only title and content fields need to be full-text searchable - but that sometimes it is also required to limit search to a certain author or a sub-forum (ie. search only those rows that have some specific values of author_id or forum_id columns in the SQL table); or to sort matches by post_date column; or to group matching posts by month of the post_date and calculate per-group match counts.

This can be achieved by specifying all the mentioned columns (excluding title and content, that are full-text fields) as attributes, indexing them, and then using API calls to setup filtering, sorting, and grouping. Here as an example.

Example sphinx.conf part:

```
...
sql_query = SELECT id, title, content, \
        author_id, forum_id, post_date FROM my_forum_posts
sql_attr_uint = author_id
sql_attr_uint = forum_id
sql_attr_timestamp = post_date
...
```

Example application code (in PHP):

```
// only search posts by author whose ID is 123
$cl->SetFilter ( "author_id", array ( 123 ) );

// only search posts in sub-forums 1, 3 and 7
$cl->SetFilter ( "forum_id", array ( 1,3,7 ) );

// sort found posts by posting date in descending order
$cl->SetSortMode ( SPH_SORT_ATTR_DESC, "post_date" );
```

Attributes are named. Attribute names are case insensitive. Attributes are *not* full-text indexed; they are stored in the index as is. Currently supported attribute types are:

- unsigned integers (1-bit to 32-bit wide);
- UNIX timestamps;
- floating point values (32-bit, IEEE 754 single precision);
- string ordinals (specially computed integers);
- MVA, multi-value attributes (variable-length lists of 32-bit unsigned integers).

The complete set of per-document attribute values is sometimes referred to as *docinfo*. Docinfos can either be

- stored separately from the main full-text index data ("extern" storage, in `.spa` file), or
- attached to each occurence of document ID in full-text index data ("inline" storage, in `.spd` file).

When using extern storage, a copy of `.spa` file (with all the attribute values for all the documents) is kept in RAM by `searchd` at all times. This is for performance reasons; random disk I/O would be too slow. On the contrary, inline storage does not require any additional RAM at all, but that comes at the cost of greatly inflating the index size: remember that it copies *all* attribute value *every* time when the document ID is mentioned, and that is exactly as many times as there are different keywords in the document. Inline may be the only viable option if you have only a few attributes and need to work with big datasets in limited RAM. However, in most cases extern storage makes both indexing and searching *much* more efficient.

Search-time memory requirements for extern storage are (1+number_of_attrs)*number_of_docs*4 bytes, ie. 10 million docs with 2 groups and 1 timestamp will take (1+2+1)*10M*4 = 160 MB of RAM. This is *PER DAEMON*, not per query. `searchd` will allocate 160 MB on startup, read the data and keep it shared between queries. The children will *NOT* allocate any additional copies of this data.

3.3. MVA (multi-valued attributes)

MVAs, or multi-valued attributes, are an important special type of per-document attributes in Sphinx. MVAs make it possible to attach lists of values to every document. They are useful for article tags, product categories, etc. Filtering and group-by (but not sorting) on MVA attributes is supported.

Currently, MVA list entries are limited to unsigned 32-bit integers. The list length is not limited, you can have an arbitrary number of values attached to each document as long as RAM permits (`.spm` file that contains the MVA values will be precached in RAM by `searchd`). The source data can be taken either from a separate query, or from a document field; see source type in sql_attr_multi. In the first case the query will have to return pairs of document ID and MVA values, in the second one the field will be parsed for integer values. There are absolutely no requirements as to incoming data order; the values will be automatically grouped by document ID (and internally sorted within the same ID) during indexing anyway.

When filtering, a document will match the filter on MVA attribute if *any* of the values satisfy the filtering condition. (Therefore, documents that pass through exclude filters will not contain any of the forbidden values.) When grouping by MVA attribute, a document will contribute to as many groups as there are different MVA values associated with that document. For instance, if the collection contains exactly 1 document having a 'tag' MVA with values 5, 7, and 11, grouping on 'tag' will produce 3 groups with '@count' equal to 1 and '@groupby' key values of 5, 7, and 11 respectively. Also note that grouping by MVA might lead to duplicate documents in the result set: because each document can participate in many groups, it can be chosen as the best one in in more than one group, leading to duplicate IDs. PHP API historically uses ordered hash on the document ID for the resulting rows; so you'll also need to use SetArrayResult() in order to employ group-by on MVA with PHP API.

3.4. Indexes

To be able to answer full-text search queries fast, Sphinx needs to build a special data structure optimized for such queries from your text data. This structure is called *index*; and the process of building index from text is called *indexing*.

Different index types are well suited for different tasks. For example, a disk-based tree-based index would be easy to update (ie. insert new documents to existing index), but rather slow to search. Therefore, Sphinx architecture allows for different *index types* to be implemented easily.

The only index type which is implemented in Sphinx at the moment is designed for maximum indexing and searching speed. This comes at a cost of updates being really slow; theoretically, it might be slower to update this type of index than than to reindex it from scratch. However, this very frequently could be worked around with muiltiple indexes, see Section 3.10, "Live index updates" for details.

It is planned to implement more index types, including the type which would be updateable in real time.

There can be as many indexes per configuration file as necessary. `indexer` utility can reindex either all of them (if `--all` option is specified), or a certain explicitly specified subset. `searchd` utility will serve all the specified indexes, and the clients can specify what indexes to search in run time.

3.5. Restrictions on the source data

There are a few different restrictions imposed on the source data which is going to be indexed by Sphinx, of which the single most important one is:

ALL DOCUMENT IDS MUST BE UNIQUE UNSIGNED NON-ZERO INTEGER NUMBERS (32-BIT OR 64-BIT, DEPENDING ON BUILD TIME SETTINGS).

If this requirement is not met, different bad things can happen. For instance, Sphinx can crash with an internal assertion while indexing; or produce strange results when searching due to conflicting IDs. Also, a 1000-pound gorilla might eventually come out of your display and start throwing barrels at you. You've been warned.

3.6. Charsets, case folding, and translation tables

When indexing some index, Sphinx fetches documents from the specified sources, splits the text into words, and does case folding so that "Abc", "ABC" and "abc" would be treated as the same word (or, to be pedantic, *term*).

To do that properly, Sphinx needs to know

- what encoding is the source text in;
- what characters are letters and what are not;
- what letters should be folded to what letters.

This should be configured on a per-index basis using `charset_type` and `charset_table` options. `charset_type` specifies whether the document encoding is single-byte (SBCS) or UTF-8. `charset_table` specifies the table that maps letter characters to their case folded versions. The characters that are not in the table are considered to be non-letters and will be treated as word separators when indexing or searching through this index.

Note that while default tables do not include space character (ASCII code 0x20, Unicode U+0020) as a letter,

it's in fact *perfectly legal* to do so. This can be useful, for instance, for indexing tag clouds, so that space-separated word sets would index as a *single* search query term.

Default tables currently include English and Russian characters. Please do submit your tables for other languages!

3.7. SQL data sources (MySQL, PostgreSQL)

With all the SQL drivers, indexing generally works as follows.

- connection to the database is established;
- pre-query (see Section 9.1.11, "sql_query_pre") is executed to perform any necessary initial setup, such as setting per-connection encoding with MySQL;
- main query (see Section 9.1.12, "sql_query") is executed and the rows it returns are indexed;
- post-query (see Section 9.1.23, "sql_query_post") is executed to perform any necessary cleanup;
- connection to the database is closed;
- indexer does the sorting phase (to be pedantic, index-type specific post-processing);
- connection to the database is established again;
- post-index query (see Section 9.1.24, "sql_query_post_index") is executed to perform any necessary final cleanup;
- connection to the database is closed again.

Most options, such as database user/host/password, are straightforward. However, there are a few subtle things, which are discussed in more detail here.

Ranged queries

Main query, which needs to fetch all the documents, can impose a read lock on the whole table and stall the concurrent queries (eg. INSERTs to MyISAM table), waste a lot of memory for result set, etc. To avoid this, Sphinx supports so-called *ranged queries*. With ranged queries, Sphinx first fetches min and max document IDs from the table, and then substitutes different ID intervals into main query text and runs the modified query to fetch another chunk of documents. Here's an example.

Example 1. Ranged query usage example

```
# in sphinx.conf

sql_query_range = SELECT MIN(id),MAX(id) FROM documents
sql_range_step = 1000
sql_query = SELECT * FROM documents WHERE id>=$start AND id<=$end
```

If the table contains document IDs from 1 to, say, 2345, then sql_query would be run three times:

1. with `$start` replaced with 1 and `$end` replaced with 1000;
2. with `$start` replaced with 1001 and `$end` replaced with 2000;
3. with `$start` replaced with 2000 and `$end` replaced with 2345.

Obviously, that's not much of a difference for 2000-row table, but when it comes to indexing 10-million-row MyISAM table, ranged queries might be of some help.

`sql_post` vs. `sql_post_index`

The difference between post-query and post-index query is in that post-query is run immediately when Sphinx received all the documents, but further indexing **may** still fail for some other reason. On the contrary, by the time the post-index query gets executed, it is **guaranteed** that the indexing was succesful. Database connection is dropped and re-established because sorting phase can be very lengthy and would just timeout otherwise.

3.8. xmlpipe data source

xmlpipe data source was designed to enable users to plug data into Sphinx without having to implement new data sources drivers themselves. It is limited to 2 fixed fields and 2 fixed attributes, and is deprecated in favor of Section 3.9, "xmlpipe2 data source" now. For new streams, use xmlpipe2.

To use xmlpipe, configure the data source in your configuration file as follows:

```
source example_xmlpipe_source
{
    type = xmlpipe
    xmlpipe_command = perl /www/mysite.com/bin/sphinxpipe.pl
}
```

The `indexer` will run the command specified in `xmlpipe_command`, and then read, parse and index the data it prints to `stdout`. More formally, it opens a pipe to given command and then reads from that pipe.

indexer will expect one or more documents in custom XML format. Here's the example document stream, consisting of two documents:

Example 2. XMLpipe document stream

```
<document>
<id>123</id>
<group>45</group>
<timestamp>1132223498</timestamp>
<title>test title</title>
<body>
this is my document body
</body>
</document>

<document>
<id>124</id>
<group>46</group>
<timestamp>1132223498</timestamp>
<title>another test</title>
<body>
this is another document
</body>
</document>
```

Legacy xmlpipe legacy driver uses a builtin parser which is pretty fast but really strict and does not actually fully support XML. It requires that all the fields *must* be present, formatted *exactly* as in this example, and occur *exactly* in the same order. The only optional field is `timestamp`; it defaults to 1.

3.9. xmlpipe2 data source

xmlpipe2 lets you pass arbitrary full-text and attribute data to Sphinx in yet another custom XML format. It also allows to specify the schema (ie. the set of fields and attributes) either in the XML stream itself, or in the source settings.

When indexing xmlpipe2 source, indexer runs the given command, opens a pipe to its stdout, and expects well-formed XML stream. Here's sample stream data:

Example 3. xmlpipe2 document stream

```
<?xml version="1.0" encoding="utf-8"?>
<sphinx:docset>

<sphinx:schema>
<sphinx:field name="subject"/>
<sphinx:field name="content"/>
<sphinx:attr name="published" type="timestamp"/>
```

```
<sphinx:attr name="author_id" type="int" bits="16" default="1"/>
</sphinx:schema>

<sphinx:document id="1234">
<content>this is the main content <![CDATA[and this <cdata> entry
must be handled properly by xml parser lib]]></content>
<published>1012325463</published>
<subject>note how field/attr tags can be
in <b class="red">randomized</b> order</subject>
<misc>some undeclared element</misc>
</sphinx:document>

<!-- ... more documents here ... -->

</sphinx:docset>
```

Arbitrary fields and attributes are allowed. They also can occur in the stream in arbitrary order within each document; the order is ignored. There is a restriction on maximum field length; fields longer than 2 MB will be truncated to 2 MB (this limit can be changed in the source).

The schema, ie. complete fields and attributes list, must be declared before any document could be parsed. This can be done either in the configuration file using `xmlpipe_field` and `xmlpipe_attr_XXX` settings, or right in the stream using <sphinx:schema> element. <sphinx:schema> is optional. It is only allowed to occur as the very first sub-element in <sphinx:docset>. If there is no in-stream schema definition, settings from the configuration file will be used. Otherwise, stream settings take precedence.

Unknown tags (which were not declared neither as fields nor as attributes) will be ignored with a warning. In the example above, <misc> will be ignored. All embedded tags and their attributes (such as <b> in <subject> in the example above) will be silently ignored.

Support for incoming stream encodings depends on whether `iconv` is installed on the system. xmlpipe2 is parsed using `libexpat` parser that understands US-ASCII, ISO-8859-1, UTF-8 and a few UTF-16 variants natively. Sphinx `configure` script will also check for `libiconv` presence, and utilize it to handle other encodings. `libexpat` also enforces the requirement to use UTF-8 charset on Sphinx side, because the parsed data it returns is always in UTF-8.

XML elements (tags) recognized by xmlpipe2 (and their attributes where applicable) are:

sphinx:docset
: Mandatory top-level element, denotes and contains xmlpipe2 document set.

sphinx:schema
: Optional element, must either occur as the very first child of sphinx:docset, or never occur at all. Declares the document schema. Contains field and attribute declarations. If present, overrides per-source settings from the configuration file.

sphinx:field
: Optional element, child of sphinx:schema. Declares a full-text field. The only recognized attribute is "name", it specifies the element name that should be treated as a full-text field in the subsequent documents.

sphinx:attr
: Optional element, child of sphinx:schema. Declares an attribute. Known attributes are:
  - "name", specifies the element name that should be treated as an attribute in the subsequent documents.
  - "type", specifies the attribute type. Possible values are "int", "timestamp", "str2ordinal", "bool", "float" and "multi".
  - "bits", specifies the bit size for "int" attribute type. Valid values are 1 to 32.
  - "default", specifies the default value for this attribute that should be used if the attribute's element is not present in the document.

sphinx:document
: Mandatory element, must be a child of sphinx:docset. Contains arbitrary other elements with field and attribute values to be indexed, as declared either using sphinx:field and sphinx:attr elements or in the configuration file. The only known attribute is "id" that must contain the unique integer document ID.

3.10. Live index updates

There's a frequent situation when the total dataset is too big to be reindexed from scratch often, but the amount of new records is rather small. Example: a forum with a 1,000,000 archived posts, but only 1,000 new posts per day.

In this case, "live" (almost real time) index updates could be implemented using so called "main+delta" scheme.

The idea is to set up two sources and two indexes, with one "main" index for the data which only changes rarely (if ever), and one "delta" for the new documents. In the example above, 1,000,000 archived posts would go to the main index, and newly inserted 1,000 posts/day would go to the delta index. Delta index could then be reindexed very frequently, and the documents can be made available to search in a matter of minutes.

Specifying which documents should go to what index and reindexing main index could also be made fully automatical. One option would be to make a counter table which would track the ID which would split the documents, and update it whenever the main index is reindexed.

Example 4. Fully automated live updates

```
# in MySQL
CREATE TABLE sph_counter
(
    counter_id INTEGER PRIMARY KEY NOT NULL,
    max_doc_id INTEGER NOT NULL
);

# in sphinx.conf
source main
{
    # ...
    sql_query_pre = SET NAMES utf8
    sql_query_pre = REPLACE INTO sph_counter SELECT 1, MAX(id) FROM documents
    sql_query = SELECT id, title, body FROM documents \
        WHERE id<=( SELECT max_doc_id FROM sph_counter WHERE counter_id=1 )
}

source delta : main
{
    sql_query_pre = SET NAMES utf8
    sql_query = SELECT id, title, body FROM documents \
        WHERE id>( SELECT max_doc_id FROM sph_counter WHERE counter_id=1 )
}

index main
{
    source = main
    path = /path/to/main
    # ... all the other settings
}

# note how all other settings are copied from main,
# but source and path are overridden (they MUST be)
index delta : main
{
    source = delta
    path = /path/to/delta
}
```

Note how we're overriding `sql_query_pre` in the delta source. We need to explicitly have that override. Otherwise `REPLACE` query would be run when indexing delta source too, effectively nullifying it. However, when we issue the directive in the inherited source for the first time, it removes *all* inherited values, so the encoding setup is also lost. So `sql_query_pre` in the delta can not just be empty; and we need to issue the encoding setup query explicitly once again.

3.11. Index merging

Merging two existing indexes can be more efficient that indexing the data from scratch, and desired in some cases (such as merging 'main' and 'delta' indexes instead of simply reindexing 'main' in 'main+delta' partitioning scheme). So `indexer` has an option to do that. Merging the indexes is normally faster than reindexing but still *not* instant on huge indexes. Basically, it will need to read the contents of both indexes once and write the result once. Merging 100 GB and 1 GB index, for example, will result in 202 GB of IO (but that's still likely less than the indexing from scratch requires).

The basic command syntax is as follows:

```
indexer --merge DSTINDEX SRCINDEX [--rotate]
```

Only the DSTINDEX index will be affected: the contents of SRCINDEX will be merged into it. `--rotate` switch will be required if DSTINDEX is already being served by `searchd`. The initially devised usage pattern is to merge a smaller update from SRCINDEX into DSTINDEX. Thus, when merging the attributes, values from SRCINDEX will win if duplicate document IDs are encountered. Note, however, that the "old" keywords will *not* be automatically removed in such cases. For example, if there's a keyword "old" associated with document 123 in DSTINDEX, and a keyword "new" associated with it in SRCINDEX, document 123 will be found by *both* keywords after the merge. You can supply an explicit condition to remove documents from DSTINDEX to mitigate that; the relevant switch is `--merge-dst-range`:

```
indexer --merge main delta --merge-dst-range deleted 0 0
```

This switch lets you apply filters to the destination index along with merging. There can be several filters; all of their conditions must be met in order to include the document in the resulting mergid index. In the example above, the filter passes only those records where 'deleted' is 0, eliminating all records that were flagged as deleted (for instance, using UpdateAttributes() call).

4. Searching

4.1. Matching modes

There are the following matching modes available:

- SPH_MATCH_ALL, matches all query words (default mode);
- SPH_MATCH_ANY, matches any of the query words;
- SPH_MATCH_PHRASE, matches query as a phrase, requiring perfect match;
- SPH_MATCH_BOOLEAN, matches query as a boolean expression (see Section 4.2, “Boolean query syntax”);
- SPH_MATCH_EXTENDED, matches query as an expression in Sphinx internal query language (see Section 4.3, “Extended query syntax”). As of 0.9.9, this has been superceded by SPH_MATCH_EXTENDED2, providing additional functionality and better performance. The ident is retained for legacy application code that will continue to be compatible once Sphinx and its components, including the API, are upgraded.
- SPH_MATCH_EXTENDED2, matches query using the second version of the Extended matching mode.
- SPH_MATCH_FULLSCAN, matches query, forcibly using the "full scan" mode as below. NB, any query terms will be ignored, such that filters, filter-ranges and grouping will still be applied, but no text-matching.

The SPH_MATCH_FULLSCAN mode will be automatically activated in place of the specified matching mode when the following conditions are met:

1. The query string is empty (ie. its length is zero).
2. docinfo storage is set to `extern`.

In full scan mode, all the indexed documents will be considered as matching. Such queries will still apply filters, sorting, and group by, but will not perform any full-text searching. This can be useful to unify full-text and non-full-text searching code, or to offload SQL server (there are cases when Sphinx scans will perform better than analogous MySQL queries). An example of using the full scan mode might be to find posts in a forum. By

selecting the forum's user ID via `SetFilter()` but not actually providing any search text, Sphinx will match every document (i.e. every post) where `SetFilter()` would match - in this case providing every post from that user. By default this will be ordered by relevancy, followed by Sphinx document ID in ascending order (earliest first).

4.2. Boolean query syntax

Boolean queries allow the following special operators to be used:

- explicit operator AND:

```
hello & world
```

- operator OR:

```
hello | world
```

- operator NOT:

```
hello -world
hello !world
```

- grouping:

```
( hello world )
```

Here's an example query which uses all these operators:

Example 5. Boolean query example

```
( cat -dog ) | ( cat -mouse)
```

There always is implicit AND operator, so "hello world" query actually means "hello & world".

OR operator precedence is higher than AND, so "looking for cat | dog | mouse" means "looking for (cat | dog | mouse)" and *not* "(looking for cat) | dog | mouse".

Queries like "-dog", which implicitly include all documents from the collection, can not be evaluated. This is both for technical and performance reasons. Technically, Sphinx does not always keep a list of all IDs. Performance-wise, when the collection is huge (ie. 10-100M documents), evaluating such queries could take very long.

4.3. Extended query syntax

The following special operators and modifiers can be used when using the extended matching mode:

- operator OR:

```
hello | world
```

- operator NOT:

```
hello -world
hello !world
```

- field search operator:

```
@title hello @body world
```

- field position limit modifier (introduced in version 0.9.9-rc1):

```
@body[50] hello
```

- multiple-field search operator:

```
@(title,body) hello world
```

- all-field search operator:

```
@* hello
```

- phrase search operator:

```
"hello world"
```

- proximity search operator:

```
"hello world"~10
```

- quorum matching operator:

```
"the world is a wonderful place"/3
```

- strict order operator (aka operator "before"):

```
aaa << bbb << ccc
```

- exact form modifier (introduced in version 0.9.9-rc1):

```
raining =cats and =dogs
```

- field-start and field-end modifier (introduced in version 0.9.9-rc2):

```
^hello world$
```

Here's an example query that uses some of these operators:

Example 6. Extended matching mode: query example

```
"hello world" @title "example program"~5 @body python -(php|perl) @* code
```

The full meaning of this search is:

- Find the words 'hello' and 'world' adjacently in any field in a document;
- Additionally, the same document must also contain the words 'example' and 'program' in the title field, with up to, but not including, 10 words between the words in question; (E.g. "example PHP program" would be matched however "example script to introduce outside data into the correct context for your program" would not because two terms have 10 or more words between them)
- Additionally, the same document must contain the word 'python' in the body field, but not contain either 'php' or 'perl';
- Additionally, the same document must contain the word 'code' in any field.

There always is implicit AND operator, so "hello world" means that both "hello" and "world" must be present in matching document.

OR operator precedence is higher than AND, so "looking for cat | dog | mouse" means "looking for (cat | dog |

mouse)" and *not* "(looking for cat) | dog | mouse".

Field limit operator limits subsequent searching to a given field. Normally, query will fail with an error message if given field name does not exist in the searched index. However, that can be suppressed by specifying "@@relaxed" option at the very beginning of the query:

```
@@relaxed @nosuchfield my query
```

This can be helpful when searching through heterogeneous indexes with different schemas.

Field position limit, introduced in version 0.9.9-rc1, additionaly restricts the searching to first N position within given field (or fields). For example, "@body[50] hello" will **not** match the documents where the keyword 'hello' occurs at position 51 and below in the body.

Proximity distance is specified in words, adjusted for word count, and applies to all words within quotes. For instance, "cat dog mouse"~5 query means that there must be less than 8-word span which contains all 3 words, ie. "CAT aaa bbb ccc DOG eee fff MOUSE" document will *not* match this query, because this span is exactly 8 words long.

Quorum matching operator introduces a kind of fuzzy matching. It will only match those documents that pass a given threshold of given words. The example above ("the world is a wonderful place"/3) will match all documents that have at least 3 of the 6 specified words.

Strict order operator (aka operator "before"), introduced in version 0.9.9-rc2, will match the document only if its argument keywords occur in the document exactly in the query order. For instance, "black << cat" query (without quotes) will match the document "black and white cat" but *not* the "that cat was black" document. Order operator has the lowest priority. It can be applied both to just keywords and more complex expressions, ie. this is a valid query:

```
(bag of words) << "exact phrase" << red|green|blue
```

Exact form keyword modifier, introduced in version 0.9.9-rc1, will match the document only if the keyword occurred in exactly the specified form. The default behaviour is to match the document if the stemmed keyword matches. For instance, "runs" query will match both the document that contains "runs" *and* the document that contains "running", because both forms stem to just "run" - while "=runs" query will only match the first document. Exact form operator requires index_exact_words option to be enabled. This is a modifier that affects the keyword and thus can be used within operators such as phrase, proximity, and quorum operators.

Field-start and field-end keyword modifiers, introduced in version 0.9.9-rc2, will make the keyword match only if it occurred at the very start or the very end of a fulltext field, respectively. For instance, the query "^hello world$" (with quotes and thus combining phrase operator and start/end modifiers) will only match documents that contain at least one field that has exactly these two keywords.

Starting with 0.9.9-rc1, arbitrarily nested brackets and negations are allowed. However, the query must be possible to compute without involving an implicit list of all documents:

```
// correct query
aaa -(bbb -(ccc ddd))

// queries that are non-computable
-aaa
aaa | -bbb
```

4.4. Weighting

Specific weighting function (currently) depends on the search mode.

There are these major parts which are used in the weighting functions:

1. phrase rank,
2. statistical rank.

Phrase rank is based on a length of longest common subsequence (LCS) of search words between document body and query phrase. So if there's a perfect phrase match in some document then its phrase rank would be the highest possible, and equal to query words count.

Statistical rank is based on classic BM25 function which only takes word frequencies into account. If the word is rare in the whole database (ie. low frequency over document collection) or mentioned a lot in specific document (ie. high frequency over matching document), it receives more weight. Final BM25 weight is a floating point number between 0 and 1.

In all modes, per-field weighted phrase ranks are computed as a product of LCS multiplied by per-field weight speficifed by user. Per-field weights are integer, default to 1, and can not be set lower than 1.

In SPH_MATCH_BOOLEAN mode, no weighting is performed at all, every match weight is set to 1.

In SPH_MATCH_ALL and SPH_MATCH_PHRASE modes, final weight is a sum of weighted phrase ranks.

In SPH_MATCH_ANY mode, the idea is essentially the same, but it also adds a count of matching words in each field. Before that, weighted phrase ranks are additionally mutliplied by a value big enough to guarantee that higher phrase rank in **any** field will make the match ranked higher, even if it's field weight is low.

In SPH_MATCH_EXTENDED mode, final weight is a sum of weighted phrase ranks and BM25 weight, multiplied by 1000 and rounded to integer.

This is going to be changed, so that MATCH_ALL and MATCH_ANY modes use BM25 weights as well. This would improve search results in those match spans where phrase ranks are equal; this is especially useful for 1-word queries.

The key idea (in all modes, besides boolean) is that better subphrase matches are ranked higher, and perfect matches are pulled to the top. Author's experience is that this phrase proximity based ranking provides noticeably better search quality than any statistical scheme alone (such as BM25, which is commonly used in other search engines).

4.5. Sorting modes

There are the following result sorting modes available:

- SPH_SORT_RELEVANCE mode, that sorts by relevance in descending order (best matches first);
- SPH_SORT_ATTR_DESC mode, that sorts by an attribute in descending order (bigger attribute values first);
- SPH_SORT_ATTR_ASC mode, that sorts by an attribute in ascending order (smaller attribute values first);
- SPH_SORT_TIME_SEGMENTS mode, that sorts by time segments (last hour/day/week/month) in descending order, and then by relevance in descending order;
- SPH_SORT_EXTENDED mode, that sorts by SQL-like combination of columns in ASC/DESC order;
- SPH_SORT_EXPR mode, that sorts by an arithmetic expression.

SPH_SORT_RELEVANCE ignores any additional parameters and always sorts matches by relevance rank. All other modes require an additional sorting clause, with the syntax depending on specific mode. SPH_SORT_ATTR_ASC, SPH_SORT_ATTR_DESC and SPH_SORT_TIME_SEGMENTS modes require simply an attribute name. SPH_SORT_RELEVANCE is equivalent to sorting by "@weight DESC, @id ASC" in extended sorting mode, SPH_SORT_ATTR_ASC is equivalent to "attribute ASC, @weight DESC, @id ASC", and SPH_SORT_ATTR_DESC to "attribute DESC, @weight DESC, @id ASC" respectively.

SPH_SORT_TIME_SEGMENTS mode

In SPH_SORT_TIME_SEGMENTS mode, attribute values are split into so-called time segments, and then sorted by time segment first, and by relevance second.

The segments are calculated according to the *current timestamp* at the time when the search is performed, so the results would change over time. The segments are as follows:

- last hour,
- last day,
- last week,

- last month,
- last 3 months,
- everything else.

These segments are hardcoded, but it is trivial to change them if necessary.

This mode was added to support searching through blogs, news headlines, etc. When using time segments, recent records would be ranked higher because of segment, but withing the same segment, more relevant records would be ranked higher - unlike sorting by just the timestamp attribute, which would not take relevance into account at all.

SPH_SORT_EXTENDED mode

In SPH_SORT_EXTENDED mode, you can specify an SQL-like sort expression with up to 5 attributes (including internal attributes), eg:

```
@relevance DESC, price ASC, @id DESC
```

Both internal attributes (that are computed by the engine on the fly) and user attributes that were configured for this index are allowed. Internal attribute names must start with magic @-symbol; user attribute names can be used as is. In the example above, `@relevance` and `@id` are internal attributes and `price` is user-specified.

Known internal attributes are:

- @id (match ID)
- @weight (match weight)
- @rank (match weight)
- @relevance (match weight)
- @random (return results in random order)

`@rank` and `@relevance` are just additional aliases to `@weight`.

SPH_SORT_EXPR mode

Expression sorting mode lets you sort the matches by an arbitrary arithmetic expression, involving attribute values, internal attributes (@id and @weight), arithmetic operations, and a number of built-in functions. Here's an example:

```
$cl->SetSortMode ( SPH_SORT_EXPR,
        "@weight + ( user_karma + ln(pageviews) )*0.1" );
```

The following operators and functions are supported. They are mimiced after MySQL. The functions take a number of arguments depending on the specific function.

- Operators: +, -, *, /, <, > <=, >=, =, <>.
- Boolean operators: AND, OR, NOT.
- 0-argument functions: NOW().
- Unary (1-argument) functions: ABS(), CEIL(), FLOOR(), SIN(), COS(), LN(), LOG2(), LOG10(), EXP(), SQRT(), BIGINT().
- Binary (2-argument) functions: MIN(), MAX(), POW(), IDIV().
- Other functions: IF(), INTERVAL(), IN(), GEODIST().

Calculations can be performed in three different modes: (a) using single-precision, 32-bit IEEE 754 floating point values (the default), (b) using signed 32-bit integers, (c) using 64-bit signed integers. The expression parser will automatically switch to integer mode if there are no operations the result in a floating point value. Otherwise, it will use the default floating point mode. For instance, "a+b" will be computed using 32-bit integers if both arguments are 32-bit integers; or using 64-bit integers if both arguments are integers but one of them is 64-bit; or in floats otherwise. However, "a/b" or "sqrt(a)" will always be computed in floats, because these operations return non-integer result. To avoid the first, you can use IDIV(). Also, "a*b" will not be automatically promoted to 64-bit when the arguments are 32-bit. To enforce 64-bit results, you can use BIGINT(). (But note that if there are non-integer operations, BIGINT() will simply be ignored.)

Comparison operators (eg. = or <=) return 1.0 when the condition is true and 0.0 otherwise. For instance,

`(a=b)+3` will evaluate to 4 when attribute 'a' is equal to attribute 'b', and to 3 when 'a' is not. Unlike MySQL, the equality comparisons (ie. = and <> operators) introduce a small equality threshold (1e-6 by default). If the difference between compared values is within the threshold, they will be considered equal.

Boolean operators (AND, OR, NOT) were introduced in 0.9.9-rc2 and behave as usual. They are left-associative and have the least priority compared to other operators. NOT has more priority than AND and OR but nevertheless less than any other operator. AND and OR have the same priority so brackets use is recommended to avoid confusion in complex expressions.

All unary and binary functions are straightforward, they behave just like their mathematical counterparts. But `IF()` behavior needs to be explained in more detail. It takes 3 arguments, check whether the 1st argument is equal to 0.0, returns the 2nd argument if it is not zero, or the 3rd one when it is. Note that unlike comparison operators, `IF()` does **not** use a threshold! Therefore, it's safe to use comparison results as its 1st argument, but arithmetic operators might produce unexpected results. For instance, the following two calls will produce *different* results even though they are logically equivalent:

```
IF ( sqrt(3)*sqrt(3)-3<>0, a, b )
IF ( sqrt(3)*sqrt(3)-3, a, b )
```

In the first case, the comparison operator <> will return 0.0 (false) because of a threshold, and `IF()` will always return 'b' as a result. In the second one, the same `sqrt(3)*sqrt(3)-3` expression will be compared with zero *without* threshold by the `IF()` function itself. But its value will be slightly different from zero because of limited floating point calculations precision. Because of that, the comparison with 0.0 done by `IF()` will not pass, and the second variant will return 'a' as a result.

BIGINT() function, introduced in version 0.9.9-rc1, forcibly promotes the integer argument to 64-bit type, and does nothing on floating point argument. It's intended to help enforce evaluation of certain expressions (such as "a*b") in 64-bit mode even though all the arguments are 32-bit.

IDIV() functions performs an integer division on its 2 arguments. The result is integer as well, unlike "a/b" result.

IN(expr,val1,val2,...), introduced in version 0.9.9-rc1, takes 2 or more arguments, and returns 1 if 1st argument (expr) is equal to any of the other arguments (val1..valN), or 0 otherwise. Currently, all the checked values (but not the expression itself!) are required to be constant. (Its technically possible to implement arbitrary expressions too, and that might be implemented in the future.) Constants are pre-sorted and then binary search is used, so IN() even against a big arbitrary list of constants will be very quick. Starting with 0.9.9-rc2, first argument can also be a MVA attribute. In that case, IN() will return 1 if any of the MVA values is equal to any of the other arguments.

INTERVAL(expr,point1,point2,point3,...), introduced in version 0.9.9-rc1, takes 2 or more arguments, and returns the index of the argument that is less than the first argument: it returns 0 if expr<point1, 1 if point1<=expr<point2, and so on. It is required that point1<point2<...<pointN for this function to work correctly.

NOW(), introduced in version 0.9.9-rc1, is a helper function that returns current timestamp as a 32-bit integer.

GEODIST(lat1,long1,lat2,long2) function, introduced in version 0.9.9-rc2, computes geosphere distance between two given points specified by their coordinates. Note that both latitudes and longitudes must be in radians and the result will be in meters. You can use arbitrary expression as any of the four coordinates. An optimized path will be selected when one pair of the arguments refers directly to a pair attributes and the other one is constant.

4.6. Grouping (clustering) search results

Sometimes it could be useful to group (or in other terms, cluster) search results and/or count per-group match counts - for instance, to draw a nice graph of how much maching blog posts were there per each month; or to group Web search results by site; or to group matching forum posts by author; etc.

In theory, this could be performed by doing only the full-text search in Sphinx and then using found IDs to group on SQL server side. However, in practice doing this with a big result set (10K-10M matches) would typically kill performance.

To avoid that, Sphinx offers so-called grouping mode. It is enabled with SetGroupBy() API call. When grouping, all matches are assigned to different groups based on group-by value. This value is computed from specified attribute using one of the following built-in functions:

- SPH_GROUPBY_DAY, extracts year, month and day in YYYYMMDD format from timestamp;
- SPH_GROUPBY_WEEK, extracts year and first day of the week number (counting from year start) in YYYYNNN format from timestamp;
- SPH_GROUPBY_MONTH, extracts month in YYYYMM format from timestamp;
- SPH_GROUPBY_YEAR, extracts year in YYYY format from timestamp;
- SPH_GROUPBY_ATTR, uses attribute value itself for grouping.

The final search result set then contains one best match per group. Grouping function value and per-group match count are returned along as "virtual" attributes named **@group** and **@count** respectively.

The result set is sorted by group-by sorting clause, with the syntax similar to `SPH_SORT_EXTENDED` sorting clause syntax. In addition to `@id` and `@weight`, group-by sorting clause may also include:

- @group (groupby function value),
- @count (amount of matches in group).

The default mode is to sort by groupby value in descending order, ie. by `"@group desc"`.

On completion, `total_found` result parameter would contain total amount of matching groups over he whole index.

WARNING: grouping is done in fixed memory and thus its results are only approximate; so there might be more groups reported in `total_found` than actually present. `@count` might also be underestimated. To reduce inaccuracy, one should raise `max_matches`. If `max_matches` allows to store all found groups, results will be 100% correct.

For example, if sorting by relevance and grouping by `"published"` attribute with `SPH_GROUPBY_DAY` function, then the result set will contain

- one most relevant match per each day when there were any matches published,
- with day number and per-day match count attached,
- sorted by day number in descending order (ie. recent days first).

Starting with version 0.9.9-rc2, aggregate functions (AVG(), MIN(), MAX(), SUM()) are supported through SetSelect() API call when using GROUP BY.

4.7. Distributed searching

To scale well, Sphinx has distributed searching capabilities. Distributed searching is useful to improve query latency (ie. search time) and throughput (ie. max queries/sec) in multi-server, multi-CPU or multi-core environments. This is essential for applications which need to search through huge amounts data (ie. billions of records and terabytes of text).

The key idea is to horizontally partition (HP) searched data accross search nodes and then process it in parallel.

Partitioning is done manually. You should

- setup several instances of Sphinx programs (`indexer` and `searchd`) on different servers;
- make the instances index (and search) different parts of data;
- configure a special distributed index on some of the `searchd` instances;
- and query this index.

This index only contains references to other local and remote indexes - so it could not be directly reindexed, and you should reindex those indexes which it references instead.

When `searchd` receives a query against distributed index, it does the following:

1. connects to configured remote agents;
2. issues the query;
3. sequentially searches configured local indexes (while the remote agents are searching);

4. retrieves remote agents' search results;
5. merges all the results together, removing the duplicates;
6. sends the merged resuls to client.

From the application's point of view, there are no differences between searching through a regular index, or a distributed index at all. That is, distributed indexes are fully transparent to the application, and actually there's no way to tell whether the index you queried was distributed or local. (Even though as of 0.9.9 Sphinx does not allow to combine searching through distributed indexes with anything else, this constraint will be lifted in the future.)

Any `searchd` instance could serve both as a master (which aggregates the results) and a slave (which only does local searching) at the same time. This has a number of uses:

1. every machine in a cluster could serve as a master which searches the whole cluster, and search requests could be balanced between masters to achieve a kind of HA (high availability) in case any of the nodes fails;
2. if running within a single multi-CPU or multi-core machine, there would be only 1 searchd instance quering itself as an agent and thus utilizing all CPUs/core.

It is scheduled to implement better HA support which would allow to specify which agents mirror each other, do health checks, keep track of alive agents, load-balance requests, etc.

4.8. `searchd` query log format

`searchd` logs all succesfully executed search queries into query log file. Here's an example:

```
[Fri Jun 29 21:17:58 2007] 0.004 sec [all/0/rel 35254 (0,20)] [lj] test
[Fri Jun 29 21:20:34 2007] 0.024 sec [all/0/rel 19886 (0,20) @channel_id] [lj] test
```

This log format is as follows:

```
[query-date] query-time [match-mode/filters-count/sort-mode
    total-matches (offset,limit) @groupby-attr] [index-name] query
```

Match mode can take one of the following values:

- "all" for SPH_MATCH_ALL mode;
- "any" for SPH_MATCH_ANY mode;
- "phr" for SPH_MATCH_PHRASE mode;
- "bool" for SPH_MATCH_BOOLEAN mode;
- "ext" for SPH_MATCH_EXTENDED mode;
- "ext2" for SPH_MATCH_EXTENDED2 mode;
- "scan" if the full scan mode was used, either by being specified with SPH_MATCH_FULLSCAN, or if the query was empty (as documented under Matching Modes)

Sort mode can take one of the following values:

- "rel" for SPH_SORT_RELEVANCE mode;
- "attr-" for SPH_SORT_ATTR_DESC mode;
- "attr+" for SPH_SORT_ATTR_ASC mode;
- "tsegs" for SPH_SORT_TIME_SEGMENTS mode;
- "ext" for SPH_SORT_EXTENDED mode.

Additionally, if `searchd` was started with `--iostats`, there will be a block of data after where the index(es) searched are listed.

A query log entry might take the form of:

```
[Fri Jun 29 21:17:58 2007] 0.004 sec [all/0/rel 35254 (0,20)] [lj]
   [ios=6 kb=111.1 ms=0.5] test
```

This additional block is information regarding I/O operations in performing the search: the number of file I/O

operations carried out, the amount of data in kilobytes read from the index files and time spent on I/O operations (although there is a background processing component, the bulk of this time is the I/O operation time)

4.9. MySQL protocol support and SphinxQL

Starting with version 0.9.9-rc2, Sphinx searchd daemon supports MySQL binary network protocol and can be accessed with regular MySQL API. For instance, 'mysql' CLI client program works well. Here's an example of querying Sphinx using MySQL client:

```
$ mysql -P 9306
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 0.9.9-dev (r1734)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SELECT * FROM test1 WHERE MATCH('test')
    -> ORDER BY group_id ASC OPTION ranker=bm25;
+------+--------+----------+------------+
| id   | weight | group_id | date_added |
+------+--------+----------+------------+
|    4 |   1442 |        2 | 1231721236 |
|    2 |   2421 |      123 | 1231721236 |
|    1 |   2421 |      456 | 1231721236 |
+------+--------+----------+------------+
3 rows in set (0.00 sec)
```

Note that mysqld was not even running on the test machine. Everything was handled by searchd itself.

The new access method is supported *in addition* to native APIs which all still work perfectly well. In fact, both access methods can be used at the same time. Also, native API is still the default access method. MySQL protocol support needs to be additionally configured. This is a matter of 1-line config change, adding a new listener with mysql41 specified as a protocol:

```
listen = localhost:9306:mysql41
```

Just supporting the protocol and not the SQL syntax would be useless so Sphinx now also supports a tiny subset of SQL that we dubbed SphinxQL. Currently implemented statements are:

- SELECT
- SHOW WARNINGS
- SHOW STATUS
- SHOW META

SELECT syntax is based upon regular SQL but adds several Sphinx-specific extensions and has a few omissions (such as (currently) missing support for JOINs). Specifically,

- Column list clause. Column names, arbitrary expressions, and star ('*') are all allowed (ie. "SELECT @id, group_id*123+456 FROM test1" will work). Unlike in regular SQL, all computed expressions must be aliased with a valid identifier. Special names such as @id and @weight should currently be used with leading at-sign. This at-sign requirement will be lifted in the future.
- FROM clause. FROM clause should contain the list of indexes to search through. Unlike in regular SQL, comma means enumeration of full-text indexes as in Query() API call rather than JOIN.
- WHERE clause. This clause will map both to fulltext query and filters. Comparison operators (=, !=, <, >, <=, >=), IN(), AND, NOT, and BETWEEN are all supported and map directly to filters. OR is not supported yet but will be in the future. MATCH('query') is supported and maps to fulltext query. Query will be interpreted according to full-text query language rules. There must be at most one MATCH() in the clause.
- GROUP BY clause. Currently only supports grouping by a single column. The column however can be a computed expression:

```
SELECT *, group_id*1000+article_type AS gkey FROM example GROUP BY gkey
```

Aggregate functions (AVG(), MIN(), MAX(), SUM()) in column list clause are supported. Arguments to aggregate functions can be either plain attributes or arbitrary expressions. COUNT(*) is implicitly supported as using GROUP BY will add @count column to result set. Explicit support might be added in the future. COUNT(DISTINCT attr) is supported. Currently there can be at most one COUNT(DISTINCT) per query and an argument needs to be an attribute. Both current restrictions on COUNT(DISTINCT) might be lifted in the future.

```
SELECT *, AVG(price) AS avgprice, COUNT(DISTINCT storeid)
FROM products
WHERE MATCH('ipod')
GROUP BY vendorid
```

- WITHIN GROUP ORDER BY clause. This is a Sphinx specific extension that lets you control how the best row within a group will to be selected. The syntax matches that of regular ORDER BY clause:

```
SELECT *, INTERVAL(posted,NOW()-7*86400,NOW()-86400) AS timeseg
FROM example WHERE MATCH('my search query')
GROUP BY siteid
WITHIN GROUP ORDER BY @weight DESC
ORDER BY timeseg DESC, @weight DESC
```

- ORDER BY clause. Unlike in regular SQL, only column names (not expressions) are allowed and explicit ASC and DESC are required. The columns however can be computed expressions:

```
SELECT *, @weight*10+docboost AS skey FROM example ORDER BY skey
```

- LIMIT clause. Both LIMIT N and LIMIT M,N forms are supported. Unlike in regular SQL (but like in Sphinx API), an implicit LIMIT 0,20 is present by default.
- OPTION clause. This is a Sphinx specific extension that lets you control a number of per-query options. The syntax is:

```
OPTION <optionname>=<value> [ , ... ]
```

Supported options and respectively allowed values are:
  - 'ranker' - any of 'proximity_bm25', 'bm25', 'none', 'wordcount', 'proximity', 'matchany', or 'fieldmask'
  - 'max_matches' - integer (per-query max matches value)
  - 'cutoff' - integer (max found matches threshold)
  - 'max_query_time' - integer (max search time threshold, msec)
  - 'retry_count' - integer (distributed retries count)
  - 'retry_delay' - integer (distributed retry delay, msec)

Example:

```
SELECT * FROM test WHERE MATCH('@title hello @body world')
OPTION ranker=bm25, max_matches=3000
```

SHOW WARNINGS statement can be used to retrieve the warning produced by the latest query. The error message will be returned along with the query itself:

```
mysql> SELECT * FROM test1 WHERE MATCH('@@title hello') \G
ERROR 1064 (42000): index test1: syntax error, unexpected TOK_FIELDLIMIT
near '@title hello'

mysql> SELECT * FROM test1 WHERE MATCH('@title -hello') \G
ERROR 1064 (42000): index test1: query is non-computable (single NOT operator)

mysql> SELECT * FROM test1 WHERE MATCH('"test doc"/3') \G
*************************** 1. row ***************************
        id: 4
    weight: 2500
  group_id: 2
date_added: 1231721236
1 row in set, 1 warning (0.00 sec)
```

```
mysql> SHOW WARNINGS \G
*************************** 1. row ***************************
  Level: warning
   Code: 1000
Message: quorum threshold too high (words=2, thresh=3); replacing quorum operator
         with AND operator
1 row in set (0.00 sec)
```

SHOW STATUS shows a number of useful performance counters. IO and CPU counters will only be available if searchd was started with --iostats and --cpustats switches respectively.

```
mysql> SHOW STATUS;
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| uptime             | 216   |
| connections        | 3     |
| maxed_out          | 0     |
| command_search     | 0     |
| command_excerpt    | 0     |
| command_update     | 0     |
| command_keywords   | 0     |
| command_persist    | 0     |
| command_status     | 0     |
| agent_connect      | 0     |
| agent_retry        | 0     |
| queries            | 10    |
| dist_queries       | 0     |
| query_wall         | 0.075 |
| query_cpu          | OFF   |
| dist_wall          | 0.000 |
| dist_local         | 0.000 |
| dist_wait          | 0.000 |
| query_reads        | OFF   |
| query_readkb       | OFF   |
| query_readtime     | OFF   |
| avg_query_wall     | 0.007 |
| avg_query_cpu      | OFF   |
| avg_dist_wall      | 0.000 |
| avg_dist_local     | 0.000 |
| avg_dist_wait      | 0.000 |
| avg_query_reads    | OFF   |
| avg_query_readkb   | OFF   |
| avg_query_readtime | OFF   |
+--------------------+-------+
29 rows in set (0.00 sec)
```

SHOW META shows additional meta-information about the latest query such as query time and keyword statistics:

```
mysql> SELECT * FROM test1 WHERE MATCH('test|one|two');
+------+--------+----------+------------+
| id   | weight | group_id | date_added |
+------+--------+----------+------------+
|    1 |   3563 |      456 | 1231721236 |
|    2 |   2563 |      123 | 1231721236 |
|    4 |   1480 |        2 | 1231721236 |
+------+--------+----------+------------+
3 rows in set (0.01 sec)

mysql> SHOW META;
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| total         | 3     |
| total_found   | 3     |
```

```
| time          | 0.005 |
| keyword[0]    | test  |
| docs[0]       | 3     |
| hits[0]       | 5     |
| keyword[1]    | one   |
| docs[1]       | 1     |
| hits[1]       | 2     |
| keyword[2]    | two   |
| docs[2]       | 1     |
| hits[2]       | 2     |
+---------------+-------+
12 rows in set (0.00 sec)
```

5. Command line tools reference

As mentioned elsewhere, Sphinx is not a single program called 'sphinx', but a collection of 4 separate programs which collectively form Sphinx. This section covers these tools and how to use them.

5.1. `indexer` command reference

`indexer` is the first of the two principle tools as part of Sphinx. Invoked from either the command line directly, or as part of a larger script, `indexer` is solely responsible for gathering the data that will be searchable.

The calling syntax for `indexer` is as follows:

```
indexer [OPTIONS] [indexname1 [indexname2 [...]]]
```

Essentially you would list the different possible indexes (that you would later make available to search) in `sphinx.conf`, so when calling `indexer`, as a minimum you need to be telling it what index (or indexes) you want to index.

If `sphinx.conf` contained details on 2 indexes, `mybigindex` and `mysmallindex`, you could do the following:

```
$ indexer mybigindex
$ indexer mysmallindex mybigindex
```

As part of the configuration file, `sphinx.conf`, you specify one or more indexes for your data. You might call `indexer` to reindex one of them, ad-hoc, or you can tell it to process all indexes - you are not limited to calling just one, or all at once, you can always pick some combination of the available indexes.

The majority of the options for `indexer` are given in the configuration file, however there are some options you might need to specify on the command line as well, as they can affect how the indexing operation is performed. These options are:

- `--config <file>` (`-c <file>` for short) tells `indexer` to use the given file as its configuration. Normally, it will look for `sphinx.conf` in the installation directory (e.g. `/usr/local/sphinx/etc/sphinx.conf` if installed into `/usr/local/sphinx`), followed by the current directory you are in when calling `indexer` from the shell. This is most of use in shared environments where the binary files are installed somewhere like `/usr/local/sphinx/` but you want to provide users with the ability to make their own custom Sphinx set-ups, or if you want to run multiple instances on a single server. In cases like those you could allow them to create their own `sphinx.conf` files and pass them to `indexer` with this option. For example:

  ```
  $ indexer --config /home/myuser/sphinx.conf myindex
  ```

- `--all` tells `indexer` to update every index listed in `sphinx.conf`, instead of listing individual indexes. This would be useful in small configurations, or `cron`-type or maintenance jobs where the entire index set will get rebuilt each day, or week, or whatever period is best. Example usage:

  ```
  $ indexer --config /home/myuser/sphinx.conf --all
  ```

- `--rotate` is used for rotating indexes. Unless you have the situation where you can take the search

function offline without troubling users, you will almost certainly need to keep search running whilst indexing new documents. `--rotate` creates a second index, parallel to the first (in the same place, simply including `.new` in the filenames). Once complete, `indexer` notifies `searchd` via sending the `SIGHUP` signal, and `searchd` will attempt to rename the indexes (renaming the existing ones to include `.old` and renaming the `.new` to replace them), and then start serving from the newer files. Depending on the setting of seamless_rotate, there may be a slight delay in being able to search the newer indexes. Example usage:

```
$ indexer --rotate --all
```

- `--quiet` tells `indexer` not to output anything, unless there is an error. Again, most used for `cron`-type, or other script jobs where the output is irrelevant or unnecessary, except in the event of some kind of error. Example usage:

```
$ indexer --rotate --all --quiet
```

- `--noprogress` does not display progress details as they occur; instead, the final status details (such as documents indexed, speed of indexing and so on are only reported at completion of indexing. In instances where the script is not being run on a console (or 'tty'), this will be on by default. Example usage:

```
$ indexer --rotate --all --noprogress
```

- `--buildstops <outputfile.text> <N>` reviews the index source, as if it were indexing the data, and produces a list of the terms that are being indexed. In other words, it produces a list of all the searchable terms that are becoming part of the index. Note; it does not update the index in question, it simply processes the data 'as if' it were indexing, including running queries defined with `sql_query_pre` or `sql_query_post`. `outputfile.txt` will contain the list of words, one per line, sorted by frequency with most frequent first, and `N` specifies the maximum number of words that will be listed; if sufficiently large to encompass every word in the index, only that many words will be returned. Such a dictionary list could be used for client application features around "Did you mean..." functionality, usually in conjunction with `--buildfreqs`, below. Example:

```
$ indexer myindex --buildstops word_freq.txt 1000
```

This would produce a document in the current directory, `word_freq.txt` with the 1,000 most common words in 'myindex', ordered by most common first. Note that the file will pertain to the last index indexed when specified with multiple indexes or `--all` (i.e. the last one listed in the configuration file)

- `--buildfreqs` works with `--buildstops` (and is ignored if `--buildstops` is not specified). As `--buildstops` provides the list of words used within the index, `--buildfreqs` adds the quantity present in the index, which would be useful in establishing whether certain words should be considered stopwords if they are too prevalent. It will also help with developing "Did you mean..." features where you can how much more common a given word compared to another, similar one. Example:

```
$ indexer myindex --buildstops word_freq.txt 1000 --buildfreqs
```

This would produce the `word_freq.txt` as above, however after each word would be the number of times it occurred in the index in question.

- `--merge <dst-index> <src-index>` is used for physically merging indexes together, for example if you have a main+delta scheme, where the main index rarely changes, but the delta index is rebuilt frequently, and `--merge` would be used to combine the two. The operation moves from right to left - the contents of `src-index` get examined and physically combined with the contents of `dst-index` and the result is left in `dst-index`. In pseudo-code, it might be expressed as: `dst-index += src-index` An example:

```
$ indexer --merge main delta --rotate
```

In the above example, where the main is the master, rarely modified index, and delta is the less frequently modified one, you might use the above to call `indexer` to combine the contents of the delta into the main index and rotate the indexes.

- `--merge-dst-range <attr> <min> <max>` runs the filter range given upon merging. Specifically, as the merge is applied to the destination index (as part of `--merge`, and is ignored if `--merge` is not specified), `indexer` will also filter the documents ending up in the destination index, and only documents will pass through the filter given will end up in the final index. This could be used for example, in an index where there is a 'deleted' attribute, where 0 means 'not deleted'. Such an index could be merged with:

  ```
  $ indexer --merge main delta --merge-dst-range deleted 0 0
  ```

  Any documents marked as deleted (value 1) would be removed from the newly-merged destination index. It can be added several times to the command line, to add successive filters to the merge, all of which must be met in order for a document to become part of the final index.

5.2. `searchd` command reference

`searchd` is the second of the two principle tools as part of Sphinx. `searchd` is the part of the system which actually handles searches; it functions as a server and is responsible for receiving queries, processing them and returning a dataset back to the different APIs for client applications.

Unlike `indexer`, `searchd` is not designed to be run either from a regular script or command-line calling, but instead either as a daemon to be called from init.d (on Unix/Linux type systems) or to be called as a service (on Windows-type systems), so not all of the command line options will always apply, and so will be build-dependent.

Calling `searchd` is simply a case of:

```
$ searchd [OPTIONS]
```

The options available to `searchd` on all builds are:

- `--help` (`-h` for short) lists all of the parameters that can be called in your particular build of `searchd`.
- `--config <file>` (`-c <file>` for short) tells `searchd` to use the given file as its configuration, just as with `indexer` above.
- `--stop` is used to stop `searchd`, using the details of the PID file as specified in the `sphinx.conf` file, so you may also need to confirm to `searchd` which configuration file to use with the `--config` option. NB, calling `--stop` will also make sure any changes applied to the indexes with `UpdateAttributes()` will be applied to the index files themselves. Example:

  ```
  $ searchd --config /home/myuser/sphinx.conf --stop
  ```

- `--status` command is used to query running `searchd` instance status, using the connection details from the (optionally) provided configuration file. It will try to connect to the running instance using the first configured UNIX socket or TCP port. On success, it will query for a number of status and performance counter values and print them. You can use Status() API call to access the very same counters from your application. Examples:

  ```
  $ searchd --status
  $ searchd --config /home/myuser/sphinx.conf --status
  ```

- `--pidfile` is used to explicitly state a PID file, where the process information is stored regarding `searchd`, used for inter-process communications (for example, `indexer` will need to know the PID to contact `searchd` for rotating indexes). Normally, `searchd` would use a PID if running in regular mode (i.e. not with `--console`), but it is possible that you will be running it in console mode whilst the index is being updated and rotated, for which a PID file will be needed.

  ```
  $ searchd --config /home/myuser/sphinx.conf --pidfile /home/myuser/sphinx.pid
  ```

- `--console` is used to force `searchd` into console mode; typically it will be running as a conventional server application, and will aim to dump information into the log files (as specified in `sphinx.conf`). Sometimes though, when debugging issues in the configuration or the daemon itself, or trying to diagnose

hard-to-track-down problems, it may be easier to force it to dump information directly to the console/command line from which it is being called. Running in console mode also means that the process will not be forked (so searches are done in sequence) and logs will not be written to. (It should be noted that console mode is not the intended method for running `searchd`) You can invoke it as such:

```
$ searchd --config /home/myuser/sphinx.conf --console
```

- `--iostats` is used in conjuction with the logging options (the `query_log` will need to have been activated in `sphinx.conf`) to provide more detailed information on a per-query basis as to the input/output operations carried out in the course of that query, with a slight performance hit and of course bigger logs. Further details are available under the query log format section. You might start `searchd` thus:

```
$ searchd --config /home/myuser/sphinx.conf --iostats
```

- `--cpustats` is used to provide actual CPU time report (in addition to wall time) in both query log file (for every given query) and status report (aggregated). It depends on clock_gettime() system call and might therefore be unavailable on certain systems. You might start `searchd` thus:

```
$ searchd --config /home/myuser/sphinx.conf --cpustats
```

- `--port portnumber` (`-p` for short) is used to specify the post that `searchd` should listen on, usually for debugging purposes. This will usually default to 9312, but sometimes you need to run it on a different port. Specifying it on the command line will override anything specified in the configuration file. The valid range is 0 to 65535, but ports numbered 1024 and below usually require a privileged account in order to run. An example of usage:

```
$ searchd --port 9313
```

- `--index <index>` forces this instance of `searchd` only to serve the specified index. Like `--port`, above, this is usually for debugging purposes; more long-term changes would generally be applied to the configuration file itself. Example usage:

```
$ searchd --index myindex
```

There are some options for `searchd` that are specific to Windows platforms, concerning handling as a service, are only be available on Windows binaries.

Note that on Windows searchd will default to `--console` mode, unless you install it as a service.

- `--install` installs `searchd` as a service into the Microsoft Management Console (Control Panel / Administrative Tools / Services). Any other parameters specified on the command line, where `--install` is specified will also become part of the command line on future starts of the service. For example, as part of calling `searchd`, you will likely also need to specify the configuration file with `--config`, and you would do that as well as specifying `--install`. Once called, the usual start/stop facilities will become available via the management console, so any methods you could use for starting, stopping and restarting services would also apply to `searchd`. Example:

```
C:\WINDOWS\system32> C:\Sphinx\bin\searchd.exe --install
   --config C:\Sphinx\sphinx.conf
```

  If you wanted to have the I/O stats every time you started `searchd`, you would specify its option on the same line as the `--install` command thus:

```
C:\WINDOWS\system32> C:\Sphinx\bin\searchd.exe --install
   --config C:\Sphinx\sphinx.conf --iostats
```

- `--delete` removes the service from the Microsoft Management Console and other places where services are registered, after previously installed with `--install`. Note, this does not uninstall the software or delete the indexes. It means the service will not be called from the services systems, and will not be started on the machine's next start. If currently running as a service, the current instance will not be

terminated (until the next reboot, or `searchd` is called with `--stop`). If the service was installed with a custom name (with `--servicename`), the same name will need to be specified with `--servicename` when calling to uninstall. Example:

```
C:\WINDOWS\system32> C:\Sphinx\bin\searchd.exe --delete
```

- `--servicename <name>` applies the given name to `searchd` when installing or deleting the service, as would appear in the Management Console; this will default to searchd, but if being deployed on servers where multiple administrators may log into the system, or a system with multiple `searchd` instances, a more descriptive name may be applicable. Note that unless combined with `--install` or `--delete`, this option does not do anything. Example:

```
C:\WINDOWS\system32> C:\Sphinx\bin\searchd.exe --install
   --config C:\Sphinx\sphinx.conf --servicename SphinxSearch
```

- `--ntservice` is the option that is passed by the Management Console to `searchd` to invoke it as a service on Windows platforms. It would not normally be necessary to call this directly; this would normally be called by Windows when the service would be started, although if you wanted to call this as a regular service from the command-line (as the complement to `--console`) you could do so in theory.

Last but not least, as every other daemon, `searchd` supports a number of signals.

SIGTERM
: Initiates a clean shutdown. New queries will not be handled; but queries that are already started will not be forcibly interrupted.

SIGHUP
: Initiates index rotation. Depending on the value of seamless_rotate setting, new queries might be shortly stalled; clients will receive temporary errors.

SIGUSR1
: Forces reopen of searchd log and query log files, letting you implement log file rotation.

5.3. `search` command reference

`search` is one of the helper tools within the Sphinx package. Whereas `searchd` is responsible for searches in a server-type environment, `search` is aimed at testing the index from the command line, and testing the index quickly without building a framework to make the connection to the server and process its response.

Note: `search` is not intended to be deployed as part of a client application; it is strongly recommended you do not write an interface to `search` instead of `searchd`, and none of the bundled client APIs support this method. (In any event, `search` will reload files each time, whereas `searchd` will cache them in memory for performance.)

That said, many types of query that you could build in the APIs could also be made with `search`, however for very complex searches it may be easier to construct them using a small script and the corresponding API. Additionally, some newer features may be available in the `searchd` system that have not yet been brought into `search`.

The calling syntax for `search` is as follows:

```
search [OPTIONS] word1 [word2 [word3 [...]]]
```

When calling `search`, it is not necessary to have `searchd` running; simply that the account running `search` has read access to the configuration file and the location and files of the indexes.

The default behaviour is to apply a search for word1 (AND word2 AND word3... as specified) to all fields in all indexes as given in the configuration file. If constructing the equivalent in the API, this would be the equivalent to passing `SPH_MATCH_ALL` to `SetMatchMode`, and specifying `*` as the indexes to query as part of `Query`.

There are many options available to `search`. Firstly, the general options:

- `--config <file>` (`-c <file>` for short) tells `search` to use the given file as its configuration, just as with

indexer above.

- --index <index> (-i <index> for short) tells search to limit searching to the specified index only; normally it would attempt to search all of the physical indexes listed in sphinx.conf, not any distributed ones.
- --stdin tells search to accept the query from the standard input, rather than the command line. This can be useful for testing purposes whereby you could feed input via pipes and from scripts.

Options for setting matches:

- --any (-a for short) changes the matching mode to match any of the words as part of the query (word1 OR word2 OR word3). In the API this would be equivalent to passing SPH_MATCH_ANY to SetMatchMode.
- --phrase (-p for short) changes the matching mode to match all of the words as part of the query, and do so in the phrase given (not including punctuation). In the API this would be equivalent to passing SPH_MATCH_PHRASE to SetMatchMode.
- --boolean (-b for short) changes the matching mode to Boolean matching. Note if using Boolean syntax matching on the command line, you may need to escape the symbols (with a backslash) to avoid the shell/command line processor applying them, such as ampersands being escaped on a Unix/Linux system to avoid it forking to the search process, although this can be resolved by using --stdin, as below. In the API this would be equivalent to passing SPH_MATCH_BOOLEAN to SetMatchMode.
- --ext (-e for short) changes the matching mode to Extended matching. In the API this would be equivalent to passing SPH_MATCH_EXTENDED to SetMatchMode, and it should be noted that use of this mode is being discouraged in favour of Extended2, below.
- --ext2 (-e2 for short) changes the matching mode to Extended matching, version 2. In the API this would be equivalent to passing SPH_MATCH_EXTENDED2 to SetMatchMode, and it should be noted that use of this mode is being recommended in favour of Extended, due to being more efficient and providing other features.
- --filter <attr> <v> (-f <attr> <v> for short) filters the results such that only documents where the attribute given (attr) matches the value given (v). For example, --filter deleted 0 only matches documents with an attribute called 'deleted' where its value is 0. You can also add multiple filters on the command line, by specifying multiple --filter multiple times, however if you apply a second filter to an attribute it will override the first defined filter.

Options for handling the results:

- --limit <count> (-l count for short) limits the total number of matches back to the number given. If a 'group' is specified, this will be the number of grouped results. This defaults to 20 results if not specified (as do the APIs)
- --offset <count> (-o <count> for short) offsets the result list by the number of places set by the count; this would be used for pagination through results, where if you have 20 results per 'page', the second page would begin at offset 20, the third page at offset 40, etc.
- --group <attr> (-g <attr> for short) specifies that results should be grouped together based on the attribute specified. Like the GROUP BY clause in SQL, it will combine all results where the attribute given matches, and returns a set of results where each returned result is the best from each group. Unless otherwise specified, this will be the best match on relevance.
- --groupsort <expr> (-gs <expr> for short) instructs that when results are grouped with -group, the expression given in <expr> shall determine the order of the groups. Note, this does not specify which is the best item within the group, only the order in which the groups themselves shall be returned.
- --sortby <clause> (-s <clause> for short) specifies that results should be sorted in the order listed in <clause>. This allows you to specify the order you wish results to be presented in, ordering by different columns. For example, you could say --sortby "@weight DESC entrytime DESC" to sort entries first by weight (or relevance) and where two or more entries have the same weight, to then sort by the time with the highest time (newest) first. You will usually need to put the items in quotes (--sortby "@weight DESC") or use commas (--sortby @weight,DESC) to avoid the items being treated separately. Additionally, like the regular sorting modes, if --group (grouping) is being used, this will state how to establish the best match within each group.
- --sortexpr expr (-S expr for short) specifies that the search results should be presented in an order determined by an arithmetic expression, stated in expr. For example: --sortexpr "@weight + (user_karma + ln(pageviews))*0.1" (again noting that this will have to be quoted to avoid the shell dealing with the asterisk). Extended sort mode is discussed in more detail under the SPH_SORT_EXTENDED entry under the Sorting modes chapter of the manual.
- --sort=date specifies that the results should be sorted by descending (i.e. most recent first) date. This

requires that there is an attribute in the index that is set as a timestamp.
- `--rsort=date` specifies that the results should be sorted by ascending (i.e. oldest first) date. This requires that there is an attribute in the index that is set as a timestamp.
- `--sort=ts` specifies that the results should be sorted by timestamp in groups; it will return all of the documents whose timestamp is within the last hour, then sorted within that bracket for relevance. After, it would return the documents from the last day, sorted by relevance, then the last week and then the last month. It is discussed in more detail under the `SPH_SORT_TIME_SEGMENTS` entry under the Sorting modes chapter of the manual.

Other options:

- `--noinfo` (`-q` for short) instructs `search` not to look-up data in your SQL database. Specifically, for debugging with MySQL and `search`, you can provide it with a query to look up the full article based on the returned document ID. It is explained in more detail under the sql_query_info directive.

5.4. `spelldump` command reference

`spelldump` is one of the helper tools within the Sphinx package.

It is used to extract the contents of a dictionary file that uses `ispell` or `MySpell` format, which can help build word lists for *wordforms* - all of the possible forms are pre-built for you.

Its general usage is:

```
spelldump [options] <dictionary> <affix> [result] [locale-name]
```

The two main parameters are the dictionary's main file and its affix file; usually these are named as `[language-prefix].dict` and `[language-prefix].aff` and will be available with most common Linux distributions, as well as various places online.

`[result]` specifies where the dictionary data should be output to, and `[locale-name]` additionally specifies the locale details you wish to use.

There is an additional option, `-c [file]`, which specifies a file for case conversion details.

Examples of its usage are:

```
spelldump en.dict en.aff
spelldump ru.dict ru.aff ru.txt ru_RU.CP1251
spelldump ru.dict ru.aff ru.txt .1251
```

The results file will contain a list of all the words in the dictionary in alphabetic order, output in the format of a wordforms file, which you can use to customise for your specific circumstances. An example of the result file:

```
zone > zone
zoned > zoned
zoning > zoning
```

5.5. `indextool` command reference

`indextool` is one of the helper tools within the Sphinx package, introduced in version 0.9.9-rc2. It is used to dump miscellaneous debug information about the physical index. (Additional functionality such as index verification is planned in the future, hence the indextool name rather than just indexdump.) Its general usage is:

```
indextool <command> [options]
```

The only currently available option applies to all commands and lets you specify the configuration file:

- `--config <file>` (`-c <file>` for short) overrides the built-in config file names.

The commands are as follows:

- `--dumpheader FILENAME.sph` quickly dumps the provided index header file without touching any other index files or even the configuration file. The report provides a breakdown of all the index settings, in particular the entire attribute and field list. Prior to 0.9.9-rc2, this command was present in CLI search utility.
- `--dumpheader INDEXNAME` dumps index header by index name with looking up the header path in the configuration file.
- `--dumpdocids INDEXNAME` dumps document IDs by index name. It takes the data from attribute (.spa) file and therefore requires docinfo=extern to work.
- `--dumphitlist INDEXNAME KEYWORD` dumps all the hits (occurences) of a given keyword in a given index.

6. API reference

There is a number of native searchd client API implementations for Sphinx. As of time of this writing, we officially support our own PHP, Python, and Java implementations. There also are third party free, open-source API implementations for Perl, Ruby, and C++.

The reference API implementation is in PHP, because (we believe) Sphinx is most widely used with PHP than any other language. This reference documentation is in turn based on reference PHP API, and all code samples in this section will be given in PHP.

However, all other APIs provide the same methods and implement the very same network protocol. Therefore the documentation does apply to them as well. There might be minor differences as to the method naming conventions or specific data structures used. But the provided functionality must not differ across languages.

6.1. General API functions

6.1.1. GetLastError

Prototype: function GetLastError()

Returns last error message, as a string, in human readable format. If there were no errors during the previous API call, empty string is returned.

You should call it when any other function (such as Query()) fails (typically, the failing function returns false). The returned string will contain the error description.

The error message is *not* reset by this call; so you can safely call it several times if needed.

6.1.2. GetLastWarning

Prototype: function GetLastWarning ()

Returns last warning message, as a string, in human readable format. If there were no warnings during the previous API call, empty string is returned.

You should call it to verify whether your request (such as Query()) was completed but with warnings. For instance, search query against a distributed index might complete succesfully even if several remote agents timed out. In that case, a warning message would be produced.

The warning message is *not* reset by this call; so you can safely call it several times if needed.

6.1.3. SetServer

Prototype: function SetServer ($host, $port)

Sets `searchd` host name and TCP port. All subsequent requests will use the new host and port settings. Default host and port are 'localhost' and 9312, respectively.

6.1.4. SetRetries

Prototype: function SetRetries ($count, $delay=0)

Sets distributed retry count and delay.

On temporary failures `searchd` will attempt up to `$count` retries per agent. `$delay` is the delay between the retries, in milliseconds. Retries are disabled by default. Note that this call will **not** make the API itself retry on temporary failure; it only tells `searchd` to do so. Currently, the list of temporary failures includes all kinds of connect() failures and maxed out (too busy) remote agents.

6.1.5. SetConnectTimeout

Prototype: function SetConnectTimeout ($timeout)

Sets the time allowed to spend connecting to the server before giving up.

Under some circumstances, the server can be delayed in responding, either due to network delays, or a query backlog. In either instance, this allows the client application programmer some degree of control over how their program interacts with `searchd` when not available, and can ensure that the client application does not fail due to exceeding the script execution limits (especially in PHP).

In the event of a failure to connect, an appropriate error code should be returned back to the application in order for application-level error handling to advise the user.

6.1.6. SetArrayResult

Prototype: function SetArrayResult ($arrayresult)

PHP specific. Controls matches format in the search results set (whether matches should be returned as an array or a hash).

`$arrayresult` argument must be boolean. If `$arrayresult` is `false` (the default mode), matches will returned in PHP hash format with document IDs as keys, and other information (weight, attributes) as values. If `$arrayresult` is true, matches will be returned as a plain array with complete per-match information including document ID.

Introduced along with GROUP BY support on MVA attributes. Group-by-MVA result sets may contain duplicate document IDs. Thus they need to be returned as plain arrays, because hashes will only keep one entry per document ID.

6.1.7. IsConnectError

Prototype: function IsConnectError ()

Checks whether the last error was a network error on API side, or a remote error reported by searchd. Returns true if the last connection attempt to searchd failed on API side, false otherwise (if the error was remote, or there were no connection attempts at all). Introduced in version 0.9.9-rc1.

6.2. General query settings

6.2.1. SetLimits

Prototype: function SetLimits ($offset, $limit, $max_matches=0, $cutoff=0)

Sets offset into server-side result set (`$offset`) and amount of matches to return to client starting from that offset (`$limit`). Can additionally control maximum server-side result set size for current query (`$max_matches`) and the threshold amount of matches to stop searching at (`$cutoff`). All parameters must be non-negative integers.

First two parameters to SetLimits() are identical in behavior to MySQL LIMIT clause. They instruct `searchd` to return at most `$limit` matches starting from match number `$offset`. The default offset and limit settings are 0 and 20, that is, to return first 20 matches.

`max_matches` setting controls how much matches `searchd` will keep in RAM while searching. **All** matching documents will be normally processed, ranked, filtered, and sorted even if `max_matches` is set to 1. But only best N documents are stored in memory at any given moment for performance and RAM usage reasons, and this

setting controls that N. Note that there are **two** places where `max_matches` limit is enforced. Per-query limit is controlled by this API call, but there also is per-server limit controlled by `max_matches` setting in the config file. To prevent RAM usage abuse, server will not allow to set per-query limit higher than the per-server limit.

You can't retrieve more than `max_matches` matches to the client application. The default limit is set to 1000. Normally, you must not have to go over this limit. One thousand records is enough to present to the end user. And if you're thinking about pulling the results to application for further sorting or filtering, that would be **much** more efficient if performed on Sphinx side.

`$cutoff` setting is intended for advanced performance control. It tells `searchd` to forcibly stop search query once `$cutoff` matches had been found and processed.

6.2.2. SetMaxQueryTime

Prototype: function SetMaxQueryTime ($max_query_time)

Sets maximum search query time, in milliseconds. Parameter must be a non-negative integer. Default valus is 0 which means "do not limit".

Similar to `$cutoff` setting from SetLimits(), but limits elapsed query time instead of processed matches count. Local search queries will be stopped once that much time has elapsed. Note that if you're performing a search which queries several local indexes, this limit applies to each index separately.

6.2.3. SetOverride

Prototype: function SetOverride ($attrname, $attrtype, $values)

Sets temporary (per-query) per-document attribute value overrides. Only supports scalar attributes. $values must be a hash that maps document IDs to overridden attribute values. Introduced in version 0.9.9-rc1.

Override feature lets you "temporary" update attribute values for some documents within a single query, leaving all other queries unaffected. This might be useful for personalized data. For example, assume you're implementing a personalized search function that wants to boost the posts that the user's friends recommend. Such data is not just dynamic, but also personal; so you can't simply put it in the index because you don't want everyone's searches affected. Overrides, on the other hand, are local to a single query and invisible to everyone else. So you can, say, setup a "friends_weight" value for every document, defaulting to 0, then temporary override it with 1 for documents 123, 456 and 789 (recommended by exactly the friends of current user), and use that value when ranking.

6.2.4. SetSelect

Prototype: function SetSelect ($clause)

Sets the select clause, listing specific attributes to fetch, and expressions to compute and fetch. Clause syntax mimics SQL. Introduced in version 0.9.9-rc1.

SetSelect() is very similar to the part of a typical SQL query between SELECT and FROM. It lets you choose what attributes (columns) to fetch, and also what expressions over the columns to compute and fetch. A certain difference from SQL is that expressions **must** always be aliased to a correct identifier (consisting of letters and digits) using 'AS' keyword. SQL also lets you do that but does not require to. Sphinx enforces aliases so that the computation results can always be returned under a "normal" name in the result set, used in other clauses, etc.

Everything else is basically identical to SQL. Star ('*') is supported. Functions are supported. Arbitrary amount of expressions is supported. Computed expressions can be used for sorting, filtering, and grouping, just as the regular attributes.

Starting with version 0.9.9-rc2, aggregate functions (AVG(), MIN(), MAX(), SUM()) are supported when using GROUP BY.

Expression sorting (Section 4.5, “SPH_SORT_EXPR mode”) and geodistance functions (Section 6.4.5, “SetGeoAnchor”) are now internally implemented using this computed expressions mechanism, using magic names '@expr' and '@geodist' respectively.

Example:

```
$cl->SetSelect ( "*, @weight+(user_karma+ln(pageviews))*0.1 AS myweight" );
$cl->SetSelect ( "exp_years, salary_gbp*{$gbp_usd_rate} AS salary_usd,
   IF(age>40,1,0) AS over40" );
$cl->SetSelect ( "*, AVG(price) AS avgprice" );
```

6.3. Full-text search query settings

6.3.1. SetMatchMode

Prototype: function SetMatchMode ($mode)

Sets full-text query matching mode, as described in Section 4.1, "Matching modes". Parameter must be a constant specifying one of the known modes.

WARNING: (PHP specific) you **must not** take the matching mode constant name in quotes, that syntax specifies a string and is incorrect:

```
$cl->SetMatchMode ( "SPH_MATCH_ANY" ); // INCORRECT! will not work as expected
$cl->SetMatchMode ( SPH_MATCH_ANY ); // correct, works OK
```

6.3.2. SetRankingMode

Prototype: function SetRankingMode ($ranker)

Sets ranking mode. Only available in SPH_MATCH_EXTENDED2 matching mode at the time of this writing. Parameter must be a constant specifying one of the known modes.

By default, Sphinx computes two factors which contribute to the final match weight. The major part is query phrase proximity to document text. The minor part is so-called BM25 statistical function, which varies from 0 to 1 depending on the keyword frequency within document (more occurrences yield higher weight) and within the whole index (more rare keywords yield higher weight).

However, in some cases you'd want to compute weight differently - or maybe avoid computing it at all for performance reasons because you're sorting the result set by something else anyway. This can be accomplished by setting the appropriate ranking mode.

Currently implemented modes are:

- SPH_RANK_PROXIMITY_BM25, default ranking mode which uses and combines both phrase proximity and BM25 ranking.
- SPH_RANK_BM25, statistical ranking mode which uses BM25 ranking only (similar to most other full-text engines). This mode is faster but may result in worse quality on queries which contain more than 1 keyword.
- SPH_RANK_NONE, disabled ranking mode. This mode is the fastest. It is essentially equivalent to boolean searching. A weight of 1 is assigned to all matches.
- SPH_RANK_WORDCOUNT, ranking by keyword occurrences count. This ranker computes the amount of per-field keyword occurrences, then multiplies the amounts by field weights, then sums the resulting values for the final result.
- SPH_RANK_PROXIMITY, added in version 0.9.9-rc1, returns raw phrase proximity value as a result. This mode is internally used to emulate SPH_MATCH_ALL queries.
- SPH_RANK_MATCHANY, added in version 0.9.9-rc1, returns rank as it was computed in SPH_MATCH_ANY mode ealier, and is internally used to emulate SPH_MATCH_ANY queries.
- SPH_RANK_FIELDMASK, added in version 0.9.9-rc2, returns a 32-bit mask with N-th bit corresponding to N-th fulltext field, numbering from 0. The bit will only be set when the respective field has any keyword occurences satisfiying the query.

6.3.3. SetSortMode

Prototype: function SetSortMode ($mode, $sortby="")

Set matches sorting mode, as described in Section 4.5, "Sorting modes". Parameter must be a constant specifying one of the known modes.

WARNING: (PHP specific) you **must not** take the matching mode constant name in quotes, that syntax specifies a string and is incorrect:

```
$cl->SetSortMode ( "SPH_SORT_ATTR_DESC" ); // INCORRECT! will not work as expected
$cl->SetSortMode ( SPH_SORT_ATTR_ASC ); // correct, works OK
```

6.3.4. SetWeights

Prototype: function SetWeights ($weights)

Binds per-field weights in the order of appearance in the index. **DEPRECATED**, use SetFieldWeights() instead.

6.3.5. SetFieldWeights

Prototype: function SetFieldWeights ($weights)

Binds per-field weights by name. Parameter must be a hash (associative array) mapping string field names to integer weights.

Match ranking can be affected by per-field weights. For instance, see Section 4.4, "Weighting" for an explanation how phrase proximity ranking is affected. This call lets you specify what non-default weights to assign to different full-text fields.

The weights must be positive 32-bit integers. The final weight will be a 32-bit integer too. Default weight value is 1. Unknown field names will be silently ignored.

There is no enforced limit on the maximum weight value at the moment. However, beware that if you set it too high you can start hitting 32-bit wraparound issues. For instance, if you set a weight of 10,000,000 and search in extended mode, then maximum possible weight will be equal to 10 million (your weight) by 1 thousand (internal BM25 scaling factor, see Section 4.4, "Weighting") by 1 or more (phrase proximity rank). The result is at least 10 billion that does not fit in 32 bits and will be wrapped around, producing unexpected results.

6.3.6. SetIndexWeights

Prototype: function SetIndexWeights ($weights)

Sets per-index weights, and enables weighted summing of match weights across different indexes. Parameter must be a hash (associative array) mapping string index names to integer weights. Default is empty array that means to disable weighting summing.

When a match with the same document ID is found in several different local indexes, by default Sphinx simply chooses the match from the index specified last in the query. This is to support searching through partially overlapping index partitions.

However in some cases the indexes are not just partitions, and you might want to sum the weights across the indexes instead of picking one. `SetIndexWeights()` lets you do that. With summing enabled, final match weight in result set will be computed as a sum of match weight coming from the given index multiplied by respective per-index weight specified in this call. Ie. if the document 123 is found in index A with the weight of 2, and also in index B with the weight of 3, and you called `SetIndexWeights ( array ( "A"=>100, "B"=>10 ) )`, the final weight return to the client will be 2*100+3*10 = 230.

6.4. Result set filtering settings

6.4.1. SetIDRange

Prototype: function SetIDRange ($min, $max)

Sets an accepted range of document IDs. Parameters must be integers. Defaults are 0 and 0; that combination means to not limit by range.

After this call, only those records that have document ID between `$min` and `$max` (including IDs exactly equal to `$min` or `$max`) will be matched.

6.4.2. SetFilter

Prototype: function SetFilter ($attribute, $values, $exclude=false)

Adds new integer values set filter.

On this call, additional new filter is added to the existing list of filters. `$attribute` must be a string with attribute name. `$values` must be a plain array containing integer values. `$exclude` must be a boolean value; it controls whether to accept the matching documents (default mode, when `$exclude` is false) or reject them.

Only those documents where `$attribute` column value stored in the index matches any of the values from `$values` array will be matched (or rejected, if `$exclude` is true).

6.4.3. SetFilterRange

Prototype: function SetFilterRange ($attribute, $min, $max, $exclude=false)

Adds new integer range filter.

On this call, additional new filter is added to the existing list of filters. `$attribute` must be a string with attribute name. `$min` and `$max` must be integers that define the acceptable attribute values range (including the boundaries). `$exclude` must be a boolean value; it controls whether to accept the matching documents (default mode, when `$exclude` is false) or reject them.

Only those documents where `$attribute` column value stored in the index is between `$min` and `$max` (including values that are exactly equal to `$min` or `$max`) will be matched (or rejected, if `$exclude` is true).

6.4.4. SetFilterFloatRange

Prototype: function SetFilterFloatRange ($attribute, $min, $max, $exclude=false)

Adds new float range filter.

On this call, additional new filter is added to the existing list of filters. `$attribute` must be a string with attribute name. `$min` and `$max` must be floats that define the acceptable attribute values range (including the boundaries). `$exclude` must be a boolean value; it controls whether to accept the matching documents (default mode, when `$exclude` is false) or reject them.

Only those documents where `$attribute` column value stored in the index is between `$min` and `$max` (including values that are exactly equal to `$min` or `$max`) will be matched (or rejected, if `$exclude` is true).

6.4.5. SetGeoAnchor

Prototype: function SetGeoAnchor ($attrlat, $attrlong, $lat, $long)

Sets anchor point for and geosphere distance (geodistance) calculations, and enable them.

`$attrlat` and `$attrlong` must be strings that contain the names of latitude and longitude attributes, respectively. `$lat` and `$long` are floats that specify anchor point latitude and longitude, in radians.

Once an anchor point is set, you can use magic `"@geodist"` attribute name in your filters and/or sorting expressions. Sphinx will compute geosphere distance between the given anchor point and a point specified by latitude and lognitude attributes from each full-text match, and attach this value to the resulting match. The latitude and longitude values both in `SetGeoAnchor` and the index attribute data are expected to be in radians. The result will be returned in meters, so geodistance value of 1000.0 means 1 km. 1 mile is approximately 1609.344 meters.

6.5. GROUP BY settings

6.5.1. SetGroupBy

Prototype: function SetGroupBy ($attribute, $func, $groupsort="@group desc")

Sets grouping attribute, function, and groups sorting mode; and enables grouping (as described in Section 4.6, "Grouping (clustering) search results ").

`$attribute` is a string that contains group-by attribute name. `$func` is a constant that chooses a function applied to the attribute value in order to compute group-by key. `$groupsort` is a clause that controls how the groups will be sorted. Its syntax is similar to that described in Section 4.5, "SPH_SORT_EXTENDED mode".

Grouping feature is very similar in nature to GROUP BY clause from SQL. Results produces by this function call are going to be the same as produced by the following pseudo code:

```
SELECT ... GROUP BY $func($attribute) ORDER BY $groupsort
```

Note that it's `$groupsort` that affects the order of matches in the final result set. Sorting mode (see Section 6.3.3, "SetSortMode") affect the ordering of matches *within* group, ie. what match will be selected as the best one from the group. So you can for instance order the groups by matches count and select the most relevant match within each group at the same time.

Starting with version 0.9.9-rc2, aggregate functions (AVG(), MIN(), MAX(), SUM()) are supported through SetSelect() API call when using GROUP BY.

6.5.2. SetGroupDistinct

Prototype: function SetGroupDistinct ($attribute)

Sets attribute name for per-group distinct values count calculations. Only available for grouping queries.

`$attribute` is a string that contains the attribute name. For each group, all values of this attribute will be stored (as RAM limits permit), then the amount of distinct values will be calculated and returned to the client. This feature is similar to `COUNT(DISTINCT)` clause in standard SQL; so these Sphinx calls:

```
$cl->SetGroupBy ( "category", SPH_GROUPBY_ATTR, "@count desc" );
$cl->SetGroupDistinct ( "vendor" );
```

can be expressed using the following SQL clauses:

```
SELECT id, weight, all-attributes,
        COUNT(DISTINCT vendor) AS @distinct,
        COUNT(*) AS @count
FROM products
GROUP BY category
ORDER BY @count DESC
```

In the sample pseudo code shown just above, `SetGroupDistinct()` call corresponds to `COUNT(DISINCT vendor)` clause only. `GROUP BY`, `ORDER BY`, and `COUNT(*)` clauses are all an equivalent of `SetGroupBy()` settings. Both queries will return one matching row for each category. In addition to indexed attributes, matches will also contain total per-category matches count, and the count of distinct vendor IDs within each category.

6.6. Querying

6.6.1. Query

Prototype: function Query ($query, $index="*", $comment="")

Connects to `searchd` server, runs given search query with current settings, obtains and returns the result set.

`$query` is a query string. `$index` is an index name (or names) string. Returns false and sets `GetLastError()` message on general error. Returns search result set on success. Additionally, the contents of `$comment` are sent to the query log, marked in square brackets, just before the search terms, which can be very useful for debugging. Currently, the comment is limited to 128 characters.

Default value for `$index` is `"*"` that means to query all local indexes. Characters allowed in index names include Latin letters (a-z), numbers (0-9), minus sign (-), and underscore (_); everything else is considered a

separator. Therefore, all of the following samples calls are valid and will search the same two indexes:

```
$cl->Query ( "test query", "main delta" );
$cl->Query ( "test query", "main;delta" );
$cl->Query ( "test query", "main, delta" );
```

Index specification order matters. If document with identical IDs are found in two or more indexes, weight and attribute values from the very last matching index will be used for sorting and returning to client (unless explicitly overridden with SetIndexWeights()). Therefore, in the example above, matches from "delta" index will always win over matches from "main".

On success, `Query()` returns a result set that contains some of the found matches (as requested by SetLimits()) and additional general per-query statistics. The result set is a hash (PHP specific; other languages might utilize other structures instead of hash) with the following keys and values:

"matches":
: Hash which maps found document IDs to another small hash containing document weight and attribute values (or an array of the similar small hashes if SetArrayResult() was enabled).

"total":
: Total amount of matches retrieved *on server* (ie. to the server side result set) by this query. You can retrieve up to this amount of matches from server for this query text with current query settings.

"total_found":
: Total amount of matching documents in index (that were found and procesed on server).

"words":
: Hash which maps query keywords (case-folded, stemmed, and otherwise processed) to a small hash with per-keyword statitics ("docs", "hits").

"error":
: Query error message reported by `searchd` (string, human readable). Empty if there were no errors.

"warning":
: Query warning message reported by `searchd` (string, human readable). Empty if there were no warnings.

It should be noted that `Query()` carries out the same actions as `AddQuery()` and `RunQueries()` without the intermediate steps; it is analoguous to a single `AddQuery()` call, followed by a corresponding `RunQueries()`, then returning the first array element of matches (from the first, and only, query.)

6.6.2. AddQuery

Prototype: function AddQuery ($query, $index="*", $comment="")

Adds additional query with current settings to multi-query batch. `$query` is a query string. `$index` is an index name (or names) string. Additionally if provided, the contents of `$comment` are sent to the query log, marked in square brackets, just before the search terms, which can be very useful for debugging. Currently, this is limited to 128 characters. Returns index to results array returned from RunQueries().

Batch queries (or multi-queries) enable `searchd` to perform internal optimizations if possible. They also reduce network connection overheads and search process creation overheads in all cases. They do not result in any additional overheads compared to simple queries. Thus, if you run several different queries from your web page, you should always consider using multi-queries.

For instance, running the same full-text query but with different sorting or group-by settings will enable `searchd` to perform expensive full-text search and ranking operation only once, but compute multiple group-by results from its output.

This can be a big saver when you need to display not just plain search results but also some per-category counts, such as the amount of products grouped by vendor. Without multi-query, you would have to run several queries which perform essentially the same search and retrieve the same matches, but create result sets differently. With multi-query, you simply pass all these querys in a single batch and Sphinx optimizes the redundant full-text search internally.

`AddQuery()` internally saves full current settings state along with the query, and you can safely change them afterwards for subsequent `AddQuery()` calls. Already added queries will not be affected; there's actually no way to change them at all. Here's an example:

```
$cl->SetSortMode ( SPH_SORT_RELEVANCE );
$cl->AddQuery ( "hello world", "documents" );

$cl->SetSortMode ( SPH_SORT_ATTR_DESC, "price" );
$cl->AddQuery ( "ipod", "products" );

$cl->AddQuery ( "harry potter", "books" );

$results = $cl->RunQueries ();
```

With the code above, 1st query will search for "hello world" in "documents" index and sort results by relevance, 2nd query will search for "ipod" in "products" index and sort results by price, and 3rd query will search for "harry potter" in "books" index while still sorting by price. Note that 2nd `SetSortMode()` call does not affect the first query (because it's already added) but affects both other subsequent queries.

Additionally, any filters set up before an `AddQuery()` will fall through to subsequent queries. So, if `SetFilter()` is called before the first query, the same filter will be in place for the second (and subsequent) queries batched through `AddQuery()` unless you call `ResetFilters()` first. Alternatively, you can add additional filters as well.

This would also be true for grouping options and sorting options; no current sorting, filtering, and grouping settings are affected by this call; so subsequent queries will reuse current query settings.

`AddQuery()` returns an index into an array of results that will be returned from `RunQueries()` call. It is simply a sequentially increasing 0-based integer, ie. first call will return 0, second will return 1, and so on. Just a small helper so you won't have to track the indexes manualy if you need then.

6.6.3. RunQueries

Prototype: function RunQueries ()

Connect to searchd, runs a batch of all queries added using `AddQuery()`, obtains and returns the result sets. Returns false and sets `GetLastError()` message on general error (such as network I/O failure). Returns a plain array of result sets on success.

Each result set in the returned array is exactly the same as the result set returned from `Query()`.

Note that the batch query request itself almost always succeds - unless there's a network error, blocking index rotation in progress, or another general failure which prevents the whole request from being processed.

However individual queries within the batch might very well fail. In this case their respective result sets will contain non-empty `"error"` message, but no matches or query statistics. In the extreme case all queries within the batch could fail. There still will be no general error reported, because API was able to succesfully connect to `searchd`, submit the batch, and receive the results - but every result set will have a specific error message.

6.6.4. ResetFilters

Prototype: function ResetFilters ()

Clears all currently set filters.

This call is only normally required when using multi-queries. You might want to set different filters for different queries in the batch. To do that, you should call `ResetFilters()` and add new filters using the respective calls.

6.6.5. ResetGroupBy

Prototype: function ResetGroupBy ()

Clears all currently group-by settings, and disables group-by.

This call is only normally required when using multi-queries. You can change individual group-by settings using `SetGroupBy()` and `SetGroupDistinct()` calls, but you can not disable group-by using those calls. `ResetGroupBy()` fully resets previous group-by settings and disables group-by mode in the current state, so that subsequent `AddQuery()` calls can perform non-grouping searches.

6.7. Additional functionality

6.7.1. BuildExcerpts

Prototype: function BuildExcerpts ($docs, $index, $words, $opts=array())

Excerpts (snippets) builder function. Connects to `searchd`, asks it to generate excerpts (snippets) from given documents, and returns the results.

`$docs` is a plain array of strings that carry the documents' contents. `$index` is an index name string. Different settings (such as charset, morphology, wordforms) from given index will be used. `$words` is a string that contains the keywords to highlight. They will be processed with respect to index settings. For instance, if English stemming is enabled in the index, "shoes" will be highlighted even if keyword is "shoe". Starting with version 0.9.9-rc1, keywords can contain wildcards, that work similarly to star-syntax available in queries. `$opts` is a hash which contains additional optional highlighting parameters:

"before_match":
: A string to insert before a keyword match. Default is "<b>".

"after_match":
: A string to insert after a keyword match. Default is "<b>".

"chunk_separator":
: A string to insert between snippet chunks (passages). Default is " ... ".

"limit":
: Maximum snippet size, in symbols (codepoints). Integer, default is 256.

"around":
: How much words to pick around each matching keywords block. Integer, default is 5.

"exact_phrase":
: Whether to highlight exact query phrase matches only instead of individual keywords. Boolean, default is false.

"single_passage":
: Whether to extract single best passage only. Boolean, default is false.

"weight_order":
: Whether to sort the extracted passages in order of relevance (decreasing weight), or in order of appearance in the document (increasing position). Boolean, default is false.

Returns false on failure. Returns a plain array of strings with excerpts (snippets) on success.

6.7.2. UpdateAttributes

Prototype: function UpdateAttributes ($index, $attrs, $values)

Instantly updates given attribute values in given documents. Returns number of actually updated documents (0 or more) on success, or -1 on failure.

`$index` is a name of the index (or indexes) to be updated. `$attrs` is a plain array with string attribute names, listing attributes that are updated. `$values` is a hash where key is document ID, and value is a plain array of new attribute values.

`$index` can be either a single index name or a list, like in `Query()`. Unlike `Query()`, wildcard is not allowed and all the indexes to update must be specified explicitly. The list of indexes can include distributed index names. Updates on distributed indexes will be pushed to all agents.

The updates only work with `docinfo=extern` storage strategy. They are very fast because they're working fully in RAM, but they can also be made persistent: updates are saved on disk on clean `searchd` shutdown initiated by SIGTERM signal. With additional restrictions, updates are also possible on MVA attributes; refer to mva_updates_pool directive for details.

Usage example:

```
$cl->UpdateAttributes ( "test1", array("group_id"), array(1=>array(456)) );
$cl->UpdateAttributes ( "products", array ( "price", "amount_in_stock" ),
        array ( 1001=>array(123,5), 1002=>array(37,11), 1003=>(25,129) ) );
```

The first sample statement will update document 1 in index "test1", setting "group_id" to 456. The second one will update documents 1001, 1002 and 1003 in index "products". For document 1001, the new price will be set to 123 and the new amount in stock to 5; for document 1002, the new price will be 37 and the new amount will be 11; etc.

6.7.3. BuildKeywords

Prototype: function BuildKeywords ($query, $index, $hits)

Extracts keywords from query using tokenizer settings for given index, optionally with per-keyword occurrence statistics. Returns an array of hashes with per-keyword information.

`$query` is a query to extract keywords from. `$index` is a name of the index to get tokenizing settings and keyword occurrence statistics from. `$hits` is a boolean flag that indicates whether keyword occurrence statistics are required.

Usage example:

```
$keywords = $cl->BuildKeywords ( "this.is.my query", "test1", false );
```

6.7.4. EscapeString

Prototype: function EscapeString ($string)

Escapes characters that are treated as special operators by the query language parser. Returns an escaped string.

`$string` is a string to escape.

This function might seem redundant because it's trivial to implement in any calling application. However, as the set of special characters might change over time, it makes sense to have an API call that is guaranteed to escape all such characters at all times.

Usage example:

```
$escaped = $cl->EscapeString ( "escaping-sample@query/string" );
```

6.7.5. Status

Prototype: function Status ()

Queries searchd status, and returns an array of status variable name and value pairs.

Usage example:

```
$status = $cl->Status ();
foreach ( $status as $row )
        print join ( ": ", $row ) . "\n";
```

6.8. Persistent connections

Persistent connections allow to use single network connection to run multiple commands that would otherwise require reconnects.

6.8.1. Open

Prototype: function Open ()

Opens persistent connection to the server.

6.8.2. Close

Prototype: function Close ()

Closes previously opened persistent connection.

7. MySQL storage engine (SphinxSE)

7.1. SphinxSE overview

SphinxSE is MySQL storage engine which can be compiled into MySQL server 5.x using its pluggable architecure. It is not available for MySQL 4.x series. It also requires MySQL 5.0.22 or higher in 5.0.x series, or MySQL 5.1.12 or higher in 5.1.x series.

Despite the name, SphinxSE does *not* actually store any data itself. It is actually a built-in client which allows MySQL server to talk to `searchd`, run search queries, and obtain search results. All indexing and searching happen outside MySQL.

Obvious SphinxSE applications include:

- easier porting of MySQL FTS applications to Sphinx;
- allowing Sphinx use with progamming languages for which native APIs are not available yet;
- optimizations when additional Sphinx result set processing on MySQL side is required (eg. JOINs with original document tables, additional MySQL-side filtering, etc).

7.2. Installing SphinxSE

You will need to obtain a copy of MySQL sources, prepare those, and then recompile MySQL binary. MySQL sources (mysql-5.x.yy.tar.gz) could be obtained from dev.mysql.com Web site.

For some MySQL versions, there are delta tarballs with already prepared source versions available from Sphinx Web site. After unzipping those over original sources MySQL would be ready to be configured and built with Sphinx support.

If such tarball is not available, or does not work for you for any reason, you would have to prepare sources manually. You will need to GNU Autotools framework (autoconf, automake and libtool) installed to do that.

7.2.1. Compiling MySQL 5.0.x with SphinxSE

Skips steps 1-3 if using already prepared delta tarball.

1. copy `sphinx.5.0.yy.diff` patch file into MySQL sources directory and run

   ```
   patch -p1 < sphinx.5.0.yy.diff
   ```

   If there's no .diff file exactly for the specific version you need to build, try applying .diff with closest version numbers. It is important that the patch should apply with no rejects.

2. in MySQL sources directory, run

   ```
   sh BUILD/autorun.sh
   ```

3. in MySQL sources directory, create `sql/sphinx` directory in and copy all files in `mysqlse` directory from Sphinx sources there. Example:

   ```
   cp -R /root/builds/sphinx-0.9.7/mysqlse /root/builds/mysql-5.0.24/sql/sphinx
   ```

4. configure MySQL and enable Sphinx engine:

   ```
   ./configure --with-sphinx-storage-engine
   ```

5. build and install MySQL:

   ```
   make
   ```

```
make install
```

7.2.2. Compiling MySQL 5.1.x with SphinxSE

Skip steps 1-2 if using already prepared delta tarball.

1. in MySQL sources directory, create `storage/sphinx` directory in and copy all files in `mysqlse` directory from Sphinx sources there. Example:

```
cp -R /root/builds/sphinx-0.9.7/mysqlse /root/builds/mysql-5.1.14/storage/sphinx
```

2. in MySQL sources directory, run

```
sh BUILD/autorun.sh
```

3. configure MySQL and enable Sphinx engine:

```
./configure --with-plugins=sphinx
```

4. build and install MySQL:

```
make
make install
```

7.2.3. Checking SphinxSE installation

To check whether SphinxSE has been succesfully compiled into MySQL, launch newly built servers, run mysql client and issue `SHOW ENGINES` query. You should see a list of all available engines. Sphinx should be present and "Support" column should contain "YES":

```
mysql> show engines;
+------------+----------+----------------------------------------------------------------+
| Engine     | Support  | Comment                                                        |
+------------+----------+----------------------------------------------------------------+
| MyISAM     | DEFAULT  | Default engine as of MySQL 3.23 with great performance      |
  ...
| SPHINX     | YES      | Sphinx storage engine                                          |
  ...
+------------+----------+----------------------------------------------------------------+
13 rows in set (0.00 sec)
```

7.3. Using SphinxSE

To search via SphinxSE, you would need to create special ENGINE=SPHINX "search table", and then SELECT from it with full text query put into WHERE clause for query column.

Let's begin with an example create statement and search query:

```
CREATE TABLE t1
(
    id          INTEGER UNSIGNED NOT NULL,
    weight      INTEGER NOT NULL,
    query       VARCHAR(3072) NOT NULL,
    group_id    INTEGER,
    INDEX(query)
) ENGINE=SPHINX CONNECTION="sphinx://localhost:9312/test";

SELECT * FROM t1 WHERE query='test it;mode=any';
```

First 3 columns of search table *must* have a types of `INTEGER UNSINGED` or `BIGINT` for the 1st column (document id), `INTEGER` or `BIGINT` for the 2nd column (match weight), and `VARCHAR` or `TEXT` for the 3rd column (your query), respectively. This mapping is fixed; you can not omit any of these three required columns, or

move them around, or change types. Also, query column must be indexed; all the others must be kept unindexed. Columns' names are ignored so you can use arbitrary ones.

Additional columns must be either `INTEGER`, `TIMESTAMP`, `BIGINT`, `VARCHAR`, or `FLOAT`. They will be bound to attributes provided in Sphinx result set by name, so their names must match attribute names specified in `sphinx.conf`. If there's no such attribute name in Sphinx search results, column will have `NULL` values.

Special "virtual" attributes names can also be bound to SphinxSE columns. `_sph_` needs to be used instead of `@` for that. For instance, to obtain the values of `@groupby`, `@count`, or `@distinct` virtual attributes, use `_sph_groupby`, `_sph_count` or `_sph_distinct` column names, respectively.

`CONNECTION` string parameter can be used to specify default searchd host, port and indexes for queries issued using this table. If no connection string is specified in `CREATE TABLE`, index name "*" (ie. search all indexes) and localhost:9312 are assumed. Connection string syntax is as follows:

```
CONNECTION="sphinx://HOST:PORT/INDEXNAME"
```

You can change the default connection string later:

```
ALTER TABLE t1 CONNECTION="sphinx://NEWHOST:NEWPORT/NEWINDEXNAME";
```

You can also override all these parameters per-query.

As seen in example, both query text and search options should be put into WHERE clause on search query column (ie. 3rd column); the options are separated by semicolons; and their names from values by equality sign. Any number of options can be specified. Available options are:

- query - query text;
- mode - matching mode. Must be one of "all", "any", "phrase", "boolean", or "extended". Default is "all";
- sort - match sorting mode. Must be one of "relevance", "attr_desc", "attr_asc", "time_segments", or "extended". In all modes besides "relevance" attribute name (or sorting clause for "extended") is also required after a colon:

  ```
  ... WHERE query='test;sort=attr_asc:group_id';
  ... WHERE query='test;sort=extended:@weight desc, group_id asc';
  ```

- offset - offset into result set, default is 0;
- limit - amount of matches to retrieve from result set, default is 20;
- index - names of the indexes to search:

  ```
  ... WHERE query='test;index=test1;';
  ... WHERE query='test;index=test1,test2,test3;';
  ```

- minid, maxid - min and max document ID to match;
- weights - comma-separated list of weights to be assigned to Sphinx full-text fields:

  ```
  ... WHERE query='test;weights=1,2,3;';
  ```

- filter, !filter - comma-separated attribute name and a set of values to match:

  ```
  # only include groups 1, 5 and 19
  ... WHERE query='test;filter=group_id,1,5,19;';

  # exclude groups 3 and 11
  ... WHERE query='test;!filter=group_id,3,11;';
  ```

- range, !range - comma-separated attribute name, min and max value to match:

  ```
  # include groups from 3 to 7, inclusive
  ... WHERE query='test;range=group_id,3,7;';
  ```

```
# exclude groups from 5 to 25
... WHERE query='test;!range=group_id,5,25;';
```

- maxmatches - per-query max matches value:

```
... WHERE query='test;maxmatches=2000;';
```

- groupby - group-by function and attribute:

```
... WHERE query='test;groupby=day:published_ts;';
... WHERE query='test;groupby=attr:group_id;';
```

- groupsort - group-by sorting clause:

```
... WHERE query='test;groupsort=@count desc;';
```

- indexweights - comma-separated list of index names and weights to use when searching through several indexes:

```
... WHERE query='test;indexweights=idx_exact,2,idx_stemmed,1;';
```

One **very important** note that it is **much** more efficient to allow Sphinx to perform sorting, filtering and slicing the result set than to raise max matches count and use WHERE, ORDER BY and LIMIT clauses on MySQL side. This is for two reasons. First, Sphinx does a number of optimizations and performs better than MySQL on these tasks. Second, less data would need to be packed by searchd, transferred and unpacked by SphinxSE.

Starting with version 0.9.9-rc1, additional query info besides result set could be retrieved with `SHOW ENGINE SPHINX STATUS` statement:

```
mysql> SHOW ENGINE SPHINX STATUS;
+--------+-------+-----------------------------------------------------+
| Type   | Name  | Status                                              |
+--------+-------+-----------------------------------------------------+
| SPHINX | stats | total: 25, total found: 25, time: 126, words: 2 |
| SPHINX | words | sphinx:591:1256 soft:11076:15945                |
+--------+-------+-----------------------------------------------------+
2 rows in set (0.00 sec)
```

This information can also be accessed through status variables. Note that this method does not require super-user privileges.

```
mysql> SHOW STATUS LIKE 'sphinx_%';
+--------------------+---------------------------------+
| Variable_name      | Value                           |
+--------------------+---------------------------------+
| sphinx_total       | 25                              |
| sphinx_total_found | 25                              |
| sphinx_time        | 126                             |
| sphinx_word_count  | 2                               |
| sphinx_words       | sphinx:591:1256 soft:11076:15945 |
+--------------------+---------------------------------+
5 rows in set (0.00 sec)
```

You could perform JOINs on SphinxSE search table and tables using other engines. Here's an example with "documents" from example.sql:

```
mysql> SELECT content, date_added FROM test.documents docs
-> JOIN t1 ON (docs.id=t1.id)
-> WHERE query="one document;mode=any";
+----------------------------------+---------------------+
| content                          | docdate             |
+----------------------------------+---------------------+
| this is my test document number two | 2006-06-17 14:04:28 |
```

```
| this is my test document number one | 2006-06-17 14:04:28 |
+--------------------------------------+---------------------+
2 rows in set (0.00 sec)

mysql> SHOW ENGINE SPHINX STATUS;
+--------+-------+-----------------------------------------------+
| Type   | Name  | Status                                        |
+--------+-------+-----------------------------------------------+
| SPHINX | stats | total: 2, total found: 2, time: 0, words: 2 |
| SPHINX | words | one:1:2 document:2:2                          |
+--------+-------+-----------------------------------------------+
2 rows in set (0.00 sec)
```

7.4. Building snippets (excerpts) via MySQL

Starting with version 0.9.9-rc2, SphinxSE also includes a UDF function that lets you create snippets through MySQL. The functionality is fully similar to BuildExcerprts API call but accesible through MySQL+SphinxSE.

The binary that provides the UDF is named `sphinx.so` and should be automatically built and installed to proper location along with SphinxSE itself. If it does not get installed automatically for some reason, look for `sphinx.so` in the build directory and copy it to the plugins directory of your MySQL instance. After that, register the UDF using the following statement:

```
CREATE FUNCTION sphinx_snippets RETURNS STRING SONAME 'sphinx.so';
```

Function name *must* be sphinx_snippets, you can not use an arbitrary name. Function arguments are as follows:

Prototype: function sphinx_snippets (document, index, words, [options]);

Document and words arguments can be either strings or table columns. Options must be specified like this: `'value' AS option_name`. For a list of supported options, refer to BuildExcerprts() API call. The only UDF-specific additional option is named `'sphinx'` and lets you specify searchd location (host and port).

Usage examples:

```
SELECT sphinx_snippets('hello world doc', 'main', 'world',
    'sphinx://192.168.1.1/' AS sphinx, true AS exact_phrase,
    '[b]' AS before_match, '[/b]' AS after_match)
FROM documents;

SELECT title, sphinx_snippets(text, 'index', 'mysql php') AS text
    FROM sphinx, documents
    WHERE query='mysql php' AND sphinx.id=documents.id;
```

8. Reporting bugs

Unfortunately, Sphinx is not yet 100% bug free (even though I'm working hard towards that), so you might occasionally run into some issues.

Reporting as much as possible about each bug is very important - because to fix it, I need to be able either to reproduce and debug the bug, or to deduce what's causing it from the information that you provide. So here are some instructions on how to do that.

Build-time issues

If Sphinx fails to build for some reason, please do the following:

1. check that headers and libraries for your DBMS are properly installed (for instance, check that `mysql-devel` package is present);
2. report Sphinx version and config file (be sure to remove the passwords!), MySQL (or PostgreSQL) configuration info, gcc version, OS version and CPU type (ie. x86, x86-64, PowerPC, etc):

```
mysql_config
gcc --version
uname -a
```

3. report the error message which is produced by `configure` or `gcc` (it should be to include error message itself only, not the whole build log).

Run-time issues

If Sphinx builds and runs, but there are any problems running it, please do the following:

1. describe the bug (ie. both the expected behavior and actual behavior) and all the steps necessary to reproduce it;
2. include Sphinx version and config file (be sure to remove the passwords!), MySQL (or PostgreSQL) version, gcc version, OS version and CPU type (ie. x86, x86-64, PowerPC, etc):

```
mysql --version
gcc --version
uname -a
```

3. build, install and run debug versions of all Sphinx programs (this is to enable a lot of additional internal checks, so-called assertions):

```
make distclean
./configure --with-debug
make install
killall -TERM searchd
```

4. reindex to check if any assertions are triggered (in this case, it's likely that the index is corrupted and causing problems);
5. if the bug does not reproduce with debug versions, revert to non-debug and mention it in your report;
6. if the bug could be easily reproduced with a small (1-100 record) part of your database, please provide a gzipped dump of that part;
7. if the problem is related to `searchd`, include relevant entries from `searchd.log` and `query.log` in your bug report;
8. if the problem is related to `searchd`, try running it in console mode and check if it dies with an assertion:

```
./searchd --console
```

9. if any program dies with an assertion, provide the assertion message.

Debugging assertions, crashes and hangups

If any program dies with an assertion, crashes without an assertion or hangs up, you would additionally need to generate a core dump and examine it.

1. enable core dumps. On most Linux systems, this is done using `ulimit`:

```
ulimit -c 32768
```

2. run the program and try to reproduce the bug;
3. if the program crashes (either with or without an assertion), find the core file in current directory (it should typically print out "Segmentation fault (core dumped)" message);
4. if the program hangs, use `kill -SEGV` from another console to force it to exit and dump core:

```
kill -SEGV HANGED-PROCESS-ID
```

5. use `gdb` to examine the core file and obtain a backtrace:

```
gdb ./CRASHED-PROGRAM-FILE-NAME CORE-DUMP-FILE-NAME
```

```
(gdb) bt
(gdb) quit
```

Note that HANGED-PROCESS-ID, CRASHED-PROGRAM-FILE-NAME and CORE-DUMP-FILE-NAME must all be replaced with specific numbers and file names. For example, hanged searchd debugging session would look like:

```
# kill -SEGV 12345
# ls *core*
core.12345
# gdb ./searchd core.12345
(gdb) bt
...
(gdb) quit
```

Note that `ulimit` is not server-wide and only affects current shell session. This means that you will not have to restore any server-wide limits - but if you relogin, you will have to set `ulimit` again.

Core dumps should be placed in current working directory (and Sphinx programs do not change it), so this is where you would look for them.

Please do not immediately remove the core file because there could be additional helpful information which could be retrieved from it. You do not need to send me this file (as the debug info there is closely tied to your system) but I might need to ask you a few additional questions about it.

9. `sphinx.conf` options reference

9.1. Data source configuration options

9.1.1. type

Data source type. Mandatory, no default value. Known types are `mysql`, `pgsql`, `mssql`, `xmlpipe` and `xmlpipe2`, and `odbc`.

All other per-source options depend on source type selected by this option. Names of the options used for SQL sources (ie. MySQL, PostgreSQL, MS SQL) start with "sql_"; names of the ones used for xmlpipe and xmlpipe2 start with "xmlpipe_". All source types except `xmlpipe` are conditional; they might or might not be supported depending on your build settings, installed client libraries, etc. `mssql` type is currently only available on Windows. `odbc` type is available both on Windows natively and on Linux through UnixODBC library.

Example:

```
type = mysql
```

9.1.2. sql_host

SQL server host to connect to. Mandatory, no default value. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

In the simplest case when Sphinx resides on the same host with your MySQL or PostgreSQL installation, you would simply specify "localhost". Note that MySQL client library chooses whether to connect over TCP/IP or over UNIX socket based on the host name. Specifically "localhost" will force it to use UNIX socket (this is the default and generally recommended mode) and "127.0.0.1" will force TCP/IP usage. Refer to MySQL manual for more details.

Example:

```
sql_host = localhost
```

9.1.3. sql_port

SQL server IP port to connect to. Optional, default is 3306 for `mysql` source type and 5432 for `pgsql` type. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only. Note that it depends on sql_host setting whether this value will actually be used.

Example:

```
sql_port = 3306
```

9.1.4. sql_user

SQL user to use when connecting to sql_host. Mandatory, no default value. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

Example:

```
sql_user = test
```

9.1.5. sql_pass

SQL user password to use when connecting to sql_host. Mandatory, no default value. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

Example:

```
sql_pass = mysecretpassword
```

9.1.6. sql_db

SQL database (in MySQL terms) to use after the connection and perform further queries within. Mandatory, no default value. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

Example:

```
sql_db = test
```

9.1.7. sql_sock

UNIX socket name to connect to for local SQL servers. Optional, default value is empty (use client library default settings). Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

On Linux, it would typically be `/var/lib/mysql/mysql.sock`. On FreeBSD, it would typically be `/tmp/mysql.sock`. Note that it depends on sql_host setting whether this value will actually be used.

Example:

```
sql_sock = /tmp/mysql.sock
```

9.1.8. mysql_connect_flags

MySQL client connection flags. Optional, default value is 0 (do not set any flags). Applies to `mysql` source type only.

This option must contain an integer value with the sum of the flags. The value will be passed to mysql_real_connect() verbatim. The flags are enumerated in mysql_com.h include file. Flags that are especially interesting in regard to indexing, with their respective values, are as follows:

- CLIENT_COMPRESS = 32; can use compression protocol
- CLIENT_SSL = 2048; switch to SSL after handshake
- CLIENT_SECURE_CONNECTION = 32768; new 4.1 authentication

For instance, you can specify 2080 (2048+32) to use both compression and SSL, or 32768 to use new authentication only. Initially, this option was introduced to be able to use compression when the `indexer` and `mysqld` are on different hosts. Compression on 1 Gbps links is most likely to hurt indexing time though it reduces network traffic, both in theory and in practice. However, enabling compression on 100 Mbps links may improve indexing time significantly (upto 20-30% of the total indexing time improvement was reported). Your mileage may vary.

Example:

```
mysql_connect_flags = 32 # enable compression
```

9.1.9. mysql_ssl_cert, mysql_ssl_key, mysql_ssl_ca

SSL certificate settings to use for connecting to MySQL server. Optional, default values are empty strings (do not use SSL). Applies to `mysql` source type only.

These directives let you set up secure SSL connection between `indexer` and MySQL. The details on creating the certificates and setting up MySQL server can be found in MySQL documentation.

Example:

```
mysql_ssl_cert = /etc/ssl/client-cert.pem
mysql_ssl_key = /etc/ssl/client-key.pem
mysql_ssl_ca = /etc/ssl/cacert.pem
```

9.1.10. odbc_dsn

ODBC DSN to connect to. Mandatory, no default value. Applies to `odbc` source type only.

ODBC DSN (Data Source Name) specifies the credentials (host, user, password, etc) to use when connecting to ODBC data source. The format depends on specific ODBC driver used.

Example:

```
odbc_dsn = Driver={Oracle ODBC Driver};Dbq=myDBName;Uid=myUsername;Pwd=myPassword
```

9.1.11. sql_query_pre

Pre-fetch query, or pre-query. Multi-value, optional, default is empty list of queries. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

Multi-value means that you can specify several pre-queries. They are executed before the main fetch query, and they will be exectued exactly in order of appeareance in the configuration file. Pre-query results are ignored.

Pre-queries are useful in a lot of ways. They are used to setup encoding, mark records that are going to be indexed, update internal counters, set various per-connection SQL server options and variables, and so on.

Perhaps the most frequent pre-query usage is to specify the encoding that the server will use for the rows it returnes. It **must** match the encoding that Sphinx expects (as specified by charset_type and charset_table options). Two MySQL specific examples of setting the encoding are:

```
sql_query_pre = SET CHARACTER_SET_RESULTS=cp1251
sql_query_pre = SET NAMES utf8
```

Also specific to MySQL sources, it is useful to disable query cache (for indexer connection only) in pre-query, because indexing queries are not going to be re-run frequently anyway, and there's no sense in caching their results. That could be achieved with:

```
sql_query_pre = SET SESSION query_cache_type=OFF
```

Example:

```
sql_query_pre = SET NAMES utf8
sql_query_pre = SET SESSION query_cache_type=OFF
```

9.1.12. sql_query

Main document fetch query. Mandatory, no default value. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

There can be only one main query. This is the query which is used to retrieve documents from SQL server. You can specify up to 32 full-text fields (formally, upto SPH_MAX_FIELDS from sphinx.h), and an arbitrary amount of attributes. All of the columns that are neither document ID (the first one) nor attributes will be full-text indexed.

Document ID **MUST** be the very first field, and it **MUST BE UNIQUE UNSIGNED POSITIVE (NON-ZERO, NON-NEGATIVE) INTEGER NUMBER**. It can be either 32-bit or 64-bit, depending on how you built Sphinx; by default it builds with 32-bit IDs support but `--enable-id64` option to `configure` allows to build with 64-bit document and word IDs support.

Example:

```
sql_query = \
        SELECT id, group_id, UNIX_TIMESTAMP(date_added) AS date_added, \
                title, content \
        FROM documents
```

9.1.13. sql_query_range

Range query setup. Optional, default is empty. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

Setting this option enables ranged document fetch queries (see Section 3.7, "Ranged queries"). Ranged queries are useful to avoid notorious MyISAM table locks when indexing lots of data. (They also help with other less notorious issues, such as reduced performance caused by big result sets, or additional resources consumed by InnoDB to serialize big read transactions.)

The query specified in this option must fetch min and max document IDs that will be used as range boundaries. It must return exactly two integer fields, min ID first and max ID second; the field names are ignored.

When ranged queries are enabled, sql_query will be required to contain `$start` and `$end` macros (because it obviously would be a mistake to index the whole table many times over). Note that the intervals specified by `$start`..`$end` will not overlap, so you should **not** remove document IDs that are exactly equal to `$start` or `$end` from your query. The example in Section 3.7, "Ranged queries") illustrates that; note how it uses greater-or-equal and less-or-equal comparisons.

Example:

```
sql_query_range = SELECT MIN(id),MAX(id) FROM documents
```

9.1.14. sql_range_step

Range query step. Optional, default is 1024. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

Only used when ranged queries are enabled. The full document IDs interval fetched by sql_query_range will be walked in this big steps. For example, if min and max IDs fetched are 12 and 3456 respectively, and the step is 1000, indexer will call sql_query several times with the following substitutions:

- $start=12, $end=1011
- $start=1012, $end=2011
- $start=2012, $end=3011
- $start=3012, $end=3456

Example:

```
sql_range_step = 1000
```

9.1.15. sql_query_killlist

Kill-list query. Optional, default is empty (no query). Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only. Introduced in version 0.9.9-rc1.

This query is expected to return a number of 1-column rows, each containing just the document ID. The returned document IDs are stored within an index. Kill-list for a given index suppresses results from *other* indexes, depending on index order in the query. The intended use is to help implement deletions and updates on existing indexes without rebuilding (actually even touching them), and especially to fight phantom results problem.

Let us dissect an example. Assume we have two indexes, 'main' and 'delta'. Assume that documents 2, 3, and 5 were deleted since last reindex of 'main', and documents 7 and 11 were updated (ie. their text contents were changed). Assume that a keyword 'test' occurred in all these mentioned documents when we were indexing 'main'; still occurs in document 7 as we index 'delta'; but does not occur in document 11 any more. We now reindex delta and then search through both these indexes in proper (least to most recent) order:

```
$res = $cl->Query ( "test", "main delta" );
```

First, we need to properly handle deletions. The result set should not contain documents 2, 3, or 5. Second, we also need to avoid phantom results. Unless we do something about it, document 11 *will* appear in search results! It will be found in 'main' (but not 'delta'). And it will make it to the final result set unless something stops it.

Kill-list, or K-list for short, is that something. Kill-list attached to 'delta' will suppress the specified rows from **all** the preceding indexes, in this case just 'main'. So to get the expected results, we should put all the updated *and* deleted document IDs into it.

Example:

```
sql_query_killlist = \
        SELECT id FROM documents WHERE updated_ts>=@last_reindex UNION \
        SELECT id FROM documents_deleted WHERE deleted_ts>=@last_reindex
```

9.1.16. sql_attr_uint

Unsigned integer attribute declaration. Multi-value (there might be multiple attributes declared), optional. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

The column value should fit into 32-bit unsigned integer range. Values outside this range will be accepted but wrapped around. For instance, -1 will be wrapped around to 2^32-1 or 4,294,967,295.

You can specify bit count for integer attributes by appending ':BITCOUNT' to attribute name (see example below). Attributes with less than default 32-bit size, or bitfields, perform slower. But they require less RAM when using extern storage: such bitfields are packed together in 32-bit chunks in `.spa` attribute data file. Bit size settings are ignored if using inline storage.

Example:

```
sql_attr_uint = group_id
sql_attr_uint = forum_id:9 # 9 bits for forum_id
```

9.1.17. sql_attr_bool

Boolean attribute declaration. Multi-value (there might be multiple attributes declared), optional. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only. Equivalent to sql_attr_uint declaration with a bit count of 1.

Example:

```
sql_attr_bool = is_deleted # will be packed to 1 bit
```

9.1.18. sql_attr_bigint

64-bit signed integer attribute declaration. Multi-value (there might be multiple attributes declared), optional. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only. Note that unlike sql_attr_uint, these values are **signed**. Introduced in version 0.9.9-rc1.

Example:

```
sql_attr_bigint = my_bigint_id
```

9.1.19. sql_attr_timestamp

UNIX timestamp attribute declaration. Multi-value (there might be multiple attributes declared), optional. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

The column value should be a timestamp in UNIX format, ie. 32-bit unsigned integer number of seconds elapsed since midnight, January 01, 1970, GMT. Timestamps are internally stored and handled as integers everywhere. But in addition to working with timestamps as integers, it's also legal to use them along with different date-based functions - such as time segments sorting mode, or day/week/month/year extraction for GROUP BY. Note that DATE or DATETIME column types in MySQL can **not** be directly used as timestamps; you need to explicitly convert such columns using UNIX_TIMESTAMP function.

Example:

```
sql_attr_timestamp = UNIX_TIMESTAMP(added_datetime) AS added_ts
```

9.1.20. sql_attr_str2ordinal

Ordinal string number attribute declaration. Multi-value (there might be multiple attributes declared), optional. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

This attribute type (so-called ordinal, for brevity) is intended to allow sorting by string values, but without storing the strings themselves. When indexing ordinals, string values are fetched from database, temporarily stored, sorted, and then replaced by their respective ordinal numbers in the array of sorted strings. So, the ordinal number is an integer such that sorting by it produces the same result as if lexicographically sorting by original strings. by string values lexicographically.

Earlier versions could consume a lot of RAM for indexing ordinals. Starting with revision r1112, ordinals accumulation and sorting also runs in fixed memory (at the cost of using additional temporary disk space), and honors mem_limit settings.

Ideally the strings should be sorted differently, depending on the encoding and locale. For instance, if the strings are known to be Russian text in KOI8R encoding, sorting the bytes 0xE0, 0xE1, and 0xE2 should produce 0xE1, 0xE2 and 0xE0, because in KOI8R value 0xE0 encodes a character that is (noticeably) after characters encoded by 0xE1 and 0xE2. Unfortunately, Sphinx does not support that at the moment and will simply sort the strings bytewise.

Note that the ordinals are by construction local to each index, and it's therefore impossible to merge ordinals while retaining the proper order. The processed strings are replaced by their sequential number in the index they occurred in, but different indexes have different sets of strings. For instance, if 'main' index contains strings "aaa", "bbb", "ccc", and so on up to "zzz", they'll be assigned numbers 1, 2, 3, and so on up to 26, respectively. But then if 'delta' only contains "zzz" the assigned number will be 1. And after the merge, the order will be broken. Unfortunately, this is impossible to workaround without storing the original strings (and once Sphinx supports storing the original strings, ordinals will not be necessary any more).

Example:

```
sql_attr_str2ordinal = author_name
```

9.1.21. sql_attr_float

Floating point attribute declaration. Multi-value (there might be multiple attributes declared), optional. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

The values will be stored in single precision, 32-bit IEEE 754 format. Represented range is approximately from 1e-38 to 1e+38. The amount of decimal digits that can be stored precisely is approximately 7. One important usage of the float attributes is storing latitude and longitude values (in radians), for further usage in query-time geosphere distance calculations.

Example:

```
sql_attr_float = lat_radians
sql_attr_float = long_radians
```

9.1.22. sql_attr_multi

Multi-valued attribute (MVA) declaration. Multi-value (ie. there may be more than one such attribute declared), optional. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

Plain attributes only allow to attach 1 value per each document. However, there are cases (such as tags or categories) when it is desired to attach multiple values of the same attribute and be able to apply filtering or grouping to value lists.

The declaration format is as follows (backslashes are for clarity only; everything can be declared in a single line as well):

```
sql_attr_multi = ATTR-TYPE ATTR-NAME 'from' SOURCE-TYPE \
        [;QUERY] \
        [;RANGE-QUERY]
```

where

- ATTR-TYPE is 'uint' or 'timestamp'
- SOURCE-TYPE is 'field', 'query', or 'ranged-query'
- QUERY is SQL query used to fetch all (docid, attrvalue) pairs
- RANGE-QUERY is SQL query used to fetch min and max ID values, similar to 'sql_query_range'

Example:

```
sql_attr_multi = uint tag from query; SELECT id, tag FROM tags
sql_attr_multi = uint tag from ranged-query; \
        SELECT id, tag FROM tags WHERE id>=$start AND id<=$end; \
        SELECT MIN(id), MAX(id) FROM tags
```

9.1.23. sql_query_post

Post-fetch query. Optional, default value is empty. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

This query is executed immediately after sql_query completes successfully. When post-fetch query produces errors, they are reported as warnings, but indexing is **not** terminated. It's result set is ignored. Note that indexing is **not** yet completed at the point when this query gets executed, and further indexing still may fail. Therefore, any permanent updates should not be done from here. For instance, updates on helper table that permanently change the last successfully indexed ID should not be run from post-fetch query; they should be run from post-index query instead.

Example:

```
sql_query_post = DROP TABLE my_tmp_table
```

9.1.24. sql_query_post_index

Post-index query. Optional, default value is empty. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

This query is executed when indexing is fully and succesfully completed. If this query produces errors, they are reported as warnings, but indexing is **not** terminated. It's result set is ignored. `$maxid` macro can be used in its text; it will be expanded to maximum document ID which was actually fetched from the database during indexing.

Example:

```
sql_query_post_index = REPLACE INTO counters ( id, val ) \
    VALUES ( 'max_indexed_id', $maxid )
```

9.1.25. sql_ranged_throttle

Ranged query throttling period, in milliseconds. Optional, default is 0 (no throttling). Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

Throttling can be useful when indexer imposes too much load on the database server. It causes the indexer to sleep for given amount of milliseconds once per each ranged query step. This sleep is unconditional, and is performed before the fetch query.

Example:

```
sql_ranged_throttle = 1000 # sleep for 1 sec before each query step
```

9.1.26. sql_query_info

Document info query. Optional, default is empty. Applies to `mysql` source type only.

Only used by CLI search to fetch and display document information, only works with MySQL at the moment, and only intended for debugging purposes. This query fetches the row that will be displayed by CLI search utility for each document ID. It is required to contain `$id` macro that expands to the queried document ID.

Example:

```
sql_query_info = SELECT * FROM documents WHERE id=$id
```

9.1.27. xmlpipe_command

Shell command that invokes xmlpipe stream producer. Mandatory. Applies to `xmlpipe` and `xmlpipe2` source types only.

Specifies a command that will be executed and which output will be parsed for documents. Refer to Section 3.8, "xmlpipe data source" or Section 3.9, "xmlpipe2 data source" for specific format description.

Example:

```
xmlpipe_command = cat /home/sphinx/test.xml
```

9.1.28. xmlpipe_field

xmlpipe field declaration. Multi-value, optional. Applies to `xmlpipe2` source type only. Refer to Section 3.9, "xmlpipe2 data source".

Example:

```
xmlpipe_field = subject
xmlpipe_field = content
```

9.1.29. xmlpipe_attr_uint

xmlpipe integer attribute declaration. Multi-value, optional. Applies to `xmlpipe2` source type only. Syntax fully matches that of sql_attr_uint.

Example:

```
xmlpipe_attr_uint = author
```

9.1.30. xmlpipe_attr_bool

xmlpipe boolean attribute declaration. Multi-value, optional. Applies to `xmlpipe2` source type only. Syntax fully matches that of sql_attr_bool.

Example:

```
xmlpipe_attr_bool = is_deleted # will be packed to 1 bit
```

9.1.31. xmlpipe_attr_timestamp

xmlpipe UNIX timestamp attribute declaration. Multi-value, optional. Applies to `xmlpipe2` source type only. Syntax fully matches that of sql_attr_timestamp.

Example:

```
xmlpipe_attr_timestamp = published
```

9.1.32. xmlpipe_attr_str2ordinal

xmlpipe string ordinal attribute declaration. Multi-value, optional. Applies to `xmlpipe2` source type only. Syntax fully matches that of sql_attr_str2ordinal.

Example:

```
xmlpipe_attr_str2ordinal = author_sort
```

9.1.33. xmlpipe_attr_float

xmlpipe floating point attribute declaration. Multi-value, optional. Applies to `xmlpipe2` source type only. Syntax fully matches that of sql_attr_float.

Example:

```
xmlpipe_attr_float = lat_radians
xmlpipe_attr_float = long_radians
```

9.1.34. xmlpipe_attr_multi

xmlpipe MVA attribute declaration. Multi-value, optional. Applies to `xmlpipe2` source type only.

This setting declares an MVA attribute tag in xmlpipe2 stream. The contents of the specified tag will be parsed and a list of integers that will constitute the MVA will be extracted, similar to how sql_attr_multi parses SQL column contents when 'field' MVA source type is specified.

Example:

```
xmlpipe_attr_multi = taglist
```

9.1.35. xmlpipe_fixup_utf8

Perform Sphinx-side UTF-8 validation and filtering to prevent XML parser from choking on non-UTF-8

documents. Optional, default is 0. Applies to `xmlpipe2` source type only.

Under certain occasions it might be hard or even impossible to guarantee that the incoming XMLpipe2 document bodies are in perfectly valid and conforming UTF-8 encoding. For instance, documents with national single-byte encodings could sneak into the stream. libexpat XML parser is fragile, meaning that it will stop processing in such cases. UTF8 fixup feature lets you avoid that. When fixup is enabled, Sphinx will preprocess the incoming stream before passing it to the XML parser and replace invalid UTF-8 sequences with spaces.

Example:

```
xmlpipe_fixup_utf8 = 1
```

9.1.36. mssql_winauth

MS SQL Windows authentication flag. Boolean, optional, default value is 0 (false). Applies to `mssql` source type only. Introduced in version 0.9.9-rc1.

Whether to use currently logged in Windows account credentials for authentication when connecting to MS SQL Server. Note that when running `searchd` as a service, account user can differ from the account you used to install the service.

Example:

```
mssql_winauth = 1
```

9.1.37. mssql_unicode

MS SQL encoding type flag. Boolean, optional, default value is 0 (false). Applies to `mssql` source type only. Introduced in version 0.9.9-rc1.

Whether to ask for Unicode or single-byte data when querying MS SQL Server. This flag **must** be in sync with charset_type directive; that is, to index Unicode data, you must set both `charset_type` in the index (to 'utf-8') and `mssql_unicode` in the source (to 1). For reference, MS SQL will actually return data in UCS-2 encoding instead of UTF-8, but Sphinx will automatically handle that.

Example:

```
mssql_unicode = 1
```

9.1.38. unpack_zlib

Columns to unpack using zlib (aka deflate, aka gunzip). Multi-value, optional, default value is empty list of columns. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only. Introduced in version 0.9.9-rc1.

Columns specified using this directive will be unpacked by `indexer` using standard zlib algorithm (called deflate and also implemented by `gunzip`). When indexing on a different box than the database, this lets you offload the database, and save on network traffic. The feature is only available if zlib and zlib-devel were both available during build time.

Example:

```
unpack_zlib = col1
unpack_zlib = col2
```

9.1.39. unpack_mysqlcompress

Columns to unpack using MySQL UNCOMPRESS() algorithm. Multi-value, optional, default value is empty list of columns. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only. Introduced in version 0.9.9-rc1.

Columns specified using this directive will be unpacked by `indexer` using modified zlib algorithm used by MySQL COMPRESS() and UNCOMPRESS() functions. When indexing on a different box than the database, this lets you

offload the database, and save on network traffic. The feature is only available if zlib and zlib-devel were both available during build time.

Example:

```
unpack_mysqlcompress = body_compressed
unpack_mysqlcompress = description_compressed
```

9.1.40. unpack_mysqlcompress_maxsize

Buffer size for UNCOMPRESS()ed data. Optional, default value is 16M. Introduced in version 0.9.9-rc1.

When using unpack_mysqlcompress, due to implementation intrincacies it is not possible to deduce the required buffer size from the compressed data. So the buffer must be preallocated in advance, and unpacked data can not go over the buffer size. This option lets you control the buffer size, both to limit `indexer` memory use, and to enable unpacking of really long data fields if necessary.

Example:

```
unpack_mysqlcompress_maxsize = 1M
```

9.2. Index configuration options

9.2.1. type

Index type. Optional, default is empty (index is plain local index). Known values are empty string or 'distributed'.

Sphinx supports two different types of indexes: local, that are stored and processed on the local machine; and distributed, that involve not only local searching but querying remote `searchd` instances over the network as well. Index type settings lets you choose this type. By default, indexes are local. Specifying 'distributed' for type enables distributed searching, see Section 4.7, “Distributed searching”.

Example:

```
type = distributed
```

9.2.2. source

Adds document source to local index. Multi-value, mandatory.

Specifies document source to get documents from when the current index is indexed. There must be at least one source. There may be multiple sources, without any restrictions on the source types: ie. you can pull part of the data from MySQL server, part from PostgreSQL, part from the filesystem using xmlpipe2 wrapper.

However, there are some restrictions on the source data. First, document IDs must be globally unique across all sources. If that condition is not met, you might get unexpected search results. Second, source schemas must be the same in order to be stored within the same index.

No source ID is stored automatically. Therefore, in order to be able to tell what source the matched document came from, you will need to store some additional information yourself. Two typical approaches include:

1. mangling document ID and encoding source ID in it:

```
source src1
{
        sql_query = SELECT id*10+1, ... FROM table1
        ...
}

source src2
```

```
{
        sql_query = SELECT id*10+2, ... FROM table2
        ...
}
```

2. storing source ID simply as an attribute:

```
source src1
{
        sql_query = SELECT id, 1 AS source_id FROM table1
        sql_attr_uint = source_id
        ...
}

source src2
{
        sql_query = SELECT id, 2 AS source_id FROM table2
        sql_attr_uint = source_id
        ...
}
```

Example:

```
source = srcpart1
source = srcpart2
source = srcpart3
```

9.2.3. path

Index files path and file name (without extension). Mandatory.

Path specifies both directory and file name, but without extension. `indexer` will append different extensions to this path when generating final names for both permanent and temporary index files. Permanent data files have several different extensions starting with '.sp'; temporary files' extensions start with '.tmp'. It's safe to remove `.tmp*` files is if indexer fails to remove them automatically.

For reference, different index files store the following data:

- `.spa` stores document attributes (used in extern docinfo storage mode only);
- `.spd` stores matching document ID lists for each word ID;
- `.sph` stores index header information;
- `.spi` stores word lists (word IDs and pointers to `.spd` file);
- `.spm` stores MVA data;
- `.spp` stores hit (aka posting, aka word occurence) lists for each word ID.

Example:

```
path = /var/data/test1
```

9.2.4. docinfo

Document attribute values (docinfo) storage mode. Optional, default is 'extern'. Known values are 'none', 'extern' and 'inline'.

Docinfo storage mode defines how exactly docinfo will be physically stored on disk and RAM. "none" means that there will be no docinfo at all (ie. no attributes). Normally you need not to set "none" explicitly because Sphinx will automatically select "none" when there are no attributes configured. "inline" means that the docinfo will be stored in the `.spd` file, along with the document ID lists. "extern" means that the docinfo will be stored separately (externally) from document ID lists, in a special `.spa` file.

Basically, externally stored docinfo must be kept in RAM when querying. for performance reasons. So in some cases "inline" might be the only option. However, such cases are infrequent, and docinfo defaults to "extern". Refer to Section 3.2, “Attributes” for in-depth discussion and RAM usage estimates.

Example:

```
docinfo = inline
```

9.2.5. mlock

Memory locking for cached data. Optional, default is 0 (do not call mlock()).

For search performance, `searchd` preloads a copy of `.spa` and `.spi` files in RAM, and keeps that copy in RAM at all times. But if there are no searches on the index for some time, there are no accesses to that cached copy, and OS might decide to swap it out to disk. First queries to such "cooled down" index will cause swap-in and their latency will suffer.

Setting mlock option to 1 makes Sphinx lock physical RAM used for that cached data using mlock(2) system call, and that prevents swapping (see man 2 mlock for details). mlock(2) is a privileged call, so it will require `searchd` to be either run from root account, or be granted enough privileges otherwise. If mlock() fails, a warning is emitted, but index continues working.

Example:

```
mlock = 1
```

9.2.6. morphology

A list of morphology preprocessors to apply. Optional, default is empty (do not apply any preprocessor).

Morphology preprocessors can be applied to the words being indexed to replace different forms of the same word with the base, normalized form. For instance, English stemmer will normalize both "dogs" and "dog" to "dog", making search results for both searches the same.

Built-in preprocessors include English stemmer, Russian stemmer (that supports UTF-8 and Windows-1251 encodings), Soundex, and Metaphone. The latter two replace the words with special phonetic codes that are equal is words are phonetically close. Additional stemmers provided by Snowball project libstemmer library can be enabled at compile time using `--with-libstemmer configure` option. Built-in English and Russian stemmers should be faster than their libstemmer counterparts, but can produce slightly different results, because they are based on an older version. Metaphone implementation is based on Double Metaphone algorithm and indexes the primary code.

Built-in values that be used in `morphology` option are: 'none', 'stem_en', 'stem_ru', 'stem_enru', 'soundex', and 'metaphone'. Additional values provided by libstemmer are in 'libstemmer_XXX' format, where XXX is libstemmer algorithm codename (refer to `libstemmer_c/libstemmer/modules.txt` for a complete list).

Several stemmers can be specified (comma-separated). They will be applied to incoming words in the order they are listed, and the processing will stop once one of the stemmers actually modifies the word. Also when wordforms feature is enabled the word will be looked up in word forms dictionary first, and if there is a matching entry in the dictionary, stemmers will not be applied at all. Or in other words, wordforms can be used to implement stemming exceptions.

Example:

```
morphology = stem_en, libstemmer_sv
```

9.2.7. min_stemming_len

Minimum word length at which to enable stemming. Optional, default is 1 (stem everything). Introduced in version 0.9.9-rc1.

Stemmers are not perfect, and might sometimes produce undesired results. For instance, running "gps" keyword through Porter stemmer for English results in "gp", which is not really the intent. `min_stemming_len` feature lets you suppress stemming based on the source word length, ie. to avoid stemming too short words. Keywords that are shorter than the given threshold will not be stemmed. Note that keywords that are exactly as long as specified **will** be stemmed. So in order to avoid stemming 3-character keywords, you should specify 4 for the

value. For more finely grained control, refer to wordforms feature.

Example:

```
min_stemming_len = 4
```

9.2.8. stopwords

Stopword files list (space separated). Optional, default is empty.

Stopwords are the words that will not be indexed. Typically you'd put most frequent words in the stopwords list because they do not add much value to search results but consume a lot of resources to process.

You can specify several file names, separated by spaces. All the files will be loaded. Stopwords file format is simple plain text. The encoding must match index encoding specified in charset_type. File data will be tokenized with respect to charset_table settings, so you can use the same separators as in the indexed data. The stemmers will also be applied when parsing stopwords file.

While stopwords are not indexed, they still do affect the keyword positions. For instance, assume that "the" is a stopword, that document 1 contains the line "in office", and that document 2 contains "in the office". Searching for "in office" as for exact phrase will only return the first document, as expected, even though "the" in the second one is stopped.

Example:

```
stopwords = /usr/local/sphinx/data/stopwords.txt
stopwords = stopwords-ru.txt stopwords-en.txt
```

9.2.9. wordforms

Word forms dictionary. Optional, default is empty.

Word forms are applied after tokenizing the incoming text by charset_table rules. They essentialy let you replace one word with another. Normally, that would be used to bring different word forms to a single normal form (eg. to normalize all the variants such as "walks", "walked", "walking" to the normal form "walk"). It can also be used to implement stemming exceptions, because stemming is not applied to words found in the forms list.

Dictionaries are used to normalize incoming words both during indexing and searching. Therefore, to pick up changes in wordforms file it's required to reindex and restart `searchd`.

Word forms support in Sphinx is designed to support big dictionaries well. They moderately affect indexing speed: for instance, a dictionary with 1 million entries slows down indexing about 1.5 times. Searching speed is not affected at all. Additional RAM impact is roughly equal to the dictionary file size, and dictionaries are shared across indexes: ie. if the very same 50 MB wordforms file is specified for 10 different indexes, additional `searchd` RAM usage will be about 50 MB.

Dictionary file should be in a simple plain text format. Each line should contain source and destination word forms, in exactly the same encoding as specified in charset_type, separated by "greater" sign. Rules from the charset_table will be applied when the file is loaded. So basically it's as case sensitive as your other full-text indexed data, ie. typically case insensitive. Here's the file contents sample:

```
walks > walk
walked > walk
walking > walk
```

There is bundled `spelldump` utility that helps you create a dictionary file in the format Sphinx can read from source `.dict` and `.aff` dictionary files in `ispell` or `MySpell` format (as bundled with OpenOffice).

Starting with version 0.9.9-rc1, you can map several source words to a single destination word. Because the work happens on tokens, not the source text, differences in whitespace and markup are ignored.

```
core 2 duo > c2d
e6600 > c2d
core 2duo > c2d
```

Example:

```
wordforms = /usr/local/sphinx/data/wordforms.txt
```

9.2.10. exceptions

Tokenizing exceptions file. Optional, default is empty.

Exceptions allow to map one or more tokens (including tokens with characters that would normally be excluded) to a single keyword. They are similar to wordforms in that they also perform mapping, but have a number of important differences.

Short summary of the differences is as follows:

- exceptions are case sensitive, wordforms are not;
- exceptions allow to detect sequences of tokens, wordforms work with single words only;
- exceptions can use special characters that are **not** in charset_table, wordforms fully obey charset_table;
- exceptions can underperform on huge dictionaries, wordforms handle millions of entries well.

The expected file format is also plain text, with one line per exception, and the line format is as follows:

```
map-from-tokens => map-to-token
```

Example file:

```
AT & T => AT&T
AT&T => AT&T
Standarten   Fuehrer => standartenfuhrer
Standarten Fuhrer => standartenfuhrer
MS Windows => ms windows
Microsoft Windows => ms windows
C++ => cplusplus
c++ => cplusplus
C plus plus => cplusplus
```

All tokens here are case sensitive: they will **not** be processed by charset_table rules. Thus, with the example exceptions file above, "At&t" text will be tokenized as two keywords "at" and "t", because of lowercase letters. On the other hand, "AT&T" will match exactly and produce single "AT&T" keyword.

Note that this map-to keyword is a) always interpereted as a *single* word, and b) is both case and space sensitive! In our sample, "ms windows" query will *not* match the document with "MS Windows" text. The query will be interpreted as a query for two keywords, "ms" and "windows". And what "MS Windows" gets mapped to is a *single* keyword "ms windows", with a space in the middle. On the other hand, "standartenfuhrer" will retrieve documents with "Standarten Fuhrer" or "Standarten Fuehrer" contents (capitalized exactly like this), or any capitalization variant of the keyword itself, eg. "staNdarTenfUhreR". (It won't catch "standarten fuhrer", however: this text does not match any of the listed exceptions because of case sensitivity, and gets indexed as two separate keywords.)

Whitespace in the map-from tokens list matters, but its amount does not. Any amount of the whitespace in the map-form list will match any other amount of whitespace in the indexed document or query. For instance, "AT & T" map-from token will match "AT & T" text, whatever the amount of space in both map-from part and the indexed text. Such text will therefore be indexed as a special "AT&T" keyword, thanks to the very first entry from the sample.

Exceptions also allow to capture special characters (that are exceptions from general charset_table rules; hence the name). Assume that you generally do not want to treat '+' as a valid character, but still want to be able search for some exceptions from this rule such as 'C++'. The sample above will do just that, totally independent of what characters are in the table and what are not.

Exceptions are applied to raw incoming document and query data during indexing and searching respectively. Therefore, to pick up changes in the file it's required to reindex and restart `searchd`.

Example:

```
exceptions = /usr/local/sphinx/data/exceptions.txt
```

9.2.11. min_word_len

Minimum indexed word length. Optional, default is 1 (index everything).

Only those words that are not shorter than this minimum will be indexed. For instance, if min_word_len is 4, then 'the' won't be indexed, but 'they' will be.

Example:

```
min_word_len = 4
```

9.2.12. charset_type

Character set encoding type. Optional, default is 'sbcs'. Known values are 'sbcs' and 'utf-8'.

Different encodings have different methods for mapping their internal characters codes into specific byte sequences. Two most common methods in use today are single-byte encoding and UTF-8. Their corresponding charset_type values are 'sbcs' (stands for Single Byte Character Set) and 'utf-8'. The selected encoding type will be used everywhere where the index is used: when indexing the data, when parsing the query against this index, when generating snippets, etc.

Note that while 'utf-8' implies that the decoded values must be treated as Unicode codepoint numbers, there's a family of 'sbcs' encodings that may in turn treat different byte values differently, and that should be properly reflected in your charset_table settings. For example, the same byte value of 224 (0xE0 hex) maps to different Russian letters depending on whether koi-8r or windows-1251 encoding is used.

Example:

```
charset_type = utf-8
```

9.2.13. charset_table

Accepted characters table, with case folding rules. Optional, default value depends on charset_type value.

charset_table is the main workhorse of Sphinx tokenizing process, ie. the process of extracting keywords from document text or query txet. It controls what characters are accepted as valid and what are not, and how the accepted characters should be transformed (eg. should the case be removed or not).

You can think of charset_table as of a big table that has a mapping for each and every of 100K+ characters in Unicode (or as of a small 256-character table if you're using SBCS). By default, every character maps to 0, which means that it does not occur within keywords and should be treated as a separator. Once mentioned in the table, character is mapped to some other character (most frequently, either to itself or to a lowercase letter), and is treated as a valid keyword part.

The expected value format is a commas-separated list of mappings. Two simplest mappings simply declare a character as valid, and map a single character to another single character, respectively. But specifying the whole table in such form would result in bloated and barely manageable specifications. So there are several syntax shortcuts that let you map ranges of characters at once. The complete list is as follows:

A->a
: Single char mapping, declares source char 'A' as allowed to occur within keywords and maps it to destination char 'a' (but does *not* declare 'a' as allowed).

A..Z->a..z
: Range mapping, declares all chars in source range as allowed and maps them to the destination range. Does *not* declare destination range as allowed. Also checks ranges' lengths (the lengths must be equal).

a
: Stray char mapping, declares a character as allowed and maps it to itself. Equivalent to a->a single char mapping.

a..z
: Stray range mapping, declares all characters in range as allowed and maps them to themselves. Equivalent to a..z->a..z range mapping.

A..Z/2
: Checkerboard range map. Maps every pair of chars to the second char. More formally, declares odd characters in range as allowed and maps them to the even ones; also declares even characters as allowed and maps them to themselves. For instance, A..Z/2 is equivalent to A->B, B->B, C->D, D->D, ..., Y->Z, Z->Z. This mapping shortcut is helpful for a number of Unicode blocks where uppercase and lowercase letters go in such interleaved order instead of contiguous chunks.

Control characters with codes from 0 to 31 are always treated as separators. Characters with codes 32 to 127, ie. 7-bit ASCII characters, can be used in the mappings as is. To avoid configuration file encoding issues, 8-bit ASCII characters and Unicode characters must be specified in U+xxx form, where 'xxx' is hexadecimal codepoint number. This form can also be used for 7-bit ASCII characters to encode special ones: eg. use U+20 to encode space, U+2E to encode dot, U+2C to encode comma.

Example:

```
# 'sbcs' defaults for English and Russian
charset_table = 0..9, A..Z->a..z, _, a..z, \
        U+A8->U+B8, U+B8, U+C0..U+DF->U+E0..U+FF, U+E0..U+FF

# 'utf-8' defaults for English and Russian
charset_table = 0..9, A..Z->a..z, _, a..z, \
        U+410..U+42F->U+430..U+44F, U+430..U+44F
```

9.2.14. ignore_chars

Ignored characters list. Optional, default is empty.

Useful in the cases when some characters, such as soft hyphenation mark (U+00AD), should be not just treated as separators but rather fully ignored. For example, if '-' is simply not in the charset_table, "abc-def" text will be indexed as "abc" and "def" keywords. On the contrary, if '-' is added to ignore_chars list, the same text will be indexed as a single "abcdef" keyword.

The syntax is the same as for charset_table, but it's only allowed to declare characters, and not allowed to map them. Also, the ignored characters must not be present in charset_table.

Example:

```
ignore_chars = U+AD
```

9.2.15. min_prefix_len

Minimum word prefix length to index. Optional, default is 0 (do not index prefixes).

Prefix indexing allows to implement wildcard searching by 'wordstart*' wildcards (refer to enable_star option for details on wildcard syntax). When mininum prefix length is set to a positive number, indexer will index all the possible keyword prefixes (ie. word beginnings) in addition to the keywords themselves. Too short prefixes (below the minimum allowed length) will not be indexed.

For instance, indexing a keyword "example" with min_prefix_len=3 will result in indexing "exa", "exam", "examp", "exampl" prefixes along with the word itself. Searches against such index for "exam" will match documents that contain "example" word, even if they do not contain "exam" on itself. However, indexing prefixes will make the index grow significantly (because of many more indexed keywords), and will degrade both indexing and searching times.

There's no automatic way to rank perfect word matches higher in a prefix index, but there's a number of tricks to achieve that. First, you can setup two indexes, one with prefix indexing and one without it, search through both, and use SetIndexWeights() call to combine weights. Second, you can enable star-syntax and rewrite your

extended-mode queries:

```
# in sphinx.conf
enable_star = 1

// in query
$cl->Query ( "( keyword | keyword* ) other keywords" );
```

Example:

```
min_prefix_len = 3
```

9.2.16. min_infix_len

Minimum infix prefix length to index. Optional, default is 0 (do not index infixes).

Infix indexing allows to implement wildcard searching by 'start*', '*end', and '*middle*' wildcards (refer to enable_star option for details on wildcard syntax). When mininum infix length is set to a positive number, indexer will index all the possible keyword infixes (ie. substrings) in addition to the keywords themselves. Too short infixes (below the minimum allowed length) will not be indexed.

For instance, indexing a keyword "test" with min_infix_len=2 will result in indexing "te", "es", "st", "tes", "est" infixes along with the word itself. Searches against such index for "es" will match documents that contain "test" word, even if they do not contain "es" on itself. However, indexing infixes will make the index grow significantly (because of many more indexed keywords), and will degrade both indexing and searching times.

There's no automatic way to rank perfect word matches higher in an infix index, but the same tricks as with prefix indexes can be applied.

Example:

```
min_infix_len = 3
```

9.2.17. prefix_fields

The list of full-text fields to limit prefix indexing to. Optional, default is empty (index all fields in prefix mode).

Because prefix indexing impacts both indexing and searching performance, it might be desired to limit it to specific full-text fields only: for instance, to provide prefix searching through URLs, but not through page contents. prefix_fields specifies what fields will be prefix-indexed; all other fields will be indexed in normal mode. The value format is a comma-separated list of field names.

Example:

```
prefix_fields = url, domain
```

9.2.18. infix_fields

The list of full-text fields to limit infix indexing to. Optional, default is empty (index all fields in infix mode).

Similar to prefix_fields, but lets you limit infix-indexing to given fields.

Example:

```
infix_fields = url, domain
```

9.2.19. enable_star

Enables star-syntax (or wildcard syntax) when searching through prefix/infix indexes. Optional, default is is 0 (do not use wildcard syntax), for compatibility with 0.9.7. Known values are 0 and 1.

This feature enables "star-syntax", or wildcard syntax, when searching through indexes which were created with prefix or infix indexing enabled. It only affects searching; so it can be changed without reindexing by simply restarting `searchd`.

The default value is 0, that means to disable star-syntax and treat all keywords as prefixes or infixes respectively, depending on indexing-time min_prefix_len and min_infix_len settings. The value of 1 means that star ('*') can be used at the start and/or the end of the keyword. The star will match zero or more characters.

For example, assume that the index was built with infixes and that enable_star is 1. Searching should work as follows:

1. "abcdef" query will match only those documents that contain the exact "abcdef" word in them.
2. "abc*" query will match those documents that contain any words starting with "abc" (including the documents which contain the exact "abc" word only);
3. "*cde*" query will match those documents that contain any words which have "cde" characters in any part of the word (including the documents which contain the exact "cde" word only).
4. "*def" query will match those documents that contain any words ending with "def" (including the documents that contain the exact "def" word only).

Example:

```
enable_star = 1
```

9.2.20. ngram_len

N-gram lengths for N-gram indexing. Optional, default is 0 (disable n-gram indexing). Known values are 0 and 1 (other lengths to be implemented).

N-grams provide basic CJK (Chinese, Japanse, Koreasn) support for unsegmented texts. The issue with CJK searching is that there could be no clear separators between the words. Ideally, the texts would be filtered through a special program called segmenter that would insert separators in proper locations. However, segmenters are slow and error prone, and it's common to index contiguous groups of N characters, or n-grams, instead.

When this feature is enabled, streams of CJK characters are indexed as N-grams. For example, if incoming text is "ABCDEF" (where A to F represent some CJK characters) and length is 1, in will be indexed as if it was "A B C D E F". (With length equal to 2, it would produce "AB BC CD DE EF"; but only 1 is supported at the moment.) Only those characters that are listed in ngram_chars table will be split this way; other ones will not be affected.

Note that if search query is segmented, ie. there are separators between individual words, then wrapping the words in quotes and using extended mode will resut in proper matches being found even if the text was **not** segmented. For instance, assume that the original query is BC DEF. After wrapping in quotes on the application side, it should look like "BC" "DEF" (*with* quotes). This query will be passed to Sphinx and internally split into 1-grams too, resulting in "B C" "D E F" query, still with quotes that are the phrase matching operator. And it will match the text even though there were no separators in the text.

Even if the search query is not segmented, Sphinx should still produce good results, thanks to phrase based ranking: it will pull closer phrase matches (which in case of N-gram CJK words can mean closer multi-character word matches) to the top.

Example:

```
ngram_len = 1
```

9.2.21. ngram_chars

N-gram characters list. Optional, default is empty.

To be used in conjunction with in ngram_len, this list defines characters, sequences of which are subject to N-gram extraction. Words comprised of other characters will not be affected by N-gram indexing feature. The value format is identical to charset_table.

Example:

```
ngram_chars = U+3000..U+2FA1F
```

9.2.22. phrase_boundary

Phrase boundary characters list. Optional, default is empty.

This list controls what characters will be treated as phrase boundaries, in order to adjust word positions and enable phrase-level search emulation through proximity search. The syntax is similar to charset_table. Mappings are not allowed and the boundary characters must not overlap with anything else.

On phrase boundary, additional word position increment (specified by phrase_boundary_step) will be added to current word position. This enables phrase-level searching through proximity queries: words in different phrases will be guaranteed to be more than phrase_boundary_step distance away from each other; so proximity search within that distance will be equivalent to phrase-level search.

Phrase boundary condition will be raised if and only if such character is followed by a separator; this is to avoid abbreviations such as S.T.A.L.K.E.R or URLs being treated as several phrases.

Example:

```
phrase_boundary = ., ?, !, U+2026 # horizontal ellipsis
```

9.2.23. phrase_boundary_step

Phrase boundary word position increment. Optional, default is 0.

On phrase boundary, current word position will be additionally incremented by this number. See phrase_boundary for details.

Example:

```
phrase_boundary_step = 100
```

9.2.24. html_strip

Whether to strip HTML markup from incoming full-text data. Optional, default is 0. Known values are 0 (disable stripping) and 1 (enable stripping).

Stripping does not work with `xmlpipe` source type (it's suggested to upgrade to xmlpipe2 anyway). It should work with properly formed HTML and XHTML, but, just as most browsers, may produce unexpected results on malformed input (such as HTML with stray <'s or unclosed >'s).

Only the tags themselves, and also HTML comments, are stripped. To strip the contents of the tags too (eg. to strip embedded scripts), see html_remove_elements option. There are no restrictions on tag names; ie. everything that looks like a valid tag start, or end, or a comment will be stripped.

Example:

```
html_strip = 1
```

9.2.25. html_index_attrs

A list of markup attributes to index when stripping HTML. Optional, default is empty (do not index markup attributes).

Specifies HTML markup attributes whose contents should be retained and indexed even though other HTML markup is stripped. The format is per-tag enumeration of indexable attributes, as shown in the example below.

Example:

```
html_index_attrs = img=alt,title; a=title;
```

9.2.26. html_remove_elements

A list of HTML elements for which to strip contents along with the elements themselves. Optional, default is empty string (do not strip contents of any elements).

This feature allows to strip element contents, ie. everything that is between the opening and the closing tags. It is useful to remove embedded scripts, CSS, etc. Short tag form for empty elements (ie.
) is properly supported; ie. the text that follows such tag will **not** be removed.

The value is a comma-separated list of element (tag) names whose contents should be removed. Tag names are case insensitive.

Example:

```
html_remove_elements = style, script
```

9.2.27. local

Local index declaration in the distributed index. Multi-value, optional, default is empty.

This setting is used to declare local indexes that will be searched when given distributed index is searched. All local indexes will be searched **sequentially**, utilizing only 1 CPU or core; to parallelize processing, you can configure `searchd` to query itself (refer to Section 9.2.28, "agent" for the details). There might be several local indexes declared per each distributed index. Any local index can be mentioned several times in other distributed indexes.

Example:

```
local = chunk1
local = chunk2
```

9.2.28. agent

Remote agent declaration in the distributed index. Multi-value, optional, default is empty.

This setting is used to declare remote agents that will be searched when given distributed index is searched. The agents can be thought of as network pointers that specify host, port, and index names. In the basic case agents would correspond to remote physical machines. More formally, that is not always correct: you can point several agents to the same remote machine; or you can even point agents to the very same single instance of `searchd` (in order to utilize many CPUs or cores).

The value format is as follows:

```
agent = specification:remote-indexes-list
specification = hostname ":" port | path
```

Where 'hostname' is remote host name; 'port' is remote TCP port; 'path' is Unix-domain socket path and 'remote-indexes-list' is a comma-separated list of remote index names.

All agents will be searched in parallel. However, all indexes specified for a given agent will be searched sequentially in this agent. This lets you fine-tune the configuration to the hardware. For instance, if two remote indexes are stored on the same physical HDD, it's better to configure one agent with several sequentially searched indexes to avoid HDD steping. If they are stored on different HDDs, having two agents will be advantageous, because the work will be fully parallelized. The same applies to CPUs; though CPU performance impact caused by two processes stepping on each other is somewhat smaller and frequently can be ignored at all.

On machines with many CPUs and/or HDDs, agents can be pointed to the same machine to utilize all of the hardware in parallel and reduce query latency. There is no need to setup several `searchd` instances for that; it's legal to configure the instance to contact itself. Here's an example setup, intended for a 4-CPU machine, that will use up to 4 CPUs in parallel to process each query:

```
index dist
{
        type = distributed
        local = chunk1
        agent = localhost:9312:chunk2
        agent = localhost:9312:chunk3
        agent = localhost:9312:chunk4
}
```

Note how one of the chunks is searched locally and the same instance of searchd queries itself to launch searches through three other ones in parallel.

Example:

```
agent = localhost:9312:chunk2 # contact itself
agent = /var/run/searchd.s:chunk2
agent = searchbox2:9312:chunk3,chunk4 # search remote indexes
```

9.2.29. agent_blackhole

Remote blackhole agent declaration in the distributed index. Multi-value, optional, default is empty. Introduced in version 0.9.9-rc1.

`agent_blackhole` lets you fire-and-forget queries to remote agents. That is useful for debugging (or just testing) production clusters: you can setup a separate debugging/testing searchd instance, and forward the requests to this instance from your production master (aggregator) instance without interfering with production work. Master searchd will attempt to connect and query blackhole agent normally, but it will neither wait nor process any responses. Also, all network errors on blackhole agents will be ignored. The value format is completely identical to regular agent directive.

Example:

```
agent_blackhole = testbox:9312:testindex1,testindex2
```

9.2.30. agent_connect_timeout

Remote agent connection timeout, in milliseconds. Optional, default is 1000 (ie. 1 second).

When connecting to remote agents, `searchd` will wait at most this much time for connect() call to complete succesfully. If the timeout is reached but connect() does not complete, and retries are enabled, retry will be initiated.

Example:

```
agent_connect_timeout = 300
```

9.2.31. agent_query_timeout

Remote agent query timeout, in milliseconds. Optional, default is 3000 (ie. 3 seconds).

After connection, `searchd` will wait at most this much time for remote queries to complete. This timeout is fully separate from connection timeout; so the maximum possible delay caused by a remote agent equals to the sum of `agent_connection_timeout` and `agent_query_timeout`. Queries will **not** be retried if this timeout is reached; a warning will be produced instead.

Example:

```
agent_query_timeout = 10000 # our query can be long, allow up to 10 sec
```

9.2.32. preopen

Whether to pre-open all index files, or open them per each query. Optional, default is 0 (do not preopen).

This option tells `searchd` that it should pre-open all index files on startup (or rotation) and keep them open while it runs. Currently, the default mode is **not** to pre-open the files (this may change in the future). Preopened indexes take a few (currently 2) file descriptors per index. However, they save on per-query `open()` calls; and also they are invulnerable to subtle race conditions that may happen during index rotation under high load. On the other hand, when serving many indexes (100s to 1000s), it still might be desired to open the on per-query basis in order to save file descriptors.

This directive does not affect `indexer` in any way, it only affects `searchd`.

Example:

```
preopen = 1
```

9.2.33. ondisk_dict

Whether to keep the dictionary file (.spi) for this index on disk, or precache it in RAM. Optional, default is 0 (precache in RAM). Introduced in version 0.9.9-rc1.

The dictionary (.spi) can be either kept on RAM or on disk. The default is to fully cache it in RAM. That improves performance, but might cause too much RAM pressure, especially if prefixes or infixes were used. Enabling `ondisk_dict` results in 1 additional disk IO per keyword per query, but reduces memory footprint.

This directive does not affect `indexer` in any way, it only affects `searchd`.

Example:

```
ondisk_dict = 1
```

9.2.34. inplace_enable

Whether to enable in-place index inversion. Optional, default is 0 (use separate temporary files). Introduced in version 0.9.9-rc1.

`inplace_enable` greatly reduces indexing disk footprint, at a cost of slightly slower indexing (it uses around 2x less disk, but yields around 90-95% the original performance).

Indexing involves two major phases. The first phase collects, processes, and partially sorts documents by keyword, and writes the intermediate result to temporary files (.tmp*). The second phase fully sorts the documents, and creates the final index files. Thus, rebuilding a production index on the fly involves around 3x peak disk footprint: 1st copy for the intermediate temporary files, 2nd copy for newly constructed copy, and 3rd copy for the old index that will be serving production queries in the meantime. (Intermediate data is comparable in size to the final index.) That might be too much disk footprint for big data collections, and `inplace_enable` allows to reduce it. When enabled, it reuses the temporary files, outputs the final data back to them, and renames them on completion. However, this might require additional temporary data chunk relocation, which is where the performance impact comes from.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
inplace_enable = 1
```

9.2.35. inplace_hit_gap

In-place inversion fine-tuning option. Controls preallocated hitlist gap size. Optional, default is 0. Introduced in version 0.9.9-rc1.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
inplace_hit_gap = 1M
```

9.2.36. inplace_docinfo_gap

In-place inversion fine-tuning option. Controls preallocated docinfo gap size. Optional, default is 0. Introduced in version 0.9.9-rc1.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
inplace_docinfo_gap = 1M
```

9.2.37. inplace_reloc_factor

In-place inversion fine-tuning option. Controls relocation buffer size within indexing memory arena. Optional, default is 0.1. Introduced in version 0.9.9-rc1.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
inplace_reloc_factor = 0.1
```

9.2.38. inplace_write_factor

In-place inversion fine-tuning option. Controls in-place write buffer size within indexing memory arena. Optional, default is 0.1. Introduced in version 0.9.9-rc1.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
inplace_write_factor = 0.1
```

9.2.39. index_exact_words

Whether to index the original keywords along with the stemmed/remapped versions. Optional, default is 0 (do not index). Introduced in version 0.9.9-rc1.

When enabled, `index_exact_words` forces `indexer` to put the raw keywords in the index along with the stemmed versions. That, in turn, enables exact form operator in the query language to work. This impacts the index size and the indexing time. However, searching performance is not impacted at all.

Example:

```
index_exact_words = 1
```

9.2.40. overshort_step

Position increment on overshort (less that min_word_len) keywords. Optional, allowed values are 0 and 1, default is 1. Introduced in version 0.9.9-rc1.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
overshort_step = 1
```

9.2.41. stopword_step

Position increment on stopwords. Optional, allowed values are 0 and 1, default is 1. Introduced in version 0.9.9-rc1.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
stopword_step = 1
```

9.3. `indexer` program configuration options

9.3.1. mem_limit

Indexing RAM usage limit. Optional, default is 32M.

Enforced memory usage limit that the `indexer` will not go above. Can be specified in bytes, or kilobytes (using K postfix), or megabytes (using M postfix); see the example. This limit will be automatically raised if set to extremely low value causing I/O buffers to be less than 8 KB; the exact lower bound for that depends on the indexed data size. If the buffers are less than 256 KB, a warning will be produced.

Maximum possible limit is 2047M. Too low values can hurt indexing speed, but 256M to 1024M should be enough for most if not all datasets. Setting this value too high can cause SQL server timeouts. During the document collection phase, there will be periods when the memory buffer is partially sorted and no communication with the database is performed; and the database server can timeout. You can resolve that either by raising timeouts on SQL server side or by lowering `mem_limit`.

Example:

```
mem_limit = 256M
# mem_limit = 262144K # same, but in KB
# mem_limit = 268435456 # same, but in bytes
```

9.3.2. max_iops

Maximum I/O operations per second, for I/O throttling. Optional, default is 0 (unlimited).

I/O throttling related option. It limits maximum count of I/O operations (reads or writes) per any given second. A value of 0 means that no limit is imposed.

`indexer` can cause bursts of intensive disk I/O during indexing, and it might desired to limit its disk activity (and keep something for other programs running on the same machine, such as `searchd`). I/O throttling helps to do that. It works by enforcing a minimum guaranteed delay between subsequent disk I/O operations performed by `indexer`. Modern SATA HDDs are able to perform up to 70-100+ I/O operations per second (that's mostly limited by disk heads seek time). Limiting indexing I/O to a fraction of that can help reduce search performance dedgradation caused by indexing.

Example:

```
max_iops = 40
```

9.3.3. max_iosize

Maximum allowed I/O operation size, in bytes, for I/O throttling. Optional, default is 0 (unlimited).

I/O throttling related option. It limits maximum file I/O operation (read or write) size for all operations performed by `indexer`. A value of 0 means that no limit is imposed. Reads or writes that are bigger than the limit will be split in several smaller operations, and counted as several operation by max_iops setting. At the time of this writing, all I/O calls should be under 256 KB (default internal buffer size) anyway, so `max_iosize` values higher than 256 KB must not affect anything.

Example:

```
max_iosize = 1048576
```

9.3.4. max_xmlpipe2_field

Maximum allowed field size for XMLpipe2 source type, bytes. Optional, default is 2 MB.

Example:

```
max_xmlpipe2_field = 8M
```

9.3.5. write_buffer

Write buffer size, bytes. Optional, default is 1 MB.

Write buffers are used to write both temporary and final index files when indexing. Larger buffers reduce the number of required disk writes. Memory for the buffers is allocated in addition to mem_limit. Note that several (currently up to 4) buffers for different files will be allocated, proportionally increasing the RAM usage.

Example:

```
write_buffer = 4M
```

9.4. searchd program configuration options

9.4.1. listen

This setting lets you specify IP address and port, or Unix-domain socket path, that searchd will listen on. Introduced in version 0.9.9-rc1.

The informal grammar for listen setting is:

```
listen = ( address ":" port | port | path ) [ ":" protocol ]
```

I.e. you can specify either an IP address (or hostname) and port number, or just a port number, or Unix socket path. If you specify port number but not the address, searchd will listen on all network interfaces. Unix path is identified by a leading slash.

Starting with version 0.9.9-rc2, you can also specify a protocol handler (listener) to be used for connections on this socket. Supported protocol values are 'sphinx' (Sphinx 0.9.x API protocol) and 'mysql41' (MySQL protocol used since 4.1 upto at least 5.1). More details on MySQL protocol support can be found in Section 4.9, “MySQL protocol support and SphinxQL” section.

Examples:

```
listen = localhost
listen = localhost:5000
listen = 192.168.0.1:5000
listen = /var/run/sphinx.s
listen = 9312
listen = localhost:9306:mysql41
```

There can be multiple listen directives, searchd will listen for client connections on all specified ports and sockets. If no listen directive is found then the server will listen on all available interfaces using the default port (which is 9312).

Unix-domain sockets are not supported on Windows.

9.4.2. address

Interface IP address to bind on. Optional, default is 0.0.0.0 (ie. listen on all interfaces). **DEPRECATED**, use

listen instead.

`address` setting lets you specify which network interface `searchd` will bind to, listen on, and accept incoming network connections on. The default value is 0.0.0.0 which means to listen on all interfaces. At the time, you can **not** specify multiple interfaces.

Example:

```
address = 192.168.0.1
```

9.4.3. port

`searchd` TCP port number. **DEPRECATED**, use listen instead. Used to be mandatory. Default port number is 9312.

Example:

```
port = 9312
```

9.4.4. log

Log file name. Optional, default is 'searchd.log'. All `searchd` run time events will be logged in this file.

Example:

```
log = /var/log/searchd.log
```

9.4.5. query_log

Query log file name. Optional, default is empty (do not log queries). All search queries will be logged in this file. The format is described in Section 4.8, "`searchd` query log format".

Example:

```
query_log = /var/log/query.log
```

9.4.6. read_timeout

Network client request read timeout, in seconds. Optional, default is 5 seconds. `searchd` will forcibly close the client connections which fail to send a query within this timeout.

Example:

```
read_timeout = 1
```

9.4.7. client_timeout

Maximum time to wait between requests (in seconds) when using persistent connections. Optional, default is five minutes.

Example:

```
client_timeout = 3600
```

9.4.8. max_children

Maximum amount of children to fork (or in other words, concurrent searches to run in parallel). Optional, default is 0 (unlimited).

Useful to control server load. There will be no more than this much concurrent searches running, at all times.

When the limit is reached, additional incoming clients are dismissed with temporarily failure (SEARCHD_RETRY) status code and a message stating that the server is maxed out.

Example:

```
max_children = 10
```

9.4.9. pid_file

`searchd` process ID file name. Mandatory.

PID file will be re-created (and locked) on startup. It will contain head daemon process ID while the daemon is running, and it will be unlinked on daemon shutdown. It's mandatory because Sphinx uses it internally for a number of things: to check whether there already is a running instance of `searchd`; to stop `searchd`; to notify it that it should rotate the indexes. Can also be used for different external automation scripts.

Example:

```
pid_file = /var/run/searchd.pid
```

9.4.10. max_matches

Maximum amount of matches that the daemon keeps in RAM for each index and can return to the client. Optional, default is 1000.

Introduced in order to control and limit RAM usage, `max_matches` setting defines how much matches will be kept in RAM while searching each index. Every match found will still be *processed*; but only best N of them will be kept in memory and return to the client in the end. Assume that the index contains 2,000,000 matches for the query. You rarely (if ever) need to retrieve *all* of them. Rather, you need to scan all of them, but only choose "best" at most, say, 500 by some criteria (ie. sorted by relevance, or price, or anything else), and display those 500 matches to the end user in pages of 20 to 100 matches. And tracking only the best 500 matches is much more RAM and CPU efficient than keeping all 2,000,000 matches, sorting them, and then discarding everything but the first 20 needed to display the search results page. `max_matches` controls N in that "best N" amount.

This parameter noticeably affects per-query RAM and CPU usage. Values of 1,000 to 10,000 are generally fine, but higher limits must be used with care. Recklessly raising `max_matches` to 1,000,000 means that `searchd` will have to allocate and initialize 1-million-entry matches buffer for *every* query. That will obviously increase per-query RAM usage, and in some cases can also noticeably impact performance.

CAVEAT EMPTOR! Note that there also is **another** place where this limit is enforced. `max_matches` can be decreased on the fly through the corresponding API call, and the default value in the API is **also** set to 1,000. So in order to retrieve more than 1,000 matches to your application, you will have to change the configuration file, restart searchd, and set proper limit in SetLimits() call. Also note that you can not set the value in the API higher than the value in the .conf file. This is prohibited in order to have some protection against malicious and/or malformed requests.

Example:

```
max_matches = 10000
```

9.4.11. seamless_rotate

Prevents `searchd` stalls while rotating indexes with huge amounts of data to precache. Optional, default is 1 (enable seamless rotation).

Indexes may contain some data that needs to be precached in RAM. At the moment, `.spa`, `.spi` and `.spm` files are fully precached (they contain attribute data, MVA data, and keyword index, respectively.) Without seamless rotate, rotating an index tries to use as little RAM as possible and works as follows:

1. new queries are temporarily rejected (with "retry" error code);
2. `searchd` waits for all currently running queries to finish;

3. old index is deallocated and its files are renamed;
4. new index files are renamed and required RAM is allocated;
5. new index attribute and dictionary data is preloaded to RAM;
6. `searchd` resumes serving queries from new index.

However, if there's a lot of attribute or dictionary data, then preloading step could take noticeble time - up to several minutes in case of preloading 1-5+ GB files.

With seamless rotate enabled, rotation works as follows:

1. new index RAM storage is allocated;
2. new index attribute and dictionary data is asynchronously preloaded to RAM;
3. on success, old index is deallocated and both indexes' files are renamed;
4. on failure, new index is deallocated;
5. at any given moment, queries are served either from old or new index copy.

Seamless rotate comes at the cost of higher **peak** memory usage during the rotation (because both old and new copies of `.spa/.spi/.spm` data need to be in RAM while preloading new copy). Average usage stays the same.

Example:

```
seamless_rotate = 1
```

9.4.12. preopen_indexes

Whether to forcibly preopen all indexes on startup. Optional, default is 0 (do not preopen). Enforces enabled preopen on all served indexes, to avoid manually specifying it in every index.

Example:

```
preopen_indexes = 1
```

9.4.13. unlink_old

Whether to unlink .old index copies on succesful rotation. Optional, default is 1 (do unlink).

Example:

```
unlink_old = 0
```

9.4.14. attr_flush_period

When calling `UpdateAttributes()` to update document attributes in real-time, changes are first written to the in-memory copy of attributes (`docinfo` must be set to `extern`). Then, once `searchd` shuts down normally (via `SIGTERM` being sent), the changes are written to disk. Introduced in version 0.9.9-rc1.

Starting with 0.9.9-rc1, it is possible to tell `searchd` to periodically write these changes back to disk, to avoid them being lost. The time between those intervals is set with `attr_flush_period`, in seconds.

It defaults to 0, which disables the periodic flushing, but flushing will still occur at normal shut-down.

Example:

```
attr_flush_period = 900 # persist updates to disk every 15 minutes
```

9.4.15. ondisk_dict_default

Instance-wide defaults for ondisk_dict directive. Optional, default it 0 (precache dictionaries in RAM). Introduced in version 0.9.9-rc1.

This directive lets you specify the default value of ondisk_dict for all the indexes served by this copy of `searchd`.

Per-index directive take precedence, and will overwrite this instance-wide default value, allowing for fine-grain control.

Example:

```
ondisk_dict_default = 1 # keep all dictionaries on disk
```

9.4.16. max_packet_size

Maximum allowed network packet size. Limits both query packets from clients, and response packets from remote agents in distributed environment. Only used for internal sanity checks, does not directly affect RAM use or performance. Optional, default is 8M. Introduced in version 0.9.9-rc1.

Example:

```
max_packet_size = 32M
```

9.4.17. mva_updates_pool

Shared pool size for in-memory MVA updates storage. Optional, default size is 1M. Introduced in version 0.9.9-rc1.

This setting controls the size of the shared storage pool for updated MVA values. Specifying 0 for the size disable MVA updates at all. Once the pool size limit is hit, MVA update attempts will result in an error. However, updates on regular (scalar) attributes will still work. Due to internal technical difficulties, currently it is **not** possible to store (flush) **any** updates on indexes where MVA were updated; though this might be implemented in the future. In the meantime, MVA updates are intended to be used as a measure to quickly catchup with latest changes in the database until the next index rebuild; not as a persistent storage mechanism.

Example:

```
mva_updates_pool = 16M
```

9.4.18. crash_log_path

Path (formally prefix) for the crash log files. Optional, default is empty (do not create crash logs). Introduced in version 0.9.9-rc1.

This is a debugging setting, to help catch rare offending queries causing crashes without otherwise affecting production instances. When enabled, `searchd` will intercept crash signals such as SIGSEGV, and dump offending query packets to files named "crash_log_path.PID", where PID is crashed process ID.

Example:

```
crash_log_path = /home/sphinx/log/crashlog
```

9.4.19. max_filters

Maximum allowed per-query filter count. Only used for internal sanity checks, does not directly affect RAM use or performance. Optional, default is 256. Introduced in version 0.9.9-rc1.

Example:

```
max_filters = 1024
```

9.4.20. max_filter_values

Maximum allowed per-filter values count. Only used for internal sanity checks, does not directly affect RAM use or performance. Optional, default is 4096. Introduced in version 0.9.9-rc1.

Example:

```
max_filter_values = 16384
```

9.4.21. listen_backlog

TCP listen backlog. Optional, default is 5.

Windows builds currently (as of 0.9.9) can only process the requests one by one. Concurrent requests will be enqueued by the TCP stack on OS level, and requests that can not be enqueued will immediately fail with "connection refused" message. listen_backlog directive controls the length of the connection queue. Non-Windows builds should work fine with the default value.

Example:

```
listen_backlog = 20
```

9.4.22. read_buffer

Per-keyword read buffer size. Optional, default is 256K.

For every keyword occurrence in every search query, there are two associated read buffers (one for document list and one for hit list). This setting lets you control their sizes, increasing per-query RAM use, but possibly decreasing IO time.

Example:

```
read_buffer = 1M
```

9.4.23. read_unhinted

Unhinted read size. Optional, default is 32K.

When querying, some reads know in advance exactly how much data is there to be read, but some currently do not. Most prominently, hit list size in not currently known in advance. This setting lest you control how much data to read in such cases. It will impact hit list IO time, reducing it for lists larger than unhinted read size, but raising it for smaller lists. It will **not** affect RAM use because read buffer will be already allocated. So it should be not greater than read_buffer.

Example:

```
read_unhinted = 32K
```