# Manticore Search Documentation

***Release 2.6.2***

**The Manticore Search team**

**Nov 07, 2018**

# Manticore Documentation

CHAPTER 1

# Introduction

## 1.1 About

Manticore Search is a full-text search engine, publicly distributed under GPL version 2, forked from 2.3 branch of open-source search engine Sphinx search.

Technically, Manticore is a standalone software package provides fast and relevant full-text search functionality to client applications. It was specially designed to integrate well with SQL databases storing the data, and to be easily accessed by scripting languages. However, Manticore does not depend on nor require any specific database to function.

Applications can access Manticore search daemon (searchd) using any of the following access methods: - Manticore own implementation of MySQL network protocol (using a small SQL subset called SphinxQL, this is recommended way) - native search API (SphinxAPI) - HTTP protocol - via MySQL server with a pluggable storage engine (SphinxSE).

Official native SphinxAPI implementations for PHP, Perl, Python, Ruby and Java are included within the distribution package. API is very lightweight so porting it to a new language is known to take a few hours or days. Third party API ports and plugins exist for Perl, C#, Haskell, Ruby-on-Rails, and possibly other languages and frameworks.

Manticore supports two different indexing backends: “disk” index backend, and “realtime” (RT) index backend. Disk indexes support online full-text index rebuilds, but online updates can only be done on non-text (attribute) data. RT indexes additionally allow for online full-text index updates.

Data can be loaded into disk indexes using a so-called data source. Built-in sources can fetch data directly from MySQL, PostgreSQL, MSSQL, ODBC compliant database (Oracle, etc) or from a pipe in TSV or XML format. Adding new data sources drivers (eg. to natively support other DBMSes) is designed to be as easy as possible. RT indexes can only be populated using SphinxQL.

## 1.2 Manticore features

Key Manticore features are:

- high indexing and searching performance;

- advanced indexing and querying tools (flexible and feature-rich text tokenizer, querying language, several different ranking modes, etc);
- advanced result set post-processing (SELECT with expressions, WHERE, ORDER BY, GROUP BY, HAVING etc over text search results);
- proven scalability up to billions of documents, terabytes of data, and thousands of queries per second;
- easy integration with SQL and XML data sources, and SphinxQL, SphinxAPI, or SphinxSE search interfaces;
- easy scaling with distributed searches.

To expand a bit, Manticore:

- has high indexing speed (upto 10-15 MB/sec per core on an internal benchmark);
- has high search speed (upto 150-250 queries/sec per core against 1,000,000 documents, 1.2 GB of data on an internal benchmark);
- has high scalability (biggest known cluster indexes over 3,000,000,000 documents, and busiest one peaks over 50,000,000 queries/day);
- provides good relevance ranking through combination of phrase proximity ranking and statistical (BM25) ranking;
- provides distributed searching capabilities;
- provides prospective searches (percolate queries)
- provides document excerpts (snippets) generation;
- provides searching from within application with SphinxQL or SphinxAPI interfaces, and from within MySQL with pluggable SphinxSE storage engine;
- supports boolean, phrase, word proximity and other types of 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 morphological word forms dictionaries;
- supports tokenizing exceptions;
- supports UTF-8 encoding;
- supports stemming (stemmers for English, Russian, Czech and Arabic are built-in; and stemmers for French, Spanish, Portuguese, Italian, Romanian, German, Dutch, Swedish, Norwegian, Danish, Finnish, Hungarian, are available by building third party libstemmer library);
- supports MySQL natively (all types of tables, including MyISAM, InnoDB, NDB, Archive, etc are supported);
- supports PostgreSQL natively;
- supports ODBC compliant databases (MS SQL, Oracle, etc) natively;
- . . . has 50+ other features not listed here, refer configuration manual!

## 1.3 Where to get Manticore

Manticore is available through its official Web site at http://manticoresearch.com/.

Currently, Manticore distribution tarball includes the following software:

- `indexer`: an utility which creates 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
- `wordbreaker`: an utility to break down compound words into separate words

## 1.4 License



## 1.5 Credits

Manticore is derived from Sphinx search engine created by Andrew Aksyonoff. More details about people involved in Sphinx development can be found on this page: http://sphinxsearch.com/docs/devel.html#credits.

Manticore is developed and maintained by Manticore Software Ltd. Current team (in alphabetical order):

- Adrian Nuta
- Alexey Vinogradov
- Gloria Vinogradova
- Ilya Kuznetsov
- Mindaugas Zukas
- Sergey Nikolaev
- Stanislav Klinov

CHAPTER 2

# Installation

## 2.1 Installing Manticore packages on Debian and Ubuntu

Supported releases:

- Debian
  - 7.0 (wheezy)
  - 8.0 (jessie)
  - 9.0 (stretch)
- Ubuntu
  - 14.04 LTS (trusty)
  - 16.05 LTS (xenial)

Supported platforms:

- x86
- x86_64

You can install Manticore with command:

```
$ wget https://github.com/manticoresoftware/manticore/releases/download/2.4.1/
↪manticore_2.4.1-171017-3b31a97-release-stemmer.jessie_amd64-bin.deb
$ sudo dpkg -i manticore_2.4.1-171017-3b31a97-release-stemmer.jessie_amd64-bin.deb
```

Manticore requires no extra libraries to be installed on Debian/Ubuntu. However if you plan to use 'indexer' tool to create indexes from different sources, you'll need to install appropriate client libraries. To know what exactly libraries, run *indexer* tool from Manticore and look at the top of it's output:

```
$ indexer
Manticore 2.4.1 4258276@171019 id64-beta
Copyright (c) 2001-2016, Andrew Aksyonoff
```

(continues on next page)

(continued from previous page)

```
Copyright (c) 2008-2016, Sphinx Technologies Inc (http://sphinxsearch.com)
Copyright (c) 2017, Manticore Software LTD (http://manticoresearch.com)

Built by gcc/clang v 6.3.0,

Built on Linux d2a57137d4f5 4.8.0-45-generic #48~16.04.1-Ubuntu SMP Fri Mar 24␣
↪12:46:56 UTC 2017 x86_64 GNU/Linux
Configured by CMake with these definitions: -DCMAKE_BUILD_TYPE=RelWithDebInfo -DDL_
↪UNIXODBC=1 -DUNIXODBC_LIB=libodbc.so.2 -DDL_EXPAT=1 -DEXPAT_LIB=libexpat.so.1 -DDL_
↪MYSQL=1 -DMYSQL_LIB=libmariadbclient.so.18 -DMYSQL_CONFIG_EXECUTABLE=/usr/bin/mysql_
↪config -DDL_PGSQL=1 -DPGSQL_LIB=libpq.so.5 -DSPLIT_SYMBOLS=ON -DUSE_BISON=ON -DUSE_
↪FLEX=ON -DUSE_SYSLOG=1 -DWITH_EXPAT=ON -DWITH_ICONV=ON -DWITH_MYSQL=ON -DWITH_
↪ODBC=ON -DWITH_PGSQL=ON -DWITH_RE2=ON -DWITH_STEMMER=ON -DWITH_ZLIB=ON
```

Here you can see mentions of *libodbc.so.2*, *libexpat.so.1*, *libmariadbclient.so.18*, and *libpq.so.5*.

Below is the reference table with list of all client libraries for different debian/ubuntu distributions:

| Distr | Mysql | PostgresQL | Xmlpipe | Unixodbc |
|---|---|---|---|---|
| trusty | libmysqlclient.so.18 | libpq.so.5 | libexpat.so.1 | libodbc.so.1 |
| xenial | libmysqlclient.so.20 | libpq.so.5 | libexpat.so.1 | libodbc.so.2 |
| wheezy | libmysqlclient.so.18 | libpq.so.5 | libexpat.so.1 | libodbc.so.1 |
| jessie | libmysqlclient.so.18 | libpq.so.5 | libexpat.so.1 | libodbc.so.2 |
| stretch | libmariadbclient.so.18 | libpq.so.5 | libexpat.so.1 | libodbc.so.2 |

To find the packages which provide the libraries you can use, for example `apt-file`:

```
$ apt-file find libmysqlclient.so.20
libmysqlclient20: /usr/lib/x86_64-linux-gnu/libmysqlclient.so.20
libmysqlclient20: /usr/lib/x86_64-linux-gnu/libmysqlclient.so.20.2.0
libmysqlclient20: /usr/lib/x86_64-linux-gnu/libmysqlclient.so.20.3.6
```

Note, that you need only libs for types of sources you're going to use. So if you plan to make indexes only from mysql source, then install only lib for mysql client (in case above - `libmysqlclient20`).

Finally install necessary packages:

```
$ sudo apt-get install libmysqlclient20 libodbc1 libpq5 libexpat1
```

If you aren't going to use `indexer` tool at all, you don't need find and install any libraries.

After preparing configuration file (see *Quick tour*), you can start searchd daemon:

```
$ systemctl manticore start
```

## 2.2 Installing Manticore packages on RedHat and CentOS

Supported releases:

- CentOS 6 and RHEL 6
- CentOS 7 and RHEL 7

Supported platforms:

- x86

- x86_64

Manticore requires no extra libraries to be installed on RedHat/CentOS. However if you plan to use 'indexer' tool to create indexes from different sources, you'll need to install appropriate client libraries. Use yum to download and install these dependencies:

```
$ yum install mysql-libs postgresql-libs expat unixODBC
```

Note, that you need only libs for types of sources you're going to use. So if you plan to make indexes only from mysql source, then installing 'mysql-libs' will be enough. If you don't going to use 'indexer' tool at all, you don't need to install these packages. Download RedHat RPM from Manticore website and install it:

```
$ wget https://github.com/manticoresoftware/manticore/releases/download/2.4.1/
↪manticore-2.4.1-171017-3b31a97-release-stemmer-rhel7-bin.rpm
$ rpm -Uhv manticore-2.4.1-171017-3b31a97-release-stemmer-rhel7-bin.rpm
```

After preparing configuration file (see *Quick tour*), you can start searchd daemon:

```
$ systemctl searchd start
```

## 2.3 Installing Manticore on Windows

To install on Windows, you need to download the zip package and unpack it first.

```
cd C:\Manticore
unzip manticore-2.4.1-171017-3b31a97-release-pgsql-stemmer-x64-bin.zip
```

Edit the contents of sphinx.conf.in - specifically entries relating to @CONFDIR@ - to paths suitable for your system.

Install the `searchd` system as a Windows service:

```
C:\Manticore\bin> C:\Manticore\bin\searchd --install --config C:\Manticore\sphinx.
↪conf.in --servicename Manticore
```

**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*.

## 2.4 Running Manticore Search in a Docker Container

Docker images of Manticore Search are hosted publicly on Docker Hub at https://hub.docker.com/r/manticoresearch/manticore/.

For more information about using Docker, see the Docker Docs.

The searchd daemon runs in nodetach mode inside the container. Default configuration includes a simple Real-Time index and listen on the default ports ( 9306 for SphinxQL and 9312 for SphinxAPI).

The image comes with MySQL and PostgreSQL client libraries for indexing data from these databases.

### 2.4.1 Starting a Manticore Search instance in a container

To start a container running the latest release of Manticore Search run:

```
docker run --name manticore -p 9306:9306 -d manticoresearch/manticore
```

Operations with utility tools over running daemon can be made with *docker exec* command:

```
docker exec -it manticore indexer --all --rotate
```

To stop the Manticore Search container you can simply do:

```
docker stop manticore
```

or (managed stop with no hard-killing):

```
docker exec -it manticore searchd --stopwait
```

Please note that any indexed data or configuration change made is lost if the container is stopped. For persistence, you need to mount the configuration and data folders.

### 2.4.2 Mounting points

The configuration folder inside the image is the usual */etc/sphinxseach*. Index files are located at */var/lib/manticore/data* and logs at */var/lib/manticore/log*. For persistence, mount these points to your local folders.

```
docker run --name manticore -v /path/to/config/:/etc/sphinxsearch/ -v /path/to/data/:/
↪var/lib/manticore/data -v /path/to/logs/:/var/lib/manticore/log -p 9306:9306 -d␣
↪manticoresearch/manticore
```

## 2.5 Compiling Manticore from source

### 2.5.1 Required tools

- a working compiler
  - on Linux - GNU gcc (4.7.2 and above) or clang can be used
  - on Windows - Microsoft Visual Studio 2015 and above (community edition is enough)
  - on Mac OS - XCode
- cmake - used on all plaftorms (version 2.8 or above)

### 2.5.2 Optional dependencies

- git, flex, bison - needed if the sources are from cloned repository and not the source tarball
- development version of MySQL client for MySQL source driver
- development version of unixODBC for the unixODBC source driver
- development version of libPQ for the PostgreSQL source driver
- development version of libexpat for the XMLpipe source driver
- RE2 (bundled in the source tarball) for *regexp_filter* feature
- lib stemmer (bundled in the source tarball ) for additional language stemmers

### 2.5.3 General building options

For compiling latest version of Manticore, recommended is checkout the latest code from the github repositiory. Alternative, for compiling a certain version, you can either checked that version from github or use it's respective source tarball. In last case avoid to use automatic tarballs from github (named there as 'Source code'), but use provided files as *manticore-2.4.1-171017-3b31a97-release.tar.gz*. When building from clone you need packages *git*, *flex*, *bison*. When building from tarball they are not necessary. This requirement may be essential to build on Windows.

```
$ git clone https://github.com/manticoresoftware/manticore.git
```

```
$ wget https://github.com/manticoresoftware/manticore/releases/download/2.4.1/
↪manticore-2.4.1-171017-3b31a97-release.tar.gz
$ tar zcvf manticore-2.4.1-171017-3b31a97-release.tar.gz
```

Next step is to configure the building with cmake. Available list of configuration options:

- `CMAKE_BUILD_TYPE` - can be Debug , Release , MinSizeRel and RelWithDebInfo (default).
- `SPLIT_SYMBOLS` (bool) - specify whenever to create separate files with debugging symbols. In the default build type,RelWithDebInfo, the binaries include the debug symbols. With this option specified, the binaries will be stripped of the debug symbols , which will be put in separate files
- `USE_BISON`, `USE_FLEX` (bool) - enabled by default, specifies whenever to enable bison and flex tools
- `LIBS_BUNDLE` - filepath to a folder with different libraries. This is mostly relevant for Windows building
- `WITH_STEMMER` (bool) - specifies if the build should include the libstemmer library. The library is searched in several places, starting with
  - libstemmer_c folder in the source directory
  - common system path. Please note that in this case, the linking is dynamic and libstemmer should be available system-wide on the installed systems
  - libstemmer_c.tgz in `LIBS_BUNDLE` folder.
  - download from snowball project website. This is done by cmake and no additional tool is required
  - NOTE: if you have libstemmer in the system, but still want to use static version, say, to build a binary for a system without such lib, provide `WITH_STEMMER_FORCE_STATIC=1` in advance.
- `WITH_RE2` (bool) - specifies if the build should include the RE2 library. The library can be taken from the following locations:
  - in the folder specified by `WITH_RE2_ROOT` parameters
  - in libre2 folder of the Manticore sources
  - system wide search, while first looking for headers specified by `WITH_RE2_INCLUDES` folder and the lib files in `WITH_RE2_LIBS` folder
  - check presence of master.zip in the `LIBS_BUNDLE` folder
  - Download from https://github.com/manticoresoftware/re2/archive/master.zip
  - NOTE: if you have RE2 in the system, but still want to use static version, say, to build a binary for a system without such lib, provide `WITH_RE2_FORCE_STATIC=1` in advance.
- `WITH_EXPAT` (bool) enabled compiling with libexpat, used XMLpipe source driver
- `WITH_MYSQL` (bool) enabled compiling with MySQL client library, used by MySQL source driver. Additional parameters `WITH_MYSQL_ROOT`, `WITH_MYSQL_LIBS` and `WITH_MYSQL_INCLUDES` can be used for custom MySQL files

- `WITH_ODBC` (bool) enabled compiling with ODBC client library, used by ODBC source driver
- `WITH_PGSQL` (bool) enabled compiling with PostgreSQL client library, used by PostgreSQL source driver
- `DISTR_BUILD` - in case the target is packaging, it specifies the target operating system. Supported values are: *centos6*, *centos7*, *wheezy*, *jessie*, *stretch*, *trusty*, *xenial*, *macos*, *default*.

### 2.5.4 Compiling on UNIX systems

To install all dependencies on Debian/Ubuntu:

```
$ apt-get install build-essential cmake unixodbc-dev libpq-dev libexpat-dev␣
↪libmysqlclient-dev git flex bison
```

Note: on Debian 9 (stretch) package `libmysqlclient-dev` is absent. Use `default-libmysqlclient-dev` there instead.

To install all dependencies on CentOS/RHEL:

```
$ yum install gcc gcc-c++ make cmake mysql-devel expat-devel postgresql-devel␣
↪unixODBC-devel rpm-build systemd-units git flex bison
```

(git, flex, bison doesn't necessary if you build from tarball)

RHEL/CentOS 6 ship with a old version of the gcc compiler, which doesn't support *-std=c++11* flag, for compiling use *devtools* repository:

```
$ wget http://people.centos.org/tru/devtools-2/devtools-2.repo -O /etc/yum.repos.d/
↪devtools-2.repo
$ yum upgrade -y
$ yum install -y devtoolset-2-gcc devtoolset-2-binutils devtoolset-2-gcc-c++
$ export PATH=/opt/rh/devtoolset-2/root/usr/bin:$PATH
```

Manticore uses *cmake* for building. We recommend to use a folder outside the sources to keep them clean.

```
$ mkdir build
$ cd build
$ cmake -D WITH_MYSQL=1 -DWITH_RE2=1 ../manticore
```

or if we use sources from tarball:

```
$ cmake -D WITH_MYSQL=1 -DWITH_RE2=1 ../manticore-2.4.1-171017-3b31a97-release
```

To simply compile:

```
$ make -j4
```

This will create the binary files, however we want to either install Manticore or more convenient to create a package. To install just do

```
$ make -j4 install
```

For packaging use `package`

```
$ make -j4 package
```

By default, if no operating system was targeted, `package` will create only a zip with the binaries. If, for example, we want to create a deb package for Debian Jessie, we need to specify to cmake the `DISTR_BUILD` parameter:

```
$ cmake -DDISTR_BUILD=jessie ../manticore
$ make -j4 package
```

This will create 2 deb packages, a manticore-x.x.x-bin.deb and a manticore-x.x.x-dbg.deb which contains the version with debug symbols. Another possible target is `tarball` , which create a tar.gz file from the sources.

### 2.5.5 Compiling on Windows

For building on Windows you need:

- Visual Studio
- Cmake for Windows
- Expat, MySQL and PostgreSQL in bundle directory.

If you build from git clone, you also need to provide *git*, *flex*, *bison* tools. They may be fond in *cygwin* framework. When building from tarball these tools are not necessary.

For a simple building on x64:

```
C:\build>"%PROGRAMW6432%\CMake\bin\cmake.exe" -G "Visual Studio 14 Win64" -DLIBS_
↪BUNDLE="C:\bundle" "C:\manticore"
C:\build>"%PROGRAMW6432%\CMake\bin\cmake.exe" -DWITH_PGSQL=1 -DWITH_RE2=1 -DWITH_
↪STEMMER=1 .
C:\build>"%PROGRAMW6432%\CMake\bin\cmake.exe" --build . --target package --config␣
↪RelWithDebInfo
```

### 2.5.6 Recompilation (update)

If you didn't change path for sources and build, just move to you build folder and run:

```
cmake .
make clean
make
```

If by any reason it doesn't work, you can delete file `CMakeCache.txt` located in build folder. After this step you have to run cmake again, pointing to source folder and configuring the options.

If it also doesn't help, just wipe out your build folder and begin clean *compiling from sources*

## 2.6 Quick Manticore usage tour

We are going to use SphinxQL protocol as it's the current recommended way and it's also easy to play with. First we connect to Manticore with the normal MySQL client:

```
$ mysql -h0 -P9306
```

The default configuration comes with a sample Real-Time. A first step to see it in action is to add several documents to it, then you can start perform searches:

```
mysql> INSERT INTO rt VALUES (1, 'this is', 'a sample text', 11);
    Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO rt VALUES (2, 'some more', 'text here', 22);
    Query OK, 1 row affected (0.00 sec)

    mysql> INSERT INTO rt VALUES (3, 'more about this text', 'can be found in this␣
↪text', 22);
    Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT *,weight() FROM rt  WHERE MATCH('text') ORDER BY WEIGHT() DESC;
    +------+------+----------+
    | id   | gid  | weight() |
    +------+------+----------+
    |    3 |   22 |     2252 |
    |    1 |   11 |     1319 |
    |    2 |   22 |     1319 |
    +------+------+----------+
    3 rows in set (0.00 sec)
```

In the sample configuration there is also a plain index with MySQL source, which needs to be indexed first in order to start using it. First, we populate the sample table in MySQL:

```
mysql> create database test;
$ mysql -u test <  /usr/share/doc/manticore/example-conf/example.sql
```

The sample config uses a `test` with no password for connecting to MySQL. Adjust the credentials, then index:

```
$ sudo -u manticore indexer -c /etc/sphinxsearch/sphinx.conf test1 --rotate
Manticore 2.3.3 9b7033e@170806 master...origin/master-id64-dev
Copyright (c) 2001-2016, Andrew Aksyonoff
Copyright (c) 2008-2016, Sphinx Technologies Inc (http://sphinxsearch.com)
Copyright (c) 2017, Manticore Software LTD (http://manticoresearch.com)

using config file '/etc/sphinxsearch/sphinx.conf'...
indexing index 'test1'...
collected 4 docs, 0.0 MB
sorted 0.0 Mhits, 100.0% done
total 4 docs, 193 bytes
total 0.002 sec, 81503 bytes/sec, 1689.18 docs/sec
total 4 reads, 0.000 sec, 8.1 kb/call avg, 0.0 msec/call avg
total 12 writes, 0.000 sec, 0.1 kb/call avg, 0.0 msec/call avg
rotating indices: successfully sent SIGHUP to searchd (pid=2947).
```

Now let's run several queries:

```
mysql> SELECT *, WEIGHT() FROM test1 WHERE MATCH('"document one"/1');SHOW META;
+------+----------+------------+----------+
| id   | group_id | date_added | weight() |
+------+----------+------------+----------+
|    1 |        1 | 1502280778 |     2663 |
|    2 |        1 | 1502280778 |     1528 |
+------+----------+------------+----------+
2 rows in set (0.00 sec)

+---------------+----------+
| Variable_name | Value    |
```

(continues on next page)

(continued from previous page)

```
+---------------+----------+
| total         | 2        |
| total_found   | 2        |
| time          | 0.000    |
| keyword[0]    | document |
| docs[0]       | 2        |
| hits[0]       | 2        |
| keyword[1]    | one      |
| docs[1]       | 1        |
| hits[1]       | 2        |
+---------------+----------+
9 rows in set (0.00 sec)
```

```
mysql>  SET profiling=1;SELECT * FROM test1 WHERE id IN (1,2,4);SHOW PROFILE;
Query OK, 0 rows affected (0.00 sec)

+------+----------+------------+
| id   | group_id | date_added |
+------+----------+------------+
|    1 |        1 | 1502280778 |
|    2 |        1 | 1502280778 |
|    4 |        2 | 1502280778 |
+------+----------+------------+
3 rows in set (0.00 sec)

+--------------+----------+----------+---------+
| Status       | Duration | Switches | Percent |
+--------------+----------+----------+---------+
| unknown      | 0.000059 | 4        | 44.70   |
| net_read     | 0.000001 | 1        | 0.76    |
| local_search | 0.000042 | 1        | 31.82   |
| sql_parse    | 0.000012 | 1        | 9.09    |
| fullscan     | 0.000001 | 1        | 0.76    |
| finalize     | 0.000007 | 1        | 5.30    |
| aggregate    | 0.000006 | 2        | 4.55    |
| net_write    | 0.000004 | 1        | 3.03    |
| eval_post    | 0.000000 | 1        | 0.00    |
| total        | 0.000132 | 13       | 0       |
+--------------+----------+----------+---------+
10 rows in set (0.00 sec)
```

```
mysql> SELECT id, id%3 idd FROM test1 WHERE MATCH('this is | nothing') GROUP BY idd;
↪SHOW PROFILE;
+------+------+
| id   | idd  |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    3 |    0 |
+------+------+
3 rows in set (0.00 sec)

+--------+----------+----------+---------+
| Status | Duration | Switches | Percent |
+--------+----------+----------+---------+
| total  | 0.000000 | 0        | 0       |
```

(continues on next page)

(continued from previous page)

```
+--------+----------+----------+---------+
1 row in set (0.00 sec)
```

```
mysql> SELECT id FROM test1 WHERE MATCH('is this a good plan?');SHOW PLAN\G
Empty set (0.00 sec)

*************************** 1. row ***************************
Variable: transformed_tree
        Value: AND(
        AND(KEYWORD(is, querypos=1)),
        AND(KEYWORD(this, querypos=2)),
        AND(KEYWORD(a, querypos=3)),
        AND(KEYWORD(good, querypos=4)),
        AND(KEYWORD(plan, querypos=5)))
1 row in set (0.00 sec)
```

```
mysql>  SELECT COUNT(*) c, id%3 idd FROM test1 GROUP BY idd HAVING COUNT(*)>1;
    +------+------+
    | c    | idd  |
    +------+------+
    |    2 |    1 |
    +------+------+
    1 row in set (0.00 sec)
```

```
mysql>  SELECT COUNT(*) FROM test1;
    +----------+
    | count(*) |
    +----------+
    |        4 |
    +----------+
    1 row in set (0.00 sec)
```

```
mysql>   CALL KEYWORDS ('one two three', 'test1', 1);
+------+-----------+------------+------+------+
| qpos | tokenized | normalized | docs | hits |
+------+-----------+------------+------+------+
| 1    | one       | one        | 1    | 2    |
| 2    | two       | two        | 1    | 2    |
| 3    | three     | three      | 0    | 0    |
+------+-----------+------------+------+------+
3 rows in set (0.00 sec)
```

CHAPTER 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 Manticore point of view, the data it indexes is a set of structured *documents*, each of which has the same set of *fields* and *attributes*. This is similar to SQL, where each row would correspond to a document, and each column to either a field or an attribute.

Depending on what source Manticore 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 built-in drivers for MySQL, PostgreSQL, MS SQL (on Windows), and ODBC. There is also a generic driver called xmlpipe2, which runs a specified command and reads the data from its `stdout`. See *xmlpipe2 data source* section for the format description. tsvpipe (Tab Separated Values) and csvpipe (Comma Separated Values) data source also available and described in *TSV/CSV data source*.

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

## 3.2 Full-text fields

Full-text fields (or just *fields* for brevity) are the textual document contents that get indexed by Manticore, and can be (quickly) searched for keywords.

Fields are named, and you can limit your searches to a single field (eg. search through "title" only) or a subset of fields (eg. to "title" and "abstract" only). Manticore index format generally supports up to 256 fields.

Note that the original contents of the fields are **not** stored in the Manticore index. The text that you send to Manticore gets processed, and a full-text index (a special data structure that enables quick searches for a keyword) gets built from that text. But the original text contents are then simply discarded. Manticore assumes that you store those contents elsewhere anyway.

Moreover, it is impossible to *fully* reconstruct the original text, because the specific whitespace, capitalization, punctuation, etc will all be lost during indexing. It is theoretically possible to partially reconstruct a given document from the Manticore full-text index, but that would be a slow process (especially if the *CRC dictionary* is used, which does not even store the original keywords and works with their hashes instead).

## 3.3 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, Manticore 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.

### 3.3.1 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
...
```

### 3.3.2 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 ) );
```

(continues on next page)

(continued from previous page)

```
// 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);
- signed big integers (64-bit wide);
- UNIX timestamps;
- floating point values (32-bit, IEEE 754 single precision);
- *strings*;
- *JSON*;
- *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 occurrence 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.

## 3.4 MVA (multi-valued attributes)

MVAs, or multi-valued attributes, are an important special type of per-document attributes in Manticore. MVAs let you attach sets of numeric values to every document. That is useful to implement article tags, product categories, etc. Filtering and group-by (but not sorting) on MVA attributes is supported.

MVA values can either be unsigned 32-bit integers (UNSIGNED INTEGER) or signed 64-bit integers (BIGINT).

The set size 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.

## 3.5 Indexes

To be able to answer full-text search queries fast, Manticore 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*.

An index identifier must be a single word, that can contain letters, numbers and underscores. It must start with a letter.

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. Manticore architecture allows internally for different *index types*, or *backends*, to be implemented comparatively easily.

Manticore provides 2 different backends: a **disk index** backend, and a **RT (realtime) index** backend.

### 3.5.1 Offline/plain indexes

**Disk indexes** are designed to provide maximum indexing and searching speed, while keeping the RAM footprint as low as possible. That comes at a cost of text index updates. You can not update an existing document or incrementally add a new document to a disk index. You only can batch rebuild the entire disk index from scratch. (Note that you still can update document's **attributes** on the fly, even with the disk indexes.)

This "rebuild only" limitation might look as a big constraint at a first glance. But in reality, it can very frequently be worked around rather easily by setting up multiple disk indexes, searching through them all, and only rebuilding the one with a fraction of the most recently changed data. See *Live index updates* for details.

### 3.5.2 Real-Time indexes

**RT indexes** enable you to implement dynamic updates and incremental additions to the full text index. RT stands for Real Time and they are indeed "soft realtime" in terms of writes, meaning that most index changes become available for searching as quick as 1 millisecond or less, but could occasionally stall for seconds. (Searches will still work even during that occasional writing stall.) Refer to *Real-time indexes* for details.

### 3.5.3 Distributed indexes

Manticore supports so-called **distributed indexes**. Compared to disk and RT indexes, those are not a real physical backend, but rather just lists of either local or remote indexes that can be searched transparently to the application, with Manticore doing all the chores of sending search requests to remote machines in the cluster, aggregating the result sets, retrying the failed requests, and even doing some load balancing. See *Distributed searching* for a discussion of distributed indexes.

### 3.5.4 Templates indexes

Template indexes are indexes with no storage backend. They can be used operations that involve only data from input, like keywords and snippets generation.

### 3.5.5 Percolate indexes

Percolate indexes are special Real-Time indexes that store queries instead of documents. They are used for prospective searches ( or "search in reverse"). Refer to *Percolate query* for more details.

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.

## 3.6 Restrictions on the source data

There are a few different restrictions imposed on the source data which is going to be indexed by Manticore, 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, Manticore 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.

## 3.7 Charsets, case folding, translation tables, and replacement rules

When indexing some index, Manticore 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, Manticore needs to know

- what encoding is the source text in (and this encoding should always be UTF-8);
- 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_table*. option. *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.

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

You can also specify text pattern replacement rules. For example, given the rules

```
regexp_filter = \**(\d+)\" => \1 inch
regexp_filter = (BLUE|RED) => COLOR
```

the text 'RED TUBE 5" LONG' would be indexed as 'COLOR TUBE 5 INCH LONG', and 'PLANK 2" x 4"'as 'PLANK 2 INCH x 4 INCH'. Rules are applied in the given order. Text in queries is also replaced; a search for"BLUE TUBE" would actually become a search for "COLOR TUBE". Note that Manticore must be built with the –with-re2 option to use this feature.

## 3.8 SQL data sources (MySQL, PostgreSQL)

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

- connection to the database is established;

- pre-query (see *sql_query_pre*) is executed to perform any necessary initial setup, such as setting per-connection encoding with MySQL;
- main query (see *sql_query*) is executed and the rows it returns are indexed;
- post-query (see *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 *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.

### 3.8.1 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, Manticore supports so-called *ranged queries*. With ranged queries, Manticore 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 3.1. 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 `$start` replaced with 2001 and `$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.

### 3.8.2 `sql_query_post` vs. `sql_query_post_index`

The difference between post-query and post-index query is in that post-query is run immediately when Manticore 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 successful. Database connection is dropped and re-established because sorting phase can be very lengthy and would just timeout otherwise.

## 3.9 xmlpipe2 data source

xmlpipe2 lets you pass arbitrary full-text and attribute data to Manticore 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 3.2. 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 <** class="red">randomized** order</subject>
<misc>some undeclared element</misc>
</sphinx:document>

<sphinx:document id="1235">
<subject>another subject</subject>
<content>here comes another document, and i am given to understand,
that in-document field order must not matter, sir</content>
<published>1012325467</published>
</sphinx:document>

<!-- ... even more sphinx:document entries here ... -->

<sphinx:killlist>
<id>1234</id>
<id>4567</id>
</sphinx:killlist>

</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 ** 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. Manticore `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 Manticore 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. Known attributes are:
  - "name", specifies the XML element name that will be treated as a full-text field in the subsequent documents.
  - "attr", specifies whether to also index this field as a string. Possible value is "string".
- 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", "bigint", "timestamp", "bool", "float", "multi" and "json".
  - "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.
- sphinx:killlist
- Optional element, child of sphinx:docset. Contains a number of "id" elements whose contents are document IDs to be put into a *kill-list* for this index.

## 3.10 TSV/CSV data source

This is the simplest way to pass data to the indexer. It was created due to xmlpipe2 limitations. Namely, indexer must map each attribute and field tag in XML file to corresponding schema element. This mapping requires some time. And time increases with increasing the number of fields and attributes in schema. There is no such issue in tsvpipe because each field and attribute is a particular column in TSV file. So, in some cases tsvpipe could work slightly faster than xmlpipe2.

The first column in TSVCSV file must be a document ID. The rest ones must mirror the declaration of fields and attributes in schema definition.

The difference between tsvpipe and csvpipe is delimiter and quoting rules. tsvpipe has tab character as hardcoded delimiter and has no quoting rules. csvpipe has option *csvpipe_delimiter* for delimiter with default value ',' and also has quoting rules, such as:

- any field may be quoted
- fields containing a line-break, double-quote or commas should be quoted
- a double quote character in a field must be represented by two double quote characters

tsvpipe and csvpipe have same field and attrribute declaration derectives as xmlpipe.

tsvpipe declarations:

*tsvpipe_command*, *tsvpipe_field*, *tsvpipe_field_string*, *tsvpipe_attr_uint*, *tsvpipe_attr_timestamp*, *tsvpipe_attr_bool*, *tsvpipe_attr_float*, *tsvpipe_attr_bigint*, *tsvpipe_attr_multi*, *tsvpipe_attr_multi_64*, *tsvpipe_attr_string*, *tsvpipe_attr_json*

csvpipe declarations:

*csvpipe_command*, *csvpipe_field*, *csvpipe_field_string*, *csvpipe_attr_uint*, *csvpipe_attr_timestamp*, *csvpipe_attr_bool*, *csvpipe_attr_float*, *csvpipe_attr_bigint*, *csvpipe_attr_multi*, *csvpipe_attr_multi_64*, *csvpipe_attr_string*, *csvpipe_attr_json*

```
source tsv_test
{
    type = tsvpipe
    tsvpipe_command = cat /tmp/rock_bands.tsv
    tsvpipe_field = name
    tsvpipe_attr_multi = genre_tags
}
```

```
1   Led Zeppelin    35,23,16
2   Deep Purple 35,92
3   Frank Zappa 35,23,16,92,33,24
```

## 3.11 Live index updates

There are two major approaches to maintaining the full-text index contents up to date. Note, however, that both these approaches deal with the task of *full-text data updates*, and not attribute updates (which are already supported, refer to *UpdateAttributes* API call description for details.)

First, you can use disk-based indexes, partition them manually, and only rebuild the smaller partitions (so-called "deltas") frequently. By minimizing the rebuild size, you can reduce the average indexing lag to something as low as 30-60 seconds. On huge collections it actually might be the most efficient one. Refer to *Delta index updates* for details.

Second, using real-time indexes (RT indexes for short) that on-the-fly updates of the full-text data. Updates on a RT index can appear in the search results in 1-2 milliseconds, ie. 0.001-0.002 seconds. However, RT index are less efficient for bulk indexing huge amounts of data. Refer to *Real-time indexes* for details.

## 3.12 Delta 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 automatic. 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 3.3. 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.

## 3.13 Index merging

Merging two existing indexes can be more efficient than 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 merged 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).

CHAPTER 4

# Real-time indexes

Real-time indexes (or RT indexes for brevity) are a backend that lets you insert, update, or delete documents (rows) on the fly. While querying of RT indexes is possible using any of the SphinxAPI, SphinxQL, or SphinxSE, updating them is only possible via SphinxQL at the moment. Full SphinxQL reference is available in *SphinxQL reference*.

## 4.1 RT indexes overview

RT indexes should be declared in `sphinx.conf`, just as every other index type. Notable differences from the regular, disk-based indexes are that a) data sources are not required and ignored, and b) you should explicitly enumerate all the text fields, not just attributes. Here's an example:

Example 4.1. RT index declaration

```
index rt
{
    type = rt
    path = /usr/local/sphinx/data/rt
    rt_field = title
    rt_field = content
    rt_attr_uint = gid
}
```

RT index can be accessed using MySQL protocol. INSERT, REPLACE, DELETE, and SELECT statements against RT index are supported. For instance, this is an example session with the sample index above:

```
$ mysql -h 127.0.0.1 -P 9306
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 1.10-dev (r2153)

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

mysql> INSERT INTO rt VALUES ( 1, 'first record', 'test one', 123 );
```

(continues on next page)

(continued from previous page)

```
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO rt VALUES ( 2, 'second record', 'test two', 234 );
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM rt;
+------+--------+------+
| id   | weight | gid  |
+------+--------+------+
|    1 |      1 |  123 |
|    2 |      1 |  234 |
+------+--------+------+
2 rows in set (0.02 sec)

mysql> SELECT * FROM rt WHERE MATCH('test');
+------+--------+------+
| id   | weight | gid  |
+------+--------+------+
|    1 |   1643 |  123 |
|    2 |   1643 |  234 |
+------+--------+------+
2 rows in set (0.01 sec)

mysql> SELECT * FROM rt WHERE MATCH('@title test');
Empty set (0.00 sec)
```

Both partial and batch INSERT syntaxes are supported, ie. you can specify a subset of columns, and insert several rows at a time. Deletions are also possible using DELETE statement; the only currently supported syntax is DELETE FROM <index> WHERE id=<id>. REPLACE is also supported, enabling you to implement updates.

```
mysql> INSERT INTO rt ( id, title ) VALUES ( 3, 'third row' ), ( 4, 'fourth entry' );
Query OK, 2 rows affected (0.01 sec)

mysql> SELECT * FROM rt;
+------+--------+------+
| id   | weight | gid  |
+------+--------+------+
|    1 |      1 |  123 |
|    2 |      1 |  234 |
|    3 |      1 |    0 |
|    4 |      1 |    0 |
+------+--------+------+
4 rows in set (0.00 sec)

mysql> DELETE FROM rt WHERE id=2;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM rt WHERE MATCH('test');
+------+--------+------+
| id   | weight | gid  |
+------+--------+------+
|    1 |   1500 |  123 |
+------+--------+------+
1 row in set (0.00 sec)

mysql> INSERT INTO rt VALUES ( 1, 'first record on steroids', 'test one', 123 );
```

(continues on next page)

(continued from previous page)

```
ERROR 1064 (42000): duplicate id '1'

mysql> REPLACE INTO rt VALUES ( 1, 'first record on steroids', 'test one', 123 );
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM rt WHERE MATCH('steroids');
+------+--------+------+
| id   | weight | gid  |
+------+--------+------+
|    1 |   1500 |  123 |
+------+--------+------+
1 row in set (0.01 sec)
```

Data stored in RT index should survive clean shutdown. When binary logging is enabled, it should also survive crash and/or dirty shutdown, and recover on subsequent startup.

## 4.2 Known caveats with RT indexes

RT indexes are currently quality feature, but there are still a few known usage quirks. Those quirks are listed in this section.

- Default conservative RAM chunk limit (`rt_mem_limit`) of 32M can lead to poor performance on bigger indexes, you should raise it to 256..1024M if you're planning to index gigabytes.
- The only attribute storage mode is 'extern' which requires at least one attribute to be present.
- High DELETE/REPLACE rate can lead to kill-list fragmentation and impact searching performance.
- No transaction size limits are currently imposed; too many concurrent INSERT/REPLACE transactions might therefore consume a lot of RAM.
- In case of a damaged binlog, recovery will stop on the first damaged transaction, even though it's technically possible to keep looking further for subsequent undamaged transactions, and recover those. This mid-file damage case (due to flaky HDD/CDD/tape?) is supposed to be extremely rare, though.
- Multiple INSERTs grouped in a single transaction perform better than equivalent single-row transactions and are recommended for batch loading of data.

## 4.3 RT index internals

RT index is internally chunked. It keeps a so-called RAM chunk that stores all the most recent changes. RAM chunk memory usage is rather strictly limited with per-index *rt_mem_limit* directive. Once RAM chunk grows over this limit, a new disk chunk is created from its data, and RAM chunk is reset. Thus, while most changes on the RT index will be performed in RAM only and complete instantly (in milliseconds), those changes that overflow the RAM chunk will stall for the duration of disk chunk creation (a few seconds).

Manticore uses double-buffering to avoid INSERT stalls. When data is being dumped to disk, the second buffer is used, so further INSERTs won't be delayed. The second buffer is defined to be 10% the size of the standard buffer, *rt_mem_limit*, but future versions of Manticore may allow configuring this further.

Disk chunks are, in fact, just regular disk-based indexes. But they're a part of an RT index and automatically managed by it, so you need not configure nor manage them manually. Because a new disk chunk is created every time RT chunk overflows the limit, and because in-memory chunk format is close to on-disk format, the disk chunks will be approximately `rt_mem_limit` bytes in size each.

Generally, it is better to set the limit bigger, to minimize both the frequency of flushes, and the index fragmentation (number of disk chunks). For instance, on a dedicated search server that handles a big RT index, it can be advised to set `rt_mem_limit` to 1-2 GB. A global limit on all indexes is also planned, but not yet implemented.

Disk chunk full-text index data can not be actually modified, so the full-text field changes (ie. row deletions and updates) suppress a previous row version from a disk chunk using a kill-list, but do not actually physically purge the data. Therefore, on workloads with high full-text updates ratio index might eventually get polluted by these previous row versions, and searching performance would degrade. Physical index purging that would improve the performance may be performed with *OPTIMIZE* command.

Data in RAM chunk gets saved to disk on clean daemon shutdown, and then loaded back on startup. However, on daemon or server crash, updates from RAM chunk might be lost. To prevent that, binary logging of transactions can be used; see *the section called :ref:'binary_logging* for details.

Full-text changes in RT index are transactional. They are stored in a per-thread accumulator until COMMIT, then applied at once. Bigger batches per single COMMIT should result in faster indexing.

## 4.4 Binary logging

Binary logs are essentially a recovery mechanism. With binary logs enabled, `searchd` writes every given transaction to the binlog file, and uses that for recovery after an unclean shutdown. On clean shutdown, RAM chunks are saved to disk, and then all the binlog files are unlinked.

During normal operation, a new binlog file will be opened every time when `binlog_max_log_size` limit is reached. Older, already closed binlog files are kept until all of the transactions stored in them (from all indexes) are flushed as a disk chunk. Setting the limit to 0 pretty much prevents binlog from being unlinked at all while `searchd` is running; however, it will still be unlinked on clean shutdown. (`binlog_max_log_size` defaults to 0.)

There are 3 different binlog flushing strategies, controlled by *binlog_flush* directive which takes the values of 0, 1, or 2. 0 means to flush the log to OS and sync it to disk every second; 1 means flush and sync every transaction; and 2 (the default mode) means flush every transaction but sync every second. Sync is relatively slow because it has to perform physical disk writes, so mode 1 is the safest (every committed transaction is guaranteed to be written on disk) but the slowest. Flushing log to OS prevents from data loss on `searchd` crashes but not system crashes. Mode 2 is the default.

On recovery after an unclean shutdown, binlogs are replayed and all logged transactions since the last good on-disk state are restored. Transactions are checksummed so in case of binlog file corruption garbage data will **not** be replayed; such a broken transaction will be detected and, currently, will stop replay. Transactions also start with a magic marker and timestamped, so in case of binlog damage in the middle of the file, it's technically possible to skip broken transactions and keep replaying from the next good one, and/or it's possible to replay transactions until a given timestamp (point-in-time recovery), but none of that is implemented yet.

One unwanted side effect of binlogs is that actively updating a small RT index that fully fits into a RAM chunk part will lead to an ever-growing binlog that can never be unlinked until clean shutdown. Binlogs are essentially append-only deltas against the last known good saved state on disk, and unless RAM chunk gets saved, they can not be unlinked. An ever-growing binlog is not very good for disk use and crash recovery time. To avoid this, you can configure `searchd` to perform a periodic RAM chunk flush to fix that problem using a *rt_flush_period* directive. With periodic flushes enabled, `searchd` will keep a separate thread, checking whether RT indexes RAM chunks need to be written back to disk. Once that happens, the respective binlogs can be (and are) safely unlinked.

Note that `rt_flush_period` only controls the frequency at which the *checks* happen. There are no *guarantees* that the particular RAM chunk will get saved. For instance, it does not make sense to regularly re-save a huge RAM chunk that only gets a few rows worth of updates. The search daemon determine whether to actually perform the flush with a few heuristics.

CHAPTER 5

# Searching

## 5.1 Matching modes

So-called matching modes are a legacy feature that used to provide (very) limited query syntax and ranking support. Currently, they are deprecated in favor of *full-text query language* and so-called *Available built-in rankers*. It is thus strongly recommended to use SPH_MATCH_EXTENDED and proper query syntax rather than any other legacy mode. All those other modes are actually internally converted to extended syntax anyway. SphinxAPI still defaults to SPH_MATCH_ALL but that is for compatibility reasons only.

There are the following matching modes available:

- SPH_MATCH_ALL, matches all query words;
- 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 *Boolean query syntax*);
- SPH_MATCH_EXTENDED, matches query as an expression in Manticore internal query language (see *Extended query syntax*);
- SPH_MATCH_EXTENDED2, an alias for SPH_MATCH_EXTENDED (default 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 Manticore 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, Manticore 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 Manticore document ID in ascending order (earliest first).

## 5.2 Boolean query syntax

Boolean queries allow the following special operators to be used:

- operator OR:

```
hello | world
```

- operator NOT:

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

- grouping:

```
( hello world )
```

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

Example 5.1. 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 may be automatically optimized if OPTION boolean_simplify=1 is specified. Some transformations performed by this optimization include:

- Excess brackets: ((A | B) | C) becomes ( A | B | C ); ((A B) C) becomes ( A B C )
- Excess AND NOT: ((A !N1) !N2) becomes (A !(N1 | N2))
- Common NOT: ((A !N) | (B !N)) becomes ((A|B) !N)
- Common Compound NOT: ((A !(N AA)) | (B !(N BB))) becomes (((A|B) !N) | (A !AA) | (B !BB)) if the cost of evaluating N is greater than the added together costs of evaluating A and B
- Common subterm: ((A (N | AA)) | (B (N | BB))) becomes (((A|B) N) | (A AA) | (B BB)) if the cost of evaluating N is greater than the added together costs of evaluating A and B
- Common keywords: (A | "A B"~N) becomes A; ("A B" | "A B C") becomes "A B"; ("A B"~N | "A B C"~N) becomes ("A B"~N)
- Common phrase: ("X A B" | "Y A B") becomes (("X|Y") "A B")
- Common AND NOT: ((A !X) | (A !Y) | (A !Z)) becomes (A !(X Y Z))
- Common OR NOT: ((A !(N | N1)) | (B !(N | N2))) becomes (( (A !N1) | (B !N2) ) !N)

Note that optimizing the queries consumes CPU time, so for simple queries -or for hand-optimized queries- you'll do better with the default boolean_simplify=0 value. Simplifications are often better for complex queries, or algorithmically generated queries.

Queries like "-dog", which implicitly include all documents from the collection, can not be evaluated. This is both for technical and performance reasons. Technically, Manticore 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.

## 5.3 Extended query syntax

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

- operator OR:

```
hello | world
```

- operator MAYBE:

```
hello MAYBE world
```

- operator NOT:

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

- field search operator:

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

- field position limit modifier:

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

- multiple-field search operator:

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

- ignore field search operator (will ignore any matches of 'hello world' from field 'title'):

```
@!title hello world
```

- ignore multiple-field search operator (if we have fields title, subject and body then @!(title) is equivalent to @(subject,body)):

```
@!(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:

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

- field-start and field-end modifier:

```
^hello world$
```

- keyword IDF boost modifier:

```
boosted^1.234 boostedfieldend$^1.234
```

- NEAR, generalized proximity operator:

```
hello NEAR/3 world NEAR/4 "my test"
```

- SENTENCE operator:

```
all SENTENCE words SENTENCE "in one sentence"
```

- PARAGRAPH operator:

```
"Bill Gates" PARAGRAPH "Steve Jobs"
```

- ZONE limit operator:

```
    ZONE:(h3,h4)

only in these titles
```

- ZONESPAN limit operator:

```
    ZONESPAN:(h2)

only in a (single) title
```

- NOTNEAR, negative assertion operator:

```
Church NOTNEAR/3 street
```

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

Example 5.2. 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, 5 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 5 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 additionally 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. Operator is limited to 255 keywords. Instead of an absolute number, you can also specify a number between 0.0 and 1.0 (standing for 0% and 100%), and Manticore will match only documents with at least the specified percentage of given words. The same example above could also have been written "the world is a wonderful place"/0.5 and it would match documents with at least 50% of the 6 words.

Strict order operator (aka operator "before") 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 will match the document only if the keyword occurred in exactly the specified form. The default behavior 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. It is possible to apply an exact form modifier to the phrase operator. It's really just syntax sugar - it adds an exact form modifier to all terms contained within the phrase.

```
="exact phrase"
```

Field-start and field-end keyword modifiers 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.

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
```

The phrase search operator may include a 'match any term' modifier. Terms within the phrase operator are position significant. When the 'match any term' modifier is implemented, the position of the subsequent terms from that phrase query will be shifted. Therefore, 'match any' has no impact on search performance.

```
"exact * phrase * * for terms"
```

**NEAR operator** is a generalized version of a proximity operator. The syntax is `NEAR/N`, it is case-sensitive, and no spaces are allowed between the NEAR keyword, the slash sign, and the distance value.

The original proximity operator only worked on sets of keywords. NEAR is more generic and can accept arbitrary subexpressions as its two arguments, matching the document when both subexpressions are found within N words of each other, no matter in which order. NEAR is left associative and has the same (lowest) precedence as BEFORE.

You should also note how a `(one NEAR/7 two NEAR/7 three)` query using NEAR is not really equivalent to a `("one two three"~7)` one using keyword proximity operator. The difference here is that the proximity operator allows for up to 6 non-matching words between all the 3 matching words, but the version with NEAR is less restrictive: it would allow for up to 6 words between 'one' and 'two' and then for up to 6 more between that two-word matching and a 'three' keyword.

**SENTENCE and PARAGRAPH operators** matches the document when both its arguments are within the same sentence or the same paragraph of text, respectively. The arguments can be either keywords, or phrases, or the instances of the same operator. Here are a few examples:

```
one SENTENCE two
one SENTENCE "two three"
one SENTENCE "two three" SENTENCE four
```

The order of the arguments within the sentence or paragraph does not matter. These operators only work on indexes built with *index_sp* (sentence and paragraph indexing feature) enabled, and revert to a mere AND otherwise. Refer to the `index_sp` directive documentation for the notes on what's considered a sentence and a paragraph.

**ZONE limit operator** is quite similar to field limit operator, but restricts matching to a given in-field zone or a list of zones. Note that the subsequent subexpressions are *not* required to match in a single contiguous span of a given zone, and may match in multiple spans. For instance, `(ZONE:th hello world)` query *will* match this example document:

```
<th>Table 1\. Local awareness of Hello Kitty brand.</th>
.. some table data goes here ..
<th>Table 2\. World-wide brand awareness.</th>
```

ZONE operator affects the query until the next field or ZONE limit operator, or the closing parenthesis. It only works on the indexes built with zones support (see *index_zones*) and will be ignored otherwise.

**ZONESPAN limit operator** is similar to the ZONE operator, but requires the match to occur in a single contiguous span. In the example above, `(ZONESPAN:th hello world)` would not match the document, since "hello" and "world" do not occur within the same span.

**MAYBE** operator works much like | operator but doesn't return documents which match only right subtree expression.

**NOTNEAR operator** is a negative assertion. It matches the document when left argument exists and either there is no right argument in document or right argument is distance away from left matched argument's end. The distance is specified in words. The syntax is `NOTNEAR/N`, it is case-sensitive, and no spaces are allowed between the NOTNEAR keyword, the slash sign, and the distance value. Both arguments of this operator might be terms or any operators or group of operators.

## 5.4 Search results ranking

### 5.4.1 Ranking overview

Ranking (aka weighting) of the search results can be defined as a process of computing a so-called relevance (aka weight) for every given matched document with regards to a given query that matched it. So relevance is in the end just a number attached to every document that estimates how relevant the document is to the query. Search results can then be sorted based on this number and/or some additional parameters, so that the most sought after results would come up higher on the results page.

There is no single standard one-size-fits-all way to rank any document in any scenario. Moreover, there can not ever be such a way, because relevance is *subjective*. As in, what seems relevant to you might not seem relevant to me. Hence, in general case it's not just hard to compute, it's theoretically impossible.

So ranking in Manticore is configurable. It has a notion of a so-called **ranker**. A ranker can formally be defined as a function that takes document and query as its input and produces a relevance value as output. In layman's terms, a ranker controls exactly how (using which specific algorithm) will Manticore assign weights to the document.

Previously, this ranking function was rigidly bound to the matching mode. So in the legacy matching modes (that is, SPH_MATCH_ALL, SPH_MATCH_ANY, SPH_MATCH_PHRASE, and SPH_MATCH_BOOLEAN) you can not choose the ranker. You can only do that in the SPH_MATCH_EXTENDED mode. (Which is the only mode in SphinxQL and the suggested mode in SphinxAPI anyway.) To choose a non-default ranker you can either use *SetRankingMode()* with SphinxAPI, or *OPTION ranker* clause in `SELECT` statement when using SphinxQL.

As a sidenote, legacy matching modes are internally implemented via the unified syntax anyway. When you use one of those modes, Manticore just internally adjusts the query and sets the associated ranker, then executes the query using the very same unified code path.

### 5.4.2 Available built-in rankers

Manticore ships with a number of built-in rankers suited for different purposes. A number of them uses two factors, phrase proximity (aka LCS) and BM25. Phrase proximity works on the keyword positions, while BM25 works on the keyword frequencies. Basically, the better the degree of the phrase match between the document body and the query, the higher is the phrase proximity (it maxes out when the document contains the entire query as a verbatim quote). And BM25 is higher when the document contains more rare words. We'll save the detailed discussion for later.

Currently implemented rankers are:

- **SPH_RANK_PROXIMITY_BM25**, the default ranking mode that 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**, no ranking mode. This mode is obviously the fastest. A weight of 1 is assigned to all matches. This is sometimes called boolean searching that just matches the documents but does not rank them.
- **SPH_RANK_WORDCOUNT**, ranking by the keyword occurrences count. This ranker computes the per-field keyword occurrence counts, then multiplies them by field weights, and sums the resulting values.

- **SPH_RANK_PROXIMITY** returns raw phrase proximity value as a result. This mode is internally used to emulate SPH_MATCH_ALL queries.
- **SPH_RANK_MATCHANY** returns rank as it was computed in SPH_MATCH_ANY mode earlier, and is internally used to emulate SPH_MATCH_ANY queries.
- **SPH_RANK_FIELDMASK** 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 occurrences satisfying the query.
- **SPH_RANK_SPH04** is generally based on the default SPH_RANK_PROXIMITY_BM25 ranker, but additionally boosts the matches when they occur in the very beginning or the very end of a text field. Thus, if a field equals the exact query, SPH04 should rank it higher than a field that contains the exact query but is not equal to it. (For instance, when the query is "Hyde Park", a document entitled "Hyde Park" should be ranked higher than a one entitled "Hyde Park, London" or "The Hyde Park Cafe".)
- **SPH_RANK_EXPR** lets you specify the ranking formula in run time. It exposes a number of internal text factors and lets you define how the final weight should be computed from those factors. You can find more details about its syntax and a reference available factors in a subsection below.

You should specify the `SPH_RANK_` prefix and use capital letters only when using the *SetRankingMode()* call from the SphinxAPI. The API ports expose these as global constants. Using SphinxQL syntax, the prefix should be omitted and the ranker name is case insensitive. Example:

```
// SphinxAPI
$client->SetRankingMode ( SPH_RANK_SPH04 );

// SphinxQL
mysql_query ( "SELECT ... OPTION ranker=sph04" );
```

### Legacy matching modes rankers

Legacy matching modes automatically select a ranker as follows:

- SPH_MATCH_ALL uses SPH_RANK_PROXIMITY ranker;
- SPH_MATCH_ANY uses SPH_RANK_MATCHANY ranker;
- SPH_MATCH_PHRASE uses SPH_RANK_PROXIMITY ranker;
- SPH_MATCH_BOOLEAN uses SPH_RANK_NONE ranker.

### 5.4.3 Quick summary of the ranking factors

| Name | Level | Type | Summary |
|---|---|---|---|
| max_lcs | query | int | maximum possible LCS value for the current query |
| bm25 | document | int | quick estimate of BM25(1.2, 0) without syntax support |
| bm25a(k1, b) | document | int | precise BM25() value with configurable K1, B constants and syntax support |
| bm25f(k1, b, {field=weight, ... }) | document | int | precise BM25F() value with extra configurable field weights |
| field_mask | document | int | bit mask of matched fields |
| query_word_count | document | int | number of unique inclusive keywords in a query |
| doc_word_count | document | int | number of unique keywords matched in the document |
| lcs | field | int | Longest Common Subsequence between query and document, in words |
| user_weight | field | int | user field weight |
| hit_count | field | int | total number of keyword occurrences |
| word_count | field | int | number of unique matched keywords |
| tf_idf | field | float | sum(tf*idf) over matched keywords == sum(idf) over occurrences |
| min_hit_pos | field | int | first matched occurrence position, in words, 1-based |
| min_best_span_pos | field | int | first maximum LCS span position, in words, 1-based |
| exact_hit | field | bool | whether query == field |
| min_idf | field | float | min(idf) over matched keywords |
| max_idf | field | float | max(idf) over matched keywords |
| sum_idf | field | float | sum(idf) over matched keywords |
| exact_order | field | bool | whether all query keywords were a) matched and b) in query order |
| min_gaps | field | int | minimum number of gaps between the matched keywords over the matching spans |
| lccs | field | int | Longest Common Contiguous Subsequence between query and document, in words |
| wlccs | field | float | Weighted Longest Common Contiguous Subsequence, sum(idf) over contiguous keyword spans |
| atc | field | float | Aggregate Term Closeness, log(1+sum(idf1*idf2*pow(distance, -1.75)) over the best pairs of keywords |

### 5.4.4 Document-level ranking factors

A **document-level factor** is a numeric value computed by the ranking engine for every matched document with regards to the current query. So it differs from a plain document attribute in that the attribute do not depend on the full text query, while factors might. Those factors can be used anywhere in the ranking expression. Currently implemented document-level factors are:

- `bm25` (integer), a document-level BM25 estimate (computed without keyword occurrence filtering).
- `max_lcs` (integer), a query-level maximum possible value that the sum(lcs*user_weight) expression can ever take. This can be useful for weight boost scaling. For instance, MATCHANY ranker formula uses this to guarantee that a full phrase match in any field ranks higher than any combination of partial matches in all fields.
- `field_mask` (integer), a document-level 32-bit mask of matched fields.
- `query_word_count` (integer), the number of unique keywords in a query, adjusted for a number of excluded

keywords. For instance, both `(one one one one)` and `(one !two)` queries should assign a value of 1 to this factor, because there is just one unique non-excluded keyword.

- `doc_word_count` (integer), the number of unique keywords matched in the entire document.

### 5.4.5 Field-level ranking factors

A **field-level factor** is a numeric value computed by the ranking engine for every matched in-document text field with regards to the current query. As more than one field can be matched by a query, but the final weight needs to be a single integer value, these values need to be folded into a single one. To achieve that, field-level factors can only be used within a field aggregation function, they can **not** be used anywhere in the expression. For example, you can not use `(lcs+bm25)` as your ranking expression, as `lcs` takes multiple values (one in every matched field). You should use `(sum(lcs)+bm25)` instead, that expression sums `lcs` over all matching fields, and then adds `bm25` to that per-field sum. Currently implemented field-level factors are:

- `lcs` (integer), the length of a maximum verbatim match between the document and the query, counted in words. LCS stands for Longest Common Subsequence (or Subset). Takes a minimum value of 1 when only stray keywords were matched in a field, and a maximum value of query keywords count when the entire query was matched in a field verbatim (in the exact query keywords order). For example, if the query is 'hello world' and the field contains these two words quoted from the query (that is, adjacent to each other, and exactly in the query order), `lcs` will be 2. For example, if the query is 'hello world program' and the field contains 'hello world', `lcs` will be 2. Note that any subset of the query keyword works, not just a subset of adjacent keywords. For example, if the query is 'hello world program' and the field contains 'hello (test program)', `lcs` will be 2 just as well, because both 'hello' and 'program' matched in the same respective positions as they were in the query. Finally, if the query is 'hello world program' and the field contains 'hello world program', `lcs` will be 3. (Hopefully that is unsurprising at this point.)
- `user_weight` (integer), the user specified per-field weight (refer to *SetFieldWeights()* in SphinxAPI and *OPTION field_weights* in SphinxQL respectively). The weights default to 1 if not specified explicitly.
- `hit_count` (integer), the number of keyword occurrences that matched in the field. Note that a single keyword may occur multiple times. For example, if 'hello' occurs 3 times in a field and 'world' occurs 5 times, `hit_count` will be 8.
- `word_count` (integer), the number of unique keywords matched in the field. For example, if 'hello' and 'world' occur anywhere in a field, `word_count` will be 2, irregardless of how many times do both keywords occur.
- `tf_idf` (float), the sum of TF/IDF over all the keywords matched in the field. IDF is the Inverse Document Frequency, a floating point value between 0 and 1 that describes how frequent is the keywords (basically, 0 for a keyword that occurs in every document indexed, and 1 for a unique keyword that occurs in just a single document). TF is the Term Frequency, the number of matched keyword occurrences in the field. As a side note, `tf_idf` is actually computed by summing IDF over all matched occurrences. That's by construction equivalent to summing TF*IDF over all matched keywords.
- `min_hit_pos` (integer), the position of the first matched keyword occurrence, counted in words. Indexing begins from position 1.
- `min_best_span_pos` (integer), the position of the first maximum LCS occurrences span. For example, assume that our query was 'hello world program' and 'hello world' subphrase was matched twice in the field, in positions 13 and 21. Assume that 'hello' and 'world' additionally occurred elsewhere in the field, but never next to each other and thus never as a subphrase match. In that case, `min_best_span_pos` will be 13. Note how for the single keyword queries `min_best_span_pos` will always equal `min_hit_pos`.
- `exact_hit` (boolean), whether a query was an exact match of the entire current field. Used in the SPH04 ranker.

- `min_idf`, `max_idf`, and `sum_idf` (float). These factors respectively represent the min(idf), max(idf) and sum(idf) over all keywords that were matched in the field.
- `exact_order` (boolean). Whether all of the query keywords were matched in the field in the exact query order. For example, `(microsoft office)` query would yield exact_order=1 in a field with the following contents: `(We use Microsoft software in our office.)`. However, the very same query in a `(Our office is Microsoft free.)` field would yield exact_order=0.
- `min_gaps` (integer), the minimum number of positional gaps between (just) the keywords matched in field. Always 0 when less than 2 keywords match; always greater or equal than 0 otherwise.

  For example, with a `[big wolf]` query, `[big bad wolf]` field would yield min_gaps=1; `[big bad hairy wolf]` field would yield min_gaps=2; `[the wolf was scary and big]` field would yield min_gaps=3; etc. However, a field like `[i heard a wolf howl]` would yield min_gaps=0, because only one keyword would be matching in that field, and, naturally, there would be no gaps between the _matched_keywords.

  Therefore, this is a rather low-level, "raw" factor that you would most likely want to *adjust* before actually using for ranking. Specific adjustments depend heavily on your data and the resulting formula, but here are a few ideas you can start with: (a) any min_gaps based boosts could be simply ignored when word_count<2; (b) non-trivial min_gaps values (i.e. when word_count>=2) could be clamped with a certain "worst case" constant while trivial values (i.e. when min_gaps=0 and word_count<2) could be replaced by that constant; (c) a transfer function like 1/(1+min_gaps) could be applied (so that better, smaller min_gaps values would maximize it and worse, bigger min_gaps values would fall off slowly); and so on.
- `lccs` (integer). Longest Common Contiguous Subsequence. A length of the longest subphrase that is common between the query and the document, computed in keywords.

  LCCS factor is rather similar to LCS but more restrictive, in a sense. While LCS could be greater than 1 though no two query words are matched next to each other, LCCS would only get greater than 1 if there are *exact*, contiguous query subphrases in the document. For example, (one two three four five) query vs (one hundred three hundred five hundred) document would yield lcs=3, but lccs=1, because even though mutual dispositions of 3 keywords (one, three, five) match between the query and the document, no 2 matching positions are actually next to each other.

  Note that LCCS still does not differentiate between the frequent and rare keywords; for that, see WLCCS.
- `wlccs` (float). Weighted Longest Common Contiguous Subsequence. A sum of IDFs of the keywords of the longest subphrase that is common between the query and the document.

  WLCCS is computed very similarly to LCCS, but every "suitable" keyword occurrence increases it by the keyword IDF rather than just by 1 (which is the case with LCS and LCCS). That lets us rank sequences of more rare and important keywords higher than sequences of frequent keywords, even if the latter are longer. For example, a query `(Zanzibar bed and breakfast)` would yield lccs=1 for a `(hotels of Zanzibar)` document, but lccs=3 against `(London bed and breakfast)`, even though "Zanzibar" is actually somewhat more rare than the entire "bed and breakfast" phrase. WLCCS factor alleviates that problem by using the keyword frequencies.
- `atc` (float). Aggregate Term Closeness. A proximity based measure that grows higher when the document contains more groups of more closely located and more important (rare) query keywords. **WARNING:** you should use ATC with OPTION idf='plain,tfidf_unnormalized'; otherwise you would get unexpected results.

  ATC basically works as follows. For every keyword *occurrence* in the document, we compute the so called *term closeness*. For that, we examine all the other closest occurrences of all the query keywords (keyword itself included too) to the left and to the right of the subject occurrence, compute a distance dampening coefficient as k = pow(distance, -1.75) for those occurrences, and sum the dampened IDFs. Thus for every occurrence of every keyword, we get a "closeness" value that describes the "neighbors" of that occurrence. We then multiply those per-occurrence closenesses by their respective subject keyword IDF, sum them all, and finally, compute a logarithm of that sum.

Or in other words, we process the best (closest) matched keyword pairs in the document, and compute pairwise "closenesses" as the product of their IDFs scaled by the distance coefficient:

```
pair_tc = idf(pair_word1) * idf(pair_word2) * pow(pair_distance, -1.75)
```

We then sum such closenesses, and compute the final, log-dampened ATC value:

```
atc = log(1+sum(pair_tc))
```

Note that this final dampening logarithm is exactly the reason you should use OPTION idf=plain, because without it, the expression inside the log() could be negative.

Having closer keyword occurrences actually contributes *much* more to ATC than having more frequent keywords. Indeed, when the keywords are right next to each other, distance=1 and k=1; when there just one word in between them, distance=2 and k=0.297, with two words between, distance=3 and k=0.146, and so on. At the same time IDF attenuates somewhat slower. For example, in a 1 million document collection, the IDF values for keywords that match in 10, 100, and 1000 documents would be respectively 0.833, 0.667, and 0.500. So a keyword pair with two rather rare keywords that occur in just 10 documents each but with 2 other words in between would yield pair_tc = 0.101 and thus just barely outweigh a pair with a 100-doc and a 1000-doc keyword with 1 other word between them and pair_tc = 0.099. Moreover, a pair of two *unique*, 1-doc keywords with 3 words between them would get a pair_tc = 0.088 and lose to a pair of two 1000-doc keywords located right next to each other and yielding a pair_tc = 0.25. So, basically, while ATC does combine both keyword frequency and proximity, it is still somewhat favoring the proximity.

### 5.4.6 Ranking factor aggregation functions

A **field aggregation function** is a single argument function that takes an expression with field-level factors, iterates it over all the matched fields, and computes the final results. Currently implemented field aggregation functions are:

- `sum`, sums the argument expression over all matched fields. For instance, `sum(1)` should return a number of matched fields.
- `top`, returns the greatest value of the argument over all matched fields.

### 5.4.7 Formula expressions for all the built-in rankers

Most of the other rankers can actually be emulated with the expression based ranker. You just need to pass a proper expression. Such emulation is, of course, going to be slower than using the built-in, compiled ranker but still might be of interest if you want to fine-tune your ranking formula starting with one of the existing ones. Also, the formulas define the nitty gritty ranker details in a nicely readable fashion.

- SPH_RANK_PROXIMITY_BM25 = sum(lcs*user_weight)*1000+bm25
- SPH_RANK_BM25 = bm25
- SPH_RANK_NONE = 1
- SPH_RANK_WORDCOUNT = sum(hit_count*user_weight)
- SPH_RANK_PROXIMITY = sum(lcs*user_weight)
- SPH_RANK_MATCHANY = sum((word_count+(lcs-1)*max_lcs*)user_weight)
- SPH_RANK_FIELDMASK = field_mask
- SPH_RANK_SPH04 = sum((4*lcs*+2(min_hit_pos==1)+exact_hit)*user_weight)*1000+bm25

## 5.5 Expressions, functions, and operators

Manticore lets you use arbitrary arithmetic expressions both via SphinxQL and SphinxAPI, involving attribute values, internal attributes (document ID and relevance weight), arithmetic operations, a number of built-in functions, and user-defined functions. This section documents the supported operators and functions. Here's the complete reference list for quick access.

- *Arithmetic operators*: +, -, *, /, %, DIV, MOD
- *Comparison operators*: <, > <=, >=, =, <>
- *Boolean operators: AND, OR, NOT*
- *Bitwise operators: &,* |
- *ABS()*
- *ALL()*
- *ANY()*
- *ATAN2()*
- *BIGINT()*
- *BITDOT()*
- *CEIL()*
- *CONTAINS()*
- *COS()*
- *CRC32()*
- *DAY()*
- *DOUBLE()*
- *EXP()*
- *FIBONACCI()*
- *FLOOR()*
- *GEODIST()*
- *GEOPOLY2D()*
- *GREATEST()*
- *HOUR()*
- *IDIV()*
- *IF()*
- *IN()*
- *INDEXOF()*
- *INTEGER()*
- *INTERVAL()*
- *LEAST()*
- *LENGTH()*

- *LN()*
- *LOG10()*
- *LOG2()*
- *MAX()*
- *MIN()*
- *MINUTE()*
- *MIN_TOP_SORTVAL()*
- *MIN_TOP_WEIGHT()*
- *MONTH()*
- *NOW()*
- *POLY2D()*
- *POW()*
- *RAND()*
- *REMAP()*
- *SECOND()*
- *SIN()*
- *SINT()*
- *SQRT()*
- *UINT()*
- *YEAR()*
- *YEARMONTH()*
- *YEARMONTHDAY()*

### 5.5.1 Operators

- Arithmetic operators: +, -, *, /, %, DIV, MOD

  The standard arithmetic operators. Arithmetic calculations involving those can be performed in three different modes: (a) using single-precision, 32-bit IEEE 754 floating point values (the default), (**) 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/**` or `sqrt(a)` will always be computed in floats, because these operations return a result of non-integer type. To avoid the first, you can either use `IDIV(a,**)` or `a DIV b` form. 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: <, > <=, >=, =, <>

  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

  Boolean operators (AND, OR, NOT) 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.

- Bitwise operators: &, |

  These operators perform bitwise AND and OR respectively. The operands must be of an integer types.

### 5.5.2 Numeric functions

- ABS()

  Returns the absolute value of the argument.

- BITDOT()

  BITDOT(mask, w0, w1, ...) returns the sum of products of an each bit of a mask multiplied with its weight. `bit0*w0 + bit1*w1 + ...`

- CEIL()

  Returns the smallest integer value greater or equal to the argument.

- CONTAINS()

  CONTAINS(polygon, x, y) checks whether the (x,y) point is within the given polygon, and returns 1 if true, or 0 if false. The polygon has to be specified using either the *POLY2D()* function or the *GEOPOLY2D()* function. The former function is intended for "small" polygons, meaning less than 500 km (300 miles) a side, and it doesn't take into account the Earth's curvature for speed. For larger distances, you should use GEOPOLY2D, which tessellates the given polygon in smaller parts, accounting for the Earth's curvature.

- COS()

  Returns the cosine of the argument.

- DOUBLE() Forcibly promotes given argument to floating point type. Intended to help enforce evaluation of numeric JSON fields.

- EXP()

  Returns the exponent of the argument (e=2.718... to the power of the argument).

- FIBONACCI()

  Returns the N-th Fibonacci number, where N is the integer argument. That is, arguments of 0 and up will generate the values 0, 1, 1, 2, 3, 5, 8, 13 and so on. Note that the computations are done using 32-bit integer math and thus numbers 48th and up will be returned modulo 2^32.

- FLOOR()

  Returns the largest integer value lesser or equal to the argument.

- GEOPOLY2D()

  GEOPOLY2D(x1,y1,x2,y2,x3,y3...) produces a polygon to be used with the *CONTAINS()* function. This function takes into account the Earth's curvature by tessellating the polygon into smaller ones, and should be used for larger areas; see the *POLY2D()* function. The function expects coordinates to be in degrees, if radians are used it will give same result as POLY2D().

- IDIV()

  Returns the result of an integer division of the first argument by the second argument. Both arguments must be of an integer type.

- LN()

  Returns the natural logarithm of the argument (with the base of e=2.718...).

- LOG10()

  Returns the common logarithm of the argument (with the base of 10).

- LOG2()

  Returns the binary logarithm of the argument (with the base of 2).

- MAX()

  Returns the bigger of two arguments.

- MIN()

  Returns the smaller of two arguments.

- POLY2D()

  POLY2D(x1,y1,x2,y2,x3,y3...) produces a polygon to be used with the *CONTAINS()* function. This polygon assumes a flat Earth, so it should not be too large; see the *POLY2D()* function.

- POW()

  Returns the first argument raised to the power of the second argument.

- SIN()

  Returns the sine of the argument.

- SQRT()

  Returns the square root of the argument.

- UINT()

  Forcibly reinterprets given argument to 64-bit unsigned type.

### 5.5.3 Date and time functions

- DAY()

  Returns the integer day of month (in 1..31 range) from a timestamp argument, according to the current timezone.

- MONTH()

  Returns the integer month (in 1..12 range) from a timestamp argument, according to the current timezone.

- NOW()

  Returns the current timestamp as an INTEGER.

- YEAR()

  Returns the integer year (in 1969..2038 range) from a timestamp argument, according to the current timezone.

- YEARMONTH()

  Returns the integer year and month code (in 196912..203801 range) from a timestamp argument, according to the current timezone.

- YEARMONTHDAY()

  Returns the integer year, month, and date code (in 19691231..20380119 range) from a timestamp argument, according to the current timezone.

- SECOND()

  Returns the integer second (in 0..59 range) from a timestamp argument, according to the current timezone.

- MINUTE()

  Returns the integer minute (in 0..59 range) from a timestamp argument, according to the current timezone.

- HOUR()

  Returns the integer hour (in 0..23 range) from a timestamp argument, according to the current timezone.

### 5.5.4 Type conversion functions

- BIGINT()

  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.

- INTEGER()

  Forcibly promotes given argument to 64-bit signed type. Intended to help enforce evaluation of numeric JSON fields.

- SINT()

  Forcibly reinterprets its 32-bit unsigned integer argument as signed, and also expands it to 64-bit type (because 32-bit type is unsigned). It's easily illustrated by the following example: 1-2 normally evaluates to 4294967295, but SINT(1-2) evaluates to -1.

### 5.5.5 Comparison functions

- IF()

  `IF()` behavior is slightly different that that of its MySQL counterpart. 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 '**' 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.
```

- IN()

  IN(expr,val1,val2,...) 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. 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. IN() also supports `IN(expr, @uservar)` syntax to check whether the value belongs to the list in the given global user variable. First argument can be JSON attribute.

- INTERVAL()

  INTERVAL(expr,point1,point2,point3,...), 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.

### 5.5.6 Miscellaneous functions

- ALL()

  ALL(cond FOR var IN json.array) applies to JSON arrays and returns 1 if condition is true for all elements in array and 0 otherwise. 'cond' is a general expression which additionally can use 'var' as current value of an array element within itself.

```
SELECT ALL(x>3 AND x<7 FOR x IN j.intarray) FROM test;
```

  ALL(mva) is a special constructor for multi value attributes. When used in conjunction with comparison operators it returns 1 if all values compared are found among the MVA values.

```
SELECT * FROM test WHERE ALL(mymva)>10;
```

- ANY()

  ANY(cond FOR var IN json.array) works similar to *ALL()* except for it returns 1 if condition is true for any element in array.

  ANY(mva) is a special constructor for multi value attributes. When used in conjunction with comparison operators it returns 1 if any of the values compared are found among the MVA values. ANY is used by default if no constructor is used, however a warning will be raised about missing constructor.

- ATAN2()

  Returns the arctangent function of two arguments, expressed in **radians**.

- CRC32()

  Returns the CRC32 value of a string argument.

- GEODIST()

  GEODIST(lat1, lon1, lat2, lon2, [...]) function computes geosphere distance between two given points specified by their coordinates. Note that by default 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.

  GEODIST() also takes an optional 5th argument that lets you easily convert between input and output units, and pick the specific geodistance formula to use. The complete syntax and a few examples are as follows:

```
GEODIST(lat1, lon1, lat2, lon2, { option=value, ... })

GEODIST(40.7643929, -73.9997683, 40.7642578, -73.9994565, {in=degrees, out=feet})

GEODIST(51.50, -0.12, 29.98, 31.13, {in=deg, out=mi}}
```

The known options and their values are:

- `in = {deg | degrees | rad | radians}`, specifies the input units;
- `out = {m | meters | km | kilometers | ft | feet | mi | miles}`, specifies the output units;
- `method = {adaptive | haversine}`, specifies the geodistance calculation method.

The default method is "adaptive". It is well optimized implementation that is both more precise *and* much faster at all times than "haversine".

- GREATEST()

  GREATEST(attr_json.some_array) function takes JSON array as the argument, and returns the greatest value in that array. Also works for MVA.

- INDEXOF()

  INDEXOF(cond FOR var IN json.array) function iterates through all elements in array and returns index of first element for which 'cond' is true and -1 if 'cond' is false for every element in array.

```
SELECT INDEXOF(name='John' FOR name IN j.peoples) FROM test;
```

- LEAST()

  LEAST(attr_json.some_array) function takes JSON array as the argument, and returns the least value in that array. Also works for MVA.

- LENGTH()

  LENGTH(attr_mva) function returns amount of elements in MVA set. It works with both 32-bit and 64-bit MVA attributes. LENGTH(attr_json) returns length of a field in JSON. Return value depends on type of a field. For example LENGTH(json_attr.some_int) always returns 1 and LENGTH(json_attr.some_array) returns number of elements in array.

- MIN_TOP_SORTVAL()

  Returns sort key value of the worst found element in the current top-N matches if sort key is float and 0 otherwise.

- MIN_TOP_WEIGHT() Returns weight of the worst found element in the current top-N matches.

- PACKEDFACTORS()

  PACKEDFACTORS() can be used in queries, either to just see all the weighting factors calculated when doing the matching, or to provide a binary attribute that can be used to write a custom ranking UDF. This function works only if expression ranker is specified and the query is not a full scan, otherwise it will return an error. PACKEDFACTORS() can take an optional argument that disables ATC ranking factor calculation:

```
PACKEDFACTORS({no_atc=1})
```

Calculating ATC slows down query processing considerably, so this option can be useful if you need to see the ranking factors, but do not need ATC. PACKEDFACTORS() can also be told to format its output as JSON:

```
PACKEDFACTORS({json=1})
```

The respective outputs in either key-value pair or JSON format would look as follows below. (Note that the examples below are wrapped for readability; actual returned values would be single-line.)

```
mysql> SELECT id, PACKEDFACTORS() FROM test1
    -> WHERE MATCH('test one') OPTION ranker=expr('1') \G
*************************** 1\. row ***************************
             id: 1
packedfactors(): bm25=569, bm25a=0.617197, field_mask=2, doc_word_count=2,
    field1=(lcs=1, hit_count=2, word_count=2, tf_idf=0.152356,
        min_idf=-0.062982, max_idf=0.215338, sum_idf=0.152356, min_hit_pos=4,
        min_best_span_pos=4, exact_hit=0, max_window_hits=1, min_gaps=2,
        exact_order=1, lccs=1, wlccs=0.215338, atc=-0.003974),
    word0=(tf=1, idf=-0.062982),
    word1=(tf=1, idf=0.215338)
1 row in set (0.00 sec)

mysql> SELECT id, PACKEDFACTORS({json=1}) FROM test1
    -> WHERE MATCH('test one') OPTION ranker=expr('1') \G
*************************** 1\. row ***************************
                      id: 1
packedfactors({json=1}):
{

    "bm25": 569,
    "bm25a": 0.617197,
    "field_mask": 2,
    "doc_word_count": 2,
    "fields": [
        {
            "lcs": 1,
            "hit_count": 2,
            "word_count": 2,
            "tf_idf": 0.152356,
            "min_idf": -0.062982,
            "max_idf": 0.215338,
            "sum_idf": 0.152356,
            "min_hit_pos": 4,
            "min_best_span_pos": 4,
            "exact_hit": 0,
            "max_window_hits": 1,
            "min_gaps": 2,
            "exact_order": 1,
            "lccs": 1,
            "wlccs": 0.215338,
            "atc": -0.003974
        }
    ],
    "words": [
        {
            "tf": 1,
            "idf": -0.062982
        },
        {
            "tf": 1,
            "idf": 0.215338
        }
    ]
```

(continues on next page)

(continued from previous page)

```
}
1 row in set (0.01 sec)
```

This function can be used to implement custom ranking functions in UDFs, as in

```
SELECT *, CUSTOM_RANK(PACKEDFACTORS()) AS r
FROM my_index
WHERE match('hello')
ORDER BY r DESC
OPTION ranker=expr('1');
```

Where CUSTOM_RANK() is a function implemented in an UDF. It should declare a SPH_UDF_FACTORS structure (defined in `sphinxudf.h`), initialize this structure, unpack the factors into it before usage, and deinitialize it afterwards, as follows:

```
SPH_UDF_FACTORS factors;
sphinx_factors_init(&factors);
sphinx_factors_unpack((DWORD*)args->arg_values[0], &factors);
// ... can use the contents of factors variable here ...
sphinx_factors_deinit(&factors);
```

PACKEDFACTORS() data is available at all query stages, not just when doing the initial matching and ranking pass. That enables another particularly interesting application of PACKEDFACTORS(), namely **re-ranking**.

In the example just above, we used an expression-based ranker with a dummy expression, and sorted the result set by the value computed by our UDF. In other words, we used the UDF to *rank* all our results. Assume now, for the sake of an example, that our UDF is extremely expensive to compute and has a throughput of just 10,000 calls per second. Assume that our query matches 1,000,000 documents. To maintain reasonable performance, we would then want to use a (much) simpler expression to do most of our ranking, and then apply the expensive UDF to only a few top results, say, top-100 results. Or, in other words, build top-100 results using a simpler ranking function and then *re-rank* those with a complex one. We can do that just as well with subselects:

```
SELECT * FROM (
    SELECT *, CUSTOM_RANK(PACKEDFACTORS()) AS r
    FROM my_index WHERE match('hello')
    OPTION ranker=expr('sum(lcs)*1000+bm25')
    ORDER BY WEIGHT() DESC
    LIMIT 100
) ORDER BY r DESC LIMIT 10
```

In this example, expression-based ranker will be called for every matched document to compute WEIGHT(). So it will get called 1,000,000 times. But the UDF computation can be postponed until the outer sort. And it also will be done for just the top-100 matches by WEIGHT(), according to the inner limit. So the UDF will only get called 100 times. And then the final top-10 matches by UDF value will be selected and returned to the application.

For reference, in the distributed case PACKEDFACTORS() data gets sent from the agents to master in a binary format, too. This makes it technically feasible to implement additional re-ranking pass (or passes) on the master node, if needed.

If used with SphinxQL but not called from any UDFs, the result of PACKEDFACTORS() is simply formatted as plain text, which can be used to manually assess the ranking factors. Note that this feature is not currently supported by the Manticore API.

- REMAP()

  REMAP(condition, expression, (cond1, cond2, ...), (expr1, expr2, ...)) function allows you to make some

exceptions of an expression values depending on condition values. Condition expression should always result integer, expression can result in integer or float.

```
SELECT REMAP(userid, karmapoints, (1, 67), (999, 0)) FROM users;
SELECT REMAP(id%10, salary, (0), (0.0)) FROM employes;
```

- rand()

  RAND(seed) function returns a random float between 0..1. Optional, an integer seed value can be specified.

## 5.6 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.

### 5.6.1 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 within 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.

### 5.6.2 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`.

### 5.6.3 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 operators and functions supported in the expressions are discussed in *Expressions, functions, and operators*.

## 5.7 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 matching 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 Manticore 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, Manticore 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 <SPH_SORT_EXTENDED_mode>` 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).

Aggregate functions (AVG(), MIN(), MAX(), SUM()) are supported through *SetSelect()* API call when using GROUP BY.

## 5.8 Distributed searching

To scale well, Manticore 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 across search nodes and then process it in parallel.

Partitioning is done manually. You should

- setup several instances of Manticore 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 results 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.

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 querying 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.

## 5.9 Query log formats

Two query log formats are supported. Plain text format is still the default one. However, while it might be more convenient for manual monitoring and review, but hard to replay for benchmarks, it only logs *search* queries but not the other types of requests, does not always contain the complete search query data, etc. The default text format is also harder (and sometimes impossible) to replay for benchmarking purposes. The `sphinxql` format alleviates that. It aims to be complete and automatable, even though at the cost of brevity and readability.

### 5.9.1 Plain log format

By default, `searchd` logs all successfully executed search queries into a query log file. Here's an example:

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

This log format is as follows:

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

- real-time is a time measured just from start to finish of the query
- wall-time like real-time but not including waiting for agents and merging result sets time

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).

### 5.9.2 SphinxQL log format

This new log format introduced with the goals begin logging everything and then some, and in a format easy to automate (for instance, automatically replay). SphinxQL log format can either be enabled via the *query_log_format* directive in the configuration file, or switched back and forth on the fly with the *SET GLOBAL query_log_format=...* statement via SphinxQL. In the new format, the example from the previous section would look as follows. (Wrapped below for readability, but with just one query per line in the actual log.)

```
/* Fri Jun 29 21:17:58.609 2007 2011 conn 2 real 0.004 wall 0.004 found 35254 */
SELECT * FROM lj WHERE MATCH('test') OPTION ranker=proximity;

/* Fri Jun 29 21:20:34 2007.555 conn 3 real 0.024 wall 0.024 found 19886 */
SELECT * FROM lj WHERE MATCH('test') GROUP BY channel_id
OPTION ranker=proximity;
```

Note that **all** requests would be logged in this format, including those sent via SphinxAPI and SphinxSE, not just those sent via SphinxQL. Also note, that this kind of logging works only with plain log files and will not work if you use 'syslog' service for logging.

The features of SphinxQL log format compared to the default text one are as follows.

- All request types should be logged. (This is still work in progress.)
- Full statement data will be logged where possible.
- Errors and warnings are logged.
- The log should be automatically replayable via SphinxQL.
- Additional performance counters (currently, per-agent distributed query times) are logged.

Use sphinxql:compact_in to shorten your IN() clauses in log if you have too much values in it.

Every request (including both SphinxAPI and SphinxQL) request must result in exactly one log line. All request types, including INSERT, CALL SNIPPETS, etc will eventually get logged, though as of time of this writing, that is a work

in progress). Every log line must be a valid SphinxQL statement that reconstructs the full request, except if the logged request is too big and needs shortening for performance reasons. Additional messages, counters, etc can be logged in the comments section after the request.

## 5.10 MySQL protocol support and SphinxQL

Manticore 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 Manticore 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 Manticore now also supports a subset of SQL that we dubbed SphinxQL. It supports the standard querying all the index types with SELECT, modifying RT indexes with INSERT, REPLACE, and DELETE, and much more. Full SphinxQL reference is available in *SphinxQL reference*.

## 5.11 Multi-queries

Multi-queries, or query batches, let you send multiple queries to Manticore in one go (more formally, one network request).

Two API methods that implement multi-query mechanism are *AddQuery()* and *RunQueries()*. You can also run multiple queries with SphinxQL, see *Multi-statement queries*. (In fact, regular *Query()* call is internally implemented as a single AddQuery() call immediately followed by RunQueries() call.) AddQuery() captures the current state of all the query settings set by previous API calls, and memorizes the query. RunQueries() actually sends all the memorized queries, and returns multiple result sets. There are no restrictions on the queries at all, except just a sanity check on a number of queries in a single batch (see *max_batch_queries*).

Why use multi-queries? Generally, it all boils down to performance. First, by sending requests to `searchd` in a batch instead of one by one, you always save a bit by doing less network roundtrips. Second, and somewhat more important, sending queries in a batch enables `searchd` to perform certain internal optimizations. As new types of optimizations are being added over time, it generally makes sense to pack all the queries into batches where possible, so that simply upgrading Manticore to a new version would automatically enable new optimizations. In the case when there aren't any possible batch optimizations to apply, queries will be processed one by one internally.

Why (or rather when) not use multi-queries? Multi-queries requires all the queries in a batch to be independent, and sometimes they aren't. That is, sometimes query B is based on query A results, and so can only be set up after executing query A. For instance, you might want to display results from a secondary index if and only if there were no results found in a primary index. Or maybe just specify offset into 2nd result set based on the amount of matches in the 1st result set. In that case, you will have to use separate queries (or separate batches).

There are two major optimizations to be aware of: common query optimization and common subtree optimization.

**Common query optimization** means that `searchd` will identify all those queries in a batch where only the sorting and group-by settings differ, and *only perform searching once*. For instance, if a batch consists of 3 queries, all of them are for "ipod nano", but 1st query requests top-10 results sorted by price, 2nd query groups by vendor ID and requests top-5 vendors sorted by rating, and 3rd query requests max price, full-text search for "ipod nano" will only be performed once, and its results will be reused to build 3 different result sets.

So-called **faceted searching** is a particularly important case that benefits from this optimization. Indeed, faceted searching can be implemented by running a number of queries, one to retrieve search results themselves, and a few other ones with same full-text query but different group-by settings to retrieve all the required groups of results (top-3 authors, top-5 vendors, etc). And as long as full-text query and filtering settings stay the same, common query optimization will trigger, and greatly improve performance.

**Common subtree optimization** is even more interesting. It lets `searchd` exploit similarities between batched full-text queries. It identifies common full-text query parts (subtrees) in all queries, and caches them between queries. For instance, look at the following query batch:

```
donald trump president
donald trump barack obama john mccain
donald trump speech
```

There's a common two-word part ("donald trump") that can be computed only once, then cached and shared across the queries. And common subtree optimization does just that. Per-query cache size is strictly controlled by *subtree_docs_cache* and *subtree_hits_cache* directives (so that caching *all* sixteen gazillions of documents that match "i am" does not exhaust the RAM and instantly kill your server).

Here's a code sample (in PHP) that fire the same query in 3 different sorting modes:

```
require ( "sphinxapi.php" );
$cl = new ManticoreClient ();
$cl->SetMatchMode ( SPH_MATCH_EXTENDED );

$cl->SetSortMode ( SPH_SORT_RELEVANCE );
$cl->AddQuery ( "the", "lj" );
$cl->SetSortMode ( SPH_SORT_EXTENDED, "published desc" );
$cl->AddQuery ( "the", "lj" );
$cl->SetSortMode ( SPH_SORT_EXTENDED, "published asc" );
$cl->AddQuery ( "the", "lj" );
$res = $cl->RunQueries();
```

How to tell whether the queries in the batch were actually optimized? If they were, respective query log will have a "multiplier" field that specifies how many queries were processed together:

```
[Sun Jul 12 15:18:17.000 2009] 0.040 sec x3 [ext/0/rel 747541 (0,20)] [lj] the
[Sun Jul 12 15:18:17.000 2009] 0.040 sec x3 [ext/0/ext 747541 (0,20)] [lj] the
[Sun Jul 12 15:18:17.000 2009] 0.040 sec x3 [ext/0/ext 747541 (0,20)] [lj] the
```

Note the "x3" field. It means that this query was optimized and processed in a sub-batch of 3 queries. For reference, this is how the regular log would look like if the queries were not batched:

```
[Sun Jul 12 15:18:17.062 2009] 0.059 sec [ext/0/rel 747541 (0,20)] [lj] the
[Sun Jul 12 15:18:17.156 2009] 0.091 sec [ext/0/ext 747541 (0,20)] [lj] the
[Sun Jul 12 15:18:17.250 2009] 0.092 sec [ext/0/ext 747541 (0,20)] [lj] the
```

Note how per-query time in multi-query case was improved by a factor of 1.5x to 2.3x, depending on a particular sorting mode. In fact, for both common query and common subtree optimizations, there were reports of 3x and even more improvements, and that's from production instances, not just synthetic tests.

## 5.12 Collations

Collations essentially affect the string attribute comparisons. They specify both the character set encoding and the strategy that Manticore uses to compare strings when doing ORDER BY or GROUP BY with a string attribute involved.

String attributes are stored as is when indexing, and no character set or language information is attached to them. That's okay as long as Manticore only needs to store and return the strings to the calling application verbatim. But when you ask Manticore to sort by a string value, that request immediately becomes quite ambiguous.

First, single-byte (ASCII, or ISO-8859-1, or Windows-1251) strings need to be processed differently that the UTF-8 ones that may encode every character with a variable number of bytes. So we need to know what is the character set type to interpret the raw bytes as meaningful characters properly.

Second, we additionally need to know the language-specific string sorting rules. For instance, when sorting according to US rules in en_US locale, the accented character 'ï' (small letter i with diaeresis) should be placed somewhere after 'z'. However, when sorting with French rules and fr_FR locale in mind, it should be placed between 'i' and 'j'. And some other set of rules might choose to ignore accents at all, allowing 'ï' and 'i' to be mixed arbitrarily.

Third, but not least, we might need case-sensitive sorting in some scenarios and case-insensitive sorting in some others.

Collations combine all of the above: the character set, the language rules, and the case sensitivity. Manticore currently provides the following four collations.

1. `libc_ci`
2. `libc_cs`
3. `utf8_general_ci`
4. `binary`

The first two collations rely on several standard C library (libc) calls and can thus support any locale that is installed on your system. They provide case-insensitive (_ci) and case-sensitive (_cs) comparisons respectively. By default they will use C locale, effectively resorting to bytewise comparisons. To change that, you need to specify a different available locale using *collation_libc_locale* directive. The list of locales available on your system can usually be obtained with the `locale` command:

```
$ locale -a
C
en_AG
en_AU.utf8
```

(continues on next page)

(continued from previous page)

```
en_BW.utf8
en_CA.utf8
en_DK.utf8
en_GB.utf8
en_HK.utf8
en_IE.utf8
en_IN
en_NG
en_NZ.utf8
en_PH.utf8
en_SG.utf8
en_US.utf8
en_ZA.utf8
en_ZW.utf8
es_ES
fr_FR
POSIX
ru_RU.utf8
ru_UA.utf8
```

The specific list of the system locales may vary. Consult your OS documentation to install additional needed locales.

`utf8_general_ci` and `binary` locales are built-in into Manticore. The first one is a generic collation for UTF-8 data (without any so-called language tailoring); it should behave similar to `utf8_general_ci` collation in MySQL. The second one is a simple bytewise comparison.

Collation can be overridden via SphinxQL on a per-session basis using `SET collation_connection` statement. All subsequent SphinxQL queries will use this collation. SphinxAPI and SphinxSE queries will use the server default collation, as specified in *collation_server* configuration directive. Manticore currently defaults to `libc_ci` collation.

Collations should affect all string attribute comparisons, including those within ORDER BY and GROUP BY, so differently ordered or grouped results can be returned depending on the collation chosen. Note that collations don't affect full-text searching, for that use *charset_table*.

## 5.13 Query cache

Query cache stores a compressed result set in memory, and then reuses it for subsequent queries where possible. You can configure it using the following directives:

- *qcache_max_bytes*, a limit on the RAM use for cached queries storage. Defaults to 16 MB. Setting qcache_max_bytes to 0 completely disables the query cache.
- *qcache_thresh_msec*, the minimum wall query time to cache. Queries that completed faster than this will *not* be cached. Defaults to 3000 msec, or 3 seconds.
- *qcache_ttl_sec*, cached entry TTL, or time to live. Queries will stay cached for this much. Defaults to 60 seconds, or 1 minute.

These settings can be changed on the fly using the *SET GLOBAL* statement:

```
mysql> SET GLOBAL qcache_max_bytes=128000000;
```

These changes are applied immediately, and the cached result sets that no longer satisfy the constraints are immediately discarded. When reducing the cache size on the fly, MRU (most recently used) result sets win.

Query cache works as follows. When it's enabled, *every* full-text search result gets *completely* stored in memory. That happens after full-text matching, filtering, and ranking, so basically we store `total_found` {docid,weight} pairs.

Compressed matches can consume anywhere from 2 bytes to 12 bytes per match on average, mostly depending on the deltas between the subsequent docids. Once the query completes, we check the wall time and size thresholds, and either save that compressed result set for reuse, or discard it.

Note how the query cache impact on RAM is thus *not* limited by `qcache_max_bytes`! If you run, say, 10 concurrent queries, each of them matching upto 1M matches (after filters), then the peak temporary RAM use will be in the 40 MB to 240 MB range, even if in the end the queries are quick enough and do not get cached.

Queries can then use cache when the index, the full-text query (ie. MATCH() contents), and the ranker are all a match, and filters are compatible. Meaning:

- The full-text part within MATCH() must be a bytewise match. Add a single extra space, and that is now a different query where the query cache is concerned.
- The ranker (and its parameters if any, for user-defined rankers) must be a bytewise match.
- The filters must be a superset of the original filters. That is, you can add extra filters and still hit the cache. (In this case, the extra filters will be applied to the cached result.) But if you remove one, that will be a new query again.

Cache entries expire with TTL, and also get invalidated on index rotation, or on TRUNCATE, or on ATTACH. Note that at the moment entries are **not** invalidated on arbitrary RT index writes! So a cached query might be returning older results for the duration of its TTL.

Current cache status can be inspected with in *SHOW STATUS* through the `qcache_XXX` variables:

```
mysql> SHOW STATUS LIKE 'qcache%';
+----------------------+----------+
| Counter              | Value    |
+----------------------+----------+
| qcache_max_bytes     | 16777216 |
| qcache_thresh_msec   | 3000     |
| qcache_ttl_sec       | 60       |
| qcache_cached_queries | 0        |
| qcache_used_bytes    | 0        |
| qcache_hits          | 0        |
+----------------------+----------+
6 rows in set (0.00 sec)
```

## 5.14 MySQL storage engine (SphinxSE)

### 5.14.1 SphinxSE overview

SphinxSE is MySQL storage engine which can be compiled into MySQL server 5.x using its pluggable architecture. 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 Manticore;
- allowing Manticore use with programming languages for which native APIs are not available yet;
- optimizations when additional Manticore result set processing on MySQL side is required (eg. JOINs with original document tables, additional MySQL-side filtering, etc).

### 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 http://dev.mysql.com Web site.

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

### Compiling MySQL 5.0.x with SphinxSE

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 Manticore 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 Manticore engine:

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

5. build and install MySQL:

```
$ make
$ make install
```

### Compiling MySQL 5.1.x with SphinxSE

1. in MySQL sources directory, create `storage/sphinx` directory in and copy all files in `mysqlse` directory from Manticore 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 Manticore engine:

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

4. build and install MySQL:

```
$ make
$ make install
```

### Checking SphinxSE installation

To check whether SphinxSE has been successfully compiled into MySQL, launch newly built servers, run mysql client and issue `SHOW ENGINES` query. You should see a list of all available engines. Manticore 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      | Manticore storage engine                                       ␣
↪   |
  ...
+------------+----------+----------------------------------------------------------------
↪+
13 rows in set (0.00 sec)
```

### 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 Manticore result set by name, so their names must match attribute names specified in `sphinx.conf`. If there's no such attribute name in Manticore 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:

```
mysql> 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 Manticore 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 (integer or bigint) Manticore attribute name, and min and max values 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;';
```

- floatrange, !floatrange - comma-separated (floating point) Manticore attribute name, and min and max values to match:

```
# filter by a float size
... WHERE query='test;floatrange=size,2,3;';

# pick all results within 1000 meter from geoanchor
... WHERE query='test;floatrange=@geodist,0,1000;';
```

- maxmatches - per-query max matches value, as in max_matches parameter to *SetLimits()* API call:

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

- cutoff - maximum allowed matches, as in cutoff parameter to *SetLimits()* API call:

```
... WHERE query='test;cutoff=10000;';
```

- maxquerytime - maximum allowed query time (in milliseconds), as in *SetMaxQueryTime()* API call:

```
... WHERE query='test;maxquerytime=1000;';
```

- groupby - group-by function and attribute, corresponding to *SetGroupBy()* API call:

```
... 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;';
```

- distinct - an attribute to compute COUNT(DISTINCT) for when doing group-by, as in *SetGroupDistinct()* API call:

```
... WHERE query='test;groupby=attr:country_id;distinct=site_id';
```

- 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;';
```

- fieldweights - comma-separated list of per-field weights that can be used by the ranker:

```
... WHERE query='test;fieldweights=title,10,abstract,3,content,1;';
```

- comment - a string to mark this query in query log (mapping to $comment parameter in *Query()* API call):

```
... WHERE query='test;comment=marker001;';
```

- select - a string with expressions to compute (mapping to *SetSelect()* API call):

```
... WHERE query='test;select=2*a+3*** as myexpr;';
```

- host, port - remote `searchd` host name and TCP port, respectively:

```
... WHERE query='test;host=sphinx-test.loc;port=7312;';
```

- ranker - a ranking function to use with "extended" matching mode, as in *SetRankingMode()* API call (the only mode that supports full query syntax). Known values are "proximity_bm25", "bm25", "none", "wordcount", "proximity", "matchany", "fieldmask", "sph04", "expr:EXPRESSION" syntax to support expression-

based ranker (where EXPRESSION should be replaced with your specific ranking formula), and "export:EXPRESSION":

```
... WHERE query='test;mode=extended;ranker=bm25;';
... WHERE query='test;mode=extended;ranker=expr:sum(lcs);';
```

**The "export" ranker works exactly like ranker=expr, but it stores the** per-document factor values, while ranker=expr discards them after computing the final WEIGHT() value. Note that ranker=export is meant to be used but rarely, only to train a ML (machine learning) function or to define your own ranking function by hand, and never in actual production. When using this ranker, you'll probably want to examine the output of the RANKFACTORS() function that produces a string with all the field level factors for each document.

```
        SELECT *, WEIGHT(), RANKFACTORS()
            FROM myindex
            WHERE MATCH('dog')
            OPTION ranker=export('100*bm25')

would produce something like
```

```
*************************** 1\. row ***************************
           id: 555617
    published: 1110067331
   channel_id: 1059819
        title: 7
      content: 428
     weight(): 69900
rankfactors(): bm25=699, bm25a=0.666478, field_mask=2,
doc_word_count=1, field1=(lcs=1, hit_count=4, word_count=1,
tf_idf=1.038127, min_idf=0.259532, max_idf=0.259532, sum_idf=0.259532,
min_hit_pos=120, min_best_span_pos=120, exact_hit=0,
max_window_hits=1), word1=(tf=4, idf=0.259532)
*************************** 2\. row ***************************
           id: 555313
    published: 1108438365
   channel_id: 1058561
        title: 8
      content: 249
     weight(): 68500
rankfactors(): bm25=685, bm25a=0.675213, field_mask=3,
doc_word_count=1, field0=(lcs=1, hit_count=1, word_count=1,
tf_idf=0.259532, min_idf=0.259532, max_idf=0.259532, sum_idf=0.259532,
min_hit_pos=8, min_best_span_pos=8, exact_hit=0, max_window_hits=1),
field1=(lcs=1, hit_count=2, word_count=1, tf_idf=0.519063,
min_idf=0.259532, max_idf=0.259532, sum_idf=0.259532, min_hit_pos=36,
min_best_span_pos=36, exact_hit=0, max_window_hits=1), word1=(tf=3,
idf=0.259532)
```

- geoanchor - geodistance anchor, as in *SetGeoAnchor()* API call. Takes 4 parameters which are latitude and longitude attribute names, and anchor point coordinates respectively:

```
... WHERE query='test;geoanchor=latattr,lonattr,0.123,0.456';
```

One **very important** note that it is **much** more efficient to allow Manticore 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, Manticore 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.

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)
```

### 5.14.2 Building snippets (excerpts) via MySQL

SphinxSE also includes a UDF function that lets you create snippets through MySQL. The functionality is fully similar to *BuildExcerprts* API call but accessible 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: `&#039; value&#039; AS option_name`. For a list of supported options, refer to *BuildExcerprts()* API call. The only UDF-specific additional option is named `&#039;sphinx&#039;` 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,
    '[**]' AS before_match, '[/**]' 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;
```

## 5.15 Percolate query

**Note:** This is a new feature, not production ready yet, just for testing purposes mostly for now. Changes will occur in future updates.

The percolate query is used to match documents against queries stored in a index. It is also called "search in reverse" as it works opposite to a regular search where documents are stored in an index and queries are issued against the index.

Queries are stored in a special RealTime index and they can be added, deleted and listed using INSERT/DELETE/SELECT statements similar way as it's done for a regular index.

Checking if a document matches any of the predefined criterias (queries) can be done with the `CALL PQ` function, which returns a list of the matched queries. Note that it does not add documents to the percolate index. You need to use another index (regular or RealTime) in which you will insert documents to perform regular searches.

### 5.15.1 Tags

A percolate query can have `tags`. `tags` can be set for the query with `INSERT` statement. Later on a user might list queries with specific `tags` with `SELECT` statement or delete query(es) with `DELETE` statement.

### 5.15.2 Filters

A percolate query can have `filters`. `filters` are set for the query with `INSERT` statement. Documents can be then filtered according to the `filters` with `CALL PQ` statement.

### 5.15.3 Index

A percolate query works only for `percolate` index *type*. Its configuration is similar to *Real-time index*, however the declaration of fields and attributes can be omitted, in this case the index is created with default field `text` and attribute `gid`.

```
index pq
{
    type = percolate
    path = path/index_name
    min_infix_len    = 4
}
```

### 5.15.4 INSERT

To store a query the `INSERT` statement looks like

```
INSERT INTO index_name (query[, tags, filters, id]) VALUES ( query_terms, tags_list,␣
↪filters, query_id );
INSERT INTO index_name (query, tags, filters) VALUES ( 'full text query terms', 'tags
↪', 'filters' );
INSERT INTO index_name (query) VALUES ( 'full text query terms');
INSERT INTO index_name VALUES ( 'full text query terms', 'tags');
INSERT INTO index_name VALUES ( 'full text query terms');
```

where `tags` and `filters` and `id` are optional fields. In case no schema declared for the `INSERT` statement te first field will be full-text `query` and the optional second field will be `tags`.

`filters` is a string and has the same format as `SphinxQL` *WHERE* clause.

To replace existed query, `REPLACE` statement with query `id` field should be used.

### 5.15.5 CALL PQ

To search for queries matching a document(s) the `CALL PQ` statement is used which looks like

```
CALL PQ ('index_name', 'single document', 0 as docs, 0 as docs_json, 0 as verbose);
CALL PQ ('index_name', ('multiple documents', 'go this way'), 0 as docs_json );
```

The document in `CALL PQ` can be `JSON` encoded string or raw string. Fields and attributes mapping are allowed for `JSON` documents only.

```
CALL PQ ('pq', (
'{"title":"header text", "body":"post context", "timestamp":11 }',
'{"title":"short post", "counter":7 }'
) );
```

`CALL PQ` can have multiple options set as `option_name`.

Here are default values for the options:

- docs_json - 1 (enabled), to treat document(s) as `JSON` encoded string or raw string otherwise
- docs - 0 (disabled), to provide per query documents matched at result set
- verbose - 0 (disabled), to provide extended info on matching at *SHOW META*
- query - 0 (disabled), to provide all query fields stored, such as query, tags, filters

`CALL PQ` performance is affected by *dist_threads*.

### 5.15.6 List stored queries

To list stored queries in index the `SELECT` statement looks like

```
SELECT * FROM index_name;
SELECT * FROM index_name WHERE tags='tags list';
SELECT * FROM index_name WHERE uid IN (11,35,101);
```

In case `tags` provided matching queries will be shown if any `tags` from the `SELECT` statement match tags in the stored query. In case `uid` provided range or value list filter will be used to filter out stored queries.

The `SELECT` supports `count(*)` and `count(*) alias` to get number of of percolate queries. Any values are just ignored there however `count(*)` should provide the total amount of queries stored.

```
mysql> select count(*) c from pq;
+------+
| c    |
+------+
|    3 |
+------+
```

The `SELECT` supports `LIMIT` clause to narrow down the number of percolate queries.

```
SELECT * FROM index_name LIMIT 5;
SELECT * FROM index_name LIMIT 1300, 45;
```

### 5.15.7 Delete query

To delete a stored percolate query(es) in index the `DELETE` statement looks like

```
DELETE FROM index_name WHERE id=1;
DELETE FROM index_name WHERE tags='tags list';
```

In case `tags` provided the query will be deleted if any `tags` from the `DELETE` statement match any of its tags.

To delete all stored query(es) in index there is `TRUNCATE` statement looks like

```
TRUNCATE RTINDEX index_name;
```

### 5.15.8 Meta

Meta information is kept for documents on "CALL PQ" and can be retrieved with `SHOW META` call.

`SHOW META` output after `CALL PQ` looks like

```
+-------------------------+-----------+
| Name                    | Value     |
+-------------------------+-----------+
| Total                   | 0.010 sec |
| Queries matched         | 950       |
| Document matches        | 1500      |
| Total queries stored    | 1000      |
```

(continues on next page)

(continued from previous page)

```
| Term only queries       | 998       |
+-------------------------+-----------+
```

With entries:

- Total - total time spent for matching the document(s)
- Queries matched - how many stored queries match the document(s)
- Document matches - how many times the documents match the queries stored in the index
- Total queries stored - how many queries are stored in the index at all
- Term only queries - how many queries in the index have terms. The rest of the queries have extended query syntax

### 5.15.9 Reconfigure

As well as for RealTime indexes `ALTER RECONFIGURE` command is also supported for percolate query index. It allows to reconfigure `percolate` index on the fly without deleting and repopulating the index with queries back.

```
mysql> desc pq1;
+-------+--------+
| Field | Type   |
+-------+--------+
| id    | bigint |
| text  | field  |
| body  | field  |
| k     | uint   |
+-------+--------+

mysql> select * from pq1;
+------+-------+------+-------------+
| UID  | Query | Tags | Filters     |
+------+-------+------+-------------+
|    1 | test  |      | k=4         |
|    2 | test  |      | k IN (4,6)  |
|    3 | test  |      |             |
+------+-------+------+-------------+
```

Add *JSON* attribute to the index config `rt_attr_json = json_data`, then issue `ALTER RECONFIGURE`

```
mysql> desc pq1;
+-----------+--------+
| Field     | Type   |
+-----------+--------+
| id        | bigint |
| text      | field  |
| body      | field  |
| k         | uint   |
| json_data | json   |
+-----------+--------+
```

# CHAPTER 6

# Extending

## 6.1 UDFs (User Defined Functions)

Our expression engine can be extended with user defined functions, or UDFs for short, like this:

```
SELECT id, attr1, myudf(attr2, attr3+attr4) ...
```

You can load and unload UDFs dynamically into `searchd` without having to restart the daemon, and used them in expressions when searching, ranking, etc. Quick summary of the UDF features is as follows.

- UDFs can take integer (both 32-bit and 64-bit), float, string, MVA, or PACKEDFACTORS() arguments.
- UDFs can return integer, float, or string values.
- UDFs can check the argument number, types, and names during the query setup phase, and raise errors.
- Aggregation UDFs are not yet supported (but might be in the future).

UDFs have a wide variety of uses, for instance:

- adding custom mathematical or string functions;
- accessing the database or files from within Manticore;
- implementing complex ranking functions.

UDFs reside in the external dynamic libraries (.so files on UNIX and .dll on Windows systems). Library files need to reside in a trusted folder specified by *plugin_dir* directive, for obvious security reasons: securing a single folder is easy; letting anyone install arbitrary code into `searchd` is a risk. You can load and unload them dynamically into searchd with *CREATE FUNCTION* and *DROP FUNCTION* SphinxQL statements respectively. Also, you can seamlessly reload UDFs (and other plugins) with *RELOAD PLUGINS* statement. Manticore keeps track of the currently loaded functions, that is, every time you create or drop an UDF, `searchd` writes its state to the *sphinxql_state* file as a plain good old SQL script.

Once you successfully load an UDF, you can use it in your SELECT or other statements just as well as any of the builtin functions:

```
SELECT id, MYCUSTOMFUNC(groupid, authorname), ... FROM myindex
```

Multiple UDFs (and other plugins) may reside in a single library. That library will only be loaded once. It gets automatically unloaded once all the UDFs and plugins from it are dropped.

In theory you can write an UDF in any language as long as its compiler is able to import standard C header, and emit standard dynamic libraries with properly exported functions. Of course, the path of least resistance is to write in either C++ or plain C. We provide an example UDF library written in plain C and implementing several functions (demonstrating a few different techniques) along with our source code, see src/udfexample.c. That example includes src/sphinxudf.h header file definitions of a few UDF related structures and types. For most UDFs and plugins, a mere `#include "sphinxudf.h"`, like in the example, should be completely sufficient, too. However, if you're writing a ranking function and need to access the ranking signals (factors) data from within the UDF, you will also need to compile and link with `src/sphinxudf.c` (also available in our source code), because the *implementations* of the fuctions that let you access the signal data from within the UDF reside in that file.

Both `sphinxudf.h` header and `sphinxudf.c` are standalone. So you can copy around those files only; they do not depend on any other bits of Manticore source code.

Within your UDF, you **must** implement and export only a couple functions, literally. First, for UDF interface version control, you **must** define a function `int LIBRARYNAME_ver()`, where LIBRARYNAME is the name of your library file, and you must return `SPH_UDF_VERSION` (a value defined in `sphinxudf.h`) from it. Here's an example.

```
#include <sphinxudf.h>

// our library will be called udfexample.so, thus, so it must define
// a version function named udfexample_ver()
int udfexample_ver()
{
    return SPH_UDF_VERSION;
}
```

That protects you from accidentally loading a library with a mismatching UDF interface version into a newer or older `searchd`. Second, yout **must** implement the actual function, too. `sphinx_int64_t testfunc ( SPH_UDF_INIT * init, SPH_UDF_ARGS * args, char * error_flag ) { return 123; }`

UDF function names in SphinxQL are case insensitive. However, the respective C function names are not, they need to be all *lower-case*, or the UDF will not load. More importantly, it is vital that a) the calling convention is C (aka __cdecl), b) arguments list matches the plugin system expectations exactly, and c) the return type matches the one you specify in `CREATE FUNCTION`. Unfortunately, there is no (easy) way for us to check for those mistakes when loading the function, and they could crash the server and/or result in unexpected results. Last but not least, all the C functions you implement need to be thread-safe.

The first argument, a pointer to SPH_UDF_INIT structure, is essentially a pointer to our function state. It is option. In the example just above the function is stateless, it simply returns 123 every time it gets called. So we do not have to define an initialization function, and we can simply ignore that argument.

The second argument, a pointer to SPH_UDF_ARGS, is the most important one. All the actual call arguments are passed to your UDF via this structure; it contians the call argument count, names, types, etc. So whether your function gets called like `SELECT id, testfunc(1)` or like `SELECT id, testfunc(&#039;abc&#039; , 1000*id+gid, WEIGHT())` or anyhow else, it will receive the very same SPH_UDF_ARGS structure in all of these cases. However, the data passed in the `args` structure will be different. In the first example `args->arg_count` will be set to 1, in the second example it will be set to 3, `args->arg_types` array will contain different type data, and so on.

Finally, the third argument is an error flag. UDF can raise it to indicate that some kinda of an internal error happened, the UDF can not continue, and the query should terminate early. You should **not** use this for argument type checks or for any other error reporting that is likely to happen during normal use. This flag is designed to report sudden critical runtime errors, such as running out of memory.

If we wanted to, say, allocate temporary storage for our function to use, or check upfront whether the arguments are of the supported types, then we would need to add two more functions, with UDF initialization and deinitialization, respectively.

```
int testfunc_init ( SPH_UDF_INIT * init, SPH_UDF_ARGS * args,
    char * error_message )
{
    // allocate and initialize a little bit of temporary storage
    init->func_data = malloc ( sizeof(int) );
    *(int*)init->func_data = 123;

    // return a success code
    return 0;
}

void testfunc_deinit ( SPH_UDF_INIT * init )
{
    // free up our temporary storage
    free ( init->func_data );
}
```

Note how `testfunc_init()` also receives the call arguments structure. By the time it is called it does not receive any actual values, so the `args->arg_values` will be NULL. But the argument names and types are known and will be passed. You can check them in the initialization function and return an error if they are of an unsupported type.

UDFs can receive arguments of pretty much any valid internal Manticore type. Refer to `sphinx_udf_argtype` enumeration in `sphinxudf.h` for a full list. Most of the types map straightforwardly to the respective C types. The most notable exception is the SPH_UDF_TYPE_FACTORS argument type. You get that type by calling your UDF with a *PACKEDFACTOR()* argument. It's data is a binary blob in a certain internal format, and to extract individual ranking signals from that blob, you need to use either of the two `sphinx_factors_XXX()` or `sphinx_get_YYY_factor()` families of functions. The first family consists of just 3 functions, `sphinx_factors_init()` that initializes the unpacked SPH_UDF_FACTORS structure, `sphinx_factors_unpack()` that unpacks a binary blob into it, and `sphinx_factors_deinit()` that cleans up an deallocates the SPH_UDF_FACTORS. So you need to call init() and unpack(), then you can use the SPH_UDF_FACTORS fields, and then you need to cleanup with deinit(). That is simple, but results in a bunch of memory allocations per each processed document, and might be slow. The other interface, consisting of a bunch of `sphinx_get_YYY_factor()` functions, is a little more wordy to use, but accesses the blob data directly and guarantees that there will be zero allocations. So for top-notch ranking UDF performance, you want to use that one.

As for the return types, UDFs can currently return a signle INTEGER, BIGINT, FLOAT, or STRING value. The C function return type should be sphinx_int64_t, sphinx_int64_t, double, or char* respectively. In the last case you **must** use `args->fn_malloc` function to allocate the returned string values. Internally in your UDF you can use whatever you want, so the `testfunc_init()` example above is correct code even though it uses malloc() directly: you manage that pointer yourself, it gets freed up using a matching free() call, and all is well. However, the returned strings values are managed by Manticore and we have our own allocator, so for the return values specifically, you need to use it too.

Depending on how your UDFs are used in the query, the main function call (`testfunc()` in our example) might be called in a rather different volume and order. Specifically,

- UDFs referenced in WHERE, ORDER BY, or GROUP BY clauses must and will be evaluated for every matched document. They will be called in the natural matching order.
- without subselects, UDFs that can be evaluated at the very last stage over the final result set will be evaluated that way, but before applying the LIMIT clause. They will be called in the result set order.
- with subselects, such UDFs will also be evaluated after applying the inner LIMIT clause.

The calling sequence of the other functions is fixed, though. Namely,

- `testfunc_init()` is called once when initializing the query. It can return a non-zero code to indicate a failure; in that case query will be terminated, and the error message from the `error_message` buffer will be returned.
- `testfunc()` is called for every eligible row (see above), whenever Manticore needs to compute the UDF value. It can also indicate an (internal) failure error by writing a non-zero byte value to `error_flag`. In that case, it is guaranteed that will no more be called for subsequent rows, and a default return value of 0 will be substituted. Manticore might or might not choose to terminate such queries early, neither behavior is currently guaranteed.
- `testfunc_deinit()` is called once when the query processing (in a given index shard) ends.

We do not yet support aggregation functions. In other words, your UDFs will be called for just a single document at a time and are expected to return some value for that document. Writing a function that can compute an aggregate value like AVG() over the entire group of documents that share the same GROUP BY key is not yet possible. However, you can use UDFs within the builtin aggregate functions: that is, even though MYCUSTOMAVG() is not supported yet, AVG(MYCUSTOMFUNC()) should work alright!

UDFs are local. In order to use them on a cluster, you have to put the same library on all its nodes and run CREATEs on all the nodes too. This might change in the future versions.

## 6.2 Plugins

Here's the complete plugin type list.

- UDF plugins;
- ranker plugins;
- indexing-time token filter plugins;
- query-time token filter plugins.

This section discusses writing and managing plugins in general; things specific to writing this or that type of a plugin are then discussed in their respective subsections.

So, how do you write and use a plugin? Four-line crash course goes as follows:

- create a dynamic library (either .so or.dll), most likely in C or C++;
- load that plugin into searchd using *CREATE PLUGIN*;
- invoke it using the plugin specific calls (typically using this or that OPTION).
- to unload or reload a plugin use *DROP PLUGIN* and *RELOAD PLUGINS* respectively.

Note that while UDFs are first-class plugins they are nevertheless installed using a separate *CREATE FUNCTION* statement. It lets you specify the return type neatly so there was especially little reason to ruin backwards compatibility *and* change the syntax.

Dynamic plugins are supported in threads and thread_pool workers. Multiple plugins (and/or UDFs) may reside in a single library file. So you might choose to either put all your project-specific plugins in a single common uber-library; or you might choose to have a separate library for every UDF and plugin; that is up to you.

Just as with UDFs, you want to include `src/sphinxudf.h` header file. At the very least, you will need the SPH_UDF_VERSION constant to implement a proper version function. Depending on the specific plugin type, you might or might not need to link your plugin with `src/sphinxudf.c`. However, all the functions implemented in `sphinxudf.c` are about unpacking the PACKEDFACTORS() blob, and no plugin types are exposed to that kind of data. So currently, you would never need to link with the C-file, just the header would be sufficient. (In fact, if you copy over the UDF version number, then for some of the plugin types you would not even need the header file.)

Formally, plugins are just sets of C functions that follow a certain naming parttern. You are typically required to define just one key function that does the most important work, but you may define a bunch of other functions, too. For example, to implement a ranker called "myrank", you must define `myrank_finalize()` function that actually returns the rank value, however, you might also define `myrank_init()`, `myrank_update()`, and `myrank_deinit()` functions. Specific sets of well-known suffixes and the call arguments do differ based on the plugin type, but _init() and _deinit() are generic, every plugin has those. Protip: for a quick reference on the known suffixes and their argument types, refer to `sphinxplugin.h`, we define the call prototoypes in the very beginning of that file.

Despite having the public interface defined in ye good olde good pure C, our plugins essentially follow the *object-oriented model*. Indeed, every `_init()` function receives a `void ** userdata` out-parameter. And the pointer value that you store at `(*userdata)` location is then be passed as a 1st argument to all the other plugin functions. So you can think of a plugin as *class* that gets instantiated every time an object of that class is needed to handle a request: the `userdata` pointer would be its `this` pointer; the functions would be its methods, and the `_init()` and `_deinit()` functions would be the constructor and destructor respectively.

Why this (minor) OOP-in-C complication? Well, plugins run in a multi-threaded environment, and some of them have to be stateful. You can't keep that state in a global variable in your plugin. So we have to pass around a userdata parameter anyway to let you keep that state. And that naturally brings us to the OOP model. And if you've got a simple, stateless plugin, the interface lets you omit the `_init()` and `_deinit()` and whatever other functions just as well.

To summarize, here goes the simplest complete ranker plugin, in just 3 lines of C code.

```
// gcc -fPIC -shared -o myrank.so myrank.c
#include "sphinxudf.h"
int myrank_ver() { return SPH_UDF_VERSION; }
int myrank_finalize(void *u, int w) { return 123; }
```

And this is how you use it:

```
mysql> CREATE PLUGIN myrank TYPE 'ranker' SONAME 'myrank.dll';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT id, weight() FROM test1 WHERE MATCH('test')
    -> OPTION ranker=myrank('');
+------+----------+
| id   | weight() |
+------+----------+
|    1 |      123 |
|    2 |      123 |
+------+----------+
2 rows in set (0.01 sec)
```

## 6.3 Ranker plugins

Ranker plugins let you implement a custom ranker that receives all the occurrences of the keywords matched in the document, and computes a WEIGHT() value. They can be called as follows:

```
SELECT id, attr1 FROM test WHERE match('hello')
OPTION ranker=myranker('option1=1');
```

The call workflow is as follows:

1. `XXX_init()` gets called once per query per index, in the very beginning. A few query-wide options are passed to it through a `SPH_RANKER_INIT` structure, including the user options strings (in the example just above, "option1=1" is that string).

2. `XXX_update()` gets called multiple times per matched document, with every matched keyword occurrence passed as its parameter, a `SPH_RANKER_HIT` structure. The occurrences within each document are guaranteed to be passed in the order of ascending `hit->hit_pos` values.
3. `XXX_finalize()` gets called once per matched document, once there are no more keyword occurrences. It must return the WEIGHT() value. This is the only mandatory function.
4. `XXX_deinit()` gets called once per query, in the very end.

CHAPTER 7

# Command line tools reference

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

## 7.1 `indexer` command reference

`indexer` is the first of the two principal tools as part of Manticore. 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 exit codes are as follows:

- 0, everything went ok
- 1, there was a problem while indexing (and if –rotate was specified, it was skipped)
- 2, indexing went ok, but –rotate attempt failed

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 Manticore 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.

- `--merge-killlists` (and its shorter alias `--merge-klists`) changes the way kill lists are processed when merging indexes. By default, both kill lists get discarded after a merge. That supports the most typical main+delta merge scenario. With this option enabled, however, kill lists from both indexes get concatenated and stored into the destination index. Note that a source (delta) index kill list will be used to suppress rows from a destination (main) index at all times.

- `--keep-attrs` allows to reuse existing attributes on reindexing. Whenever the index is rebuilt, each new document id is checked for presence in the "old" index, and if it already exists, its attributes are transferred to the "new" index; if not found, attributes from the new index are used. If the user has updated attributes in the index, but not in the actual source used for the index, all updates will be lost when reindexing; using –keep-attrs enables saving the updated attribute values from the previous index. It is possible to specify a path for index files to used instead of reference path from config:

```
indexer myindex --keep-attrs=/path/to/index/files
```

- `--keep-attrs-names=<attributes list>` allows to specify attributes to reuse from existing index on reindexing. By default all attributes from existed index reused at new "index"

```
indexer myindex --keep-attrs=/path/to/index/files --keep-attrs-names=update,state
```

- `--dump-rows <FILE>` dumps rows fetched by SQL source(s) into the specified file, in a MySQL compatible syntax. Resulting dumps are the exact representation of data as received by `indexer` and help to repeat indexing-time issues.

- `--verbose` guarantees that every row that caused problems indexing (duplicate, zero, or missing document ID; or file field IO issues; etc) will be reported. By default, this option is off, and problem summaries may be reported instead.
- `--sighup-each` is useful when you are rebuilding many big indexes, and want each one rotated into `searchd` as soon as possible. With `--sighup-each`, `indexer` will send a SIGHUP signal to searchd after successfully completing the work on each index. (The default behavior is to send a single SIGHUP after all the indexes were built.)
- `--nohup` is useful when you want to check your index with indextool before actually rotating it. indexer won't send SIGHUP if this option is on.
- `--print-queries` prints out SQL queries that `indexer` sends to the database, along with SQL connection and disconnection events. That is useful to diagnose and fix problems with SQL sources.
- `--help` (`-h` for short) lists all of the parameters that can be called in your particular build of `indexer`.
- `-v` show version information of your particular build of `indexer`.

## 7.2 `indextool` command reference

`indextool` is one of the helper tools within the Manticore package. 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]
```

Options apply to all commands:

- `--config <file>` (`-c <file>` for short) overrides the built-in config file names.
- `--quiet` (`-q` for short) keep indextool quiet - it will not output banner, etc.
- `--help` (`-h` for short) lists all of the parameters that can be called in your particular build of `indextool`.
- `-v` show version information of your particular build of `indextool`.

The commands are as follows:

- `--checkconfig` just loads and verifies the config file to check if it's valid, without syntax errors.
- `--build-infixes INDEXNAME` build infixes for an existing dict=keywords index (upgrades .sph, .spi in place). You can use this option for legacy index files that already use dict=keywords, but now need to support infix searching too; updating the index files with indextool may prove easier or faster than regenerating them from scratch with indexer.
- `--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.
- `--dumpconfig FILENAME.sph` dumps the index definition from the given index header file in (almost) compliant `sphinx.conf` file format.
- `--dumpheader INDEXNAME` dumps index header by index name with looking up the header path in the configuration file.
- `--dumpdict INDEXNAME` dumps dictionary.
- `--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 (occurrences) of a given keyword in a given index, with keyword specified as text.
- `--dumphitlist INDEXNAME --wordid ID` dumps all the hits (occurrences) of a given keyword in a given index, with keyword specified as internal numeric ID.
- `--fold INDEXNAME OPTFILE` This options is useful too see how actually tokenizer proceeds input. You can feed indextool with text from file if specified or from stdin otherwise. The output will contain spaces instead of separators (accordingly to your charset_table settings) and lowercased letters in words.
- `--html_strip INDEXNAME` filters stdin using HTML stripper settings for a given index, and prints the filtering results to stdout. Note that the settings will be taken from sphinx.conf, and not the index header.
- `--morph INDEXNAME` applies morphology to the given stdin and prints the result to stdout.
- `--check INDEXNAME` checks the index data files for consistency errors that might be introduced either by bugs in `indexer` and/or hardware faults. `--check` also works on RT indexes, RAM and disk chunks.
- `--strip-path` strips the path names from all the file names referenced from the index (stopwords, wordforms, exceptions, etc). This is useful for checking indexes built on another machine with possibly different path layouts.
- `--optimize-rt-klists` optimizes the kill list memory use in the disk chunk of a given RT index. That is a one-off optimization intended for rather old RT indexes. In last releases this kill list optimization (purging) should happen automatically, and there should never be a need to use this option.
- `--rotate` works only with `--check` and defines whether to check index waiting for rotation, i.e. with .new extension. This is useful when you want to check your index before actually using it.

## 7.3 `searchd` command reference

`searchd` is the second of the two principle tools as part of Manticore. `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`.
- `-v` show version information of 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 asynchronously 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 :ref:`UpdateAttributes() <update_attributes>` will be applied to the index files themselves. Example:

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

- `--stopwait` is used to synchronously stop `searchd`. `--stop` essentially tells the running instance to exit (by sending it a SIGTERM) and then immediately returns. `--stopwait` will also attempt to wait until the running `searchd` instance actually finishes the shutdown (eg. saves all the pending attribute changes) and exits. Example:

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

  Possible exit codes are as follows:

  - 0 on success;
  - 1 if connection to running searchd daemon failed;
  - 2 if daemon reported an error during shutdown;
  - 3 if daemon crashed during shutdown.

- `--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 force using a PID file (where the `searchd` process number is stored) despite any other debugging options that say otherwise (for instance, `--console`). This is a debugging option.

```
$ searchd --console --pidfile
```

- `--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
```

- `--logdebug`, `--logdebugv`, and `--logdebugvv` options enable additional debug output in the daemon log. They differ by the logging verboseness level. These are debugging options, they pollute the log a lot, and thus they should *not* be normally enabled. (The normal use case for these is to enable them temporarily on request, to assist with some particularly complicated debugging session.)

- `--iostats` is used in conjunction 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 <README>* 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 port 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
```

- `--listen ( address ":" port | port | path ) [ ":" protocol ]` (or `-l` for short) Works as `--port`, but allow you to specify not only the port, but full path, as IP address and port, or Unix-domain socket path, that `searchd` will listen on. Otherwords, 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. As the last param you can also specify a protocol handler (listener) to be used for connections on this socket. Supported protocol values are 'sphinx' and 'mysql41' (MySQL protocol used since 4.1 upto at least 5.1).

- `--index <index>` (or `-i <index>` for short) 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
```

- `--strip-path` strips the path names from all the file names referenced from the index (stopwords, wordforms, exceptions, etc). This is useful for picking up indexes built on another machine with possibly different path layouts.

- `--replay-flags=<OPTIONS>` switch can be used to specify a list of extra binary log replay options. The supported options are:

  - `accept-desc-timestamp`, ignore descending transaction timestamps and replay such transactions anyway (the default behavior is to exit with an error).

  Example:

```
$ searchd --replay-flags=accept-desc-timestamp
```

- `--coredump` is used to enable save of core file or minidump of daemon on crash. Disabled by default to speed up of daemon restart on crash. This is useful for debugging purposes.

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

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:\Manticore\bin\searchd.exe --install
   --config C:\Manticore\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:\Manticore\bin\searchd.exe --install
   --config C:\Manticore\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:\Manticore\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:\Manticore\bin\searchd.exe --install
   --config C:\Manticore\sphinx.conf --servicename ManticoreSearch
```

- `--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.
- `--safetrace` forces `searchd` to only use system backtrace() call in crash reports. In certain (rare) scenarios, this might be a "safer" way to get that report. This is a debugging option.
- `--nodetach` switch (Linux only) tells `searchd` not to detach into background. This will also cause log entry to be printed out to console. Query processing operates as usual. This is a debugging option.

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.

## 7.4 `spelldump` command reference

`spelldump` is one of the helper tools within the Manticore 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 alphabetical order, output in the format of a wordforms file, which you can use to customize for your specific circumstances. An example of the result file:

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

## 7.5 `wordbreaker` command reference

`wordbreaker` is one of the helper tools within the Manticore package. It is used to split compound words, as usual in URLs, into its component words. For example, this tool can split "lordoftherings" into its four component words, or "http://manofsteel.warnerbros.com" into "man of steel warner bros". This helps searching, without requiring prefixes or infixes: searching for "sphinx" wouldn't match "sphinxsearch" but if you break the compound word and index the separate components, you'll get a match without the costs of prefix and infix larger index files.

Examples of its usage are:

```
echo manofsteel | bin/wordbreaker -dict dict.txt split
    man of steel
```

The input stream will be separated in words using the `-dict` dictionary file. In no dictionary specified, wordbreaker looks in the working folder for a wordbreaker-dict.txt file. (The dictionary should match the language of the compound word.) The `split` command breaks words from the standard input, and outputs the result in the standard output. There are also `test` and `bench` commands that let you test the splitting quality and benchmark the splitting functionality.

Wordbreaker Wordbreaker needs a dictionary to recognize individual substrings within a string. To differentiate between different guesses, it uses the relative frequency of each word in the dictionary: higher frequency means higher split probability. You can generate such a file using the `indexer` tool, as in

```
indexer --buildstops dict.txt 100000 --buildfreqs myindex -c /path/to/sphinx.conf
```

which will write the 100,000 most frequent words, along with their counts, from myindex into dict.txt. The output file is a text file, so you can edit it by hand, if need be, to add or remove words.

# CHAPTER 8

# SphinxQL reference

SphinxQL is our SQL dialect that exposes all of the search daemon functionality using a standard SQL syntax with a few Manticore-specific extensions. Everything available via the SphinxAPI is also available via SphinxQL but not vice versa; for instance, writes into RT indexes are only available via SphinxQL. This chapter documents supported SphinxQL statements syntax.

## 8.1 ALTER syntax

```
ALTER TABLE index {ADD|DROP} COLUMN column_name [
↪{INTEGER|INT|BIGINT|FLOAT|BOOL|MULTI|MULTI64|JSON|STRING}]
```

It supports adding one attribute at a time for both plain and RT indexes. The int, bigint, float, bool, multi-valued, multi-valued 64bit, json and string attribute types are supported. You can add json and string attributes, but you cannot modify their values.

Implementation details. The querying of an index is impossible (because of a write lock) while adding a column. This may change in the future. The newly created attribute values are set to 0. ALTER will not work for distributed indexes and indexes without any attributes. DROP COLUMN will fail if an index has only one attribute.

```
ALTER RTINDEX index RECONFIGURE
```

ALTER can also reconfigure an existing RT index, so that new tokenization, morphology, and other text processing settings from sphinx.conf take effect on the newly INSERT-ed rows, while retaining the existing rows as they were. Internally, it forcibly saves the current RAM chunk as a new disk chunk, and adjusts the index header, so that the new rows are tokenized using the new rules. Note that as the queries are currently parsed separately for every disk chunk, this might result in warnings regarding the keyword sets mismatch.

```
mysql> desc plain;
+------------+-----------+
| Field      | Type      |
+------------+-----------+
| id         | bigint    |
```

(continues on next page)

(continued from previous page)

```
| text       | field     |
| group_id   | uint      |
| date_added | timestamp |
+------------+-----------+
4 rows in set (0.01 sec)

mysql> alter table plain add column test integer;
Query OK, 0 rows affected (0.04 sec)

mysql> desc plain;
+------------+-----------+
| Field      | Type      |
+------------+-----------+
| id         | bigint    |
| text       | field     |
| group_id   | uint      |
| date_added | timestamp |
| test       | uint      |
+------------+-----------+
5 rows in set (0.00 sec)

mysql> alter table plain drop column group_id;
Query OK, 0 rows affected (0.01 sec)

mysql> desc plain;
+------------+-----------+
| Field      | Type      |
+------------+-----------+
| id         | bigint    |
| text       | field     |
| date_added | timestamp |
| test       | uint      |
+------------+-----------+
4 rows in set (0.00 sec)
```

## 8.2 ATTACH INDEX syntax

```
ATTACH INDEX diskindex TO RTINDEX rtindex
```

ATTACH INDEX statement lets you move data from a regular disk index to a RT index.

After a successful ATTACH, the data originally stored in the source disk index becomes a part of the target RT index, and the source disk index becomes unavailable (until the next rebuild). ATTACH does not result in any index data changes. Basically, it just renames the files (making the source index a new disk chunk of the target RT index), and updates the metadata. So it is a generally quick operation which might (frequently) complete as fast as under a second.

Note that when an index is attached to an empty RT index, the fields, attributes, and text processing settings (tokenizer, wordforms, etc) from the *source* index are copied over and take effect. The respective parts of the RT index definition from the configuration file will be ignored.

ATTACH INDEX comes with a number of restrictions. Most notably, the target RT index is currently required to be empty, making ATTACH INDEX a one-time conversion operation only. Those restrictions may be lifted in future releases, as we add the needed functionality to the RT indexes. The complete list is as follows.

- Target RT index needs to be empty. (See *TRUNCATE RTINDEX syntax*)

- Source disk index needs to have index_sp=0, boundary_step=0, stopword_step=1.
- Source disk index needs to have an empty index_zones setting.

```
mysql> DESC rt;
+-----------+---------+
| Field     | Type    |
+-----------+---------+
| id        | integer |
| testfield | field   |
| testattr  | uint    |
+-----------+---------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM rt;
Empty set (0.00 sec)

mysql> SELECT * FROM disk WHERE MATCH('test');
+------+--------+----------+-------------+
| id   | weight | group_id | date_added |
+------+--------+----------+-------------+
|    1 |   1304 |        1 | 1313643256 |
|    2 |   1304 |        1 | 1313643256 |
|    3 |   1304 |        1 | 1313643256 |
|    4 |   1304 |        1 | 1313643256 |
+------+--------+----------+-------------+
4 rows in set (0.00 sec)

mysql> ATTACH INDEX disk TO RTINDEX rt;
Query OK, 0 rows affected (0.00 sec)

mysql> DESC rt;
+------------+-----------+
| Field      | Type      |
+------------+-----------+
| id         | integer   |
| title      | field     |
| content    | field     |
| group_id   | uint      |
| date_added | timestamp |
+------------+-----------+
5 rows in set (0.00 sec)

mysql> SELECT * FROM rt WHERE MATCH('test');
+------+--------+----------+-------------+
| id   | weight | group_id | date_added |
+------+--------+----------+-------------+
|    1 |   1304 |        1 | 1313643256 |
|    2 |   1304 |        1 | 1313643256 |
|    3 |   1304 |        1 | 1313643256 |
|    4 |   1304 |        1 | 1313643256 |
+------+--------+----------+-------------+
4 rows in set (0.00 sec)

mysql> SELECT * FROM disk WHERE MATCH('test');
ERROR 1064 (42000): no enabled local indexes to search
```

## 8.3 BEGIN, COMMIT, and ROLLBACK syntax

```
START TRANSACTION | BEGIN
COMMIT
ROLLBACK
SET AUTOCOMMIT = {0 | 1}
```

BEGIN statement (or its START TRANSACTION alias) forcibly commits pending transaction, if any, and begins a new one. COMMIT statement commits the current transaction, making all its changes permanent. ROLLBACK statement rolls back the current transaction, canceling all its changes. SET AUTOCOMMIT controls the autocommit mode in the active session.

AUTOCOMMIT is set to 1 by default, meaning that every statement that performs any changes on any index is implicitly wrapped in BEGIN and COMMIT.

Transactions are limited to a single RT index, and also limited in size. They are atomic, consistent, overly isolated, and durable. Overly isolated means that the changes are not only invisible to the concurrent transactions but even to the current session itself.

## 8.4 BEGIN syntax

```
START TRANSACTION | BEGIN
```

BEGIN syntax is discussed in detail in *BEGIN, COMMIT, and ROLLBACK syntax*.

## 8.5 CALL KEYWORDS syntax

```
CALL KEYWORDS(text, index [, options])
```

CALL KEYWORDS statement splits text into particular keywords. It returns tokenized and normalized forms of the keywords, and, optionally, keyword statistics. It also returns the position of each keyword in the query and all forms of tokenized keywords in the case that lemmatizers were used.

`text` is the text to break down to keywords. `index` is the name of the index from which to take the text processing settings. `options`, is an optional boolean parameter that specifies whether to return document and hit occurrence statistics. `options` can also accept parameters for configuring folding depending on tokenization settings:

- `stats` - show statistics of keywords, default is 0
- `fold_wildcards` - fold wildcards, default is 1
- `fold_lemmas` - fold morphological lemmas, default is 0
- `fold_blended` - fold blended words, default is 0
- `expansion_limit` - override expansion_limit defined in configuration, default is 0 (use value from configuration)

```
call keywords(
    'que*',
    'myindex',
    1 as fold_wildcards,
    1 as fold_lemmas,
    1 as fold_blended,
```

(continues on next page)

(continued from previous page)

```
    1 as expansion_limit,
    1 as stats);
```

Default values to match previous CALL KEYWORDS output are:

```
call keywords(
    'que*',
    'myindex',
    1 as fold_wildcards,
    0 as fold_lemmas,
    0 as fold_blended,
    0 as expansion_limit,
    0 as stats);
```

## 8.6 CALL PQ syntax

```
CALL PQ(data, index[, opt_value AS opt_name[, ...]])
```

CALL PQ statement performs a prospective search. It returns stored queries from a `percolate``index` that match documents from provided``data`. For more information, see *Percolate Query* section.

`data` can be a document in plain text, a JSON object containing a document or a list of documents in one of the two formats. The JSON object can contain pairs of text field names and values as well as attribute names and values.

Example:

```
CALL PQ ('index_name', 'single document', 0 as docs_json);
CALL PQ ('index_name', ('first document', 'second document'), 0 as docs_json );
CALL PQ ('index_name', '{"title":"single document","content":"Add your content here",
↪"category":10,"timestamp":1513725448}');
CALL PQ ('index_name', (
                          '{"title":"first document","content":"Add your content here
↪","category":10,"timestamp":1513725448}',
                          '{"title":"second document","content":"Add more content here
↪","category":20,"timestamp":1513758240}'
                        )
 );
```

A number of options can be set:

- docs_json - 1 ( default enabled), specify if the `data` provides document(s) as raw string or encapsulated as JSON object
- docs - 0 (default disabled), provide per query documents matched at result set
- verbose - 0 (default disabled), provide extended info in *SHOW META*
- query - 0 (default disabled), if true returns all information of matched stored query, otherwise it returns just the stored query ID

Example:

```
MySQL [(none)]> CALL PQ('pq','catch me if you can',0 AS docs_json,1 AS query);
+------+----------+------+---------+
| UID  | Query    | Tags | Filters |
+------+----------+------+---------+
```

(continues on next page)

(continued from previous page)

```
|      6 | catch me |        |          |
+--------+----------+--------+----------+
1 row in set (0.00 sec)
```

`CALL PQ` can be followed by a *SHOW META* statement which provides additional meta-information about the executed prospective search.

## 8.7 CALL QSUGGEST syntax

```
CALL QSUGGEST(word, index [,options])
```

CALL QSUGGEST statement enumerates for a giving word all suggestions from the dictionary. This statement works only on indexes with infixing enabled and dict=keywords. It returns the suggested keywords, Levenshtein distance between the suggested and original keyword and the docs statistic of the suggested keyword. If the first parameter is a bag of words, the function will return suggestions only for the last word, ignoring the rest. Several options are supported for customization:

- `limit` - returned N top matches, default is 5
- `max_edits` - keep only dictionary words which Levenshtein distance is less or equal, default is 4
- `result_stats` - provide Levenshtein distance and document count of the found words, default is 1 (enabled)
- `delta_len` - keep only dictionary words whose length difference is less, default is 3
- `max_matches` - number of matches to keep, default is 25
- `reject` - defaults to 4; rejected words are matches that are not better than those already in the match queue. They are put in a rejected queue that gets reset in case one actually can go in the match queue. This parameter defines the size of the rejected queue (as reject*max(max_matched,limit)). If the rejected queue is filled, the engine stops looking for potential matches.
- `result_line` - alternate mode to display the data by returning all suggests, distances and docs each per one row, default is 0
- `non_char` - do not skip dictionary words with non alphabet symbols, default is 0 (skip such words)

```
mysql> CALL QSUGGEST('automaticlly ','forum', 5 as limit, 4 as max_edits,1 as result_
↪stats,3 as delta_len,0 as result_line,25 as max_matches,4 as reject );
+---------------+----------+------+
| suggest       | distance | docs |
+---------------+----------+------+
| automatically | 1        | 282  |
| automaticly   | 1        | 6    |
| automaticaly  | 1        | 3    |
| automagically | 2        | 14   |
| automtically  | 2        | 1    |
+---------------+----------+------+
5 rows in set (0.00 sec)
```

## 8.8 CALL SNIPPETS syntax

```
CALL SNIPPETS(data, index, query[, opt_value AS opt_name[, ...]])
```

CALL SNIPPETS statement builds a snippet from provided data and query, using specified index settings.

`data` is the source data to extract a snippet from. It could be a single string, or the list of the strings enclosed in curly brackets. `index` is the name of the index from which to take the text processing settings. `query` is the full-text query to build snippets for. Additional options are documented in *BuildExcerpts*. Usage example:

```
CALL SNIPPETS('this is my document text', 'test1', 'hello world',
    5 AS around, 200 AS limit);
CALL SNIPPETS(('this is my document text','this is my another text'), 'test1', 'hello␣
↪world',
    5 AS around, 200 AS limit);
CALL SNIPPETS(('data/doc1.txt','data/doc2.txt','/home/sphinx/doc3.txt'), 'test1',
↪'hello world',
    5 AS around, 200 AS limit, 1 AS load_files);
```

## 8.9 CALL SUGGEST syntax

```
CALL SUGGEST(word, index [,options])
```

CALL SUGGEST statement works the same as CALL QUSUGGEST, except that if a bag of words is present, the statement will return suggestions only for the first word, ignoring the rest. If the first paramenter is a word, the functionality of CALL SUGGEST and CALL QSUGGEST is the same.

## 8.10 Comment syntax

SphinxQL supports C-style comment syntax. Everything from an opening `/*` sequence to a closing `*/` sequence is ignored. Comments can span multiple lines, can not nest, and should not get logged. MySQL specific `/*! ... */` comments are also currently ignored. (As the comments support was rather added for better compatibility with `mysqldump` produced dumps, rather than improving general query interoperability between Manticore and MySQL.)

```
SELECT /*! SQL_CALC_FOUND_ROWS */ col1 FROM table1 WHERE ...
```

## 8.11 CREATE FUNCTION syntax

```
CREATE FUNCTION udf_name
    RETURNS {INT | INTEGER | BIGINT | FLOAT | STRING}
    SONAME 'udf_lib_file'
```

CREATE FUNCTION statement installs a *user-defined function (UDF)* with the given name and type from the given library file. The library file must reside in a trusted *plugin_dir* directory. On success, the function is available for use in all subsequent queries that the server receives. Example:

```
mysql> CREATE FUNCTION avgmva RETURNS INTEGER SONAME 'udfexample.dll';
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT *, AVGMVA(tag) AS q from test1;
+------+--------+---------+------------+
| id   | weight | tag     | q          |
+------+--------+---------+------------+
|    1 |      1 | 1,3,5,7 | 4.000000   |
```

(continues on next page)

(continued from previous page)

```
|    2 |       1 | 2,4,6    | 4.000000  |
|    3 |       1 | 15       | 15.000000 |
|    4 |       1 | 7,40     | 23.500000 |
+------+---------+----------+-----------+
```

## 8.12 CREATE PLUGIN syntax

```
CREATE PLUGIN plugin_name TYPE 'plugin_type' SONAME 'plugin_library'
```

Loads the given library (if it is not loaded yet) and loads the specified plugin from it. The known plugin types are:

- ranker
- index_token_filter
- query_token_filter

Refer to *Plugins* for more information regarding writing the plugins.

```
mysql> CREATE PLUGIN myranker TYPE 'ranker' SONAME 'myplugins.so';
Query OK, 0 rows affected (0.00 sec)
```

## 8.13 DELETE syntax

```
DELETE FROM index WHERE where_condition
```

DELETE statement is only supported for RT indexes and for distributed which contains only RT indexes as agents It deletes existing rows (documents) from an existing index based on ID.

`index` is the name of RT index from which the row should be deleted.

`where_condition` has the same syntax as in the SELECT statement (see *SELECT syntax* for details).

```
mysql> select * from rt;
+------+------+-------------+------+
| id   | gid  | mva1        | mva2 |
+------+------+-------------+------+
|  100 | 1000 | 100,201     | 100  |
|  101 | 1001 | 101,202     | 101  |
|  102 | 1002 | 102,203     | 102  |
|  103 | 1003 | 103,204     | 103  |
|  104 | 1004 | 104,204,205 | 104  |
|  105 | 1005 | 105,206     | 105  |
|  106 | 1006 | 106,207     | 106  |
|  107 | 1007 | 107,208     | 107  |
+------+------+-------------+------+
8 rows in set (0.00 sec)

mysql> delete from rt where match ('dumy') and mva1>206;
Query OK, 2 rows affected (0.00 sec)

mysql> select * from rt;
+------+------+-------------+------+
```

(continues on next page)

(continued from previous page)

```
| id   | gid  | mva1        | mva2 |
+------+------+-------------+------+
|  100 | 1000 | 100,201     | 100  |
|  101 | 1001 | 101,202     | 101  |
|  102 | 1002 | 102,203     | 102  |
|  103 | 1003 | 103,204     | 103  |
|  104 | 1004 | 104,204,205 | 104  |
|  105 | 1005 | 105,206     | 105  |
+------+------+-------------+------+
6 rows in set (0.00 sec)

mysql> delete from rt where id in (100,104,105);
Query OK, 3 rows affected (0.01 sec)

mysql> select * from rt;
+------+------+---------+------+
| id   | gid  | mva1    | mva2 |
+------+------+---------+------+
|  101 | 1001 | 101,202 | 101  |
|  102 | 1002 | 102,203 | 102  |
|  103 | 1003 | 103,204 | 103  |
+------+------+---------+------+
3 rows in set (0.00 sec)

mysql> delete from rt where mva1 in (102,204);
Query OK, 2 rows affected (0.01 sec)

mysql> select * from rt;
+------+------+---------+------+
| id   | gid  | mva1    | mva2 |
+------+------+---------+------+
|  101 | 1001 | 101,202 | 101  |
+------+------+---------+------+
1 row in set (0.00 sec)
```

## 8.14 DESCRIBE syntax

```
{DESC | DESCRIBE} index [ LIKE pattern ]
```

DESCRIBE statement lists index columns and their associated types. Columns are document ID, full-text fields, and attributes. The order matches that in which fields and attributes are expected by INSERT and REPLACE statements. Column types are `field`, `integer`, `timestamp`, `ordinal`, `bool`, `float`, `bigint`, `string`, and `mva`. ID column will be typed as `bigint`. Example:

```
mysql> DESC rt;
+---------+---------+
| Field   | Type    |
+---------+---------+
| id      | bigint  |
| title   | field   |
| content | field   |
| gid     | integer |
+---------+---------+
4 rows in set (0.00 sec)
```

An optional LIKE clause is supported. Refer to *SHOW META syntax* for its syntax details.

## 8.15 DROP FUNCTION syntax

```
DROP FUNCTION udf_name
```

DROP FUNCTION statement deinstalls a *user-defined function (UDF)* with the given name. On success, the function is no longer available for use in subsequent queries. Pending concurrent queries will not be affected and the library unload, if necessary, will be postponed until those queries complete. Example:

```
mysql> DROP FUNCTION avgmva;
Query OK, 0 rows affected (0.00 sec)
```

## 8.16 DROP PLUGIN syntax

```
DROP PLUGIN plugin_name TYPE 'plugin_type'
```

Markes the specified plugin for unloading. The unloading is **not** immediate, because the concurrent queries might be using it. However, after a DROP new queries will not be able to use it. Then, once all the currently executing queries using it are completed, the plugin will be unloaded. Once all the plugins from the given library are unloaded, the library is also automatically unloaded.

```
mysql> DROP PLUGIN myranker TYPE 'ranker';
Query OK, 0 rows affected (0.00 sec)
```

## 8.17 FLUSH ATTRIBUTES syntax

```
FLUSH ATTRIBUTES
```

Flushes all in-memory attribute updates in all the active disk indexes to disk. Returns a tag that identifies the result on-disk state (basically, a number of actual disk attribute saves performed since the daemon startup).

```
mysql> UPDATE testindex SET channel_id=1107025 WHERE id=1;
Query OK, 1 row affected (0.04 sec)

mysql> FLUSH ATTRIBUTES;
+------+
| tag  |
+------+
|    1 |
+------+
1 row in set (0.19 sec)
```

## 8.18 FLUSH HOSTNAMES syntax

```
FLUSH HOSTNAMES
```

Renew IPs associates to agent host names. To always query the DNS for getting the host name IP, see *hostname_lookup* directive.

```
mysql> FLUSH HOSTNAMES;
Query OK, 5 rows affected (0.01 sec)
```

## 8.19 FLUSH LOGS syntax

```
FLUSH LOGS
```

Works same as system USR1 signal. Initiate reopen of searchd log and query log files, letting you implement log file rotation. Command is non-blocking (i.e., returns immediately).

```
mysql> FLUSH LOGS;
Query OK, 0 rows affected (0.01 sec)
```

## 8.20 FLUSH RAMCHUNK syntax

```
FLUSH RAMCHUNK rtindex
```

FLUSH RAMCHUNK forcibly creates a new disk chunk in a RT index.

Normally, RT index would flush and convert the contents of the RAM chunk into a new disk chunk automatically, once the RAM chunk reaches the maximum allowed *rt_mem_limit* size. However, for debugging and testing it might be useful to forcibly create a new disk chunk, and FLUSH RAMCHUNK statement does exactly that.

Note that using FLUSH RAMCHUNK increases RT index fragmentation. Most likely, you want to use FLUSH RTINDEX instead. We suggest that you abstain from using just this statement unless you're absolutely sure what you're doing. As the right way is to issue FLUSH RAMCHUNK with following *OPTIMIZE* command. Such combo allows to keep RT index fragmentation on minimum.

```
mysql> FLUSH RAMCHUNK rt;
Query OK, 0 rows affected (0.05 sec)
```

## 8.21 FLUSH RTINDEX syntax

```
FLUSH RTINDEX rtindex
```

FLUSH RTINDEX forcibly flushes RT index RAM chunk contents to disk.

Backing up a RT index is as simple as copying over its data files, followed by the binary log. However, recovering from that backup means that all the transactions in the log since the last successful RAM chunk write would need to be replayed. Those writes normally happen either on a clean shutdown, or periodically with a (big enough!) interval between writes specified in *rt_flush_period* directive. So such a backup made at an arbitrary point in time just might end up with way too much binary log data to replay.

FLUSH RTINDEX forcibly writes the RAM chunk contents to disk, and also causes the subsequent cleanup of (now-redundant) binary log files. Thus, recovering from a backup made just after FLUSH RTINDEX should be almost instant.

```
mysql> FLUSH RTINDEX rt;
Query OK, 0 rows affected (0.05 sec)
```

## 8.22 INSERT and REPLACE syntax

```
{INSERT | REPLACE} INTO index [(column, ...)]
    VALUES (value, ...)
    [, (...)]
```

INSERT statement is only supported for RT indexes. It inserts new rows (documents) into an existing index, with the provided column values.

ID column must be present in all cases. Rows with duplicate IDs will **not** be overwritten by INSERT; use REPLACE to do that. REPLACE works exactly like INSERT, except that if an old row has the same ID as a new row, the old row is deleted before the new row is inserted.

`index` is the name of RT index into which the new row(s) should be inserted. The optional column names list lets you only explicitly specify values for some of the columns present in the index. All the other columns will be filled with their default values (0 for scalar types, empty string for text types).

Expressions are not currently supported in INSERT and values should be explicitly specified.

Multiple rows can be inserted using a single INSERT statement by providing several comma-separated, parentheses-enclosed lists of rows values.

## 8.23 List of SphinxQL reserved keywords

A complete alphabetical list of keywords that are currently reserved in SphinxQL syntax (and therefore can not be used as identifiers).

```
AND, AS, BY, DIV, FACET, FALSE, FROM, ID, IN, INDEXES, IS, LIMIT,
LOGS, MOD, NOT, NULL, OR, ORDER, RELOAD, SELECT, SYSFILTERS, TRUE
```

## 8.24 Multi-statement queries

SphinxQL supports multi-statement queries, or batches. Possible inter-statement optimizations described in *Multi-queries* do apply to SphinxQL just as well. The batched queries should be separated by a semicolon. Your MySQL client library needs to support MySQL multi-query mechanism and multiple result set. For instance, mysqli interface in PHP and DBI/DBD libraries in Perl are known to work.

Here's a PHP sample showing how to utilize mysqli interface with Manticore.

```
<?php

$link = mysqli_connect ( "127.0.0.1", "root", "", "", 9306 );
if ( mysqli_connect_errno() )
    die ( "connect failed: " . mysqli_connect_error() );

$batch = "SELECT * FROM test1 ORDER BY group_id ASC;";
$batch .= "SELECT * FROM test1 ORDER BY group_id DESC";

```

(continues on next page)

(continued from previous page)

```
if ( !mysqli_multi_query ( $link, $batch ) )
    die ( "query failed" );

do
{
    // fetch and print result set
    if ( $result = mysqli_store_result($link) )
    {
        while ( $row = mysqli_fetch_row($result) )
            printf ( "id=%s\n", $row[0] );
        mysqli_free_result($result);
    }

    // print divider
    if ( mysqli_more_results($link) )
        printf ( "------\n" );

} while ( mysqli_next_result($link) );
```

Its output with the sample `test1` index included with Manticore is as follows.

```
$ php test_multi.php
id=1
id=2
id=3
id=4
------
id=3
id=4
id=1
id=2
```

The following statements can currently be used in a batch: SELECT, SHOW WARNINGS, SHOW STATUS, and SHOW META. Arbitrary sequence of these statements are allowed. The results sets returned should match those that would be returned if the batched queries were sent one by one.

## 8.25 OPTIMIZE INDEX syntax

```
OPTIMIZE INDEX index_name
```

OPTIMIZE statement enqueues a RT index for optimization in a background thread.

Over time, RT indexes can grow fragmented into many disk chunks and/or tainted with deleted, but unpurged data, impacting search performance. When that happens, they can be optimized. Basically, the optimization pass merges together disk chunks pairs, purging off documents suppressed by K-list as it goes.

That is a lengthy and IO intensive process, so to limit the impact, all the actual merge work is executed serially in a special background thread, and the OPTIMIZE statement simply adds a job to its queue. Currently, there is no way to check the index or queue status (that might be added in the future to the SHOW INDEX STATUS and SHOW STATUS statements respectively). The optimization thread can be IO-throttled, you can control the maximum number of IOs per second and the maximum IO size with *rt_merge_iops* and *rt_merge_maxiosize* directives respectively. The optimization jobs queue is lost on daemon crash.

The RT index being optimized stays online and available for both searching and updates at (almost) all times during the optimization. It gets locked (very) briefly every time that a pair of disk chunks is merged successfully, to rename

the old and the new files, and update the index header.

At the moment, OPTIMIZE needs to be issued manually, the indexes will *not* be optimized automatically. That might change in the future releases.

```
mysql> OPTIMIZE INDEX rt;
Query OK, 0 rows affected (0.00 sec)
```

## 8.26 RELOAD INDEX syntax

```
RELOAD INDEX idx [ FROM '/path/to/index_files' ]
```

RELOAD INDEX allows you to rotate indexes using SphinxQL.

It has two modes of operation. First one (without specifying a path) makes Manticore daemon check for new index files in directory specified in *path*. New index files must have a idx.new.sp? names.

And if you additionally specify a path, daemon will look for index files in specified directory, move them to index *path*, rename from index_files.sp? to idx.new.sp? and rotate them.

```
mysql> RELOAD INDEX plain_index;
mysql> RELOAD INDEX plain_index FROM '/home/mighty/new_index_files';
```

## 8.27 RELOAD INDEXES syntax

```
RELOAD INDEXES
```

Works same as system HUP signal. Initiates index rotation. Depending on the value of *seamless_rotate* setting, new queries might be shortly stalled; clients will receive temporary errors. Command is non-blocking (i.e., returns immediately).

```
mysql> RELOAD INDEXES;
Query OK, 0 rows affected (0.01 sec)
```

## 8.28 RELOAD PLUGINS syntax

```
RELOAD PLUGINS FROM SONAME 'plugin_library'
```

Reloads all plugins (UDFs, rankers, etc) from a given library. Reload is, in a sense, transactional: a successful reload guarantees that a) all the plugins were successfully updated with their new versions; b) the update was atomic, all the plugins were replaced at once. Atomicity means that queries using multiple functions from a reloaded library will never mix the old and new versions. The set of plugins is guaranteed to always be consistent during the RELOAD, it will be either all old, or all new.

Reload also is seamless, meaning that some version of a reloaded plugin will be available to concurrent queries at all times, and there will be no temporary disruptions. Note how this improves on using a pair of DROP and CREATE statements for reloading: with those, there is a tiny window between the DROP and the subsequent CREATE, during which the queries technically refer to an unknown plugin and will thus fail.

In case of any failure RELOAD PLUGINS does absolutely nothing, keeps the old plugins, and reports an error.

On Windows, either overwriting or deleting a DLL library currently in use seems to be an issue. However, you can still rename it, then put a new version under the old name, and RELOAD will then work. After a succesful reload you will also be able to delete the renamed old library, too.

```
mysql> RELOAD PLUGINS FROM SONAME 'udfexample.dll';
Query OK, 0 rows affected (0.00 sec)
```

## 8.29 REPLACE syntax

```
{INSERT | REPLACE} INTO index [(column, ...)]
    VALUES (value, ...)
    [, (...)]
```

REPLACE syntax is identical to INSERT syntax and is described in *INSERT and REPLACE syntax*.

## 8.30 ROLLBACK syntax

```
ROLLBACK
```

ROLLBACK syntax is discussed in detail in *BEGIN, COMMIT, and ROLLBACK syntax*.

## 8.31 SELECT syntax

```
SELECT
    select_expr [, select_expr ...]
    FROM index [, index2 ...]
    [WHERE where_condition]
    [GROUP [N] BY {col_name | expr_alias} [, {col_name | expr_alias}]]
    [WITHIN GROUP ORDER BY {col_name | expr_alias} {ASC | DESC}]
    [HAVING having_condition]
    [ORDER BY {col_name | expr_alias} {ASC | DESC} [, ...]]
    [LIMIT [offset,] row_count]
    [OPTION opt_name = opt_value [, ...]]
    [FACET facet_options[ FACET facet_options][ ...]]
```

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

### 8.31.1 Column list

Column list clause. Column names, arbitrary expressions, and star ('*') are all allowed (ie. `SELECT id, group_id*123+456 AS expr1 FROM test1` will work). Unlike in regular SQL, all computed expressions must be aliased with a valid identifier. `AS` is optional.

### 8.31.2 EXIST()

EXIST ( "attr-name", default-value ) replaces non-existent columns with default values. It returns either a value of an attribute specified by 'attr-name', or 'default-value' if that attribute does not exist. It does not support STRING or

MVA attributes. This function is handy when you are searching through several indexes with different schemas.

```
SELECT *, EXIST('gid', 6) as cnd FROM i1, i2 WHERE cnd>5
```

### 8.31.3 SNIPPET()

This is a wrapper around the snippets functionality, similar to what is available via CALL SNIPPETS. The first two arguments are: the text to highlight, and a query. It's possible to pass *options* to function. The intended use is as follows:

```
SELECT id, SNIPPET(myUdf(id), 'my.query', 'limit=100')
FROM myIndex WHERE MATCH('my.query')
```

where myUdf() would be a UDF that fetches a document by its ID from some external storage. This enables applications to fetch the entire result set directly from Manticore in one query, without having to separately fetch the documents in the application and then send them back to Manticore for highlighting.

SNIPPET() is a so-called "post limit" function, meaning that computing snippets is postponed not just until the entire final result set is ready, but even after the LIMIT clause is applied. For example, with a LIMIT 20,10 clause, SNIPPET() will be called at most 10 times.

Table functions is a mechanism of post-query result set processing. Table functions take an arbitrary result set as their input, and return a new, processed set as their output. The first argument should be the input result set, but a table function can optionally take and handle more arguments. Table functions can completely change the result set, including the schema. For now, only built in table functions are supported. UDFs are planned when the internal call interface is stabilized. Table functions work for both outer SELECT and nested SELECT.

### 8.31.4 REMOVE_REPEATS()

REMOVE_REPEATS ( result_set, column, offset, limit ) - removes repeated adjusted rows with the same 'column' value.

```
SELECT REMOVE_REPEATS((SELECT * FROM dist1), gid, 0, 10)
```

### 8.31.5 FROM

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. Index name should be according to the rules of a C identifier.

### 8.31.6 WHERE

This clause will map both to fulltext query and filters. Comparison operators (=, !=, <, >, <=, >=), IN, AND, OR, NOT, and BETWEEN are all supported and map directly to filters. 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. `{col_name | expr_alias} [NOT] IN @uservar` condition syntax is supported. (Refer to *SET syntax* for a description of global user variables.)

### 8.31.7 GROUP BY

Supports grouping by multiple columns or computed expressions:

```
SELECT *, group_id*1000+article_type AS gkey FROM example GROUP BY gkey
SELECT id FROM products GROUP BY region, price
```

Implicit grouping supported when using aggregate functions without specifiying a GROUP BY clause. Consider these two queries:

```
SELECT MAX(id), MIN(id), COUNT(*) FROM books
SELECT MAX(id), MIN(id), COUNT(*), 1 AS grp FROM books GROUP BY grp
```

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(*), COUNT(DISTINCT attr) are 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. A special GROUPBY() function is also supported. It returns the GROUP BY key. That is particularly useful when grouping by an MVA value, in order to pick the specific value that was used to create the current group.

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

GROUP BY on a string attribute is supported, with respect for current collation (see *Collations*).

You can query Manticore to return (no more than) N top matches for each group accordingly to WITHIN GROUP ORDER BY.

```
SELECT id FROM products GROUP 3 BY category
```

You can sort the result set by (an alias of) the aggregate value.

```
SELECT group_id, MAX(id) AS max_id
FROM my_index WHERE MATCH('the')
GROUP BY group_id ORDER BY max_id DESC
```

### 8.31.8 GROUP_CONCAT()

When you group by an attribute, the result set only shows attributes from a single document representing the whole group. GROUP_CONCAT() produces a comma-separated list of the attribute values of all documents in the group.

```
SELECT id, GROUP_CONCAT(price) as pricesList, GROUPBY() AS name FROM shops GROUP BY␣
↪shopName;
```

### 8.31.9 ZONESPANLIST()

ZONESPANLIST() function returns pairs of matched zone spans. Each pair contains the matched zone span identifier, a colon, and the order number of the matched zone span. For example, if a document reads <emphasis role=”bold”>text the text</emphasis>, and you query for ‘ZONESPAN:(i,b) text’, then ZONESPANLIST() will return the string “1:1 1:2 2:1” meaning that the first zone span matched “text” in spans 1 and 2, and the second zone span in span 1 only.

### 8.31.10 WITHIN GROUP ORDER BY

This is a Manticore 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, WEIGHT() AS w
    FROM example WHERE MATCH('my search query')
    GROUP BY siteid
    WITHIN GROUP ORDER BY w DESC
    ORDER BY timeseg DESC, w DESC

WITHIN GROUP ORDER BY on a string attribute is supported, with
respect for current collation (see :ref:`collations`).
```

### 8.31.11 HAVING

This is used to filter on GROUP BY values. Currently supports only one filtering condition.

```
SELECT id FROM plain GROUP BY title HAVING group_id=16;
SELECT id FROM plain GROUP BY attribute HAVING COUNT(*)>1;
```

Because of HAVING is implemented as a whole result set post-processing, result set for query with HAVING could be less than *max_matches* allows.

### 8.31.12 ORDER BY

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
```

You can use subqueries to speed up specific searches, which involve reranking, by postponing hard (slow) calculations as late as possible. For example, SELECT id,a_slow_expression() AS cond FROM an_index ORDER BY id ASC, cond DESC LIMIT 100; could be better written as SELECT * FROM (SELECT id,a_slow_expression() AS cond FROM an_index ORDER BY id ASC LIMIT 100) ORDER BY cond DESC; because in the first case the slow expression would be evaluated for the whole set, while in the second one it would be evaluated just for a subset of values.

ORDER BY on a string attribute is supported, with respect for current collation (see *Collations*).

ORDER BY RAND() syntax is supported. Note that this syntax is actually going to randomize the weight values and then order matches by those randomized weights.

### 8.31.13 LIMIT

Both LIMIT N and LIMIT M,N forms are supported. Unlike in regular SQL (but like in Manticore API), an implicit LIMIT 0,20 is present by default.

### 8.31.14 OPTION

This is a Manticore 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:

- `agent_query_timeout` - integer (max time in milliseconds to wait for remote queries to complete, see *agent_query_timeout* under Index configuration options for details)
- `boolean_simplify` - 0 or 1, enables simplifying the query to speed it up
- `comment` - string, user comment that gets copied to a query log file
- `cutoff` - integer (max found matches threshold)
- `field_weights` - a named integer list (per-field user weights for ranking)
- `global_idf` - use global statistics (frequencies) from the *global_idf file* for IDF computations, rather than the local index statistics.
- `idf` - a quoted, comma-separated list of IDF computation flags. Known flags are:
  - normalized: BM25 variant, idf = log((N-n+1)/n), as per Robertson et al
  - plain: plain variant, idf = log(N/n), as per Sparck-Jones
  - tfidf_normalized: additionally divide IDF by query word count, so that TF*IDF fits into [0, 1] range
  - tfidf_unnormalized: do not additionally divide IDF by query word count

  where **N** is the collection size and **n** is the number of matched documents.

  The historically default IDF (Inverse Document Frequency) in Manticore is equivalent to `OPTION idf='normalized,tfidf_normalized'`, and those normalizations may cause several undesired effects.

  First, idf=normalized causes keyword penalization. For instance, if you search for [the | something] and [the] occurs in more than 50% of the documents, then documents with both keywords [the] and [something] will get **less** weight than documents with just one keyword [something]. Using `OPTION idf=plain` avoids this. Plain IDF varies in [0, log(N)] range, and keywords are never penalized; while the normalized IDF varies in [-log(N), log(N)] range, and too frequent keywords are penalized.

  Second, idf=tfidf_normalized causes IDF drift over queries. Historically, we additionally divided IDF by query keyword count, so that the entire sum(tf*idf) over all keywords would still fit into [0,1] range. However, that means that queries [word1] and [word1 | nonmatchingword2] would assign different weights to the exactly same result set, because the IDFs for both "word1" and "nonmatchingword2" would be divided by 2. `OPTION idf='tfidf_unnormalized'` fixes that. Note that BM25, BM25A, BM25F() ranking factors will be scale accordingly once you disable this normalization.

  IDF flags can be mixed; `plain` and `normalized` are mutually exclusive; `tfidf_unnormalized` and `tfidf_normalized` are mutually exclusive; and unspecified flags in such a mutually exclusive group take their defaults. That means that `OPTION idf=plain` is equivalent to a complete `OPTION idf='plain, tfidf_normalized'` specification.
- `local_df` - 0 or 1,automatically sum DFs over all the local parts of a distributed index, so that the IDF is consistent (and precise) over a locally sharded index.
- `index_weights` - a named integer list (per-index user weights for ranking)
- `max_matches` - integer (per-query max matches value)

  Maximum amount of matches that the daemon keeps in RAM for each index and can return to the client. 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.

- `max_query_time` - integer (max search time threshold, msec)
- `max_predicted_time` - integer (max predicted search time, see *predicted_time_costs*)
- `ranker` - any of `proximity_bm25`, `bm25`, `none`, `wordcount`, `proximity`, `matchany`, `fieldmask`, `sph04`, `expr`, or `export` (refer to *Search results ranking* for more details on each ranker)
- `retry_count` - integer (distributed retries count)
- `retry_delay` - integer (distributed retry delay, msec)
- `reverse_scan` - 0 or 1, lets you control the order in which full-scan query processes the rows
- `sort_method` - `pq` (priority queue, set by default) or `kbuffer` (gives faster sorting for already pre-sorted data, e.g. index data sorted by id). The result set is in both cases the same; picking one option or the other may just improve (or worsen!) performance.
- `rand_seed` - lets you specify a specific integer seed value for an `ORDER BY RAND()` query, for example: ... OPTION `rand_seed=1234`. By default, a new and different seed value is autogenerated for every query.
- `low_priority` - runs the query with idle priority.
- `expand_keywords` - 0 or 1, expand keywords with exact forms and/or stars when possible (refer to *expand_keywords* for more details).

Example:

```
SELECT * FROM test WHERE MATCH('@title hello @body world')
OPTION ranker=bm25, max_matches=3000,
    field_weights=(title=10, body=3), agent_query_timeout=10000
```

### 8.31.15 FACET

This Manticore specific extension enables faceted search with subtree optimization. It is capable of returning multiple result sets with a single SQL statement, without the need for complicated *multi-queries*. FACET clauses should be written at the very end of SELECT statements with spaces between them.

```
FACET {expr_list} [BY {expr_list}] [ORDER BY {expr | FACET()} {ASC | DESC}] [LIMIT␣
↪[offset,] count]
SELECT * FROM test FACET brand_id FACET categories;
SELECT * FROM test FACET brand_name BY brand_id ORDER BY brand_name ASC FACET␣
↪property;
```

Working example:

```
mysql> SELECT *, IN(brand_id,1,2,3,4) AS b FROM facetdemo WHERE MATCH('Product') AND b=1 LIMIT 0,10
FACET brand_name, brand_id BY brand_id ORDER BY brand_id ASC
FACET property ORDER BY COUNT(*) DESC
FACET INTERVAL(price,200,400,600,800) ORDER BY FACET() ASC
FACET categories ORDER BY FACET() ASC;
+------+-------+----------+-------------------+------------+----------+------------+-----+
| id   | price | brand_id | title             | brand_name | property | categories | b** |
+------+-------+----------+-------------------+------------+----------+------------+-----+
|    1 |   668 |        3 | Product Four Six  | Brand Three | Three    | 11,12,13   |   1 |
|    2 |   101 |        4 | Product Two Eight | Brand Four  | One      | 12,13,14   |   1 |
|    8 |   750 |        3 | Product Ten Eight | Brand Three | Five     | 13         |   1 |
|    9 |    49 |        1 | Product Ten Two   | Brand One   | Three    | 13,14,15   |   1 |
|   13 |   613 |        1 | Product Six Two   | Brand One   | Eight    | 13         |   1 |
|   20 |   985 |        2 | Product Two Six   | Brand Two   | Nine     | 10         |   1 |
|   22 |   501 |        3 | Product Five Two  | Brand Three | Four     | 12,13,14   |   1 |
|   23 |   765 |        1 | Product Six Seven | Brand One   | Nine     | 11,12      |   1 |
|   28 |   992 |        1 | Product Six Eight | Brand One   | Two      | 12,13      |   1 |
|   29 |   259 |        1 | Product Nine Ten  | Brand One   | Five     | 12,13,14   |   1 |
+------+-------+----------+-------------------+------------+----------+------------+-----+
+-------------+----------+----------+
| brand_name  | brand_id | count(*) |
+-------------+----------+----------+
| Brand One   |        1 |     1012 |
| Brand Two   |        2 |     1025 |
| Brand Three |        3 |      994 |
| Brand Four  |        4 |      973 |
+-------------+----------+----------+
+----------+----------+
| property | count(*) |
+----------+----------+
| One      |      427 |
| Five     |      420 |
| Seven    |      420 |
| Two      |      418 |
| Three    |      407 |
| Six      |      401 |
| Nine     |      396 |
| Eight    |      387 |
| Four     |      371 |
| Ten      |      357 |
+----------+----------+
+---------------------------------+----------+
```

(continues on next page)

(continued from previous page)

```
| interval(price,200,400,600,800) | count(*) |
+----------------------------------+----------+
|                                0 |      799 |
|                                1 |      795 |
|                                2 |      757 |
|                                3 |      833 |
|                                4 |      820 |
+----------------------------------+----------+
+------------+----------+
| categories | count(*) |
+------------+----------+
|         10 |      961 |
|         11 |     1653 |
|         12 |     1998 |
|         13 |     2090 |
|         14 |     1058 |
|         15 |      347 |
+------------+----------+
```

### 8.31.16 Subselects

In format `SELECT * FROM (SELECT ... ORDER BY cond1 LIMIT X) ORDER BY cond2 LIMIT Y`. The outer select allows only ORDER BY and LIMIT clauses. Subselects currently have 2 usage cases:

1. We have a query with 2 ranking UDFs, one very fast and the other one slow and we perform a full-text search will a big match result set. Without subselect the query would look like

   ```
   SELECT id,slow_rank() as slow,fast_rank() as fast FROM index
           WHERE MATCH('some common query terms') ORDER BY fast DESC, slow DESC␣
   ↪LIMIT 20
           OPTION max_matches=1000;
   ```

   With subselects the query can be rewritten as :

   ```
   SELECT * FROM
           (SELECT id,slow_rank() as slow,fast_rank() as fast FROM index WHERE
                   MATCH('some common query terms')
                   ORDER BY fast DESC LIMIT 100 OPTION max_matches=1000)
   ORDER BY slow DESC LIMIT 20;
   ```

   In the initial query the slow_rank() UDF is computed for the entire match result set. With subselects, only fast_rank() is computed for the entire match result set, while slow_rank() is only computed for a limited set.

2. The second case comes handy for large result set coming from a distributed index.

   For this query:

   ```
   SELECT * FROM my_dist_index WHERE some_conditions LIMIT 50000;
   ```

   If we have 20 nodes, each node can send back to master a number of 50K records, resulting in **20 x 50K = 1M records**, however as the master sends back only 50K (out of 1M), it might be good enough for us for the nodes to send only the top 10K records. With subselect we can rewrite the query as:

   ```
   SELECT * FROM
            (SELECT * FROM my_dist_index WHERE some_conditions LIMIT 10000)
    ORDER by some_attr LIMIT 50000;
   ```

In this case, the nodes receive only the inner query and execute. This means the master will receive only *20x10K=200K records*. The master will take all the records received, reorder them by the OUTER clause and return the best 50K records. The subselect help reducing the traffic between the master and the nodes and also reduce the master's computation time (as it process only 200K instead of 1M).

## 8.32 SELECT @@system_variable syntax

```
SELECT @@system_variable [LIMIT [offset,] row_count]
```

This is currently a placeholder query that does nothing and reports success. That is in order to keep compatibility with frameworks and connectors that automatically execute this statement.

## 8.33 SET syntax

```
SET [GLOBAL] server_variable_name = value
SET [INDEX index_name] GLOBAL @user_variable_name = (int_val1 [, int_val2, ...])
SET NAMES value
SET @@dummy_variable = ignored_value
```

SET statement modifies a variable value. The variable names are case-insensitive. No variable value changes survive server restart.

SET NAMES statement and SET @@variable_name syntax, both introduced do nothing. They were implemented to maintain compatibility with 3rd party MySQL client libraries, connectors, and frameworks that may need to run this statement when connecting.

There are the following classes of the variables:

1. per-session server variable
2. global server variable
3. global user variable
4. global distributed variable

Global user variables are shared between concurrent sessions. Currently, the only supported value type is the list of BIGINTs, and these variables can only be used along with IN() for filtering purpose. The intended usage scenario is uploading huge lists of values to `searchd` (once) and reusing them (many times) later, saving on network overheads. Global user variables might be either transferred to all agents of distributed index or set locally in case of local index defined at distibuted index. Example:

```
// in session 1
mysql> SET GLOBAL @myfilter=(2,3,5,7,11,13);
Query OK, 0 rows affected (0.00 sec)

// later in session 2
mysql> SELECT * FROM test1 WHERE group_id IN @myfilter;
+------+--------+----------+------------+-----------------+------+
| id   | weight | group_id | date_added | title           | tag  |
+------+--------+----------+------------+-----------------+------+
|    3 |      1 |        2 | 1299338153 | another doc     | 15   |
|    4 |      1 |        2 | 1299338153 | doc number four | 7,40 |
+------+--------+----------+------------+-----------------+------+
2 rows in set (0.02 sec)
```

Per-session and global server variables affect certain server settings in the respective scope. Known per-session server variables are:

- `AUTOCOMMIT = {0 | 1}`
- Whether any data modification statement should be implicitly wrapped by BEGIN and COMMIT.
- `COLLATION_CONNECTION = collation_name`
- Selects the collation to be used for ORDER BY or GROUP BY on string values in the subsequent queries. Refer to *the section called "Collations" <collations>* for a list of known collation names.
- `CHARACTER_SET_RESULTS = charset_name`
- Does nothing; a placeholder to support frameworks, clients, and connectors that attempt to automatically enforce a charset when connecting to a Manticore server.
- `SQL_AUTO_IS_NULL = value`
- Does nothing; a placeholder to support frameworks, clients, and connectors that attempt to automatically enforce a charset when connecting to a Manticore server.
- `SQL_MODE = value`
- Does nothing; a placeholder to support frameworks, clients, and connectors that attempt to automatically enforce a charset when connecting to a Manticore server.
- `PROFILING = {0 | 1}`
- Enables query profiling in the current session. Defaults to 0. See also *SHOW PROFILE syntax*.

Known global server variables are:

- `QUERY_LOG_FORMAT = {plain | sphinxql}`
- Changes the current log format.
- `LOG_LEVEL = {info | debug | debugv | debugvv}`
- Changes the current log verboseness level.
- `QCACHE_MAX_BYTES = <value>`
- Changes the *query cache* RAM use limit to a given value.
- `QCACHE_THRESH_MSEC = <value>`
- Changes the *query cache* minimum wall time threshold to a given value.
- `QCACHE_TTL_SEC = <value>`
- Changes the *query cache* TTL for a cached result to a given value.
- `MAINTENANCE = {0 | 1}`
- When set to 1, puts the server in maintenance mode. Only clients with vip connections can execute queries in this mode. All new non-vip incoming connections are refused.
- `GROUPING_IN_UTC = {0 | 1}`
- When set to 1, cause timed grouping functions (day(), month(), year(), yearmonth(), yearmonthday()) to be calculated in utc. Read the doc for *grouping_in_utc* config params for more details.

Examples:

```
mysql> SET autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> SET GLOBAL query_log_format=sphinxql;
Query OK, 0 rows affected (0.00 sec)
```

## 8.34 SET TRANSACTION syntax

```
SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SERIALIZABLE }
```

SET TRANSACTION statement does nothing. It was implemented to maintain compatibility with 3rd party MySQL client libraries, connectors, and frameworks that may need to run this statement when connecting.

Example:

```
mysql> SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
Query OK, 0 rows affected (0.00 sec)
```

## 8.35 SHOW AGENT STATUS

```
SHOW AGENT ['agent'|'index'] STATUS [ LIKE pattern ]
```

Displays the statistic of *remote agents* or distributed index. It includes the values like the age of the last request, last answer, the number of different kind of errors and successes, etc. The statistic is shown for every agent for last 1, 5 and 15 intervals, each of them of *ha_period_karma* seconds. The command exists only in sphinxql.

```
mysql> SHOW AGENT STATUS;
+-----------------------------------+----------------------------+
| Variable_name                     | Value                      |
+-----------------------------------+----------------------------+
| status_period_seconds             | 60                         |
| status_stored_periods             | 15                         |
| ag_0_hostname                     | 192.168.0.202:6713         |
| ag_0_references                   | 2                          |
| ag_0_lastquery                    | 0.41                       |
| ag_0_lastanswer                   | 0.19                       |
| ag_0_lastperiodmsec               | 222                        |
| ag_0_errorsarow                   | 0                          |
| ag_0_1periods_query_timeouts      | 0                          |
| ag_0_1periods_connect_timeouts    | 0                          |
| ag_0_1periods_connect_failures    | 0                          |
| ag_0_1periods_network_errors      | 0                          |
| ag_0_1periods_wrong_replies       | 0                          |
| ag_0_1periods_unexpected_closings | 0                          |
| ag_0_1periods_warnings            | 0                          |
| ag_0_1periods_succeeded_queries   | 27                         |
| ag_0_1periods_msecsperquery       | 232.31                     |
| ag_0_5periods_query_timeouts      | 0                          |
| ag_0_5periods_connect_timeouts    | 0                          |
```

(continues on next page)

(continued from previous page)

```
| ag_0_5periods_connect_failures     | 0                          |
| ag_0_5periods_network_errors       | 0                          |
| ag_0_5periods_wrong_replies        | 0                          |
| ag_0_5periods_unexpected_closings  | 0                          |
| ag_0_5periods_warnings             | 0                          |
| ag_0_5periods_succeeded_queries    | 146                        |
| ag_0_5periods_msecsperquery        | 231.83                     |
| ag_1_hostname                      | 192.168.0.202:6714         |
| ag_1_references                    | 2                          |
| ag_1_lastquery                     | 0.41                       |
| ag_1_lastanswer                    | 0.19                       |
| ag_1_lastperiodmsec                | 220                        |
| ag_1_errorsarow                    | 0                          |
| ag_1_1periods_query_timeouts       | 0                          |
| ag_1_1periods_connect_timeouts     | 0                          |
| ag_1_1periods_connect_failures     | 0                          |
| ag_1_1periods_network_errors       | 0                          |
| ag_1_1periods_wrong_replies        | 0                          |
| ag_1_1periods_unexpected_closings  | 0                          |
| ag_1_1periods_warnings             | 0                          |
| ag_1_1periods_succeeded_queries    | 27                         |
| ag_1_1periods_msecsperquery        | 231.24                     |
| ag_1_5periods_query_timeouts       | 0                          |
| ag_1_5periods_connect_timeouts     | 0                          |
| ag_1_5periods_connect_failures     | 0                          |
| ag_1_5periods_network_errors       | 0                          |
| ag_1_5periods_wrong_replies        | 0                          |
| ag_1_5periods_unexpected_closings  | 0                          |
| ag_1_5periods_warnings             | 0                          |
| ag_1_5periods_succeeded_queries    | 146                        |
| ag_1_5periods_msecsperquery        | 230.85                     |
+------------------------------------+----------------------------+
50 rows in set (0.01 sec)
```

An optional LIKE clause is supported. Refer to *SHOW META syntax* for its syntax details.

```
mysql> SHOW AGENT STATUS LIKE '%5period%msec%';
+----------------------------+--------+
| Key                        | Value  |
+----------------------------+--------+
| ag_0_5periods_msecsperquery | 234.72 |
| ag_1_5periods_msecsperquery | 233.73 |
| ag_2_5periods_msecsperquery | 343.81 |
+----------------------------+--------+
3 rows in set (0.00 sec)
```

You can specify a particular agent by its address. In this case only that agent's data will be displayed. Also, `agent_` prefix will be used instead of `ag_N_`:

```
mysql> SHOW AGENT '192.168.0.202:6714' STATUS LIKE '%15periods%';
+-----------------------------------+--------+
| Variable_name                     | Value  |
+-----------------------------------+--------+
| agent_15periods_query_timeouts    | 0      |
| agent_15periods_connect_timeouts  | 0      |
| agent_15periods_connect_failures  | 0      |
```

(continues on next page)

(continued from previous page)

```
| agent_15periods_network_errors      | 0      |
| agent_15periods_wrong_replies       | 0      |
| agent_15periods_unexpected_closings | 0      |
| agent_15periods_warnings            | 0      |
| agent_15periods_succeeded_queries   | 439    |
| agent_15periods_msecsperquery       | 231.73 |
+-------------------------------------+--------+
9 rows in set (0.00 sec)
```

Finally, you can check the status of the agents in a specific distributed index. It can be done with a `SHOW AGENT 'index' STATUS` statement. That statement shows the index HA status (ie. whether or not it uses agent mirrors at all), and then the mirror information (specifically: address, blackhole and persistent flags, and the mirror selection probability used when one of the *weighted-probability strategies* is in effect).

```
mysql> SHOW AGENT dist_index STATUS;
+-------------------------------------+--------------------------------+
| Variable_name                       | Value                          |
+-------------------------------------+--------------------------------+
| dstindex_1_is_ha                    | 1                              |
| dstindex_1mirror1_id                | 192.168.0.202:6713:loc         |
| dstindex_1mirror1_probability_weight | 0.372864                       |
| dstindex_1mirror1_is_blackhole      | 0                              |
| dstindex_1mirror1_is_persistent     | 0                              |
| dstindex_1mirror2_id                | 192.168.0.202:6714:loc         |
| dstindex_1mirror2_probability_weight | 0.374635                       |
| dstindex_1mirror2_is_blackhole      | 0                              |
| dstindex_1mirror2_is_persistent     | 0                              |
| dstindex_1mirror3_id                | dev1.sphinxsearch.com:6714:loc |
| dstindex_1mirror3_probability_weight | 0.252501                       |
| dstindex_1mirror3_is_blackhole      | 0                              |
| dstindex_1mirror3_is_persistent     | 0                              |
+-------------------------------------+--------------------------------+
13 rows in set (0.00 sec)
```

## 8.36 SHOW CHARACTER SET syntax

```
SHOW CHARACTER SET
```

This is currently a placeholder query that does nothing and reports that a UTF-8 character set is available. It was added in order to keep compatibility with frameworks and connectors that automatically execute this statement.

```
mysql> SHOW CHARACTER SET;
+---------+---------------+-------------------+--------+
| Charset | Description   | Default collation | Maxlen |
+---------+---------------+-------------------+--------+
| utf8    | UTF-8 Unicode | utf8_general_ci   | 3      |
+---------+---------------+-------------------+--------+
1 row in set (0.00 sec)
```

## 8.37 SHOW COLLATION syntax

```
SHOW COLLATION
```

This is currently a placeholder query that does nothing and reports success. That is in order to keep compatibility with frameworks and connectors that automatically execute this statement.

```
mysql> SHOW COLLATION;
Query OK, 0 rows affected (0.00 sec)
```

## 8.38 SHOW DATABASES syntax

```
SHOW DATABASES
```

This is a dummy statement to support MySQL Workbench and other clients that require it. Currently, it does absolutely nothing.

## 8.39 SHOW INDEX SETTINGS syntax

```
SHOW INDEX index_name[.N | CHUNK N] SETTINGS
```

Displays per-index settings in a `sphinx.conf` compliant file format, similar to the *–dumpconfig* option of the indextool. The report provides a breakdown of all the index settings, including tokenizer and dictionary options. You may also specify a particular *chunk number* for the RT indexes.

## 8.40 SHOW INDEX STATUS syntax

```
SHOW INDEX index_name STATUS
```

Displays various per-index statistics. Currently, those include:

- **indexed_documents** and **indexed_bytes**, number of the documents indexed and their text size in bytes, respectively.
- **field_tokens_XXX**, sums of per-field lengths (in tokens) over the entire index (that is used internally in BM25A and BM25F functions for ranking purposes). Only available for indexes built with index_field_lengths=1.
- **ram_bytes**, total size (in bytes) of the RAM-resident index portion.
- queries time statistics of last 1 minute, 5 minutes, 15 minutes and total since daemon start;data is encapsulated as a JSON object which includes number of queries, min,max,avg,95 and 99 percentile values.
- queries found rows statistics of last 1 minute, 5 minutes, 15 minutes and total since daemon start;data is encapsulated as a JSON object which includes number of queries, min,max,avg,95 and 99 percentile values.

```
mysql> SHOW INDEX lj STATUS;
+--------------------+--------------+
| Variable_name      | Value        |
+--------------------+--------------+
| index_type         | disk         |
```

(continues on next page)

(continued from previous page)

```
| indexed_documents   | 2495219     |
| indexed_bytes       | 10380483879 |
| field_tokens_title | 6999145     |
| field_tokens_body   | 1501825050  |
| total_tokens        | 1508824195  |
| ram_bytes           | 305963599   |
| disk_bytes          | 5455804365  |
| mem_limit           | 536870912   |
+---------------------+-------------+
8 rows in set (0.00 sec)
```

## 8.41 SHOW META syntax

```
SHOW META [ LIKE pattern ]
```

**SHOW META** shows additional meta-information about the latest query such as query time and keyword statistics. IO and CPU counters will only be available if searchd was started with –iostats and –cpustats switches respectively. Additional predicted_time, dist_predicted_time, [{local|dist}]*fetched*[{docs|hits|skips}] counters will only be available if searchd was configured with *predicted time costs* and query had predicted_time in OPTION clause.

```
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     |
| cpu_time             | 0.350 |
| io_read_time         | 0.004 |
| io_read_ops          | 2     |
| io_read_kbytes       | 0.4   |
| io_write_time        | 0.000 |
| io_write_ops         | 0     |
| io_write_kbytes      | 0.0   |
```

(continues on next page)

(continued from previous page)

```
| agents_cpu_time       | 0.000 |
| agent_io_read_time    | 0.000 |
| agent_io_read_ops     | 0     |
| agent_io_read_kbytes  | 0.0   |
| agent_io_write_time   | 0.000 |
| agent_io_write_ops    | 0     |
| agent_io_write_kbytes | 0.0   |
+-----------------------+-------+
12 rows in set (0.00 sec)
```

You can also use the optional LIKE clause. It lets you pick just the variables that match a pattern. The pattern syntax is that of regular SQL wildcards, that is, '%' means any number of any characters, and '_' means a single character:

```
mysql> SHOW META LIKE 'total%';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| total                | 3     |
| total_found          | 3     |
+----------------------+-------+
2 rows in set (0.00 sec)
```

SHOW META can be used after executing a *CALL PQ* statement. In this case, it provides a different *output*.

## 8.42 SHOW PLAN syntax

```
SHOW PLAN
```

SHOW PLAN displays the execution plan of the previous SELECT statement. The plan gets generated and stored during the actual execution, so profiling must be enabled in the current session **before** running that statement. That can be done with a `SET profiling=1` statement.

Here's a complete instrumentation example:

```
mysql> SET profiling=1 \G
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT id FROM lj WHERE MATCH('the i') LIMIT 1 \G
*************************** 1\. row ***************************
id: 39815
1 row in set (1.53 sec)

mysql> SHOW PLAN \G
*************************** 1\. row ***************************
Variable: transformed_tree
   Value: AND(
  AND(KEYWORD(the, querypos=1)),
  AND(KEYWORD(i, querypos=2)))
1 row in set (0.00 sec)
```

And here's a less trivial example that shows how the actually evaluated query tree can be rather different from the original one because of expansions and other transformations:

```
mysql> SELECT * FROM test WHERE MATCH('@title abc* @body hey') \G SHOW PLAN \G
...
*************************** 1\. row ***************************
Variable: transformed_tree
   Value: AND(
  OR(fields=(title), KEYWORD(abcx, querypos=1, expanded), KEYWORD(abcm, querypos=1,␣
↪expanded)),
  AND(fields=(body), KEYWORD(hey, querypos=2)))
1 row in set (0.00 sec)
```

## 8.43 SHOW PLUGINS syntax

```
SHOW PLUGINS
```

Displays all the loaded plugins and UDFs. "Type" column should be one of the udf, ranker, index_token_filter, or query_token_filter. "Users" column is the number of thread that are currently using that plugin in a query. "Extra" column is intended for various additional plugin-type specific information; currently, it shows the return type for the UDFs and is empty for all the other plugin types.

```
mysql> SHOW PLUGINS;
+------+----------+-----------------+-------+-------+
| Type | Name     | Library         | Users | Extra |
+------+----------+-----------------+-------+-------+
| udf  | sequence | udfexample.dll | 0     | INT   |
+------+----------+-----------------+-------+-------+
1 row in set (0.00 sec)
```

## 8.44 SHOW PROFILE syntax

```
SHOW PROFILE
```

SHOW PROFILE shows a detailed execution profile of the previous SQL statement executed in the current SphinxQL session. Also, profiling must be enabled in the current session **before** running the statement to be instrumented. That can be done with a `SET profiling=1` statement. By default, profiling is disabled to avoid potential performance implications, and therefore the profile will be empty.

Here's a complete instrumentation example:

```
mysql> SET profiling=1;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT id FROM lj WHERE MATCH('the test') LIMIT 1;
+--------+
| id     |
+--------+
| 946418 |
+--------+
1 row in set (0.05 sec)

mysql> SHOW PROFILE;
+--------------+----------+----------+
```

(continues on next page)

(continued from previous page)

```
| Status       | Duration | Switches |
+--------------+----------+----------+
| unknown      | 0.000610 | 6        |
| net_read     | 0.000007 | 1        |
| dist_connect | 0.000036 | 1        |
| sql_parse    | 0.000048 | 1        |
| dict_setup   | 0.000001 | 1        |
| parse        | 0.000023 | 1        |
| transforms   | 0.000002 | 1        |
| init         | 0.000401 | 3        |
| open         | 0.000104 | 1        |
| read_docs    | 0.001570 | 71       |
| read_hits    | 0.003936 | 222      |
| get_docs     | 0.029837 | 1347     |
| get_hits     | 0.000548 | 1433     |
| filter       | 0.000619 | 1274     |
| rank         | 0.009892 | 2909     |
| sort         | 0.001562 | 52       |
| finalize     | 0.000250 | 1        |
| dist_wait    | 0.000000 | 1        |
| aggregate    | 0.000145 | 1        |
| net_write    | 0.000031 | 1        |
+--------------+----------+----------+
20 rows in set (0.00 sec)
```

Status column briefly describes where exactly (in which state) was the time spent. Duration column shows the wall clock time, in seconds. Switches column displays the number of times query engine changed to the given state. Those are just logical engine state switches and **not** any OS level context switches nor function calls (even though some of the sections can actually map to function calls) and they do **not** have any direct effect on the performance. In a sense, number of switches is just a number of times when the respective instrumentation point was hit.

States in the profile are returned in a prerecorded order that roughly maps (but is **not** identical) to the actual query order.

A list of states may (and will) vary over time, as we refine the states. Here's a brief description of the currently profiled states.

- **unknown**, generic catch-all state. Accounts for both not-yet-instrumented code, or just small miscellaneous tasks that do not really belong in any other state, but are too small to deserve their own state.
- **net_read**, reading the query from the network (that is, the application).
- **io**, generic file IO time.
- **dist_connect**, connecting to remote agents in the distributed index case.
- **sql_parse**, parsing the SphinxQL syntax.
- **dict_setup**, dictionary and tokenizer setup.
- **parse**, parsing the full-text query syntax.
- **transforms**, full-text query transformations (wildcard and other expansions, simplification, etc).
- **init**, initializing the query evaluation.
- **open**, opening the index files.
- **read_docs**, IO time spent reading document lists.
- **read_hits**, IO time spent reading keyword positions.

- **get_docs**, computing the matching documents.
- **get_hits**, computing the matching positions.
- **filter**, filtering the full-text matches.
- **rank**, computing the relevance rank.
- **sort**, sorting the matches.
- **finalize**, finalizing the per-index search result set (last stage expressions, etc).
- **dist_wait**, waiting for the remote results from the agents in the distributed index case.
- **aggregate**, aggregating multiple result sets.
- **net_write**, writing the result set to the network.

## 8.45 SHOW STATUS syntax

```
SHOW STATUS [ LIKE pattern ]
```

**SHOW STATUS** displays 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;
+--------------------+-------+
| Counter            | 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   |
```

(continues on next page)

(continued from previous page)

```
+--------------------+-------+
29 rows in set (0.00 sec)
```

An optional LIKE clause is supported. Refer to *SHOW META syntax* for its syntax details.

## 8.46 SHOW TABLES syntax

```
SHOW TABLES [ LIKE pattern ]
```

SHOW TABLES statement enumerates all currently active indexes along with their types. Existing index types are `local`, `distributed`, `rt`,and `template` respectively. Example:

```
mysql> SHOW TABLES;
+-------+-------------+
| Index | Type        |
+-------+-------------+
| dist1 | distributed |
| rt    | rt          |
| test1 | local       |
| test2 | local       |
+-------+-------------+
4 rows in set (0.00 sec)
```

An optional LIKE clause is supported. Refer to *SHOW META syntax* for its syntax details.

```
mysql> SHOW TABLES LIKE '%4';
+-------+-------------+
| Index | Type        |
+-------+-------------+
| dist4 | distributed |
+-------+-------------+
1 row in set (0.00 sec)
```

## 8.47 SHOW THREADS syntax

```
SHOW THREADS [ OPTION columns=width ]
```

SHOW THREADS lists all currently active client threads, not counting system threads. It returns a table with columns that describe:

- **thread id**
- **connection protocol**, possible values are sphinxapi and sphinxql
- **thread state**, possible values are handshake, net_read, net_write, query, net_idle
- **time** since the current state was changed (in seconds, with microsecond precision)
- **information** about queries

The 'Info' column will be cut at the width you've specified in the 'columns=width' option (notice the third row in the example table below). This column will contain raw SphinxQL queries and, if there are API queries, full text syntax and comments will be displayed. With an API-snippet, the data size will be displayed along with the query. This

column will also contain active system thread started with SYSTEM and time since current iteration started in system endless loop.

```
mysql> SHOW THREADS OPTION columns=50;
+------+----------+-------+----------+--------------------------------------------------
↪----+
| Tid  | Proto    | State | Time     | Info                                                ⌟
↪    |
+------+----------+-------+----------+--------------------------------------------------
↪----+
| 5168 | sphinxql | query | 0.000002 | show threads option columns=50                      ⌟
↪    |
| 5175 | sphinxql | query | 0.000002 | select * from rt where match ( 'the box' )          ⌟
↪    |
| 1168 | sphinxql | query | 0.000002 | select * from rt where match ( 'the box and         ⌟
↪faximi |
| 9580 | -        | -     | 0.019280 | SYSTEM OPTIMIZE                                     ⌟
↪    |
+------+----------+-------+----------+--------------------------------------------------
↪----+
3 row in set (0.00 sec)
```

## 8.48 SHOW VARIABLES syntax

```
SHOW [{GLOBAL | SESSION}] VARIABLES [WHERE variable_name='xxx']
```

**SHOW VARIABLES** statement was added to improve compatibility with 3rd party MySQL connectors and frameworks that automatically execute this statement.

It returns the current values of a few server-wide variables. Also, support for GLOBAL and SESSION clauses was added.

```
mysql> SHOW GLOBAL VARIABLES;
+----------------------+----------+
| Variable_name        | Value    |
+----------------------+----------+
| autocommit           | 1        |
| collation_connection | libc_ci  |
| query_log_format     | sphinxql |
| log_level            | info     |
+----------------------+----------+
4 rows in set (0.00 sec)
```

Support for WHERE variable_name clause was added, to help certain connectors.

## 8.49 SHOW WARNINGS syntax

```
SHOW WARNINGS
```

**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)
```

## 8.50 TRUNCATE RTINDEX syntax

```
TRUNCATE RTINDEX rtindex
```

TRUNCATE RTINDEX clears the RT index completely. It disposes the in-memory data, unlinks all the index data files, and releases the associated binary logs.

```
mysql> TRUNCATE RTINDEX rt;
Query OK, 0 rows affected (0.05 sec)
```

You may want to use this if you are using RT indices as "delta index" files; when you build the main index, you need to wipe the delta index, and thus TRUNCATE RTINDEX. You also need to use this command before attaching an index; see *ATTACH INDEX syntax*.

## 8.51 UPDATE syntax

```
UPDATE index SET col1 = newval1 [, ...] WHERE where_condition [OPTION opt_name = opt_
↪value [, ...]]
```

Multiple attributes and values can be specified in a single statement. Both RT and disk indexes are supported.

All attributes types (int, bigint, float, MVA), except for strings and JSON attributes, can be dynamically updated.

`where_condition` has the same syntax as in the SELECT statement (see *SELECT syntax* for details).

When assigning the out-of-range values to 32-bit attributes, they will be trimmed to their lower 32 bits without a prompt. For example, if you try to update the 32-bit unsigned int with a value of 4294967297, the value of 1 will actually be stored, because the lower 32 bits of 4294967297 (0x100000001 in hex) amount to 1 (0x00000001 in hex).

MVA values sets for updating (and also for INSERT or REPLACE, refer to *INSERT and REPLACE syntax*) must be specified as comma-separated lists in parentheses. To erase the MVA value, just assign () to it.

UPDATE can be used to update integer and float values in JSON array. No strings, arrays and other types yet.

```
mysql> UPDATE myindex SET enabled=0 WHERE id=123;
Query OK, 1 rows affected (0.00 sec)

mysql> UPDATE myindex
  SET bigattr=-100000000000,
    fattr=3465.23,
    mvattr1=(3,6,4),
    mvattr2=()
  WHERE MATCH('hehe') AND enabled=1;
Query OK, 148 rows affected (0.01 sec)
```

OPTION clause. This is a Manticore specific extension that lets you control a number of per-update options. The syntax is:

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

The list of allowed options are the same as for *SELECT* statement. Specifically for UPDATE statement you can use these options:

- 'ignore_nonexistent_columns' - points that the update will silently ignore any warnings about trying to update a column which is not exists in current index schema.

  'strict' - this option is used while updating JSON attributes. It's possible to update just some types in JSON. And if you try to update, for example, array type you'll get error with 'strict' option on and warning otherwise.

# CHAPTER 9

# HTTP API reference

Manticore search daemon supports HTTP protocol and can be accessed with regular HTTP clients. Available only with `workers = thread_pool` (see *workers*). To enabled the HTTP protocol, a *listen* directive with http specified as a protocol needs to be declared:

```
listen = localhost:8080:http
```

Supported endpoints:

## 9.1 /search API

Allows a simple full-text search, parameters can be : * index (index or list of indexes) * match (equivalent of MATCH()) * select (as SELECT clause) * group (grouping attribute) * order (SQL-like sorting) * limit (equivalent of LIMIT 0,N)

Response is a JSON document containing an array of attrs,matches and meta similar with the SphinxAPI response.

```
curl -X POST 'http://manticoresearch:9308/search/'
-d 'index=forum&match=@subject php manticore&select=id,subject,author_id&limit=5'
```

```
{
   "attrs":[
          "forum_id",
          "author_id",
          "subject",
          "id"
   ],
   "matches":[

   ],
   "meta":{
          "total":0,
          "total_found":0,
```

(continues on next page)

(continued from previous page)

```
            "time":0.000,
            "words":[
                    {
                            "word":"php",
                            "docs":3252,
                            "hits":11166
                    },
                    {
                            "word":"manticore",
                            "docs":0,
                            "hits":0
                    }
            ]
    }
}
```

## 9.2 /sql API

Allows running a SELECT SphinxQL, set as query parameter.

Response is a JSON document containing an array of attrs,matches and meta similar with the SphinxAPI response.

```
 curl -X POST 'http://manticoresearch:9308/sql/'
-d "query=select id,subject,author_id  from forum where match('@subject php manticore
↪') group by
 author_id order by id desc limit 0,5"
```

```
{
    "attrs":[
            "forum_id",
            "author_id",
            "subject",
            "id",
            "@groupby",
            "@count"
    ],
    "matches":[

    ],
    "meta":{
            "total":123,
            "total_found":123,
            "time":0.087,
            "words":[
                    {
                            "word":"php",
                            "docs":3252,
                            "hits":11166
                    },
                    {
                            "word":"manticore",
                            "docs":1242,
                            "hits":4352
                    }
```

(continues on next page)

(continued from previous page)

```
            ]
    }
}
```

## 9.3 /json API

This endpoint expects request body with queries defined as JSON document. Responds with JSON documents containing result and/or information about executed query.

**Warning:** Please note that this endpoint is in preview stage. Some functionalities are not yet complete and syntax may suffer changes in future. Read careful changelog of future updates to avoid possible breakages.

### 9.3.1 json/bulk

The `json/bulk` endpoint allows you to perform several insert, update or delete operations in a single call. This endpoint only works with data that has `Content-Type` set to `application/x-ndjson`. The data itself should be formatted as a newline-delimited json (NDJSON). Basically it means that each line should contain exactly one json statement and end with a newline `\n` and maybe a `\r`.

Example:

```
{ "insert" : { "index" : "test", "id" : 1, "doc": { "gid" : 10, "content" : "doc one"␣
↪} } }
{ "insert" : { "index" : "test", "id" : 2, "doc": { "gid" : 20, "content" : "doc two"␣
↪} } }
```

This inserts two documents to index `test`. Each statement starts with an action type (in this case, `insert`). Here's a list of the supported actions:

- `insert`: Inserts a document. Syntax is the same as in `json/insert`.
- `create`: a synonym for `insert`
- `replace`: Replaces a document. Syntax is the same as in `json/replace`.
- `index`: a synonym for `replace`
- `update`: Updates a document. Syntax is the same as in `json/update`.
- `delete`: Deletes a document. Syntax is the same as in `json/delete`.

Updates by query and deletes by query are also supported.

Example:

```
{ "update" : { "index" : "test", "doc": { "tag" : 1000 }, "query": { "range": { "price
↪": { "gte": 1000 } } } } }
{ "delete" : { "index" : "test", "query": { "range": { "price": { "lt": 1000 } } } } }
```

Note that the bulk operation stops at the first query that results in an error.

### 9.3.2 json/delete

This endpoint allows you to delete documents from indexes, similar to SphinxQL's *DELETE syntax*.

Example:

```
{
  "index":"test",
  "id":1
}
```

The daemon will respond with a JSON object stating if the operation was successfull or not:

```
 {
  "_index": "test",
  "_id": 1,
  "found": true,
  "result": "deleted"
}
```

This deletes a document that has and id of 1 from an index named `test`.

As in `json/update`, you can do a delete by query.

```
{
  "index":"test",

  "query":
  {
    "match": { "*": "apple" }
  }
}
```

This deletes all documents that match a given query.

### 9.3.3 json/insert

Documents can be inserted into RT indexes using the `/json/insert` endpoint. As with SphinxQL's *INSERT and REPLACE syntax*, documents with ids that already exist will not be overwritten. You can also use the `/json/create` endpoint, it's a synonym for `json/insert`.

Here's how you can index a simple document:

```
{
  "index":"test",
  "id":1
}
```

This creates a document with an id specified by `id` in an index specified by the `index` property. This document has empty fulltext fields and all attributes are set to their default values. However, you can use the optional `doc` property to set field and attribute values:

```
{
  "index":"test",
  "id":1,
  "doc":
  {
```

(continues on next page)

(continued from previous page)

```
    "gid" : 10,
    "content" : "new document"
  }
}
```

The daemon will respond with a JSON object stating if the operation was successfull or not:

```
 {
  "_index": "test",
  "_id": 1,
  "created": true,
  "result": "created"
}
```

MVA attributes are inserted as arrays of numbers. JSON attributes can be inserted either as JSON objects or as strings containing escaped JSON:

```
{
  "index":"test",
  "id":1,
  "doc":
  {
    "mva" : [1,2,3,4,5],
    "json1":
    {
      "string": "name1",
      "int": 1,
      "array" : [100,200],
      "object": {}
    },
    "json2": "{\"string\":\"name2\",\"int\":2,\"array\":[300,400],\"object\":{}}",
    "content" : "new document"
  }
}
```

### 9.3.4 json/pq

Percolate are accepted at `/json/pq` endpoint. Here is an example:

```
curl -X POST 'http://manticoresearch:9308//json/pq/index_name/search'
    -d '{}'
```

to list of stored queries at `"index_name"` percolate index.

#### Store query

Query might be inserted:

- with `ID` auto generated - at endpoint `json/pq/index_name/doc`
- with `ID` explicitly set - at endpoint `json/pq/index_name/doc/ID`

To replace already stored query `ID` should be provided along with `refresh=1` argument, such as `json/pq/index_pq_1/doc/2?refresh=1`

There is 2 formats of full-text queries that might be stored into index:

- query in `json\search` compartible format, described at *json/search*
- query in `SpinxQL` compartible format, described at *extended query syntax*

`tags` and `filters` also might be stored along with query, for details refer to *Tags* However there is no way to mix `json\search` native filters with `filters` field, only one type of filter might be used per query.

Example of `json\search` query with `tags`:

```
PUT json/pq/idx_pq_1/doc
{
        "query": { "match": { "title": "test" } },
        "tags": ["fid", "fid_x1"]
}
```

Example of `json\search` query there terms combined via `and` operator:

```
PUT json/pq/idx_pq_1/doc
{
        "query": { "match": { "title": { "query": "cat test", "operator": "and" } } }
}
```

Example of `json\search` query with native filters:

```
PUT json/pq/idx_pq_1/doc
{
        "query":
        {
                "match": { "title": "tree" },
                "range": { "gid": { "lt": 3 } }
        }
}
```

Example of `json\search` boolean query:

```
PUT json/pq/idx_pq_1/doc
{
        "query":
        {
                "bool":
                {
                        "must": [
                                { "match": { "title": "tree" } },
                                { "match": { "title": "test" } } ]
                }
        }
}
```

Example of `json\search` query with `SphinxQL` filters and `ID` set:

```
PUT json/pq/idx_pq_1/doc/17
{
        "query":
        {
                "match": { "title": "tree" }
        },
        "filters": "gid < 3 or zip = 049"
}
```

Example of `Sphinx` query with filters and tags that repalces already stored query with 2nd `ID`:

```
PUT json/pq/idx_pq_1/doc/2?refresh=1
{
        "query":
        {
                "ql": "(test me !he) || (testing place)"
        },
        "filters": "zip IN (1,7,9)",
        "tags": ["zip", "location", "city"]
}
```

The responce:

```
{
        "index": "idx_pq_1",
        "type": "doc",
        "_id": "2",
        "result": "created"
}
```

there `result` field got value `created` for inserted query or value `updated` for query that got successfully replaced.

### Search matching document

To search for queries matching document(s) the *_search* endpoint with body should be queried

Example of single document matching:

```
POST json/pq/idx_pq_1/_search
{
        "query":
        {
                "percolate":
                {
                        "document" : { "title" : "some text to match" }
                }
        }
}
```

The responce:

```
{
        "timed_out": false,
        "hits": {
                "total": 2,
                "max_score": 1,
                "hits": [
                        {
                                "_index": "idx_pq_1",
                                "_type": "doc",
                                "_id": "2",
                                "_score": "1",
                                "_source": {
                                        "query": {
                                                "match": {
                                                        "title": "some"
                                                },
```

(continues on next page)

(continued from previous page)

```
                                        }
                                }
                        },
                        {
                                "_index": "idx_pq_1",
                                "_type": "doc",
                                "_id": "5",
                                "_score": "1",
                                "_source": {
                                        "query": {
                                                "ql": "some | none"
                                        }
                                }
                        }
                ]
        }
}
```

there queries matched located at `hits` array with their `ID` at `_id` field and full-text part at `_source` field.

Example of multiple documents matching:

```
POST json/pq/idx_pq_1/_search
{
        "query":
        {
                "percolate":
                {
                        "documents" :
                        [
                                { "title" : "some text to match" },
                                { "title" : "another text to match" },
                                { "title" : "new document to match" }
                        ]
                }
        }
}
```

The responce:

```
{
        "timed_out": false,
        "hits": {
                "total": 1,
                "max_score": 1,
                "hits": [
                {
                        "_index": "idx_pq_1",
                        "_type": "doc",
                        "_id": "3",
                        "_score": "1",
                        "_source": {
                                "query": {
                                        "match": {
                                                "title": "text"
                                        }
                                }
```

(continues on next page)

(continued from previous page)

```
                        },
                        "fields": {
                                "_percolator_document_slot": [
                                        1,
                                        2
                                ]
                        }
                } ]
        }
}
```

there queries matched located at `hits` array and documents matched for each query is located at `fields` object `_percolator_document_slot` array.

### List stored queries

*_search* endpoint without body shows all stored queries in index, similar to SphinxQL's *List stored queries*.

Example:

```
POST /json/pq/idx_pq_1/search
{
}
```

The responce:

```
{
        "timed_out": false,
        "hits": {
                "total": 4,
                "max_score": 1,
                "hits": [
                        {
                                "_index": "idx_pq_1",
                                "_type": "doc",
                                "_id": "1",
                                "_score": "1",
                                "_source": {
                                        "query": {
                                                "bool": {
                                                        "must": [
                                                                {
                                                                        "match": {
                                                                                "title
↪": "tree"
                                                                        }
                                                                },
                                                                {
                                                                        "match": {
                                                                                "title
↪": "test"
                                                                        }
                                                                }
                                                        ]
                                                }
                                        }
```

(continues on next page)

(continued from previous page)

```
                        }
                },
                {
                        "_index": "idx_pq_1",
                        "_type": "doc",
                        "_id": "2",
                        "_score": "1",
                        "_source": {
                                "query": {
                                        "match": {
                                                "title": "tree"
                                        },
                                        "range": {
                                                "gid": {
                                                        "lt": 3
                                                }
                                        }
                                }
                        }
                },
                {
                        "_index": "idx_pq_1",
                        "_type": "doc",
                        "_id": "4",
                        "_score": "1",
                        "_source": {
                                "query": {
                                        "ql": "tree !new"
                                }
                        }
                },
                {
                        "_index": "idx_pq_1",
                        "_type": "doc",
                        "_id": "5",
                        "_score": "1",
                        "_source": {
                                "query": {
                                        "ql": "new | old"
                                }
                        }
                }
        ]
    }
}
```

There `hits` contains queries stored at percolate index with query `ID` at `_id` field and `_source` field is full text query in `SpinxQL` compartible format, described at *extended query syntax* or `json\search` compartible format, described at *jsonsearch*

### Delete stored queries

This endpoint allows to delete queries from index, similar to SphinxQL's *Delete query*. Either id or tags lists supported

Example:

```
DELETE json/pq/idx_pq_1/_delete_by_query
{
        "id": [2, 10]
}
```

The daemon will respond with a JSON object stating if the operation was successful or not:

```
{
        "timed_out": false,
        "deleted": 2,
        "total": 2,
        "failures": []
}
```

This deletes 2 documents from an index named `idx_pq_1`.

### 9.3.5 json/replace

`json/replace` works similar to SphinxQL's *INSERT and REPLACE syntax*. It inserts a new document into an index and if the index already has a document with the same id, it is deleted before the new document is inserted. There's also a synonym endpoint, `json/index`.

```
{
  "index":"test",
  "id":1,
  "doc":
  {
    "gid" : 10,
    "content" : "updated document"
  }
}
```

The daemon will respond with a JSON object stating if the operation was successfull or not:

```
 {
  "_index": "test",
  "_id": 1,
  "created": false,
  "result": "updated"
}
```

### 9.3.6 json/search

Searches are accepted at `/json/search` endpoint. Here's an example of a simple query:

```
curl -X POST 'http://manticoresearch:9308/json/search'
    -d '{"index":"test","query":{"match":{"title":"keyword"}}}'
```

`"index"` clause sets the list of indexes to search through. You can specify a single index: `"index": "test"`, a comma-separated list of indexes: `"index": "test1,test2,test3"` or use `_all` or `*` to issue the query to all available indexes:

```
"index": "_all"
"index": "*"
```

"query" clause contains fulltext queries (if any) and filters. It can be used to organize queries and filters into a tree (using the bool query).

### Fulltext queries

The following fulltext queries are supported:

match

"match" is a simple query that matches the specified keywords in the specified fields

```
"query":
{
  "match": { "field": "keyword" }
}
```

Just as in case of indexes, you can specify a list of fields:

```
"match":
{
  "field1,field2": "keyword"
}
```

Or you can use "_all" or "*" to search all fields.

You can search all fields except one using "!field":

```
"match":
{
  "!field1": "keyword"
}
```

By default keywords are combined using the OR operator. However, you can change that behaviour using the "operator" clause:

```
"query":
{
  "match":
  {
    "content,title":
    {
      "query":"keyword",
      "operator":"or"
    }
  }
}
```

"operator" can be set to "or" or "and".

match_phrase

"match_phrase" is a query that matches the entire phrase. It is similar to a phrase operator in SphinxQL. Here's an example:

```
"query":
{
  "match_phrase": { "_all" : "had grown quite" }
}
```

match_all

"match_all" is a query that matches all documents. The syntax looks like this:

```
"query":
{
  "match_all": {}
}
```

It can be used to create fullscan queries. However, it is not required as you can just specify the filters without a fulltext query.

### Bool queries

A bool query matches documents matching boolean combinations of other queries and/or filters. Queries and filters must be specified in "must", "should" or "must_not" sections. Example:

```
{
  "index":"test",
  "query":
  {
    "bool":
    {
      "must":
      [
        { "match": {"_all":"keyword"} },
        { "range": { "int_col": { "gte": 14 } } }
      ]
    }
  }
}
```

"must"

Queries and filters specified in the "must" section must match the documents. If several fulltext queries or filters are specified, all of them. This is the equivalent of AND queries in SphinxQL.

"should"

Queries and filters specified in the "should" section should match the documents. If some queries are specified in "must" or "must_not", "should" queries are ignored. On the other hand, if there are no queries other than "should", then at least one of these queries must match a document for it to match the bool query. This is the equivalent of OR queries.

"must_not"

Queries and filters specified in the "must_not" section must not match the documents. If several queries are specified under "must_not", the document matches if none of them match.

Example:

```
{
  "index": "test1",
  "query":
  {
    "bool":
    {
      "must":
      {
```

(continues on next page)

(continued from previous page)

```
        "match" : { "_all" : "product" }
      },
      "must_not":
      [
        { "match": {"_all":"phone"} },
        { "range": { "price": { "gte": 500 } } }
      ]
    }
  }
}
```

### Filters

JSON queries have two distinct entities: fulltext queries and filters. Both can be organised in a tree (using a bool query), but for now filters work only for the root element of the query. For example:

```
{
  "index":"test",
  "query": { "range": { "price": { "lte": 11 } } }
}
```

Here's an example of several filters in a `bool` query:

```
{
  "index": "test1",
  "query":
  {
    "bool":
    {
      "must":
      [
        { "match" : { "_all" : "product" } },
        { "range": { "price": { "gte": 500, "lte": 1000 } } },
      ],
      "must_not":
      {
        "range": { "revision": { "lt": 15 } }
      }
    }
  }
}
```

This is a fulltext query that matches all the documents containing `product` in any field. These documents must have a `price` greather or equal than 500 (`"gte"`) and less or equal than 1000 (`"lte"`). All of these documents must not have a `revision` less than 15 (`"lt"`).

The following types of filters are supported:

**Equality filters**

Equality filters are the simplest filters that work with integer, float and string attributes. Example:

```
{
  "index":"test1",
  "query":
  {
```

(continues on next page)

(continued from previous page)

```
        "equals": { "price": 500 }
  }
}
```

**Range filters**

Range filters match documents that have attribute values within a specified range. Example:

```
{
  "index":"test1",
  "query":
  {
    "range":
    {
      "price":
      {
        "gte": 500,
        "lte": 1000
      }
    }
  }
}
```

Range filters support the following properties:

- `gte`: value must be greater than or equal to
- `gt`: value must be greater than
- `lte`: value must be less than or equal to
- `lt`: value must be less

**Geo distance filters**

geo_distance filters are used to filter the documents that are within a specific distance from a geo location.

Example:

```
{
  "index":"test",
  "query":
  {
    "geo_distance":
    {
      "location_anchor": {"lat":49, "lon":15},
      "location_source": {"attr_lat, attr_lon"},
      "distance_type": "adaptive",
      "distance":"100 km"
    }
      }
}
```

- `location_anchor`: specifies the pin location, in degrees. Distances are calculated from this point.
- `location_source`: specifies the attributes that contain latitude and longitude.
- `distance_type`: specifies distance calculation function. Can be either `adaptive` or `haversine`. `adaptive` is faster and more precise, for more details see *GEODIST()*. Optional, defaults to `adaptive`.

- `distance`: specifies the maximum distance from the pin locations. All documents within this distance match. The distance can be specified in various units. If no unit is specified, the distance is assumed to be in meters. Here is a list of supported distance units:
  - Meter: `m` or `meters`
  - Kilometer: `km` or `kilometers`
  - Centimeter: `cm` or `centimeters`
  - Millimeter: `mm` or `millimeters`
  - Mile: `mi` or `miles`
  - Yard: `yd` or `yards`
  - Feet: `ft` or `feet`
  - Inch: `in` or `inch`
  - Nautical mile: `NM`, `nmi` or `nauticalmiles`

`location_anchor` and `location_source` properties accept the following latitude/longitude formats:

- an object with lat and lon keys: `{ "lat":"attr_lat", "lon":"attr_lon" }`
- a string of the following structure: `"attr_lat,attr_lon"`
- an array with the latitude and longitude in the following order: `[attr_lon, attr_lat]`

Latitude and longitude are specified in degrees.

### Sorting

#### Sorting by attributes

Query results can be sorted by one or more attributes. Example:

```
{
  "index":"test",
  "query":
  {
    "match": { "title": "what was" }
  },
  "sort": [ "_score", "id" ]
}
```

`"sort"` specifies an array of attibutes and/or additional properties. Each element of the array can be an attribute name or `"_score"` if you want to sort by match weights. In that case sort order defaults to ascending for attributes and descending for `_score`.

You can also specify sort order explicitly. Example:

```
"sort":
[
  { "price":"asc" },
  "id"
]
```

- `asc`: sort in ascending order
- `desc`: sort in descending order

You can also use another syntax and specify sort order via the `order` property:

```
"sort":
[
  { "gid": { "order":"desc" } }
]
```

Sorting by MVA attributes is also supported in JSON queries. Sorting mode can be set via the `mode` property. The following modes are supported:

- `min`: sort by minimum value
- `max`: sort by maximum value

Example:

```
"sort":
[
  { "attr_mva": { "order":"desc", "mode":"max" } }
]
```

When sorting on an attribute, match weight (score) calculation is disabled by default (no ranker is used). You can enable weight calculation by setting the `track_scores` property to true:

```
{
  "index":"test",
  "track_scores":true,
  "query": { "match": { "title": "what was" } },
  "sort": [ { "gid": { "order":"desc" } } ]
}
```

**Sorting by geo distance**

Matches can be sorted by their distance from a specified location. Example:

```
{
  "index": "test",
  "query": { "match_all": {} },
  "sort":
  [
    {
      "_geo_distance":
      {
        "location_anchor": {"lat":41, "lon":32},
        "location_source": [ "attr_lon", "attr_lat" ],
        "distance_type": "adaptive"
      }
    }
  ]
}
```

`location_anchor` property specifies the pin location, `location_source` specifies the attributes that contain latitude and longitude and `distance_type` selects distance computation function (optional, defaults to "arc").

### Expressions

Expressions are supported via `script_fields`:

```
{
  "index": "test",
  "query": { "match_all": {} },
  "script_fields":
  {
    "add_all": { "script": { "inline": "( gid * 10 ) | crc32(title)" } },
    "title_len": { "script": { "inline": "crc32(title)" } }
  }
}
```

In this example two expressions are created: `add_all` and `title_len`. First expression calculates `( gid * 10 ) | crc32(title)` and stores the result in the `add_all` attribute. Second expression calculates `crc32(title)` and stores the result in the `title_len` attribute.

Only `inline` expressions are supported for now. The value of `inline` property (the expression to compute) has the same syntax as SphinxQL expressions.

### Text highlighting

Fulltext query search results can be highlighted on one or more fields. Field contents has to be stored in string attributes (for now). Here's an example:

```
{
  "index": "test",
  "query": { "match": { "content": "and first" } },
  "highlight":
  {
    "fields":
    {
      "content": {},
      "title": {}
    }
  }
}
```

As a result of this query, the values of string attributes called `content` and `title` are highlighted against the query specified in `query` clause. Highlighted snippets are added in the `highlight` property of the `hits` array:

```
{
  "took":1,
  "timed_out": false,
  "hits":
  {
    "total": 1,
    "hits":
    [
      {
        "_id": "1",
        "_score": 1625,
        "_source":
        {
          "gid": 1,
          "title": "it was itself in this way",
          "content": "first now and then at"
         },
         "highlight":
```

(continues on next page)

(continued from previous page)

```
          {
            "content": [ "<b>first</b> now <b>and</b> then at" ],
            "title": [ "" ]
          }
        }
      ]
    }
}
```

The following options are supported:

- `fields` object contains attribute names with options.
- `encoder` can be set to `default` or `html`. When set to `html`, retains html markup when highlighting. Works similar to `html_strip_mode=retain` in `CALL SNIPPETS`.
- `highlight_query` makes it possible to highlight against a query other than our search query. Syntax is the same as in the main `query`:

```
{
  "index": "test",
  "query": { "match": { "content": "and first" } },
  "highlight":
  {
    "fields": { "content": {}, "title": {} },
    "highlight_query": { "match": { "_all":"on and not" } }
  }
}
```

- `pre_tags` and `post_tags` set opening and closing tags for highlighted text snippets. They work similar to `before_match` and `after_match` options in `CALL SNIPPETS`. Optional, defaults are `<b>` and `</b>`. Example:

```
"highlight":
{
  "fields": { "content": {} },
  "pre_tags": "before_",
  "post_tags": "_after"
}
```

- `no_match_size` works similar to `allow_empty` in `CALL SNIPPETS`. If set to zero value, acts as `allow_empty=1`, e.g. allows empty string to be returned as highlighting result when a snippet could not be generated. Otherwise, the beginning of the field will be returned. Optional, default is 0. Example:

```
"highlight":
{
  "fields": { "content": {} },
  "no_match_size": 0
}
```

- `order`: if set to `"score"`, sorts the extracted passages in order of relevance. Optional. Works similar to `weight_order` in `CALL SNIPPETS`. Example:

```
"highlight":
{
  "fields": { "content": {} },
```

(continues on next page)

(continued from previous page)

```
  "order": "score"
}
```

- `fragment_size` sets maximum fragment size in symbols. Can be global or per-field. Per-field options override global options. Optional, default is 256. Works similar to `limit` in `CALL SNIPPETS`. Example of per-field usage:

```
"highlight":
{
  "fields": { "content": { "fragment_size": 100 } },
}
```

  Example of global usage:

```
"highlight":
{
  "fields": { "content": {} },
  "fragment_size": 100
}
```

- `number_of_fragments`: Limits the maximum number of fragments in a snippet. Just as `fragment_size`, can be global or per-field. Optional, default is 0 (no limit). Works similar to `limit_passages` in `CALL SNIPPETS`.

### Result set format

Query result is sent as a JSON document. Example:

```
{
  "took":10
  "timed_out": false,
  "hits":
  {
    "total": 2,
    "hits":
    [
      {
        "_id": "1",
        "_score": 1,
        "_source": { "gid": 11 }
      },
      {
        "_id": "2",
        "_score": 1,
        "_source": { "gid": 12 }
      }
    ]
  }
}
```

- `took`: time in milliseconds it took to execute the search
- `timed_out`: if the query timed out or not
- `hits`: search results. has the following properties:
  - `total`: total number of matching documents

  - `hits`: an array containing matches

Query result can also include query profile information, see *Query profile*.

Each match in the `hits` array has the following properties:

- `_id`: match id
- `_score`: match weight, calculated by ranker
- `_source`: an array containing the attributes of this match. By default all attributes are included. However, this behaviour can be changed, see below

You can use the `_source` property to select the fields you want to be included in the result set. Example:

```
{
  "index":"test",
  "_source":"attr*",
  "query": { "match_all": {} }
}
```

You can specify the attributes which you want to include in the query result as a string (`"_source": "attr*"`) or as an array of strings (`"_source": [ "attr1", "attri*" ]`). Each entry can be an attribute name or a wildcard (`*`, `%` and `?` symbols are supported).

You can also explicitly specify which attributes you want to include and which to exlude from the result set using the `includes` and `excludes` properties:

```
"_source":
{
  "includes": [ "attr1", "attri*" ],
  "excludes": [ "*desc*" ]
}
```

An empty list of includes is interpreted as "include all attributes" while an empty list of excludes does not match anything. If an attribute matches both the includes and excludes, then the excludes win.

### Query profile

You can view the final transformed query tree with all normalized keywords by adding a `"profile":true` property:

```
{
  "index":"test",
  "profile":true,
  "query":
  {
    "match_phrase": { "_all" : "had grown quite" }
  }
}
```

This feature is somewhat similar to `SHOW PLAN` statement in SphinxQL. The result appears as a `profile` property in the result set. For example:

```
"profile":
{
  "query":
  {
    "type": "PHRASE",
    "description": "PHRASE( AND(KEYWORD(had, querypos=1)), AND(KEYWORD(grown,␣
↪querypos=2)), AND(KEYWORD(quite, querypos=3)))",
```

(continues on next page)

(continued from previous page)


```
    "children":
    [
      {
        "type": "AND",
        "description": "AND(KEYWORD(had, querypos=1))",
        "max_field_pos": 0,
        "children":
        [
          {
            "type": "KEYWORD",
            "word": "had",
            "querypos": 1
           }
        ]
      },
      {
        "type": "AND",
        "description": "AND(KEYWORD(grown, querypos=2))",
        "max_field_pos": 0,
        "children":
        [
          {
            "type": "KEYWORD",
            "word": "grown",
            "querypos": 2
          }
        ]
      },
      {
        "type": "AND",
        "description": "AND(KEYWORD(quite, querypos=3))",
        "max_field_pos": 0,
        "children":
        [
          {
            "type": "KEYWORD",
            "word": "quite",
            "querypos": 3
          }
        ]
      }
    ]
  }
}
```


`query` property contains the transformed fulltext query tree. Each node contains:

- `type`: node type. Can be `AND`, `OR`, `PHRASE`, `KEYWORD` etc.
- `description`: query subtree for this node shown as a string (in `SHOW PLAN` format)
- `children`: child nodes, if any
- `max_field_pos`: maximum position within a field
- `word`: transformed keyword. Keyword nodes only.
- `querypos`: position of this keyword in a query. Keyword nodes only.
- `excluded`: keyword excluded from query. Keyword nodes only.

- `expanded`: keyword added by prefix expansion. Keyword nodes only.
- `field_start`: keyword must occur at the very start of the field. Keyword nodes only.
- `field_end`: keyword must occur at the very end of the field. Keyword nodes only.
- `boost`: keyword IDF will be multiplied by this. Keyword nodes only.

### 9.3.7 json/update

This endpoint allows you to update attribute values in documents, same as SphinxQL's *UPDATE syntax*. Syntax is similar to `json/insert`, but this time the `doc` property is mandatory.

Example:

```
{
  "index":"test",
  "id":1,
  "doc":
  {
    "gid" : 100,
    "price" : 1000
  }
}
```

The daemon will respond with a JSON object stating if the operation was successfull or not:

```
 {
  "_index": "test",
  "_id": 1,
  "result": "updated"
}
```

The id of the document that needs to be updated can be set directly using the `id` property (as in the example above) or you can do an update by query and apply the update to all the documents that match the query:

```
{
  "index":"test",
  "doc":
  {
    "price" : 1000
  },

  "query":
  {
    "match": { "*": "apple" }
  }
}
```

Query syntax is the same as in the `json/search` endpoint. Note that you can't specify `id` and `query` at the same time.

# CHAPTER 10

# API reference

There is a number of native searchd client API implementations for Manticore. 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) Manticore 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.

## 10.1 General API functions

### 10.1.1 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.

### 10.1.2 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 successfully 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.

### 10.1.3 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.

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.

### 10.1.4 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.

### 10.1.5 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.

### 10.1.6 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).

## 10.2 General query settings

### 10.2.1 SetSelect

**Prototype:** function SetSelect ( $clause )

Sets the select clause, listing specific attributes to fetch, and *Sorting modes* to compute and fetch. Clause syntax mimics SQL.

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. Manticore 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.

When using GROUP BY agregate functions (AVG(), MIN(), MAX(), SUM()) are supported.

Expression sorting (*Sorting modes*) and geodistance functions (*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" );
```

### 10.2.2 SetLimits

**Prototype:** function SetLimits ( $offset, $limit, $max_matches=1000, $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 Manticore 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.

### 10.2.3 SetMaxQueryTime

**Prototype:** function SetMaxQueryTime ( $max_query_time )

Sets maximum search query time, in milliseconds. Parameter must be a non-negative integer. Default value 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.

### 10.2.4 SetOverride

**DEPRECATED**

**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.

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.

## 10.3 Full-text search query settings

### 10.3.1 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 *Search results ranking* 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 *search_results_ranking*) 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.

### 10.3.2 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 Manticore 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.

### 10.3.3 SetMatchMode

**DEPRECATED**

**Prototype:** function SetMatchMode ( $mode )

Sets full-text query matching mode, as described in *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
```

### 10.3.4 SetRankingMode

**Prototype:** function SetRankingMode ( $ranker, $rankexpr="" )

Sets ranking mode (aka ranker). Only available in SPH_MATCH_EXTENDED matching mode. Parameter must be a constant specifying one of the known rankers.

By default, in the EXTENDED matching mode Manticore computes two factors which contribute to the final match weight. The major part is a phrase proximity value between the document text and the query. 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. The list of the modes is available in *Search results ranking*.

`$rankexpr` argument lets you specify a ranking formula to use with the *expression based ranker <expression_based_ranker_sphrank_expr>*, that is, when `$ranker` is set to SPH_RANK_EXPR. In all other cases, `$rankexpr` is ignored.

### 10.3.5 SetSortMode

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

Set matches sorting mode, as described in *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
```

### 10.3.6 SetWeights

**Prototype:** function SetWeights ( $weights )

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

## 10.4 Result set filtering settings

### 10.4.1 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).

### 10.4.2 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).

### 10.4.3 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).

### 10.4.4 SetFilterString

**Prototype:** function SetFilterString ( $attribute, $value, $exclude=false )

Adds new string value filter.

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

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

### 10.4.5 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.

### 10.4.6 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. Manticore will compute geosphere distance between the given anchor point and a point specified by latitude and longitude 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.

## 10.5 GROUP BY settings

### 10.5.1 SetGroupBy

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

Sets grouping attribute, function, and groups sorting mode; and enables grouping (as described in *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 *Sorting modes*.

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 *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.

Aggregate functions (AVG(), MIN(), MAX(), SUM()) are supported through *SetSelect()* API call when using GROUP BY.

Grouping on string attributes is supported, with respect to current collation.

### 10.5.2 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 Manticore 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.

## 10.6 Querying

### 10.6.1 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 queries in a single batch and Manticore 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 manually if you need then.

### 10.6.2 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) and underscore (_); everything else is considered a separator. Note that index name should not start with underscore character. 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 processed on server).
- "words":
- Hash which maps query keywords (case-folded, stemmed, and otherwise processed) to a small hash with per-keyword statistics ("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 analogous to a single `AddQuery()` call, followed by a corresponding `RunQueries()`, then returning the first array element of matches (from the first, and only, query.)

### 10.6.3 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 succeeds - 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 `&quot;error&quot;` 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 successfully connect to `searchd`, submit the batch, and receive the results - but every result set will have a specific error message.

### 10.6.4 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.

### 10.6.5 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.

## 10.7 Additional functionality

### 10.7.1 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. `$words` is a string that contains the keywords to highlight. They will be processed with respect to index settings. For instance, if English stemming is enabled in the index, `shoes` will be highlighted even if keyword is `shoe`. Keywords can contain wildcards, that work similarly to star-syntax available in queries. `$opts` is a hash which contains additional optional highlighting parameters:

- `before_match`: A string to insert before a keyword match. A %PASSAGE_ID% macro can be used in this string. The first match of the macro is replaced with an incrementing passage number within a current snippet. Numbering starts at 1 by default but can be overridden with `start_passage_id` option. In a multi-document call, %PASSAGE_ID% would restart at every given document. Default is `**`.
- `after_match`: A string to insert after a keyword match. Starting with version 1.10-beta, a %PASSAGE_ID% macro can be used in this string. Default is `**`.
- `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.
- `use_boundaries`: Whether to additionally break passages by phrase boundary characters, as configured in index settings with *phrase_boundary* directive. 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.
- `query_mode`: Whether to handle $words as a query in *extended syntax*, or as a bag of words (default behavior). For instance, in query mode (`one two|three four`) will only highlight and include those occurrences `one two` or `three four` when the two words from each pair are adjacent to each other. In default mode, any single occurrence of `one`, `two`, `three`, or `four` would be highlighted. Boolean, default is false.
- `force_all_words`: Ignores the snippet length limit until it includes all the keywords. Boolean, default is false.
- `limit_passages`: Limits the maximum number of passages that can be included into the snippet. Integer, default is 0 (no limit).
- `limit_words`: Limits the maximum number of words that can be included into the snippet. Note the limit applies to any words, and not just the matched keywords to highlight. For example, if we are highlighting `Mary` and a passage `Mary had a little lamb` is selected, then it contributes 5 words to this limit, not just 1. Integer, default is 0 (no limit).
- `start_passage_id`: Specifies the starting value of %PASSAGE_ID% macro (that gets detected and expanded in `before_match`, `after_match` strings). Integer, default is 1.
- `load_files`: Whether to handle $docs as data to extract snippets from (default behavior), or to treat it as file names, and load data from specified files on the server side. Up to *dist_threads* worker threads per request will be created to parallelize the work when this flag is enabled. Boolean, default is false. Building of the snippets could be parallelized between remote agents. Just set the *'dist_threads'* param in the config to the value greater than 1, and then invoke the snippets generation over the distributed index, which contain only one(!) *local* agent and several remotes. The *snippets_file_prefix* option is also in the game and the final filename is calculated by concatenation of the prefix with given name. Otherwords, when snippets_file_prefix is '/var/data' and filename is 'text.txt' the sphinx will try to generate the snippets from the file '/var/datatext.txt', which is exactly '/var/data' + 'text.txt'.
- `load_files_scattered`: It works only with distributed snippets generation with remote agents. The source files for snippets could be distributed among different agents, and the main daemon will merge together all non-erroneous results. So, if one agent of the distributed index has 'file1.txt', another has 'file2.txt' and you call for the snippets with both these files, the sphinx will merge results from the agents together, so you will get the snippets from both 'file1.txt' and 'file2.txt'. Boolean, default is false.

  If the `load_files` is also set, the request will return the error in case if any of the files is not available anywhere. Otherwise (if `load_files` is not set) it will just return the empty strings for all absent files. The master instance reset this flag when distributes the snippets among agents. So, for agents the absence of a file is

not critical error, but for the master it might be so. If you want to be sure that all snippets are actually created, set both `load_files_scattered` and `load_files`. If the absence of some snippets caused by some agents is not critical for you - set just `load_files_scattered`, leaving `load_files` not set.

- `html_strip_mode`: HTML stripping mode setting. Defaults to `index`, which means that index settings will be used. The other values are `none` and `strip`, that forcibly skip or apply stripping irregardless of index settings; and `retain`, that retains HTML markup and protects it from highlighting. The `retain` mode can only be used when highlighting full documents and thus requires that no snippet size limits are set. String, allowed values are `none`, `strip`, `index`, and `retain`.
- `allow_empty`: Allows empty string to be returned as highlighting result when a snippet could not be generated (no keywords match, or no passages fit the limit). By default, the beginning of original text would be returned instead of an empty string. Boolean, default is false.
- `passage_boundary`: Ensures that passages do not cross a sentence, paragraph, or zone boundary (when used with an index that has the respective indexing settings enabled). String, allowed values are `sentence`, `paragraph`, and `zone`.
- `emit_zones`: Emits an HTML tag with an enclosing zone name before each passage. Boolean, default is false.

Snippets extraction algorithm currently favors better passages (with closer phrase matches), and then passages with keywords not yet in snippet. Generally, it will try to highlight the best match with the query, and it will also to highlight all the query keywords, as made possible by the limits. In case the document does not match the query, beginning of the document trimmed down according to the limits will be return by default. You can also return an empty snippet instead case by setting `allow_empty` option to true.

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

### 10.7.2 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 );
```

### 10.7.3 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" );
```

### 10.7.4 FlushAttributes

**Prototype:** function FlushAttributes ()

Forces `searchd` to flush pending attribute updates to disk, and blocks until completion. Returns a non-negative internal `flush tag` on success. Returns -1 and sets an error message on error.

Attribute values updated using *UpdateAttributes()* API call are only kept in RAM until a so-called flush (which writes the current, possibly updated attribute values back to disk). FlushAttributes() call lets you enforce a flush. The call will block until `searchd` finishes writing the data to disk, which might take seconds or even minutes depending on the total data size (.spa file size). All the currently updated indexes will be flushed.

Flush tag should be treated as an ever growing magic number that does not mean anything. It's guaranteed to be non-negative. It is guaranteed to grow over time, though not necessarily in a sequential fashion; for instance, two calls that return 10 and then 1000 respectively are a valid situation. If two calls to FlushAttrs() return the same tag, it means that there were no actual attribute updates in between them, and therefore current flushed state remained the same (for all indexes).

Usage example:

```
$status = $cl->FlushAttributes ();
if ( $status<0 )
    print "ERROR: " . $cl->GetLastError();
```

### 10.7.5 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";
```

### 10.7.6 UpdateAttributes

**Prototype:** function UpdateAttributes ( $index, $attrs, $values, $mva=false, $ignorenonexistent=false )

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. `$values` is a hash where key is document ID, and value is a plain array of new attribute values. Optional boolean parameter `mva` points that there is update of MVA attributes. In this case the values must be a dict with int key (document ID) and list of lists of int values (new MVA attribute values). Optional boolean parameter `$ignorenonexistent` points that the update will silently ignore any warnings about trying to update a column which is not exists in current index schema.

`$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 ini-

tiated 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.

## 10.8 Persistent connections

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

### 10.8.1 Open

**Prototype:** function Open ()

Opens persistent connection to the server.

### 10.8.2 Close

**Prototype:** function Close ()

Closes previously opened persistent connection.

# CHAPTER 11

# Configuration reference

## 11.1 Common section configuration options

### 11.1.1 lemmatizer_base

Lemmatizer dictionaries base path. Optional, default is /usr/local/share (as in –datadir switch to ./configure script).

Our lemmatizer implementation (see *morphology* for a discussion of what lemmatizers are) is dictionary driven. lemmatizer_base directive configures the base dictionary path. File names are hardcoded and specific to a given lemmatizer; the Russian lemmatizer uses ru.pak dictionary file. The dictionaries can be obtained from the Manticore website.

Example:

```
lemmatizer_base = /usr/local/share/sphinx/dicts/
```

### 11.1.2 progressive_merge

Merge Real-Time index chunks during OPTIMIZE operation from smaller to bigger. Progressive merge merger faster and reads/write less data. Enabled by default. If disabled, chunks are merged from first to last created.

### 11.1.3 json_autoconv_keynames

Whether and how to auto-convert key names within JSON attributes. Known value is 'lowercase'. Optional, default value is unspecified (do not convert anything).

When this directive is set to 'lowercase', key names within JSON attributes will be automatically brought to lower case when indexing. This conversion applies to any data source, that is, JSON attributes originating from either SQL or XMLpipe2 sources will all be affected.

Example:

```
json_autoconv_keynames = lowercase
```

### 11.1.4 json_autoconv_numbers

Automatically detect and convert possible JSON strings that represent numbers, into numeric attributes. Optional, default value is 0 (do not convert strings into numbers).

When this option is 1, values such as "1234" will be indexed as numbers instead of strings; if the option is 0, such values will be indexed as strings. This conversion applies to any data source, that is, JSON attributes originating from either SQL or XMLpipe2 sources will all be affected.

Example:

```
json_autoconv_numbers = 1
```

### 11.1.5 on_json_attr_error

What to do if JSON format errors are found. Optional, default value is `ignore_attr` (ignore errors). Applies only to `sql_attr_json` attributes.

By default, JSON format errors are ignored (`ignore_attr`) and the indexer tool will just show a warning. Setting this option to `fail_index` will rather make indexing fail at the first JSON format error.

Example:

```
on_json_attr_error = ignore_attr
```

### 11.1.6 plugin_dir

Trusted location for the dynamic libraries (UDFs). Optional, default is empty (no location).

Specifies the trusted directory from which the *UDF libraries* can be loaded. Requires *workers* = *thread* <*workers*> to take effect.

Example:

```
plugin_dir = /usr/local/sphinx/lib
```

### 11.1.7 rlp_environment

RLP environment configuration file. Mandatory if RLP is used.

Example:

```
rlp_environment = /home/myuser/RLP/rlp-environment.xml
```

### 11.1.8 rlp_max_batch_docs

Maximum number of documents batched before processing them by the RLP. Optional, default is 50. This option has effect only if `morphology = rlp_chinese_batched` is specified.

Example:

```
rlp_max_batch_docs = 100
```

### 11.1.9 rlp_max_batch_size

Maximum total size of documents batched before processing them by the RLP. Optional, default is 51200. Do not set this value to more than 10Mb because sphinx splits large documents to 10Mb chunks before processing them by the RLP. This option has effect only if `morphology = rlp_chinese_batched` is specified.

Example:

```
rlp_max_batch_size = 100k
```

### 11.1.10 rlp_root

Path to the RLP root folder. Mandatory if RLP is used.

Example:

```
rlp_root = /home/myuser/RLP
```

## 11.2 Data source configuration options

### 11.2.1 csvpipe_delimiter

csvpipe source fields delimiter. Optional, default value is ‘,’.

Example:

```
csvpipe_delimiter = ;
```

### 11.2.2 mssql_winauth

MS SQL Windows authentication flag. Boolean, optional, default value is 0 (false). Applies to `mssql` source type only.

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
```

### 11.2.3 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
```

### 11.2.4 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
```

### 11.2.5 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
```

### 11.2.6 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**.

Example:

```
sql_attr_bigint = my_bigint_id
```

### 11.2.7 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
```

### 11.2.8 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
```

### 11.2.9 sql_attr_json

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

When indexing JSON attributes, Manticore expects a text field with JSON formatted data. JSON attributes supports arbitrary JSON data with no limitation in nested levels or types.

```
{
    "id": 1,
    "gid": 2,
    "title": "some title",
    "tags": [
        "tag1",
        "tag2",
        "tag3"
        {
            "one": "two",
            "three": [4, 5]
        }
    ]
}
```

These attributes allow Manticore to work with documents without a fixed set of attribute columns. When you filter on a key of a JSON attribute, documents that don't include the key will simply be ignored.

Example:

```
sql_attr_json = properties
```

### 11.2.10 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] \
    [;RANGED-QUERY]
```

where

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

if using 'ranged-main-query' SOURCE-TYPE then omit the RANGED-QUERY and it will automatically use the same query from 'sql_query_range' (useful option in complex inheritance setups to save having to manually duplicate the same query many times)

Example:

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

### 11.2.11 sql_attr_string

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

String attributes can store arbitrary strings attached to every document. There's a fixed size limit of 4 MB per value. Also, `searchd` will currently cache all the values in RAM, which is an additional implicit limit.

String attributes can be used for sorting and grouping(ORDER BY, GROUP BY, WITHIN GROUP ORDER BY). Note that attributes declared using `sql_attr_string` will **not** be full-text indexed; you can use *sql_field_string* directive for that.

Example:

```
sql_attr_string = title # will be stored but will not be indexed
```

### 11.2.12 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.

Timestamps can store date and time in the range of Jan 01, 1970 to Jan 19, 2038 with a precision of one second. The expected 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 timestamp attributes in Manticore; you need to explicitly convert such columns using UNIX_TIMESTAMP function (if data is in range).

Note timestamps can not represent dates before January 01, 1970, and UNIX_TIMESTAMP() in MySQL will not return anything expected. If you only needs to work with dates, not times, consider TO_DAYS() function in MySQL instead.

Example:

```
# sql_query = ... UNIX_TIMESTAMP(added_datetime) AS added_ts ...
sql_attr_timestamp = added_ts
```

### 11.2.13 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
```

### 11.2.14 sql_column_buffers

Per-column buffer sizes. Optional, default is empty (deduce the sizes automatically). Applies to `odbc`, `mssql` source types only.

ODBC and MS SQL drivers sometimes can not return the maximum actual column size to be expected. For instance, NVARCHAR(MAX) columns always report their length as 2147483647 bytes to `indexer` even though the actually used length is likely considerably less. However, the receiving buffers still need to be allocated upfront, and their sizes have to be determined. When the driver does not report the column length at all, Manticore allocates default 1 KB buffers for each non-char column, and 1 MB buffers for each char column. Driver-reported column length also gets clamped by an upper limit of 8 MB, so in case the driver reports (almost) a 2 GB column length, it will be clamped and a 8 MB buffer will be allocated instead for that column. These hard-coded limits can be overridden using the `sql_column_buffers` directive, either in order to save memory on actually shorter columns, or overcome the 8 MB limit on actually longer columns. The directive values must be a comma-separated lists of selected column names and sizes:

```
sql_column_buffers = <colname>=<size>[K|M] [, ...]
```

Example:

```
sql_query = SELECT id, mytitle, mycontent FROM documents
sql_column_buffers = mytitle=64K, mycontent=10M
```

### 11.2.15 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
```

### 11.2.16 sql_field_string

Combined string attribute and full-text field declaration. Multi-value (ie. there may be more than one such attribute declared), optional. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

*sql_attr_string* only stores the column value but does not full-text index it. In some cases it might be desired to both full-text index the column and store it as attribute. `sql_field_string` lets you do exactly that. Both the field and the attribute will be named the same.

Example:

```
sql_field_string = title # will be both indexed and stored
```

### 11.2.17 sql_file_field

File based field declaration. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only. Introduced in version 1.10-beta.

This directive makes `indexer` interpret field contents as a file name, and load and index the referred file. Files larger than *max_file_field_buffer* in size are skipped. Any errors during the file loading (IO errors, missed limits, etc) will be reported as indexing warnings and will **not** early terminate the indexing. No content will be indexed for such files.

Example:

```
sql_file_field = my_file_path # load and index files referred to by my_file_path
```

### 11.2.18 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 Manticore 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
```

### 11.2.19 sql_joined_field

Joined/payload field fetch query. Multi-value, optional, default is empty list of queries. Applies to SQL source types (`mysql`, `pgsql`, `mssql`) only.

`sql_joined_field` lets you use two different features: joined fields, and payloads (payload fields). It's syntax is as follows:

```
sql_joined_field = FIELD-NAME 'from'  ( 'query' | 'payload-query' | 'ranged-query' |
↪'ranged-main-query' ); \
            QUERY [ ; RANGE-QUERY ]
```

where

- FIELD-NAME is a joined/payload field name;
- QUERY is an SQL query that must fetch values to index.
- RANGE-QUERY is an optional SQL query that fetches a range of values to index.

**Joined fields** let you avoid JOIN and/or GROUP_CONCAT statements in the main document fetch query (sql_query). This can be useful when SQL-side JOIN is slow, or needs to be offloaded on Manticore side, or simply to emulate MySQL-specific GROUP_CONCAT functionality in case your database server does not support it.

The query must return exactly 2 columns: document ID, and text to append to a joined field. Document IDs can be duplicate, but they **must** be in ascending order. All the text rows fetched for a given ID will be concatenated together, and the concatenation result will be indexed as the entire contents of a joined field. Rows will be concatenated in the order returned from the query, and separating whitespace will be inserted between them. For instance, if joined field query returns the following rows:

```
( 1, 'red' )
( 1, 'right' )
( 1, 'hand' )
( 2, 'mysql' )
( 2, 'sphinx' )
```

then the indexing results would be equivalent to that of adding a new text field with a value of 'red right hand' to document 1 and 'mysql sphinx' to document 2, including the keyword positions inside the field in the order they come from the query. If the rows needs to be in a specific order, that needs to be explicitly defined in the query.

Joined fields are only indexed differently. There are no other differences between joined fields and regular text fields.

When a single query is not efficient enough or does not work because of the database driver limitations, **ranged queries** can be used. It works similar to the ranged queries in the main indexing loop, see *Ranged queries*. The range will be queried for and fetched upfront once, then multiple queries with different `$start` and `$end` substitutions will be run to fetch the actual data.

When using `ranged-main-query` query then omit the `ranged-query` and it will automatically use the same query from :ref:`sql_query_range` (useful option in complex inheritance setups to save having to manually duplicate the same query many times).

**Payloads** let you create a special field in which, instead of keyword positions, so-called user payloads are stored. Payloads are custom integer values attached to every keyword. They can then be used in search time to affect the ranking.

The payload query must return exactly 3 columns: document ID; keyword; and integer payload value. Document IDs can be duplicate, but they **must** be in ascending order. Payloads must be unsigned integers within 24-bit range, ie. from 0 to 16777215. For reference, payloads are currently internally stored as in-field keyword positions, but that is not guaranteed and might change in the future.

Currently, the only method to account for payloads is to use SPH_RANK_PROXIMITY_BM25 ranker. On indexes with payload fields, it will automatically switch to a variant that matches keywords in those fields, computes a sum of matched payloads multiplied by field weights, and adds that sum to the final rank.

Example:

```
sql_joined_field = \
    tagstext from query; \
    SELECT docid, CONCAT('tag',tagid) FROM tags ORDER BY docid ASC

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

### 11.2.20 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
```

### 11.2.21 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
```

### 11.2.22 sql_query_killlist

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

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.

Note that in the distributed index setup, K-lists are **local to every node in the cluster**. They are **not** get transmitted over the network when sending queries. (Because that might be too much of an impact when the K-list is huge.) You will need to setup a separate per-server K-lists in that case.

Example:

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

### 11.2.23 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 successfully 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. If no documents were indexed, $maxid will be expanded to 0.

Example:

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

### 11.2.24 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
```

### 11.2.25 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 executed exactly in order of appearance 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 returns. Note that Manticore accepts only UTF-8 texts. Two MySQL specific examples of setting the encoding are:

```
sql_query_pre = SET CHARACTER_SET_RESULTS=utf8
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
```

### 11.2.26 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 *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 *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
```

### 11.2.27 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**.

Example:

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

### 11.2.28 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
```

### 11.2.29 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
```

### 11.2.30 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
```

### 11.2.31 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
```

### 11.2.32 type

Data source type. Mandatory, no default value. Known types are `mysql`, `pgsql`, `mssql`, `xmlpipe2`, `tsvpipe`, `csvpipe` 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 xmlpipe2 or tsvpipe, csvpipe start with `xmlpipe_` and `tsvpipe_`, `csvpipe_` correspondingly. All source types 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
```

### 11.2.33 unpack_mysqlcompress_maxsize

Buffer size for UNCOMPRESS()ed data. Optional, default value is 16M.

When using *unpack_mysqlcompress*, due to implementation intricacies 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
```

### 11.2.34 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.

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
```

### 11.2.35 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.

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
```

### 11.2.36 xmlpipe_attr_bigint

xmlpipe signed 64-bit integer attribute declaration. Multi-value, optional. Applies to `xmlpipe2` source type only. Syntax fully matches that of *sql_attr_bigint*.

Example:

```
xmlpipe_attr_bigint = my_bigint_id
```

### 11.2.37 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
```

### 11.2.38 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
```

### 11.2.39 xmlpipe_attr_json

JSON attribute declaration. Multi-value (ie. there may be more than one such attribute declared), optional.

This directive is used to declare that the contents of a given XML tag are to be treated as a JSON document and stored into a Manticore index for later use. Refer to *sql_attr_json* for more details on the JSON attributes.

Example:

```
xmlpipe_attr_json = properties
```

### 11.2.40 xmlpipe_attr_multi_64

xmlpipe MVA attribute declaration. Declares the BIGINT (signed 64-bit integer) MVA attribute. 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_64 = taglist
```

### 11.2.41 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
```

### 11.2.42 xmlpipe_attr_string

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

This setting declares a string attribute tag in xmlpipe2 stream. The contents of the specified tag will be parsed and stored as a string value.

Example:

```
xmlpipe_attr_string = subject
```

### 11.2.43 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
```

### 11.2.44 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_id
```

### 11.2.45 xmlpipe_command

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

Specifies a command that will be executed and which output will be parsed for documents. Refer to *xmlpipe2 data source* for specific format description.

Example:

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

### 11.2.46 xmlpipe_field

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

Example:

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

### 11.2.47 xmlpipe_field_string

xmlpipe field and string attribute declaration. Multi-value, optional. Applies to `xmlpipe2` source type only. Refer to *xmlpipe2 data source*.

Makes the specified XML element indexed as both a full-text field and a string attribute. Equivalent to <sphinx:field name="field" attr="string"/> declaration within the XML file.

Example:

```
xmlpipe_field_string = subject
```

### 11.2.48 xmlpipe_fixup_utf8

Perform Manticore-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, Manticore 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
```

## 11.3 Index configuration options

### 11.3.1 agent_blackhole

Remote blackhole agent declaration in the *distributed index*. Multi-value, optional, default is empty.

`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
```

### 11.3.2 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 successfully. If the timeout is reached but connect() does not complete, and *retries* are enabled, retry will be initiated.

Example:

```
agent_connect_timeout = 300
```

### 11.3.3 agent_persistent

Persistently connected remote agent declaration. Multi-value, optional, default is empty.

`agent_persistent` directive syntax matches that of the *agent* directive. The only difference is that the master will **not** open a new connection to the agent for every query and then close it. Rather, it will keep a connection open and attempt to reuse for the subsequent queries. The maximal number of such persistent connections per one agent host is limited by *persistent_connections_limit* option of searchd section.

Note, that you **have** to set the last one in something greater than 0 if you want to use persistent agent connections. Otherwise - when *persistent_connections_limit* is not defined, it assumes the zero num of persistent connections, and 'agent_persistent' acts exactly as simple 'agent'.

Persistent master-agent connections reduce TCP port pressure, and save on connection handshakes. As of time of this writing, they are supported **only** in workers=threads and workers=threadpool mode. In other modes, simple non-persistent connections (i.e., one connection per operation) will be used, and a warning will show up in the console.

Example:

```
agent_persistent = remotebox:9312:index2
```

### 11.3.4 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
```

### 11.3.5 agent_retry_count

Integer, specifies how many times manticore will try to connect and query remote agents in distributed index before reporting fatal query error. It works the same as *agent_retry_count* in searchd section, but define the value for concrete index. See also *mirror_retry_count* option.

### 11.3.6 agent

Remote agent declaration in the *distributed index*. Multi-value, optional, default is empty.

`agent` directive declares remote agents that are searched every time when the enclosing distributed index is searched. The agents are, essentially, pointers to networked indexes. The value specifies address, and also can additionally specify multiple alternatives (agent mirrors) for either the address only, or the address and index list:

```
agent = address1 [ | address2 [...] ][:index-list]
agent = address1[:index-list [ | address2[:index-list [...] ] ] ]
```

In both cases the address specification must be one of the following:

```
address = hostname[:port] # eg. server2:9312
address = /absolute/unix/socket/path # eg. /var/run/sphinx2.sock
```

Where `hostname` is the remote host name, `port` is the remote TCP port number, `index-list` is a comma-separated list of index names, and square braces [] designate an optional clause.

When index name is omited, it is assumed the same index as the one where this line is defined. I.e. when defining agents for distributed index 'mycoolindex' you can just point the address, and it is assumed to calll 'mycoolindex' index on agent's endpoints.

When port number is omited, it is assumed to be default SphinxQL IANA port (9312). However when portnumber is pointed, but invalid (say, port 70000), it will fail (skip) such agent.

In other words, you can point every single agent to one or more remote indexes, residing on one or more networked servers. There are absolutely no restrictions on the pointers. To point out a couple important things, the host can be localhost, and the remote index can be a distributed index in turn, all that is legal. That enables a bunch of very different usage modes:

- sharding over multiple agent servers, and creating an arbitrary cluster topology;
- sharding over multiple agent servers, mirrored for HA/LB (High Availability and Load Balancing) purposes;
- sharding within localhost, to utilize multiple cores (however, it is simpler just to use multiple local indexes and dist_threads directive instead);

All agents are searched in parallel. An index list is passed verbatim to the remote agent. How exactly that list is searched within the agent (ie. sequentially or in parallel too) depends solely on the agent configuration (ie. dist_threads directive). Master has no remote control over that.

The value can additionally enumerate per agent options such as:

- *ha_strategy* - random, roundrobin, nodeads, noerrors (replaces index *ha_strategy* for particular agent)
- *conn* - pconn, persistent (same as *agent_persistent* agent declaration)
- *blackhole* - 0,1 (same as *agent_blackhole* agent declaration)
- *retry_count* - integer (same as *agent_retry_count*, but provided num will not be multiplied to number of mirrors)

```
agent = address1:index-list[[ha_strategy=value] | [conn=value] | [blackhole=value]]
```

Example:

```
# config on box2
# sharding an index over 3 servers
agent = box2:9312:chunk2
agent = box3:9312:chunk3
```

(continues on next page)

(continued from previous page)

```
# config on box2
# sharding an index over 3 servers
agent = box1:9312:chunk2
agent = box3:9312:chunk3

# config on box3
# sharding an index over 3 servers
agent = box1:9312:chunk2
agent = box2:9312:chunk3

# per agent options
agent = box1:9312:chunk1[ha_strategy=nodeads]
agent = box2:9312:chunk2[conn=pconn]
agent = test:9312:any[blackhole=1]
agent = test:9312|box2:9312|box3:9312:any2[retry_count=2]
```

### Agent mirrors

The syntax lets you define so-called **agent mirrors** that can be used interchangeably when processing a search query. Master server keeps track of mirror status (alive or dead) and response times, and does automatic failover and load balancing based on that. For example, this line:

```
agent = box1:9312|box2:9312|box3:9312:chunk2
```

declares that box1:9312, box2:9312, and box3:9312 all have an index called chunk2, and can be used as interchangeable mirrors. If any single of those servers go down, the queries will be distributed between the other two. When it gets back up, master will detect that and begin routing queries to all three boxes again.

Another way to define the mirrors is to explicitly specify the index list for every mirror:

```
agent = box1:9312:box1chunk2|box2:9312:box2chunk2
```

This works essentially the same as the previous example, but different index names will be used when querying different severs: box1chunk2 when querying box1:9312, and box2chunk when querying box2:9312.

By default, all queries are routed to the best of the mirrors. The best one is picked based on the recent statistics, as controlled by the *ha_period_karma* config directive. Master stores a number of metrics (total query count, error count, response time, etc) recently observed for every agent. It groups those by time spans, and karma is that time span length. The best agent mirror is then determined dynamically based on the last 2 such time spans. Specific algorithm that will be used to pick a mirror can be configured *ha_strategy* directive.

The karma period is in seconds and defaults to 60 seconds. Master stores up to 15 karma spans with per-agent statistics for instrumentation purposes (see *SHOW AGENT STATUS* statement). However, only the last 2 spans out of those are ever used for HA/LB logic.

When there are no queries, master sends a regular ping command every *ha_ping_interval* milliseconds in order to have some statistics and at least check, whether the remote host is still alive. ha_ping_interval defaults to 1000 msec. Setting it to 0 disables pings and statistics will only be accumulated based on actual queries.

Example:

```
# sharding index over 4 servers total
# in just 2 chunks but with 2 failover mirrors for each chunk
# box1, box2 carry chunk1 as local
# box3, box4 carry chunk2 as local
```

(continues on next page)

(continued from previous page)

```
# config on box1, box2
agent = box3:9312|box4:9312:chunk2

# config on box3, box4
agent = box1:9312|box2:9312:chunk1
```

### 11.3.7 bigram_freq_words

A list of keywords considered "frequent" when indexing bigrams. Optional, default is empty.

Bigram indexing is a feature to accelerate phrase searches. When indexing, it stores a document list for either all or some of the adjacent words pairs into the index. Such a list can then be used at searching time to significantly accelerate phrase or sub-phrase matching.

Some of the bigram indexing modes (see *bigram_index*) require to define a list of frequent keywords. These are **not** to be confused with stopwords! Stopwords are completely eliminated when both indexing and searching. Frequent keywords are only used by bigrams to determine whether to index a current word pair or not.

`bigram_freq_words` lets you define a list of such keywords.

Example:

```
bigram_freq_words = the, a, you, i
```

### 11.3.8 bigram_index

Bigram indexing mode. Optional, default is none.

Bigram indexing is a feature to accelerate phrase searches. When indexing, it stores a document list for either all or some of the adjacent words pairs into the index. Such a list can then be used at searching time to significantly accelerate phrase or sub-phrase matching.

`bigram_index` controls the selection of specific word pairs. The known modes are:

- `all`, index every single word pair. (NB: probably totally not worth it even on a moderately sized index, but added anyway for the sake of completeness.)
- `first_freq`, only index word pairs where the *first* word is in a list of frequent words (see *bigram_freq_words*). For example, with `bigram_freq_words = the, in, i, a`, indexing "alone in the dark" text will result in "in the" and "the dark" pairs being stored as bigrams, because they begin with a frequent keyword (either "in" or "the" respectively), but "alone in" would **not** be indexed, because "in" is a *second* word in that pair.
- `both_freq`, only index word pairs where both words are frequent. Continuing with the same example, in this mode indexing "alone in the dark" would only store "in the" (the very worst of them all from searching perspective) as a bigram, but none of the other word pairs.

For most usecases, `both_freq` would be the best mode, but your mileage may vary.

Example:

```
bigram_freq_words = both_freq
```

### 11.3.9 blend_chars

Blended characters list. Optional, default is empty.

Blended characters are indexed both as separators and valid characters. For instance, assume that & is configured as blended and AT&T occurs in an indexed document. Three different keywords will get indexed, namely "at&t", treating blended characters as valid, plus "at" and "t", treating them as separators.

Positions for tokens obtained by replacing blended characters with whitespace are assigned as usual, so regular keywords will be indexed just as if there was no `blend_chars` specified at all. An additional token that mixes blended and non-blended characters will be put at the starting position. For instance, if the field contents are "AT&T company" occurs in the very beginning of the text field, "at" will be given position 1, "t" position 2, "company" position 3, and "AT&T" will also be given position 1 ("blending" with the opening regular keyword). Thus, querying for either AT&T or just AT will match that document, and querying for "AT T" as a phrase also match it. Last but not least, phrase query for "AT&T company" will *also* match it, despite the position

Blended characters can overlap with special characters used in query syntax (think of T-Mobile or @twitter). Where possible, query parser will automatically handle blended character as blended. For instance, "hello @twitter" within quotes (a phrase operator) would handle @-sign as blended, because @-syntax for field operator is not allowed within phrases. Otherwise, the character would be handled as an operator. So you might want to escape the keywords.

Blended characters can be remapped, so that multiple different blended characters could be normalized into just one base form. This is useful when indexing multiple alternative Unicode codepoints with equivalent glyphs.

Example:

```
blend_chars = +, &, U+23
blend_chars = +, &->+
```

### 11.3.10 blend_mode

Blended tokens indexing mode. Optional, default is `trim_none`.

By default, tokens that mix blended and non-blended characters get indexed in there entirety. For instance, when both at-sign and an exclamation are in `blend_chars`, "@dude!" will get result in two tokens indexed: "@dude!" (with all the blended characters) and "dude" (without any). Therefore "@dude" query will *not* match it.

`blend_mode` directive adds flexibility to this indexing behavior. It takes a comma-separated list of options.

```
blend_mode = option [, option [, ...]]
option = trim_none | trim_head | trim_tail | trim_both | skip_pure
```

Options specify token indexing variants. If multiple options are specified, multiple variants of the same token will be indexed. Regular keywords (resulting from that token by replacing blended with whitespace) are always be indexed.

- trim_none
- Index the entire token.
- trim_head
- Trim heading blended characters, and index the resulting token.
- trim_tail
- Trim trailing blended characters, and index the resulting token.
- trim_both
- Trim both heading and trailing blended characters, and index the resulting token.

- skip_pure
- Do not index the token if it's purely blended, that is, consists of blended characters only.

Returning to the "@dude!" example above, setting `blend_mode = trim_head, trim_tail` will result in two tokens being indexed, "@dude" and"dude!". In this particular example, `trim_both` would have no effect, because trimming both blended characters results in"dude" which is already indexed as a regular keyword. Indexing "@U.S.A." with `trim_both` (and assuming that dot is blended two) would result in "U.S.A" being indexed. Last but not least, `skip_pure` enables you to fully ignore sequences of blended characters only. For example, "one @@@ two" would be indexed exactly as "one two", and match that as a phrase. That is not the case by default because a fully blended token gets indexed and offsets the second keyword position.

Default behavior is to index the entire token, equivalent to `blend_mode = trim_none`.

Example:

```
blend_mode = trim_tail, skip_pure
```

### 11.3.11 charset_table

Accepted characters table, with case folding rules. Optional, default value are latin and cyrillic characters.

charset_table is the main workhorse of Manticore tokenizing process, ie. the process of extracting keywords from document text or query text. 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. 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.

Aliases "english" and "russian" are allowed at control character mapping.

Example:

```
# default are English and Russian letters
charset_table = 0..9, A..Z->a..z, _, a..z, \
    U+410..U+42F->U+430..U+44F, U+430..U+44F, U+401->U+451, U+451

# english charset defined with alias
charset_table = 0..9, english, _
```

### 11.3.12 dict

The keywords dictionary type. Known values are 'crc' and 'keywords'. 'crc' is DEPRECATED. Use 'keywords' instead. Optional, default is 'keywords'.

Keywords dictionary mode (dict=keywords), (greatly) reduces indexing impact and enable substring searches on huge collections. That mode is supported both for disk and RT indexes.

CRC dictionaries never store the original keyword text in the index. Instead, keywords are replaced with their control sum value (calculated using FNV64) both when searching and indexing, and that value is used internally in the index.

That approach has two drawbacks. First, there is a chance of control sum collision between several pairs of different keywords, growing quadratically with the number of unique keywords in the index. However, it is not a big concern as a chance of a single FNV64 collision in a dictionary of 1 billion entries is approximately 1:16, or 6.25 percent. And most dictionaries will be much more compact that a billion keywords, as a typical spoken human language has in the region of 1 to 10 million word forms.) Second, and more importantly, substring searches are not directly possible with control sums. Manticore alleviated that by pre-indexing all the possible substrings as separate keywords (see *min_prefix_len*, *min_infix_len* directives). That actually has an added benefit of matching substrings in the quickest way possible. But at the same time pre-indexing all substrings grows the index size a lot (factors of 3-10x and even more would not be unusual) and impacts the indexing time respectively, rendering substring searches on big indexes rather impractical.

Keywords dictionary fixes both these drawbacks. It stores the keywords in the index and performs search-time wildcard expansion. For example, a search for a 'test*'prefix could internally expand to 'test|tests|testing' query based on the dictionary contents. That expansion is fully transparent to the application, except that the separate per-keyword statistics for all the actually matched keywords would now also be reported.

For substring (infix) search extended wildcards may be used. Special symbols like '?' and '%' are supported along with substring (infix) search (e.g. "t?st*","*run%*","*abc*"). Note, however, these wildcards work only with dict=keywords, and not elsewhere.

Indexing with keywords dictionary should be 1.1x to 1.3x slower compared to regular, non-substring indexing - but times faster compared to substring indexing (either prefix or infix). Index size should only be slightly bigger that than of the regular non-substring index, with a 1..10% percent total difference. Regular keyword searching time must be very close or identical across all three discussed index kinds (CRC non-substring, CRC substring, keywords). Substring searching time can vary greatly depending on how many actual keywords match the given substring (in other words, into how many keywords does the search term expand). The maximum number of keywords matched is restricted by the *expansion_limit* directive.

Essentially, keywords and CRC dictionaries represent the two different trade-off substring searching decisions. You can choose to either sacrifice indexing time and index size in favor of top-speed worst-case searches (CRC dictionary), or only slightly impact indexing time but sacrifice worst-case searching time when the prefix expands into very many keywords (keywords dictionary).

Example:

```
dict = keywords
```

### 11.3.13 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 Manticore 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 *Attributes* for in-depth discussion and RAM usage estimates.

Example:

```
docinfo = inline
```

### 11.3.14 embedded_limit

Embedded exceptions, wordforms, or stopwords file size limit. Optional, default is 16K.

Indexer can either save the file name, or embed the file contents directly into the index. Files sized under `embedded_limit` get stored into the index. For bigger files, only the file names are stored. This also simplifies moving index files to a different machine; you may get by just copying a single file.

With smaller files, such embedding reduces the number of the external files on which the index depends, and helps maintenance. But at the same time it makes no sense to embed a 100 MB wordforms dictionary into a tiny delta index. So there needs to be a size threshold, and `embedded_limit` is that threshold.

Example:

```
embedded_limit = 32K
```

### 11.3.15 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.

Small enough files are stored in the index header, see *embedded_limit* for details.

Short summary of the differences is as follows:

- exceptions are case sensitive, wordforms are not;
- 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 interpreted 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
```

### 11.3.16 expand_keywords

Expand keywords with exact forms and/or stars when possible. The value can additionally enumerate options such us exact and star. Optional, default is 0 (do not expand keywords).

Queries against indexes with `expand_keywords` feature enabled are internally expanded as follows. If the index was built with prefix or infix indexing enabled, every keyword gets internally replaced with a disjunction of keyword itself and a respective prefix or infix (keyword with stars). If the index was built with both stemming and

*index_exact_words* enabled, exact form is also added. Here's an example that shows how internal expansion works when all of the above (infixes, stemming, and exact words) are combined:

```
running -> ( running | *running* | =running )
```

or expansion limited by exact option even infixes enabled for index

```
running -> ( running | =running )
```

Expanded queries take naturally longer to complete, but can possibly improve the search quality, as the documents with exact form matches should be ranked generally higher than documents with stemmed or infix matches.

Note that the existing query syntax does not allow to emulate this kind of expansion, because internal expansion works on keyword level and expands keywords within phrase or quorum operators too (which is not possible through the query syntax).

This directive does not affect `indexer` in any way, it only affects `searchd`.

Example:

```
expand_keywords = 1
```

### 11.3.17 global_idf

The path to a file with global (cluster-wide) keyword IDFs. Optional, default is empty (use local IDFs).

On a multi-index cluster, per-keyword frequencies are quite likely to differ across different indexes. That means that when the ranking function uses TF-IDF based values, such as BM25 family of factors, the results might be ranked slightly different depending on what cluster node they reside.

The easiest way to fix that issue is to create and utilize a global frequency dictionary, or a global IDF file for short. This directive lets you specify the location of that file. It it suggested (but not required) to use a .idf extension. When the IDF file is specified for a given index *and* and OPTION global_idf is set to 1, the engine will use the keyword frequencies and collection documents count from the global_idf file, rather than just the local index. That way, IDFs and the values that depend on them will stay consistent across the cluster.

IDF files can be shared across multiple indexes. Only a single copy of an IDF file will be loaded by `searchd`, even when many indexes refer to that file. Should the contents of an IDF file change, the new contents can be loaded with a SIGHUP.

You can build an .idf file using `indextool` utility, by dumping dictionaries using `--dumpdict` switch first, then converting those to .idf format using `--buildidf`, then merging all .idf files across cluser using `--mergeidf`. Refer to *indextool command reference* for more information.

Example:

```
global_idf = /usr/local/sphinx/var/global.idf
```

### 11.3.18 ha_strategy

Agent mirror selection strategy, for load balancing. Optional, default is random.

The strategy used for mirror selection, or in other words, choosing a specific *agent mirror* in a distributed index. Essentially, this directive controls how exactly master does the load balancing between the configured mirror agent nodes. The following strategies are implemented:

### Simple random balancing

```
ha_strategy = random
```

The default balancing mode. Simple linear random distribution among the mirrors. That is, equal selection probability are assigned to every mirror. Kind of similar to round-robin (RR), but unlike RR, does not impose a strict selection order.

### Adaptive randomized balancing

The default simple random strategy does not take mirror status, error rate, and, most importantly, actual response latencies into account. So to accommodate for heterogeneous clusters and/or temporary spikes in agent node load, we have a group of balancing strategies that dynamically adjusts the probabilities based on the actual query latencies observed by the master.

The adaptive strategies based on **latency-weighted probabilities** basically work as follows:

- latency stats are accumulated, in blocks of ha_period_karma seconds;
- once per karma period, latency-weighted probabilities get recomputed;
- once per request (including ping requests), "dead or alive" flag is adjusted.

Currently, we begin with equal probabilities (or percentages, for brevity), and on every step, scale them by the inverse of the latencies observed during the last "karma" period, and then renormalize them. For example, if during the first 60 seconds after the master startup 4 mirrors had latencies of 10, 5, 30, and 3 msec/query respectively, the first adjustment step would go as follow:

- initial percentages: 0.25, 0.25, 0.25, 0.2%;
- observed latencies: 10 ms, 5 ms, 30 ms, 3 ms;
- inverse latencies: 0.1, 0.2, 0.0333, 0.333;
- scaled percentages: 0.025, 0.05, 0.008333, 0.0833;
- renormalized percentages: 0.15, 0.30, 0.05, 0.50.

Meaning that the 1st mirror would have a 15% chance of being chosen during the next karma period, the 2nd one a 30% chance, the 3rd one (slowest at 30 ms) only a 5% chance, and the 4th and the fastest one (at 3 ms) a 50% chance. Then, after that period, the second adjustment step would update those chances again, and so on.

The rationale here is, once the **observed latencies** stabilize, the **latency weighted probabilities** stabilize as well. So all these adjustment iterations are supposed to converge at a point where the average latencies are (roughly) equal over all mirrors.

```
ha_strategy = nodeads
```

Latency-weighted probabilities, but dead mirrors are excluded from the selection. "Dead" mirror is defined as a mirror that resulted in multiple hard errors (eg. network failure, or no answer, etc) in a row.

```
ha_strategy = noerrors
```

Latency-weighted probabilities, but mirrors with worse errors/success ratio are excluded from the selection.

### Round-robin balancing

```
ha_strategy = roundrobin
```

Simple round-robin selection, that is, selecting the 1st mirror in the list, then the 2nd one, then the 3rd one, etc, and then repeating the process once the last mirror in the list is reached. Unlike with the randomized strategies, RR imposes a strict querying order (1, 2, 3, .., N-1, N, 1, 2, 3, ... and so on) and *guarantees* that no two subsequent queries will be sent to the same mirror.

### 11.3.19 hitless_words

Hitless words list. Optional, allowed values are 'all', or a list file name.

By default, Manticore full-text index stores not only a list of matching documents for every given keyword, but also a list of its in-document positions (aka hitlist). Hitlists enables phrase, proximity, strict order and other advanced types of searching, as well as phrase proximity ranking. However, hitlists for specific frequent keywords (that can not be stopped for some reason despite being frequent) can get huge and thus slow to process while querying. Also, in some cases we might only care about boolean keyword matching, and never need position-based searching operators (such as phrase matching) nor phrase ranking.

`hitless_words` lets you create indexes that either do not have positional information (hitlists) at all, or skip it for specific keywords.

Hitless index will generally use less space than the respective regular index (about 1.5x can be expected). Both indexing and searching should be faster, at a cost of missing positional query and ranking support. When searching, positional queries (eg. phrase queries) will be automatically converted to respective non-positional (document-level) or combined queries. For instance, if keywords "hello" and "world" are hitless, "hello world" phrase query will be converted to (hello & world) bag-of-words query, matching all documents that mention either of the keywords but not necessarily the exact phrase. And if, in addition, keywords "simon" and "says" are not hitless, "simon says hello world" will be converted to ("simon says" & hello & world) query, matching all documents that contain "hello" and "world" anywhere in the document, and also "simon says" as an exact phrase.

Example:

```
hitless_words = all
```

### 11.3.20 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;
```

### 11.3.21 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. <br />) 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
```

### 11.3.22 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).

Both HTML tags and entities and considered markup and get processed.

HTML tags are removed, their contents (i.e., everything between <P> and </P>) are left intact by default. You can choose to keep and index attributes of the tags (e.g., HREF attribute in an A tag, or ALT in an IMG one). Several well-known inline tags are completely removed, all other tags are treated as block level and replaced with whitespace. For example, 'te<B>st</B>' text will be indexed as a single keyword 'test', however, 'te<P>st</P>' will be indexed as two keywords 'te' and 'st'. Known inline tags are as follows: A, B, I, S, U, BASEFONT, BIG, EM, FONT, IMG, LABEL, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TT.

HTML entities get decoded and replaced with corresponding UTF-8 characters. Stripper supports both numeric forms (such as ï) and text forms (such as ó or  ). All entities as specified by HTML4 standard are supported.

Stripping 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
```

### 11.3.23 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
```

### 11.3.24 index_exact_words

Whether to index the original keywords along with the stemmed/remapped versions. Optional, default is 0 (do not index).

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
```

### 11.3.25 index_field_lengths

Enables computing and storing of field lengths (both per-document and average per-index values) into the index. Optional, default is 0 (do not compute and store).

When `index_field_lengths` is set to 1, `indexer` will 1) create a respective length attribute for every full-text field, sharing the same name but with __len_ suffix; 2) compute a field length (counted in keywords) for every document and store in to a respective attribute; 3) compute the per-index averages. The lengths attributes will have a special TOKENCOUNT type, but their values are in fact regular 32-bit integers, and their values are generally accessible.

BM25A() and BM25F() functions in the expression ranker are based on these lengths and require `index_field_lengths` to be enabled. Historically, Manticore used a simplified, stripped-down variant of BM25 that, unlike the complete function, did **not** account for document length. (We later realized that it should have been called BM15 from the start.) Also we added support for both a complete variant of BM25, and its extension towards multiple fields, called BM25F. They require per-document length and per-field lengths, respectively. Hence the additional directive.

Example:

```
index_field_lengths = 1
```

### 11.3.26 index_sp

Whether to detect and index sentence and paragraph boundaries. Optional, default is 0 (do not detect and index).

This directive enables sentence and paragraph boundary indexing. It's required for the SENTENCE and PARAGRAPH operators to work. Sentence boundary detection is based on plain text analysis, so you only need to set `index_sp = 1` to enable it. Paragraph detection is however based on HTML markup, and happens in the *HTML stripper*. So to index paragraph locations you also need to enable the stripper by specifying `html_strip = 1`. Both types of boundaries are detected based on a few built-in rules enumerated just below.

Sentence boundary detection rules are as follows.

- Question and exclamation signs (? and !) are always a sentence boundary.
- Trailing dot (.) is a sentence boundary, except:
  - When followed by a letter. That's considered a part of an abbreviation (as in "S.T.A.L.K.E.R" or "Goldman Sachs S.p.A.").
  - When followed by a comma. That's considered an abbreviation followed by a comma (as in "Telecom Italia S.p.A., founded in 1994").
  - When followed by a space and a small letter. That's considered an abbreviation within a sentence (as in "News Corp. announced in February").
  - When preceded by a space and a capital letter, and followed by a space. That's considered a middle initial (as in "John D. Doe").

Paragraph boundaries are inserted at every block-level HTML tag. Namely, those are (as taken from HTML 4 standard) ADDRESS, BLOCKQUOTE, CAPTION, CENTER, DD, DIV, DL, DT, H1, H2, H3, H4, H5, LI, MENU, OL, P, PRE, TABLE, TBODY, TD, TFOOT, TH, THEAD, TR, and UL.

Both sentences and paragraphs increment the keyword position counter by 1.

Example:

```
index_sp = 1
```

### 11.3.27 index_zones

A list of in-field HTML/XML zones to index. Optional, default is empty (do not index zones).

Zones can be formally defined as follows. Everything between an opening and a matching closing tag is called a span, and the aggregate of all spans corresponding sharing the same tag name is called a zone. For instance, everything between the occurrences of <H1> and </H1> in the document field belongs to H1 zone.

Zone indexing, enabled by `index_zones` directive, is an optional extension of the HTML stripper. So it will also require that the *stripper* is enabled (with `html_strip = 1`). The value of the `index_zones` should be a comma-separated list of those tag names and wildcards (ending with a star) that should be indexed as zones.

Zones can nest and overlap arbitrarily. The only requirement is that every opening tag has a matching tag. You can also have an arbitrary number of both zones (as in unique zone names, such as H1) and spans (all the occurrences of those H1 tags) in a document. Once indexed, zones can then be used for matching with the ZONE operator, see *Extended query syntax*.

Example:

```
index_zones = h*, th, title
```

### 11.3.28 infix_fields

The list of full-text fields to limit infix indexing to. Applies to dict=crc only. 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
```

### 11.3.29 inplace_docinfo_gap

*In-place inversion* fine-tuning option. Controls preallocated docinfo gap size. Optional, default is 0.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
inplace_docinfo_gap = 1M
```

### 11.3.30 inplace_enable

Whether to enable in-place index inversion. Optional, default is 0 (use separate temporary files).

`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
```

### 11.3.31 inplace_hit_gap

*In-place inversion* fine-tuning option. Controls preallocated hitlist gap size. Optional, default is 0.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
inplace_hit_gap = 1M
```

### 11.3.32 inplace_reloc_factor

*inplace_reloc_factor* fine-tuning option. Controls relocation buffer size within indexing memory arena. Optional, default is 0.1.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
inplace_reloc_factor = 0.1
```

### 11.3.33 inplace_write_factor

*inplace_write_factor* fine-tuning option. Controls in-place write buffer size within indexing memory arena. Optional, default is 0.1.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
inplace_write_factor = 0.1
```

### 11.3.34 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. Many local indexes can be declared per each distributed index. Any local index can also be mentioned several times in different distributed indexes.

Note that by default all local indexes will be searched **sequentially**, utilizing only 1 CPU or core. To parallelize processing of the local parts in the distributed index, you should use *dist_threads*.

Before `dist_threads`, there also was a legacy solution to configure `searchd` to query itself instead of using local indexes (refer to *agent* for the details). However, that creates redundant CPU and network load, and `dist_threads` is now strongly suggested instead.

Example:

```
local = chunk1
local = chunk2
```

### 11.3.35 max_substring_len

Maximum substring (either prefix or infix) length to index. Optional, default is 0 (do not limit indexed substrings). Applies to dict=crc only.

By default, substring (either prefix or infix) indexing in the *dict=crc mode* will index **all** the possible substrings as separate keywords. That might result in an overly large index. So the `max_substring_len` directive lets you limit the impact of substring indexing by skipping too-long substrings (which, chances are, will never get searched for anyway).

For example, a test index of 10,000 blog posts takes this much disk space depending on the settings:

- 6.4 MB baseline (no substrings)
- 24.3 MB (3.8x) with min_prefix_len = 3
- 22.2 MB (3.5x) with min_prefix_len = 3, max_substring_len = 8
- 19.3 MB (3.0x) with min_prefix_len = 3, max_substring_len = 6
- 94.3 MB (14.7x) with min_infix_len = 3
- 84.6 MB (13.2x) with min_infix_len = 3, max_substring_len = 8
- 70.7 MB (11.0x) with min_infix_len = 3, max_substring_len = 6

So in this test limiting the max substring length saved us 10-15% on the index size.

There is no performance impact associated with substring length when using dict=keywords mode, so this directive is not applicable and intentionally forbidden in that case. If required, you can still limit the length of a substring that you search for in the application code.

Example:

```
max_substring_len = 12
```

### 11.3.36 min_infix_len

Minimum infix prefix length to index and search. Optional, default is 0 (do not index infixes), and minimum allowed non-zero value is 2.

Infix length setting enables wildcard searches with term patterns like 'start', 'end', '*middle*', and so on. It also lets you disable too short wildcards if those are too expensive to search for.

Perfect word matches can be differentiated from infix matches, and ranked higher, by utilizing all of the following options: a) dict=keywords (on by default), b) index_exact_words=1 (off by default), and c) expand_keywords=1 (also off by default). Note that either with the legacy dict=crc mode (which you should ditch anyway!), or with any of the above options disable, there is no data to differentiate between the infixes and full words, and thus perfect word matches can't be ranked higher.

However, query time might vary greatly, depending on how many keywords the substring will actually expand to. Short and frequent syllables like '*in*' or '*ti*' just might expand to way too many keywords, all of which would need to be matched and processed. Therefore, to generally enable substring searches you would set min_infix_len to 2; and to limit the impact from wildcard searches with too short wildcards, you might set it higher.

Infixes must be at least 2 characters long, wildcards like '*a*' are not allowed for performance reasons. (While in theory it is possible to scan the entire dictionary, identify keywords matching on just a single character, expand '*a*' to an OR operator over 100,000+ keywords, and evaluate that expanded query, in practice this will very definitely kill your server.)

Example:

```
min_infix_len = 3
```

### 11.3.37 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. 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.

Perfect word matches can be differentiated from prefix matches, and ranked higher, by utilizing all of the following options: a) dict=keywords (on by default), b) index_exact_words=1 (off by default), and c) expand_keywords=1 (also off by default). Note that either with the legacy dict=crc mode (which you should ditch anyway!), or with any of the above options disable, there is no data to differentiate between the prefixes and full words, and thus perfect word matches can't be ranked higher.

Example:

```
min_prefix_len = 3
```

### 11.3.38 min_stemming_len

Minimum word length at which to enable stemming. Optional, default is 1 (stem everything).

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
```

### 11.3.39 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
```

### 11.3.40 mirror_retry_count

Same as *index_agent_retry_count*. If both values provided, `mirror_retry_count` will be taken, and the warning about it will be fired.

### 11.3.41 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 Manticore 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
```

### 11.3.42 morphology

A list of morphology preprocessors (stemmers or lemmatizers) 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.

There are 3 different morphology preprocessors that Manticore implements: lemmatizers, stemmers, and phonetic algorithms.

- Lemmatizer reduces a keyword form to a so-called lemma, a proper normal form, or in other words, a valid natural language root word. For example, "running" could be reduced to "run", the infinitive verb form, and "octopi" would be reduced to "octopus", the singular noun form. Note that sometimes a word form can have multiple corresponding root words. For instance, by looking at "dove" it is not possible to tell whether this is a

past tense of "dive" the verb as in "He dove into a pool.", or "dove" the noun as in "White dove flew over the cuckoo's nest." In this case lemmatizer can generate all the possible root forms.

- Stemmer reduces a keyword form to a so-called stem by removing and/or replacing certain well-known suffixes. The resulting stem is however *not* guaranteed to be a valid word on itself. For instance, with a Porter English stemmers "running" would still reduce to "run", which is fine, but "business" would reduce to "busi", which is not a word, and "octopi" would not reduce at all. Stemmers are essentially (much) simpler but still pretty good replacements of full-blown lemmatizers.
- Phonetic algorithms replace the words with specially crafted phonetic codes that are equal even when the words original are different, but phonetically close.

The morphology processors that come with our own built-in Manticore implementations are:

- English, Russian, and German lemmatizers;
- English, Russian, Arabic, and Czech stemmers;
- SoundEx and MetaPhone phonetic algorithms.

You can also link with **libstemmer** library for even more stemmers (see details below). With libstemmer, Manticore also supports morphological processing for more than 15 other languages. Binary packages should come prebuilt with libstemmer support, too.

Lemmatizers require a dictionary that needs to be additionally downloaded from the Manticore website. That dictionary needs to be installed in a directory specified by *lemmatizer_base* directive. Also, there is a *lemmatizer_cache* directive that lets you speed up lemmatizing (and therefore indexing) by spending more RAM for, basically, an uncompressed cache of a dictionary.

Chinese segmentation using Rosette Linguistics Platform is also available. It is a much more precise but slower way (compared to n-grams) to segment Chinese documents. *charset_table* must contain all Chinese characters except Chinese punctuation marks because incoming documents are first processed by sphinx tokenizer and then the result is processed by RLP. Manticore performs per-token language detection on the incoming documents. If token language is identified as Chinese, it will only be processed the RLP, even if multiple morphology processors are specified. Otherwise, it will be processed by all the morphology processors specified in the "morphology" option. Rosette Linguistics Platform must be installed and configured and sphinx must be built with a –with-rlp switch. See also *rlp_root*, *rlp_environment* and *rlp_context* options. A batched version of RLP segmentation is also available (`rlp_chinese_batched`). It provides the same functionality as the basic `rlp_chinese` segmentation, but enables batching documents before processing them by the RLP. Processing several documents at once can result in a substantial indexing speedup if the documents are small (for example, less than 1k). See also *rlp_max_batch_size* and *rlp_max_batch_docs* options.

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.

Soundex implementation matches that of MySQL. Metaphone implementation is based on Double Metaphone algorithm and indexes the primary code.

Built-in values that are available for use in `morphology` option are as follows:

- none - do not perform any morphology processing;
- lemmatize_ru - apply Russian lemmatizer and pick a single root form;
- lemmatize_en - apply English lemmatizer and pick a single root form;
- lemmatize_de - apply German lemmatizer and pick a single root form;
- lemmatize_ru_all - apply Russian lemmatizer and index all possible root forms;
- lemmatize_en_all - apply English lemmatizer and index all possible root forms;

- lemmatize_de_all - apply German lemmatizer and index all possible root forms;
- stem_en - apply Porter's English stemmer;
- stem_ru - apply Porter's Russian stemmer;
- stem_enru - apply Porter's English and Russian stemmers;
- stem_cz - apply Czech stemmer;
- stem_ar - apply Arabic stemmer;
- soundex - replace keywords with their SOUNDEX code;
- metaphone - replace keywords with their METAPHONE code.
- rlp_chinese - apply Chinese text segmentation using Rosette Linguistics Platform
- rlp_chinese_batched - apply Chinese text segmentation using Rosette Linguistics Platform with document batching

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
```

### 11.3.43 morphology_skip_fields

A list of fields there morphology preprocessors do not apply. Optional, default is empty (apply preprocessors to all fields).

Used on indexing there only exact form of words got stored for defined fields.

Example:

```
morphology_skip_fields = tags, name
```

### 11.3.44 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
```

### 11.3.45 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, Japanese, Korean) 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 result 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 Manticore 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, Manticore 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
```

### 11.3.46 ondisk_attrs

Allows for fine-grain control over how attributes are loaded into memory when using indexes with external storage. It is possible to keep attributes on disk. Although, the daemon does map them to memory and the OS loads small chunks of data on demand. This allows use of docinfo = extern instead of docinfo = inline, but still leaves plenty of free memory for cases when you have large collections of pooled attributes (string/JSON/MVA) or when you're using many indexes per daemon that don't consume memory. It is not possible to update attributes left on disk when this option is enabled and the constraint of 4Gb of entries per pool is still in effect.

Note that this option also affects RT indexes. When it is enabled, all attribute updates will be disabled, and also all disk chunks of RT indexes will behave described above. However inserting and deleting of docs from RT indexes is still possible with enabled ondisk_attrs.

Possible values:

- 0 - disabled and default value, all attributes are loaded in memory (the normal behaviour of docinfo = extern)
- 1 - all attributes stay on disk. Daemon loads no files (spa, spm, sps). This is the most memory conserving mode, however it is also the slowest as the whole doc-id-list and block index doesn't load.
- pool - only pooled attributes stay on disk. Pooled attributes are string, MVA, and JSON attributes (sps, spm files). Scalar attributes stored in docinfo (spa file) load as usual.

This option does not affect indexing in any way, it only requires daemon restart.

Example:

```
ondisk_attrs = pool #keep pooled attributes on disk
```

### 11.3.47 overshort_step

Position increment on overshort (less that *min_word_len*) keywords. Optional, allowed values are 0 and 1, default is 1.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
overshort_step = 1
```

### 11.3.48 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);
- `.spk` stores kill-lists;
- `.spm` stores MVA data;
- `.spp` stores hit (aka posting, aka word occurrence) lists for each word ID;
- `.sps` stores string attribute data.
- `.spe` stores skip-lists to speed up doc-list filtering

Example:

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

### 11.3.49 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
```

### 11.3.50 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
```

### 11.3.51 prefix_fields

The list of full-text fields to limit prefix indexing to. Applies to dict=crc only. 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
```

### 11.3.52 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
```

### 11.3.53 regexp_filter

Regular expressions (regexps) to filter the fields and queries with. Optional, multi-value, default is an empty list of regexps.

In certain applications (like product search) there can be many different ways to call a model, or a product, or a property, and so on. For instance, 'iphone 3gs' and 'iphone 3 gs' (or even 'iphone3 gs') are very likely to mean the same product. Or, for a more tricky example, '13-inch', '13 inch', '13"', and '13in' in a laptop screen size descriptions do mean the same.

Regexps provide you with a mechanism to specify a number of rules specific to your application to handle such cases. In the first 'iphone 3gs' example, you could possibly get away with a wordforms files tailored to handle a handful of iPhone models. However even in a comparatively simple second '13-inch' example there is just way too many individual forms and you are better off specifying rules that would normalize both '13-inch' and '13in' to something identical.

Regular expressions listed in `regexp_filter` are applied in the order they are listed. That happens at the earliest stage possible, before any other processing, even before tokenization. That is, regexps are applied to the raw source fields when indexing, and to the raw search query text when searching.

We use the RE2 engine to implement regexps. So when building from the source, the library must be installed in the system and Manticore must be configured built with a `--with-re2` switch. Binary packages should come with RE2 builtin.

Example:

```
# index '13-inch' as '13inch'
regexp_filter = \**(\d+)\" => \1inch

# index 'blue' or 'red' as 'color'
regexp_filter = (blue|red) => color
```

### 11.3.54 rlp_context

RLP context configuration file. Mandatory if RLP is used.

Example:

```
rlp_context = /home/myuser/RLP/rlp-context.xml
```

### 11.3.55 rt_attr_bigint

BIGINT attribute declaration. Multi-value (an arbitrary number of attributes is allowed), optional. Declares a signed 64-bit attribute.

Example:

```
rt_attr_bigint = guid
```

### 11.3.56 rt_attr_bool

Boolean attribute declaration. Multi-value (there might be multiple attributes declared), optional. Declares a 1-bit unsigned integer attribute.

Example:

```
rt_attr_bool = available
```

### 11.3.57 rt_attr_float

Floating point attribute declaration. Multi-value (an arbitrary number of attributes is allowed), optional. Declares a single precision, 32-bit IEEE 754 format float attribute.

Example:

```
rt_attr_float = gpa
```

### 11.3.58 rt_attr_json

JSON attribute declaration. Multi-value (ie. there may be more than one such attribute declared), optional.

Refer to *sql_attr_json* for more details on the JSON attributes.

Example:

```
rt_attr_json = properties
```

### 11.3.59 rt_attr_multi_64

*Multi-valued attribute* (MVA) declaration. Declares the BIGINT (signed 64-bit) MVA attribute. Multi-value (ie. there may be more than one such attribute declared), optional. Applies to RT indexes only.

Example:

```
rt_attr_multi_64 = my_wide_tags
```

### 11.3.60 rt_attr_multi

*Multi-valued attribute* (MVA) declaration. Declares the UNSIGNED INTEGER (unsigned 32-bit) MVA attribute. Multi-value (ie. there may be more than one such attribute declared), optional. Applies to RT indexes only.

Example:

```
rt_attr_multi = my_tags
```

### 11.3.61 rt_attr_string

String attribute declaration. Multi-value (an arbitrary number of attributes is allowed), optional.

Example:

```
rt_attr_string = author
```

### 11.3.62 rt_attr_timestamp

Timestamp attribute declaration. Multi-value (an arbitrary number of attributes is allowed), optional.

Example:

```
rt_attr_timestamp = date_added
```

### 11.3.63 rt_attr_uint

Unsigned integer attribute declaration. Multi-value (an arbitrary number of attributes is allowed), optional. Declares an unsigned 32-bit attribute.

Example:

```
rt_attr_uint = gid
```

### 11.3.64 rt_field

Full-text field declaration. Multi-value, mandatory

Full-text fields to be indexed are declared using `rt_field` directive. The names must be unique. The order is preserved; and so field values in INSERT statements without an explicit list of inserted columns will have to be in the same order as configured.

Example:

```
rt_field = author
rt_field = title
rt_field = content
```

### 11.3.65 rt_mem_limit

RAM chunk size limit. Optional, default is 128M.

RT index keeps some data in memory (so-called RAM chunk) and also maintains a number of on-disk indexes (so-called disk chunks). This directive lets you control the RAM chunk size. Once there's too much data to keep in RAM, RT index will flush it to disk, activate a newly created disk chunk, and reset the RAM chunk.

The limit is pretty strict; RT index should never allocate more memory than it's limited to. The memory is not preallocated either, hence, specifying 512 MB limit and only inserting 3 MB of data should result in allocating 3 MB, not 512 MB.

Example:

```
rt_mem_limit = 512M
```

### 11.3.66 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
```

### 11.3.67 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 be UTF-8. 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 normally be applied when parsing stopwords file. That might however lead to undesired results. You can turn that off with *stopwords_unstemmed*.

Small enough files are stored in the index header, see *embedded_limit* for details.

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. That behavior can be tweaked through the *stopword_step* directive.

Stopwords files can either be created manually, or semi-automatically. `indexer` provides a mode that creates a frequency dictionary of the index, sorted by the keyword frequency, see `--buildstops` and `--buildfreqs` switch in *indexer command reference*. Top keywords from that dictionary can usually be used as stopwords.

Example:

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

### 11.3.68 stopword_step

Position increment on *stopwords*. Optional, allowed values are 0 and 1, default is 1.

This directive does not affect `searchd` in any way, it only affects `indexer`.

Example:

```
stopword_step = 1
```

### 11.3.69 stopwords_unstemmed

Whether to apply stopwords before or after stemming. Optional, default is 0 (apply stopword filter after stemming).

By default, stopwords are stemmed themselves, and applied to tokens *after* stemming (or any other morphology processing). In other words, by default, a token is stopped when stem(token) == stem(stopword). That can lead to unexpected results when a token gets (erroneously) stemmed to a stopped root. For example, 'Andes' gets stemmed to 'and' by our current stemmer implementation, so when 'and' is a stopword, 'Andes' is also stopped.

stopwords_unstemmed directive fixes that issue. When it's enabled, stopwords are applied before stemming (and therefore to the original word forms), and the tokens are stopped when token == stopword.

Example:

```
stopwords_unstemmed = 1
```

### 11.3.70 type

Index type. Known values are `plain`, `distributed`, `rt`, `template` and `percolate`. Optional, default is 'plain' (plain local index).

Manticore supports several different types of indexes. Plain local indexes are stored and processed on the local machine. Distributed indexes involve not only local searching but querying remote `searchd` instances over the network as well (see *Distributed searching*). Real-time indexes (or RT indexes for short) are also stored and processed locally, but additionally allow for on-the-fly updates of the full-text index (see *Real-time indexes*). Note that *attributes* can be updated on-the-fly using either plain local indexes or RT ones. Template indexes are actually a pseudo-indexes because they do not store any data. That means they do not create any files on your hard drive. But you can use them for keywords and snippets generation, which may be useful in some cases, and also as templates to inherit real indexes from them.

Index type setting lets you choose the needed type. By default, plain local index type will be assumed.

Example:

```
type = distributed
```

### 11.3.71 wordforms

Word forms dictionary. Optional, default is empty.

Word forms are applied after tokenizing the incoming text by *charset_table* rules. They essentially 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.

Small enough files are stored in the index header, see *embedded_limit* for details.

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

Word forms support in Manticore 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 UTF-8 encoding, 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 a bundled `spelldump` utility that helps you create a dictionary file in the format Manticore can read from source `.dict` and `.aff` dictionary files in `ispell` or `MySpell` format (as bundled with OpenOffice).

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.

You can use "=>" instead of ">". Comments (starting with "#" are also allowed. Finally, if a line starts with a tilde ("~") the wordform will be applied after morphology, instead of before.

```
core 2 duo > c2d
e6600 > c2d
core 2duo => c2d # Some people write '2duo' together...
```

You can specify multiple destination tokens:

```
s02e02 > season 2 episode 2
s3 e3 > season 3 episode 3
```

Example:

```
wordforms = /usr/local/sphinx/data/wordforms.txt
wordforms = /usr/local/sphinx/data/alternateforms.txt
wordforms = /usr/local/sphinx/private/dict*.txt
```

You can specify several files and not only just one. Masks can be used as a pattern, and all matching files will be processed in simple ascending order. (If multi-byte codepages are used, and file names can include foreign characters, the resulting order may not be exactly alphabetic.) If a same wordform definition is found in several files, the latter one is used, and it overrides previous definitions.

## 11.4 `indexer` program configuration options

### 11.4.1 lemmatizer_cache

Lemmatizer cache size. Optional, default is 256K.

Our lemmatizer implementation (see *morphology* for a discussion of what lemmatizers are) uses a compressed dictionary format that enables a space/speed tradeoff. It can either perform lemmatization off the compressed data, using more CPU but less RAM, or it can decompress and precache the dictionary either partially or fully, thus using less CPU but more RAM. And the lemmatizer_cache directive lets you control how much RAM exactly can be spent for that uncompressed dictionary cache.

Currently, the only available dictionaries are ru.pak, en.pak, and de.pak. These are the russian, english and german dictionaries. The compressed dictionary is approximately 2 to 10 MB in size. Note that the dictionary stays in memory at all times, too. The default cache size is 256 KB. The accepted cache sizes are 0 to 2047 MB. It's safe to raise the cache size too high; the lemmatizer will only use the needed memory. For instance, the entire Russian dictionary decompresses to approximately 110 MB; and thus setting lemmatizer_cache anywhere higher than that will not affect the memory use: even when 1024 MB is allowed for the cache, if only 110 MB is needed, it will only use those 110 MB.

On our benchmarks, the total indexing time with different cache sizes was as follows:

- 9.07 sec, morphology = lemmatize_ru, lemmatizer_cache = 0
- 8.60 sec, morphology = lemmatize_ru, lemmatizer_cache = 256K
- 8.33 sec, morphology = lemmatize_ru, lemmatizer_cache = 8M
- 7.95 sec, morphology = lemmatize_ru, lemmatizer_cache = 128M
- 6.85 sec, morphology = stem_ru (baseline)

Your mileage may vary, but a simple rule of thumb would be to either go with the small default 256 KB cache when pressed for memory, or spend 128 MB extra RAM and cache the entire dictionary for maximum indexing performance.

Example:

```
lemmatizer_cache = 256M # cache it all
```

### 11.4.2 max_file_field_buffer

Maximum file field adaptive buffer size, bytes. Optional, default is 8 MB, minimum is 1 MB.

File field buffer is used to load files referred to from *sql_file_field* columns. This buffer is adaptive, starting at 1 MB at first allocation, and growing in 2x steps until either file contents can be loaded, or maximum buffer size, specified by `max_file_field_buffer` directive, is reached.

Thus, if there are no file fields are specified, no buffer is allocated at all. If all files loaded during indexing are under (for example) 2 MB in size, but `max_file_field_buffer` value is 128 MB, peak buffer usage would still be only 2 MB. However, files over 128 MB would be entirely skipped.

Example:

```
max_file_field_buffer = 128M
```

### 11.4.3 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 degradation caused by indexing.

Example:

```
max_iops = 40
```

### 11.4.4 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
```

### 11.4.5 max_xmlpipe2_field

Maximum allowed field size for XMLpipe2 source type, bytes. Optional, default is 2 MB.

Example:

```
max_xmlpipe2_field = 8M
```

### 11.4.6 mem_limit

Indexing RAM usage limit. Optional, default is 128M.

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
```

### 11.4.7 on_file_field_error

How to handle IO errors in file fields. Optional, default is `ignore_field`.

When there is a problem indexing a file referenced by a file field (*sql_file_field*), `indexer` can either index the document, assuming empty content in this particular field, or skip the document, or fail indexing entirely. `on_file_field_error` directive controls that behavior. The values it takes are:

- `ignore_field`, index the current document without field;
- `skip_document`, skip the current document but continue indexing;
- `fail_index`, fail indexing with an error message.

The problems that can arise are: open error, size error (file too big), and data read error. Warning messages on any problem will be given at all times, irregardless of the phase and the `on_file_field_error` setting.

Note that with `on_file_field_error = skip_document` documents will only be ignored if problems are detected during an early check phase, and **not** during the actual file parsing phase. `indexer` will open every referenced file and check its size before doing any work, and then open it again when doing actual parsing work. So in case a file goes away between these two open attempts, the document will still be indexed.

Example:

```
on_file_field_error = skip_document
```

### 11.4.8 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
```

## 11.5 `searchd` program configuration options

### 11.5.1 agent_connect_timeout

Instance-wide defaults for *agent_connect_timeout* parameter. The last defined in distributed (network) indexes.

### 11.5.2 agent_query_timeout

Instance-wide defaults for *agent_query_timeout* parameter. The last defined in distributed (network) indexes, or also may be overrided per-query using OPTION clause.

### 11.5.3 agent_retry_count

Integer, specifies how many times manticore will try to connect and query remote agents in distributed index before reporting fatal query error. Default is 0 (i.e. no retries). This value may be also specified on per-query basis using 'OPTION retry_count=XXX' clause. If per-query option exists, it will override the one specified in config.

Note, that if you use *agent mirrors* in definition of your distributed index, then before every attempt of connect daemon will select different mirror, according to specified *ha_strategy*specified. In this case `agent_retry_count` will be aggregated for all mirrors in a set.

For example, if you have 10 mirrors, and set `agent_retry_count=5`, then daemon will retry up to 50 times, assuming average 5 tries per every of 10 mirrors. (in case of option `ha_strategy = roundrobin` it will be actually so).

In the same time value provided as `retry_count` option of *agent* definition serves as absolute limit. Otherwords, `[retry_count=2]` option in agent definition means always at most 2 tries, no mean if you have 1 or 10 mirrors in a line.

### 11.5.4 agent_retry_delay

Integer, in milliseconds. Specifies the delay sphinx rest before retrying to query a remote agent in case it fails. The value has sense only if non-zero *agent_retry_count* or non-zero per-query OPTION retry_count specified. Default is 500. This value may be also specified on per-query basis using 'OPTION retry_delay=XXX' clause. If per-query option exists, it will override the one specified in config.

### 11.5.5 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.

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
```

### 11.5.6 binlog_flush

Binary log transaction flush/sync mode. Optional, default is 2 (flush every transaction, sync every second).

This directive controls how frequently will binary log be flushed to OS and synced to disk. Three modes are supported:

- 0, flush and sync every second. Best performance, but up to 1 second worth of committed transactions can be lost both on daemon crash, or OS/hardware crash.
- 1, flush and sync every transaction. Worst performance, but every committed transaction data is guaranteed to be saved.
- 2, flush every transaction, sync every second. Good performance, and every committed transaction is guaranteed to be saved in case of daemon crash. However, in case of OS/hardware crash up to 1 second worth of committed transactions can be lost.

For those familiar with MySQL and InnoDB, this directive is entirely similar to `innodb_flush_log_at_trx_commit`. In most cases, the default hybrid mode 2 provides a nice balance of speed and safety, with full RT index data protection against daemon crashes, and some protection against hardware ones.

Example:

```
binlog_flush = 1 # ultimate safety, low speed
```

### 11.5.7 binlog_max_log_size

Maximum binary log file size. Optional, default is 0 (do not reopen binlog file based on size).

A new binlog file will be forcibly opened once the current binlog file reaches this limit. This achieves a finer granularity of logs and can yield more efficient binlog disk usage under certain borderline workloads.

Example:

```
binlog_max_log_size = 16M
```

### 11.5.8 binlog_path

Binary log (aka transaction log) files path. Optional, default is build-time configured data directory.

Binary logs are used for crash recovery of RT index data, and also of attributes updates of plain disk indices that would otherwise only be stored in RAM until flush. When logging is enabled, every transaction COMMIT-ted into RT index gets written into a log file. Logs are then automatically replayed on startup after an unclean shutdown, recovering the logged changes.

`binlog_path` directive specifies the binary log files location. It should contain just the path; `searchd` will create and unlink multiple binlog.* files in that path as necessary (binlog data, metadata, and lock files, etc).

Empty value disables binary logging. That improves performance, but puts RT index data at risk.

WARNING! It is strongly recommended to always explicitly define 'binlog_path' option in your config. Otherwise, the default path, which in most cases is the same as working folder, may point to the folder with no write access (for example, /usr/local/var/data). In this case, the searchd will not start at all.

Example:

```
binlog_path = # disable logging
binlog_path = /var/data # /var/data/binlog.001 etc will be created
```

### 11.5.9 client_timeout

Maximum time to wait between requests (in seconds) when using persistent connections. Optional, default is five minutes.

Example:

```
client_timeout = 3600
```

### 11.5.10 collation_libc_locale

Server libc locale. Optional, default is C.

Specifies the libc locale, affecting the libc-based collations. Refer to *Collations* section for the details.

Example:

```
collation_libc_locale = fr_FR
```

### 11.5.11 collation_server

Default server collation. Optional, default is libc_ci.

Specifies the default collation used for incoming requests. The collation can be overridden on a per-query basis. Refer to *Collations* section for the list of available collations and other details.

Example:

```
collation_server = utf8_ci
```

### 11.5.12 dist_threads

Max local worker threads to use for parallelizable requests (searching a distributed index; building a batch of snippets). Optional, default is 0, which means to disable in-request parallelism.

Distributed index can include several local indexes. `dist_threads` lets you easily utilize multiple CPUs/cores for that (previously existing alternative was to specify the indexes as remote agents, pointing searchd to itself and paying some network overheads).

When set to a value N greater than 1, this directive will create up to N threads for every query, and schedule the specific searches within these threads. For example, if there are 7 local indexes to search and dist_threads is set to 2, then 2 parallel threads would be created: one that sequentially searches 4 indexes, and another one that searches the other 3 indexes.

In case of CPU bound workload, setting `dist_threads` to 1x the number of cores is advised (creating more threads than cores will not improve query time). In case of mixed CPU/disk bound workload it might sometimes make sense to use more (so that all cores could be utilizes even when there are threads that wait for I/O completion).

Building a batch of snippets with `load_files` flag enabled can also be parallelized. Up to `dist_threads` threads are be created to process those files. That speeds up snippet extraction when the total amount of document data to process is significant (hundreds of megabytes).

Up to `dist_threads` threads can be created to handle *CALL PQ* calls.

Example:

```
index dist_test
{
    type = distributed
    local = chunk1
    local = chunk2
    local = chunk3
    local = chunk4
}

# ...
```

(continues on next page)

(continued from previous page)

```
dist_threads = 4
```

### 11.5.13 expansion_limit

The maximum number of expanded keywords for a single wildcard. Optional, default is 0 (no limit).

When doing substring searches against indexes built with `dict = keywords` enabled, a single wildcard may potentially result in thousands and even millions of matched keywords (think of matching 'a*' against the entire Oxford dictionary). This directive lets you limit the impact of such expansions. Setting `expansion_limit = N` restricts expansions to no more than N of the most frequent matching keywords (per each wildcard in the query).

Example:

```
expansion_limit = 16
```

### 11.5.14 grouping_in_utc

Specifies whether timed grouping in API and SphinxQL will be calculated in local timezone, or in UTC. Optional, default is 0 (means 'local tz').

By default all 'group by time' expressions (like group by day, week, month and year in API, also group by day, month, year, yearmonth, yearmonthday in SphinxQL) is done using local time. I.e. when you have docs with attributes timed `13:00 utc` and `15:00 utc` - in case of grouping they both will fall into facility group according to your local tz setting. Say, if you live in `utc`, it will be one day, but if you live in `utc+10`, then these docs will be matched into different `group by day` facility groups (since 13:00 utc in UTC+10 tz 23:00 local time, but 15:00 is 01:00 of the next day). Sometimes such behavior is unacceptable, and it is desirable to make time grouping not dependent from timezone. Of course, you can run the daemon with defined global TZ env variable, but it will affect not only grouping, but also timestamping in the logs, which may be also undesirable. Switching 'on' this option (either in config, either using *set global* statement in sphinxql) will cause all time grouping expressions to be calculated in UTC, leaving the rest of time-dependend functions (i.e. logging of the daemon) in local TZ.

### 11.5.15 ha_period_karma

Agent mirror statistics window size, in seconds. Optional, default is 60.

For a distributed index with agent mirrors in it (see more in *remote agents*), master tracks several different per-mirror counters. These counters are then used for failover and balancing. (Master picks the best mirror to use based on the counters.) Counters are accumulated in blocks of `ha_period_karma` seconds.

After beginning a new block, master may still use the accumulated values from the previous one, until the new one is half full. Thus, any previous history stops affecting the mirror choice after 1.5 times ha_period_karma seconds at most.

Despite that at most 2 blocks are used for mirror selection, upto 15 last blocks are actually stored, for instrumentation purposes. They can be inspected using *SHOW AGENT STATUS* statement.

Example:

```
ha_period_karma = 120
```

### 11.5.16 ha_ping_interval

Interval between agent mirror pings, in milliseconds. Optional, default is 1000.

For a distributed index with agent mirrors in it (see more in *remote agents*), master sends all mirrors a ping command during the idle periods. This is to track the current agent status (alive or dead, network roundtrip, etc). The interval between such pings is defined by this directive.

To disable pings, set ha_ping_interval to 0.

Example:

```
ha_ping_interval = 0
```

### 11.5.17 hostname_lookup

Hostnames renew strategy. By default, IP addresses of agent host names are cached at daemon start to avoid extra flood to DNS. In some cases the IP can change dynamically (e.g. cloud hosting) and it might be desired to don't cache the IPs. Setting this option to 'request' disabled the caching and queries the DNS at each query. The IP addresses can also be manually renewed with FLUSH HOSTNAMES command.

### 11.5.18 listen_backlog

TCP listen backlog. Optional, default is 5.

Windows builds currently 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
```

### 11.5.19 listen

This setting lets you specify IP address and port, or Unix-domain socket path, that `searchd` will listen on.

The informal grammar for `listen` setting is:

```
listen = ( address ":" port | port | path ) [ ":" protocol ] [ "_vip" ]
```

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.

You can also specify a protocol handler (listener) to be used for connections on this socket. Supported protocol values are 'sphinx' (Manticore 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 *MySQL protocol support and SphinxQL* section.

Adding a "_vip" suffix to a protocol (for instance "sphinx_vip" or "mysql41_vip") makes all connections to that port bypass the thread pool and always forcibly create a new dedicated thread. That's useful for managing in case of a severe overload when the daemon would either stall or not let you connect via a regular port.

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` directives are found then the server will listen on all available interfaces using the default SphinxAPI port 9312, and also on default SphinxQL port 9306. Both port numbers are assigned by IANA (see http://www.iana.org/assignments/port-numbers for details) and should therefore be available.

Unix-domain sockets are not supported on Windows.

### 11.5.20 log

Log file name. Optional, default is 'searchd.log'. All `searchd` run time events will be logged in this file.

Also you can use the 'syslog' as the file name. In this case the events will be sent to syslog daemon. To use the syslog option the sphinx must be configured '–with-syslog' on building.

Example:

```
log = /var/log/searchd.log
```

### 11.5.21 max_batch_queries

Limits the amount of queries per batch. Optional, default is 32.

Makes searchd perform a sanity check of the amount of the queries submitted in a single batch when using *multi-queries*. Set it to 0 to skip the check.

Example:

```
max_batch_queries = 256
```

### 11.5.22 max_children

Maximum amount of worker threads (or in other words, concurrent queries to run in parallel). Optional, default is 0 (unlimited) in workers=threads, or 1.5 times the CPU cores count in workers=thread_pool mode.

max_children imposes a hard limit on the number of threads working on user queries. There might still be additional internal threads doing maintenance tasks, but when it comes to user queries, it is no more than max_children concurrent threads (and queries) at all times.

Note that in workers=threads mode a thread is allocated for every connection rather than an active query. So when there are 100 idle connections but only 2 active connections with currently running queries, that still accounts for 102 threads, all of them subject to max_children limit. So with a max_children set way too low, and pooled connections not reused well enough on the application side, you can effectively DOS your own server. When the limit is reached, any additional incoming connections will be dismissed with a temporary "maxed out" error immediately as they attempt to connect. Thus, in threads mode you should choose the max_children limit rather carefully, with not just the concurrent queries but also with potentially idle *network connections* in mind.

Now, in workers=thread_pool mode the network connections are separated from query processing, so in the example above, 100 idle connections will all be handled by an internal network thread, and only the 2 actually active queries

will be subject to max_children limit. When the limit is reached, any additional incoming *connections* will still be accepted, and any additional *queries* will *get enqueued* until there are free worker threads. The queries will only start failing with a temporary. Thus, in thread_pool mode it makes little sense to raise max_children much higher than the amount of CPU cores. Usually that will only hurt CPU contention and *decrease* the general throughput.

Example:

```
max_children = 10
```

### 11.5.23 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.

Example:

```
max_filters = 1024
```

### 11.5.24 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.

Example:

```
max_filter_values = 16384
```

### 11.5.25 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.

Example:

```
max_packet_size = 32M
```

### 11.5.26 mva_updates_pool

Shared pool size for in-memory MVA updates storage. Optional, default size is 1M.

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
```

### 11.5.27 mysql_version_string

A server version string to return via MySQL protocol. Optional, default is empty (return Manticore version).

Several picky MySQL client libraries depend on a particular version number format used by MySQL, and moreover, sometimes choose a different execution path based on the reported version number (rather than the indicated capabilities flags). For instance, Python MySQLdb 1.2.2 throws an exception when the version number is not in X.Y.ZZ format; MySQL .NET connector 6.3.x fails internally on version numbers 1.x along with a certain combination of flags, etc. To workaround that, you can use `mysql_version_string` directive and have `searchd` report a different version to clients connecting over MySQL protocol. (By default, it reports its own version.)

Example:

```
mysql_version_string = 5.0.37
```

### 11.5.28 net_workers

Number of network threads for workers=thread_pool mode, default is 1.

Useful for extremely high query rates, when just 1 thread is not enough to manage all the incoming queries.

### 11.5.29 net_wait_tm

Control busy loop interval of a network thread for workers=thread_pool mode, default is 1, might be set to -1, 0, positive integer.

In case daemon configured as pure master and routes requests to agents it is important to handle requests without delays and do not allow net-thread to sleep or cut out from CPU. Here is busy loop to do that. After incoming request, network thread use CPU poll for `10 * net_wait_tm` milliseconds in case `net_wait_tm` is positive number or polls only with CPU in case `net_wait_tm` is `0`. Also busy loop might be disabled with `net_wait_tm = -1` - this way poller set timeout of `1ms` for system poll call.

### 11.5.30 net_throttle_accept net_throttle_action

Control network thread for workers=thread_pool mode, default is 0.

These options define how many clients got accepted and how many requests processed on each iteration of network loop, in case of value above zero. Zero value means do not constrain network loop. These options might help to fine tune network loop throughput at high load scenario.

### 11.5.31 ondisk_attrs_default

Instance-wide defaults for *ondisk_attrs* directive. Optional, default is 0 (all attributes are loaded in memory). This directive lets you specify the default value of ondisk_attrs for all indexes served by this copy of searchd. Per-index directives take precedence, and will overwrite this instance-wide default value, allowing for fine-grain control.

### 11.5.32 persistent_connections_limit

The maximum # of simultaneous persistent connections to remote *persistent agents*. Each time connecting agent defined under 'agent_persistent' we try to reuse existing connection (if any), or connect and save the connection for the future. However we can't hold unlimited # of such persistent connections, since each one holds a worker on agent

size (and finally we'll receive the 'maxed out' error, when all of them are busy). This very directive limits the number. It affects the num of connections to each agent's host, across all distributed indexes.

It is reasonable to set the value equal or less than *max_children* option of the agents.

Example:

```
persistent_connections_limit = 29 # assume that each host of agents has max_children␣
↪= 30 (or 29).
```

### 11.5.33 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 Manticore 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
```

### 11.5.34 predicted_time_costs

Costs for the query time prediction model, in nanoseconds. Optional, default is "doc=64, hit=48, skip=2048, match=64" (without the quotes).

Terminating queries before completion based on their execution time (via either *SetMaxQueryTime()* API call, or *SELECT ... OPTION max_query_time* SphinxQL statement) is a nice safety net, but it comes with an inborn drawback: indeterministic (unstable) results. That is, if you repeat the very same (complex) search query with a time limit several times, the time limit will get hit at different stages, and you will get *different* result sets.

There is a new option, *SELECT ... OPTION max_predicted_time*, that lets you limit the query time *and* get stable, repeatable results. Instead of regularly checking the actual current time while evaluating the query, which is indeterministic, it predicts the current running time using a simple linear model instead:

```
predicted_time =
    doc_cost * processed_documents +
    hit_cost * processed_hits +
    skip_cost * skiplist_jumps +
    match_cost * found_matches
```

The query is then terminated early when the `predicted_time` reaches a given limit.

Of course, this is not a hard limit on the actual time spent (it is, however, a hard limit on the amount of *processing* work done), and a simple linear model is in no way an ideally precise one. So the wall clock time *may* be either below or over the target limit. However, the error margins are quite acceptable: for instance, in our experiments with a 100 msec target limit the majority of the test queries fell into a 95 to 105 msec range, and *all* of the queries were in a 80 to 120 msec range. Also, as a nice side effect, using the modeled query time instead of measuring actual run time results in somewhat less gettimeofday() calls, too.

No two server makes and models are identical, so `predicted_time_costs` directive lets you configure the costs for the model above. For convenience, they are integers, counted in nanoseconds. (The limit in max_predicted_time is counted in milliseconds, and having to specify cost values as 0.000128 ms instead of 128 ns is somewhat more error prone.) It is not necessary to specify all 4 costs at once, as the missed one will take the default values. However, we strongly suggest to specify all of them, for readability.

Example:

```
predicted_time_costs = doc=128, hit=96, skip=4096, match=128
```

### 11.5.35 preopen_indexes

Whether to forcibly preopen all indexes on startup. Optional, default is 1 (preopen everything).

When set to 1, this directive overrides and enforces *preopen* on all indexes. They will be preopened, no matter what is the per-index `preopen` setting. When set to 0, per-index settings can take effect. (And they default to 0.)

Pre-opened indexes avoid races between search queries and rotations that can cause queries to fail occasionally. They also make `searchd` use more file handles. In most scenarios it's therefore preferred and recommended to preopen indexes.

Example:

```
preopen_indexes = 1
```

### 11.5.36 qcache_max_bytes

Integer, in bytes. The maximum RAM allocated for cached result sets. Default is 0, meaning disabled. Refer to *query cache* for details.

```
qcache_max_bytes = 16777216
```

### 11.5.37 qcache_thresh_msec

Integer, in milliseconds. The minimum wall time threshold for a query result to be cached. Defaults to 3000, or 3 seconds. 0 means cache everything. Refer to *query cache* for details.

### 11.5.38 qcache_ttl_sec

Integer, in seconds. The expiration period for a cached result set. Defaults to 60, or 1 minute. The minimum possible value is 1 second. Refer to *query cache* for details.

### 11.5.39 query_log_format

Query log format. Optional, allowed values are 'plain' and 'sphinxql', default is 'plain'.

The default one logs queries in a custom text format. The 'sphinxql' logs valid SphinxQL statements. This directive allows to switch between the two formats on search daemon startup. The log format can also be altered on the fly, using `SET GLOBAL query_log_format=sphinxql` syntax. Refer to *Query log formats* for more discussion and format details.

Example:

```
query_log_format = sphinxql
```

### 11.5.40 query_log_min_msec

Limit (in milliseconds) that prevents the query from being written to the query log. Optional, default is 0 (all queries are written to the query log). This directive specifies that only queries with execution times that exceed the specified limit will be logged.

### 11.5.41 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 *Query log formats*. In case of 'plain' format, you can use the 'syslog' as the path to the log file. In this case all search queries will be sent to syslog daemon with LOG_INFO priority, prefixed with '[query]' instead of timestamp. To use the syslog option the sphinx must be configured '–with-syslog' on building.

Example:

```
query_log = /var/log/query.log
```

### 11.5.42 query_log_mode

By default the searchd and query log files are created with 600 permission, so only the user under which daemon runs and root users can read the log files. query_log_mode allows settings a different permission. This can be handy to allow other users to be able to read the log files (for example monitoring solutions running on non-root users).

Example:

```
query_log_mode  = 666
```

### 11.5.43 queue_max_length

Maximum pending queries queue length for workers=thread_pool mode, default is 0 (unlimited).

In case of high CPU load thread pool queries queue may grow all the time. This directive lets you constrain queue length and start rejecting incoming queries at some point with a "maxed out" message.

### 11.5.44 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
```

### 11.5.45 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
```

### 11.5.46 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
```

### 11.5.47 rt_flush_period

RT indexes RAM chunk flush check period, in seconds. Optional, default is 10 hours.

Actively updated RT indexes that however fully fit in RAM chunks can result in ever-growing binlogs, impacting disk use and crash recovery time. With this directive the search daemon performs periodic flush checks, and eligible RAM chunks can get saved, enabling consequential binlog cleanup. See *Binary logging* for more details.

Example:

```
rt_flush_period = 3600 # 1 hour
```

### 11.5.48 rt_merge_iops

A maximum number of I/O operations (per second) that the RT chunks merge thread is allowed to start. Optional, default is 0 (no limit).

This directive lets you throttle down the I/O impact arising from the `OPTIMIZE` statements. It is guaranteed that all the RT optimization activity will not generate more disk iops (I/Os per second) than the configured limit. Modern SATA drives can perform up to around 100 I/O operations per second, and limiting rt_merge_iops can reduce search performance degradation caused by merging.

Example:

```
rt_merge_iops = 40
```

### 11.5.49 rt_merge_maxiosize

A maximum size of an I/O operation that the RT chunks merge thread is allowed to start. Optional, default is 0 (no limit).

This directive lets you throttle down the I/O impact arising from the `OPTIMIZE` statements. I/Os bigger than this limit will be broken down into 2 or more I/Os, which will then be accounted as separate I/Os with regards to the *rt_merge_iops* limit. Thus, it is guaranteed that all the optimization activity will not generate more than (rt_merge_iops * rt_merge_maxiosize) bytes of disk I/O per second.

Example:

```
rt_merge_maxiosize = 1M
```

### 11.5.50 seamless_rotate

Prevents `searchd` stalls while rotating indexes with huge amounts of data to precache. Optional, default is 1 (enable seamless rotation). On Windows systems seamless rotation is disabled by default.

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 noticeable 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
```

### 11.5.51 shutdown_timeout

searchd –stopwait wait time, in seconds. Optional, default is 3 seconds.

When you run searchd –stopwait your daemon needs to perform some activities before stopping like finishing queries, flushing RT RAM chunk, flushing attributes and updating binlog. And it requires some time. searchd –stopwait will wait up to shutdown_time seconds for daemon to finish its jobs. Suitable time depends on your index size and load.

Example:

```
shutdown_timeout = 5 # wait for up to 5 seconds
```

### 11.5.52 snippets_file_prefix

A prefix to prepend to the local file names when generating snippets. Optional, default is empty.

This prefix can be used in distributed snippets generation along with `load_files` or `load_files_scattered` options.

Note how this is a prefix, and **not** a path! Meaning that if a prefix is set to "server1" and the request refers to "file23", `searchd` will attempt to open "server1file23" (all of that without quotes). So if you need it to be a path, you have to mention the trailing slash.

Note also that this is a local option, it does not affect the agents anyhow. So you can safely set a prefix on a master server. The requests routed to the agents will not be affected by the master's setting. They will however be affected by the agent's own settings.

This might be useful, for instance, when the document storage locations (be those local storage or NAS mountpoints) are inconsistent across the servers.

Example:

```
snippets_file_prefix = /mnt/common/server1/
```

### 11.5.53 sphinxql_state

Path to a file where current SphinxQL state will be serialized.

On daemon startup, this file gets replayed. On eligible state changes (eg. SET GLOBAL), this file gets rewritten automatically. This can prevent a hard-to-diagnose problem: If you load UDF functions, but Manticore crashes, when it gets (automatically) restarted, your UDF and global variables will no longer be available; using persistent state helps a graceful recovery with no such surprises.

Example:

```
sphinxql_state = uservars.sql
```

### 11.5.54 sphinxql_timeout

Maximum time to wait between requests (in seconds) when using sphinxql interface. Optional, default is 15 minutes.

Example:

```
sphinxql_timeout = 900
```

### 11.5.55 subtree_docs_cache

Max common subtree document cache size, per-query. Optional, default is 0 (disabled).

Limits RAM usage of a common subtree optimizer (see *Multi-queries*). At most this much RAM will be spent to cache document entries per each query. Setting the limit to 0 disables the optimizer.

Example:

```
subtree_docs_cache = 8M
```

### 11.5.56 subtree_hits_cache

Max common subtree hit cache size, per-query. Optional, default is 0 (disabled).

Limits RAM usage of a common subtree optimizer (see *Multi-queries*). At most this much RAM will be spent to cache keyword occurrences (hits) per each query. Setting the limit to 0 disables the optimizer.

Example:

```
subtree_hits_cache = 16M
```

### 11.5.57 thread_stack

Per-thread stack size. Optional, default is 1M.

In the `workers = threads` mode, every request is processed with a separate thread that needs its own stack space. By default, 1M per thread are allocated for stack. However, extremely complex search requests might eventually exhaust the default stack and require more. For instance, a query that matches a thousands of keywords (either directly or through term expansion) can eventually run out of stack. `searchd` attempts to estimate the expected stack use, and blocks the potentially dangerous queries. To process such queries, you can either set the thread stack size by using the `thread_stack` directive (or switch to a different `workers` setting if that is possible).

A query with N levels of nesting is estimated to require approximately 30+0.16*N KB of stack, meaning that the default 64K is enough for queries with upto 250 levels, 150K for upto 700 levels, etc. If the stack size limit is not met, `searchd` fails the query and reports the required stack size in the error message.

Example:

```
thread_stack = 256K
```

### 11.5.58 unlink_old

Whether to unlink .old index copies on successful rotation. Optional, default is 1 (do unlink).

Example:

```
unlink_old = 0
```

### 11.5.59 watchdog

Threaded server watchdog. Optional, default is 1 (watchdog enabled).

A crashed query in `threads` multi-processing mode (`:ref:`workers` = threads`) can take down the entire server. With watchdog feature enabled, `searchd` additionally keeps a separate lightweight process that monitors the main server process, and automatically restarts the latter in case of abnormal termination. Watchdog is enabled by default.

Example:

```
watchdog = 0 # disable watchdog
```

### 11.5.60 workers

Multi-processing mode (MPM). Optional; allowed values are thread_pool, and threads. Default is thread_pool.

Lets you choose how `searchd` processes multiple concurrent requests. The possible values are:

- threads
- A new dedicated thread is created on every incoming network connection. Subsequent queries on that connection are handled by that thread. When a client disconnected, the thread gets killed.
- thread_pool
- A worker threads pool is created on daemon startup. An internal network thread handles all the incoming network connections. Subsequent queries on any connection are then put into a queue, and processed in order by the first avaialble worker thread from the pool. No threads are normally created or killed, neither for new connections, nor for new queries. Network thread uses epoll() and poll() on Linux, kqueue() on Mac OS/BSD, and WSAPoll on Windows (Vista and later). This is the default mode.

Thread pool is a newer, better, faster implementation of threads mode which does not suffer from overheads of creating a new thread per every new connection and managing a lot of parallel threads. We still retain workers=threads for the transition period, but thread pool is scheduled to become the only MPM mode.

Example:

```
workers = thread_pool
```

# CHAPTER 12

# Reporting bugs

Unfortunately, Manticore is not yet 100% bug free (even though we're 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, we need to be able either to reproduce and fix 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.

## 12.1 Bug-tracker

Please issue the Github issue tracker https://github.com/manticoresoftware/manticore/issues. Create a new ticket and describe your bug in details so both you and developers can save their time.

## 12.2 Crashes

In case of crashes we sometimes can get enough info to fix from backtrace.

Manticore tries to write crash backtrace to its log file. It may look like this:

```
./indexer(_Z12sphBacktraceib+0x2d6)[0x5d337e]
./indexer(_Z7sigsegvi+0xbc)[0x4ce26a]
/lib64/libpthread.so.0[0x3f75a0dd40]
/lib64/libc.so.6(fwrite+0x34)[0x3f74e5f564]
./indexer(_
↪ZN27CSphCharsetDefinitionParser5ParseEPKcR10CSphVectorI14CSphRemapRange16CSphVe
ctorPolicyIS3_EE+0x5b)[0x51701b]
./indexer(_ZN13ISphTokenizer14SetCaseFoldingEPKcR10CSphString+0x62)[0x517e4c]
./indexer(_ZN17CSphTokenizerBase14SetCaseFoldingEPKcR10CSphString+0xbd)[0x518283]
./indexer(_ZN18CSphTokenizer_SBCSILb0EEC1Ev+0x3f)[0x5b312b]
./indexer(_Z22sphCreateSBCSTokenizerv+0x20)[0x51835c]
./indexer(_
↪ZN13ISphTokenizer6CreateERK21CSphTokenizerSettingsPK17CSphEmbeddedFilesR10CSphS
```

(continues on next page)

(continued from previous page)

```
tring+0x47)[0x5183d7]
./indexer(_Z7DoIndexRK17CSphConfigSectionPKcRK17SmallStringHash_TIS_EbP8_IO_
↪FILE+0x494)[0x
4d31c8]
./indexer(main+0x1a17)[0x4d6719]
/lib64/libc.so.6(__libc_start_main+0xf4)[0x3f74e1d8a4]
./indexer(__gxx_personality_v0+0x231)[0x4cd779]
```

This is an example of a good backtrace - we can see mangled function names here.

But sometimes backtrace may look like this:

```
/opt/piler/bin/indexer[0x4c4919]
/opt/piler/bin/indexer[0x405cf0]
/lib/x86_64-linux-gnu/libpthread.so.0(+0xfcb0)[0x7fc659cb6cb0]
/opt/piler/bin/indexer[0x4237fd]
/opt/piler/bin/indexer[0x491de6]
/opt/piler/bin/indexer[0x451704]
/opt/piler/bin/indexer[0x40861a]
/opt/piler/bin/indexer[0x40442c]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xed)[0x7fc6588aa76d]
/opt/piler/bin/indexer[0x405b89]
```

Developers can get nothing useful from those cryptic numbers. They're ordinary humans and want to see function names. To help them you need to provide symbols (function and variable names). If you've installed sphinx by building from the sources, run the following command over your binary:

```
nm -n indexer > indexer.sym
```

Attach this file to bug report along with backtrace. You should however ensure that the binary is not stripped. Our official binary packages should be fine. (That, or we have the symbols stored.) However, if you manually build Manticore from the source tarball, do not run `strip` utility on that binary, and/or do not let your build/packaging system do that!

## 12.3 Uploading your data

To fix your bug developers often need to reproduce it on their machines. To do this they need your sphinx.conf, index files, binlog (if present), sometimes data to index (like SQL tables or XMLpipe2 data files) and queries.

Attach your data to ticket. In case it's too big to attach ask developers and they give you an address to write-only FTP created exactly for such puproses.

# CHAPTER 13

# Release notes

## 13.1 Version 2.6.2 GA, 23 February 2018

### 13.1.1 Improvements

- improved *Percolate Queries* performance in case of using NOT operator and for batched documents.
- *CALL PQ* can use multiple threads depending on *dist_threads*
- new full-text matching operator NOTNEAR/N
- LIMIT for SELECT on percolate indexes
- *expand_keywords* can accept 'start','exact' (where 'star,exact' has same effect as '1')
- ranged-main-query for *joined fields* which uses the ranged query defined by sql_query_range

### 13.1.2 Bugfixes

- 72dcf66 fixed crash on searching ram segments; deadlock on save disk chunk with double buffer; deadlock on save disk chunk during optimize
- 3613714 fixed indexer crash on xml embeded schema with empty attribute name
- 48d7e80 fixed erroneous unlinking of not-owned pid-file
- a5563a4 fixed orphaned fifos sometimes left in temp folder
- 2376e8f fixed empty FACET result set with wrong NULL row
- 4842b67 fixed broken index lock when running daemon as windows service
- be35fee fixed wrong iconv libs on mac os
- 83744a9 fixed wrong count(*)

## 13.2 Version 2.6.1 GA, 26 January 2018

### 13.2.1 Improvements

- *agent_retry_count* in case of agents with mirrors gives the value of retries per mirror instead of per agent, the total retries per agent being agent_retry_count*mirrors.
- *agent_retry_count* can now be specified per index, overriding global value. An alias *mirror_retry_count* is added.
- a retry_count can be specified in agent definition and the value represents retries per agent
- Percolate Queries are now in HTTP JSON API at */json/pq*.
- Added -h and -v options (help and version) to executables
- *morphology_skip_fields* support for Real-Time indexes

### 13.2.2 Bugfixes

- a40b079 fixed ranged-main-query to correctly work with sql_range_step when used at MVA field
- f2f5375 fixed issue with blackhole system loop hung and blackhole agents seems disconnected
- 84e1f54 fixed query id to be consistent, fixed duplicated id for stored queries
- 1948423 fixed daemon crash on shutdown from various states
- 9a706b 3495fd7 timeouts on long queries
- 3359bcd8 refactored master-agent network polling on kqueue-based systems (Mac OS X, BSD).

## 13.3 Version 2.6.0, 29 December 2017

### 13.3.1 Features and improvements

- *HTTP JSON*: JSON queries can now do equality on attributes, MVA and JSON attributes can be used in inserts and updates, updates and deletes via JSON API can be performed on distributed indexes
- *Percolate Queries*
- Removed support for 32-bit docids from the code. Also removed all the code that converts/loads legacy indexes with 32-bit docids.
- Morphology only for certain fields . A new index directive morphology_skip_fields allows defining a list of fields for which morphology does not apply.
- expand_keywords can now be a query runtime directive set using the OPTION statement

### 13.3.2 Bugfixes

- 0cfae4c fixed crash on debug build of daemon (and m.b. UB on release) when built with rlp
- 324291e fixed RT index optimize with progressive option enabled that merges kill-lists with wrong order
- ac0efee minor crash on mac
- lots of minor fixes after thorough static code analysis

- other minor bugfixes

### 13.3.3 Upgrade

In this release we've changed internal protocol used by masters and agents to speak with each other. In case you run Manticoresearch in a distributed environment with multiple instances make sure your first upgrade agents, then the masters.

## 13.4 Version 2.5.1, 23 November 2017

### 13.4.1 Features and improvements

- JSON queries on *HTTP API protocol*. Supported search, insert, update, delete, replace operations. Data manipulation commands can be also bulked, also there are some limitations currently as MVA and JSON attributes can't be used for inserts, replaces or updates.
- *RELOAD INDEXES* command
- *FLUSH LOGS* command
- *SHOW THREADS* can show progress of optimize, rotation or flushes.
- GROUP N BY work correctly with MVA attributes
- blackhole agents are run on separate thread to not affect master query anymore
- implemented reference count on indexes, to avoid stalls caused by rotations and high load
- SHA1 hashing implemented, not exposed yet externally
- fixes for compiling on FreeBSD, macOS and Alpine

### 13.4.2 Bugfixes

- 989752b filter regression with block index
- b1c3864 rename PAGE_SIZE -> ARENA_PAGE_SIZE for compatibility with musl
- f2133cc disable googletests for cmake < 3.1.0
- f30ec53 failed to bind socket on daemon restart
- 0807240 fixed crash of daemon on shutdown
- 3e3acc3 fixed show threads for system blackhole thread
- 262c3fe Refactored config check of iconv, fixes building on FreeBSD and Darwin

## 13.5 Version 2.4.1 GA, 16 October 2017

### 13.5.1 Features and improvements

- OR operator in WHERE clause between attribute filters
- Maintenance mode ( SET MAINTENANCE=1)

- *CALL KEYWORDS* available on distributed indexes
- *Grouping in UTC*
- *query_log_mode* for custom log files permissions
- Field weights can be zero or negative
- *max_query_time* can now affect full-scans
- added *net_wait_tm*, *net_throttle_accept net_throttle_action* and *net_throttle_accept net_throttle_action* for network thread fine tuning (in case of workers=thread_pool)
- COUNT DISTINCT works with facet searches
- IN can be used with JSON float arrays
- multi-query optimization is not broken anymore by integer/float expressions
- *SHOW META* shows a `multiplier` row when multi-query optimization is used

### 13.5.2 Compiling

Manticore Search is built using cmake and the minimum gcc version required for compiling is 4.7.2.

### 13.5.3 Folders and service

Manticore Search runs under `manticore` user.

Default data folder is now `/var/lib/manticore/`.

Default log folder is now `/var/log/manticore/`.

Default pid folder is now `/var/run/manticore/`.

### 13.5.4 Bugfixes

- a58c619 fixed SHOW COLLATION statement that breaks java connector
- 631cf4e fixed crashes on processing distributed indexes; added locks to distributed index hash; removed move and copy operators from agent
- 942bec0 fixed crashes on processing distributed indexes due to parallel reconnects
- e5c1ed2 fixed crash at crash handler on store query to daemon log
- 4a4bda5 fixed a crash with pooled attributes in multiqueries
- 3873bfb fixed reduced core size by prevent index pages got included into core file
- 11e6254 fixed searchd crashes on startup when invalid agents are specified
- 4ca6350 fixed indexer reports error in sql_query_killlist query
- 123a9f0 fixed fold_lemmas=1 vs hit count
- cb99164 fixed inconsistent behavior of html_strip
- e406761 fixed optimize rt index loose new settings; fixed optimize with sync option lock leaks;
- 86aeb82 Fixed processing erroneous multiqueries
- 2645230 fixed result set depends on multi-query order

- 72395d9 fixed daemon crash on multi-query with bad query
- f353326 fixed shared to exclusive lock
- 3754785 fixed daemon crash for query without indexes
- 29f360e fixed dead lock of daemon

## 13.6 Version 2.3.3, 06 July 2017

- Manticore branding