# Free/open source information retrieval libraries

# What are they and why using one

Information retrieval libraries are software libraries that provides functionality for searching within databases and documents within them. In particular, this often refers to searching text document for combinations of words and/or phrases.

Database software such as MySQL are very good at storing and managing data but many of them are poor at searching text. One can bundle an information retrieval library with database software to add good searching capability to databases. Take websites for an example. The main database can handle insert/update/delete/view operations of articles and the search library can handle the in-site search engine.

# The choices

# MySQL

Although mainly a database server, MySQL actually has built-in full-text search functions.

## How it runs

Full-text search is different from “LIKE” and “RLIKE” operations. For the former, MySQL breaks down the text into words, removes stopwords and operates at the word level. The columns can be full-text searched by MATCH() ... AGAINST syntax. See MySQL’s doc for details.

## Supported queries

1. Word and phrase search

2. Word prefix search (word*)
3. Simple boosting (marking a word/phrase as having an increased/decreased contribution)
4. Boolean operations (AND, OR, NOT)
5. Default or customizable stopword list
6. Sorting (scores are returned as floating-point numbers so the result can be sorted by that "field" using ORDER BY)

# Data storage

For MyISAM tables, full-text indices can be created for CHAR, VARCHAR or TEXT columns. A column can be full-text indexed by creating an index of it with the type FULLTEXT. MySQL creates, updates and stores the full-text indices automatically in the .MYI file along with other indices.

# Performance

Indexing is fast (~4 minutes for about 30000 documents). However, searching is slow, as each query takes seconds (from less than 1 second up to about 35 seconds in our test).

# Resource usage

Full-text search is part of MySQL server so mearsuring its resource usage is hard. We haven't noticed increased memory usage when full-text queries are used though.

# Sphinx

Sphinx is relatively new and written by a single programmer (Andrew Aksyonoff). Features are added in an ad-hoc matter and are limited. But it's very fast (probably faster than anything else out there) for both indexing and searching. For a majority of document-retrieval systems (e.g. websites, knowledge bases), Sphinx probably has everything they need.

# How it runs

Sphinx is written in C++. Other programs can link against Sphinx and use its functions. Alternatively, Sphinx provides a search daemon (searchd). Programs can talk to it and query against its indices using socket communication.

There are APIs for PHP (standard), Ruby (see UltraSphinx) and Java (standard). The APIs provide functions in these languages that when called, serialize and send the queries to searchd (and then receives answers and unserialize the result).

There is a plug-in (storage engine to be exact) for MySQL 5 called SphinxSE. You can query against a Sphinx index just like querying a MySQL database.

There is no plug-in architecture within Sphinx yet. If you want to extending its functionality, you have to modify the source and re-compile.

# Supported queries

1. Word and phrase search
2. Boolean operations (AND, OR, NOT), although nesting is not supported
3. Word-word proximity search (i.e. a number of words within a distance of each other)
4. Multi-field query
5. Grouping of records by date on fields that are defined as UNIX timestamps (like SQL's GROUP BY)
6. Sorting by fields, ascending or descending, single or multi-field

# Data storage

Sphinx stores integers (including UNIX timestamps, which Sphinx has special support for). Although Sphinx indexes strings, tokenized or not, it does not store them. Therefore, the documents have to be stored in MySQL. The work flow for searching is like this:

1. Translate a system query into a Sphinx query.
2. Give Sphinx the query and get back a list of document IDs.
3. Ask MySQL for the document records with the given document IDs.
4. Process the document records.

Sphinx supports updating the index, but according to the doc, it's not faster than rebuilding the index. An alternative is to use a "main+delta" scheme, as described in the doc.

# Performance

Both indexing and searching are very fast. In our case, it indexes 40000 documents (1 GB) in about 3 minutes. Queries are executed in the order of 10 ms each (although some are in the 500ms range). Queries seem to be cached and repeated queries are executed even faster.

# Resource usage

Sphinx is easy on system resource. You can specify how much memory the indexer uses. The default (32 MB) satisfies our need. The search daemon uses about 10 MB of memory. CPU usage is also very low.

# Lucene

Lucene is older and considered more mature. It is used in many big projects such as Wikipedia. It might take more time to fine-tune (partly because there are so many ways to do things). If you need the extra functionality, Lucene is worth trying.

Lucene In Action is a must-read if you want to use this library.

# How it runs

Lucene is embedded inside the programs that use it (think Berkeley DB).

There are ports of Lucene to PHP, Ruby, C, C++, C#, Delphi and others. See Lucene's website for details. Ports usually lag behind the main Lucene project. To use the main project with other programming languages, refer to PHP/Java Bridge (which has been renamed to VMBridge and is adding support for other languages).

For ready-to-use search-daemon-like service, refer to "Solr:http://lucene.apache.org/solr/ , which is "open source enterprise search server based on the Lucene search library, with XML/HTTP and JSON APIs, hit highlighting, faceted search, caching, replication, and a web administration interface."

Lucene does not do specific processing such as keyword-highlighting or web-crawling. Related projects (e.g. Solr, Nutch) fill in these gaps.

# Supported queries

Lucene provides the "building-blocks" for a search system. In other words, you build the system yourself, but you can build it any way you want. For example, the several primitive types of queries are sufficient to build about any kind of query you want.

1. Word and phrase search
2. Boolean operations (AND, OR, NOT) with or without nesting
3. Span queries (a span is a [document, starting location, ending location] tuple) that can build proximity search and more
4. Multi-field query
5. Grouping is not build-in but you can build it yourself
6. Sorting by scores, by field values or using your own function

The biggest gain of building thing at this low level is flexibilty (of building whatever you like) and the additional information about the documents and the searh process that other "ready-to-run" libraries don't give you. For example:

1. In addition to TermDocs ([term, document] tuples grouped by term), you have access to TermPositions ([term, document, position] tuples grouped by term) and term vectors (document, term, position] tuples grouped by document). Such information is very useful for doing statistics things fast (from word count to something more complex).
2. You can control how Lucene selects (using Query and Filter classes) and scores (using Scorer and Similarity classes) the documents.
3. You can trace how Lucene selects and scores the documents (using the Explanation class).

# Data storage

Lucene supports storing and/or indexing and/or tokenizing text. Note that it treats everything as strings.

In theory, it can replace the database and still perform well on many kinds of applications. Not sure how it would perform though.

# Performance

Indexing is moderately fast (about 8 minutes for the same 40000-document database) Query time varies greatly from 10ms to about 500ms. Repeated queries also take less time.

# Resource usage

Lucene is more resource-intensive than Sphinx, especially on the memory side. You will have to measure the difference youself. Our Lucene service (Lucene loaded by PHP/Java bridge) process is about 100 MB all the time, though we haven’t tested how much of that memory is actually used. Java VMs usually have large startup-lags, so starting a “service” is much better than starting a new Java VM every time. However, watch for “memory leak” problem if you are going to do that. Prepared to spend some time tuning GC behavior too.

---

Revision #3
Created 2008-03-06 13:24:45 UTC by Chan, Wing Kai
Updated 2008-07-20 22:22:00 UTC by Chan, Wing Kai