Getting Started with Xapian Documentation

Release 0.1

Gaurav Arora, James Aylett, Jenny Black, Olly Betts, Richard Bou

October 28, 2014

Contents:

CHAPTER 1

Introduction

Xapian is an open source search engine library, which allows developers to add advanced indexing and search facilities to their own applications.

This document aims to be a guide to getting up and running with your first database, explaining basic concepts and providing code examples of the library's core functionality.

If you just want to follow our code examples, you can skip the chapter on "Core Concepts" and go straight to *A practical example* - but you should probably make sure you have Xapian installed first!

Todo

Mention Omega here and link to its documentation

1.1 Installation

There are two pieces of Xapian you need to following this guide: the library itself, and support for the language you're going to be using. This guide was originally written with examples in Python, and we're also working on full translations into PHP and C++. Help with translating the examples into other languages would be most welcome.

This guide documents Xapian 1.2 (except where a different version is explicitly mentioned) so you'll find it easier to follow if you use a version from the 1.2 release series. So let's get that onto your system.

1.1.1 Installation on Debian or Ubuntu

Recent versions of both Debian and Ubuntu have Xapian 1.2 packaged: if you're using Debian 6.0 (squeeze) or later, or Ubuntu 11.10 (Oneiric Ocelot) or later you can just do one of the following depending on whether you want to work through the examples in Python or C++:

```
$ sudo apt-get install python-xapian
$ sudo apt-get install libxapian-dev
```

If you're using Ubuntu 11.04 (Natty Narwhal) or earlier, you'll need to install from our PPA.

Packages of the PHP bindings aren't available due to a licence compatibility issue, but you can build your own packages.

1.1.2 Installation on other systems

Many operating systems have packages available to make Xapian easy to install; information is available on our download page. This covers most popular Linux distributions, FreeBSD, Mac OS (Python and C++ only) and Windows using Microsoft Visual Studio. If you're using a different operating system, you will need to compile from source, which should work on any Unix-like operating system, and Windows using any one of Cygwin, MSYS+mingw or MSVC. Source code is again available from our download page, as are additional Makefiles for building using MSVC on Windows.

1.2 Datasets and example code

If you want to run the code we use to demonstrate Xapian's features (and we recommend you do), you'll need both the code itself and the two datasets we use.

The example code is available in Python, PHP, and C++ so far, although there's only a complete set of examples for Python at present.

Todo

finalise datasets and code and link to them from here

For now, you'll want to grab the documentation source from github which contains the example code in each language, and also the data files listed in the next paragraph (both are in the "code" subdirectory).

The first dataset is the first 100 objects taken from museum catalogue data released by the Science Museum, and the second we have curated ourselves from information on Wikipedia about the 50 US States. Both are provided as gzipped CSV files. The first dataset is released under the Creative Commons license Attribution-NonCommercial-ShareAlike license, and the second under Creative Commons Attribution-Share Alike 3.0.

Todo

link to here from every howto and everything that needs the data files and example code

CHAPTER 2

Core concepts

2.1 Concurrency

2.1.1 Threading

Xapian does not provide explicit support for multi-threading, though it can be used in a multi-threaded program if you are aware of the details described below.

Xapian doesn't maintain any global state, so you can safely use Xapian in a multi-threaded program provided you don't share objects between threads. In practice this restriction is often not a problem - each thread can create its own `xapian.Database` object, and everything will work fine.

If you really want to access the same Xapian object from multiple threads, then you need to ensure that it won't ever be accessed concurrently (if you don't ensure this bad things are likely to happen - for example crashes or even data corruption). One way to prevent concurrent access is to require that a thread gets an exclusive lock on a mutex while the access is made.

Xapian doesn't include thread locking code to avoid imposing an overhead when it isn't needed. And in practice the caller can often lock over several operations, which wouldn't work if the locking code was in Xapian itself.

Be aware that some Xapian objects will keep internal references to others - for example, if you call `xapian.Database.get_document()`, the resulting `xapian.Document` object will keep a reference to the `xapian.Database` object, and so you can't safely use the `xapian.Database` object in one thread at the same time as using the `xapian.Document` object in another.

2.2 Indexing concepts

2.2.1 Databases

Pretty much all Xapian operations revolve around a Xapian database. Before searches can be performed, details of the documents to be searched need to be put into a database; the search process then refers to the database to efficiently determine the best matches for a given query. The process of putting documents into the database is usually referred to as *indexing*.

The main information stored in a database is a mapping from each term to a list of all the documents it occurs in, together with various statistics about these occurrences. It may also store the full text, or extracts, from the documents, so that result summaries can be displayed. Databases can also contain additional data for spelling correction and synonym expansion, and developers can even store arbitrary key-value pairs in part of the database.

Todo

split into sections

Todo

list backends

Todo

disk-based brass/chert/(flint quartz how to upgrade); autodetecton

---

Todo

inmemory; use ram disk for serious use

Todo

remotes

Todo

multiple databases

Todo

stubs

Xapian databases store data in custom formats which allow searches to be performed extremely quickly; Xapian does not use a relational database as its datastore. There are several database backends; the main backend in the 1.2 release series of Xapian is called the *Chert* backend. This stores information in the filesystem (under a given path). If you're familiar with data storage structures, you might be interested to know that this backend uses a copy-on-write B+-tree structure, but don't worry if that doesn't mean anything to you!

Most backend formats (and certainly the main backend format for each release) will allow updates to be grouped into transactions, and will allow at least some old versions of the database to be searched while new ones are being written. Currently, all the backends only support a single writer existing at a given time; attempting to open another writer on the same database will throw `xapian.DatabaseLockError` to indicate that it wasn't possible to acquire a lock. Multiple concurrent readers are supported (in addition to the writer).

When a database is opened for reading, a fixed snapshot of the database is referenced by the reader, (essentially Multi-Version Concurrency Control). Updates which are made to the database will not be visible to the reader unless it calls `xapian.Database.reopen()`.

Currently Xapian's disk based backends have a limitation to their *multi-version concurrency* implementation - specifically, at most two versions can exist concurrently. Therefore a reader will be able to access its snapshot of the database without limitations when only one change has been made and committed by the writer, but after the writer has made two changes, readers will receive a `xapian.DatabaseModifiedError` if they attempt to access a part of the database which has changed. In this situation, the reader can be updated to the latest version using the `xapian.Database.reopen()` method.

With the disk-based Xapian backends, when a database is opened for writing, a lock is obtained on the database to ensure that no further writers are opened concurrently. This lock will be released when the database writer is closed (or automatically if the writer process dies).

One unusual feature of Xapian's locking mechanism (at least on POSIX operating systems) is that Xapian forks a subprocess to hold the lock, rather than holding it in the main process. This is to avoid the lock being accidentally released due to the slightly unhelpful semantics of fcntl locks.

There's also a *remote* database backend which allows the database to be located on a different machine and accessed via a custom network protocol.

It is possible to perform searches across multiple databases at once, and Xapian will handle merging the results together appropriately. This feature can be combined with remote databases to handle datasets which are too large for a single machine, by performing searches across multiple remote databases.

Xapian also has special support for replicating databases to multiple machines, such that only the parts of the database which have been modified are copied; this can be useful for redundancy and load-balancing purposes.

Xapian also supports a simple text file format for listing the locations of a set of databases (either on the local file system, or remote databases). Such files are called *stub-databases*, and can be used to point to a database when the physical database location may vary; for example, because a new database is being built nightly, and is named according to the date on which it was built.

2.2.2 Documents

A document in Xapian is simply an item which is returned by a search. When building a new search system, a key thing to decide is what the documents in your system are going to be. There's often an obvious choice here, but in many cases there are alternatives. For example, for a search over a website, it seems natural to have one document for each page of the site. However, you could instead choose to use one document for each paragraph of each page, or to group pages together into subjects and have one document for each subject.

Documents are identified in a database by a unique positive integer id, known as the *document ID*. Currently this is a 32 bit quantity.

Documents have three components: *data*, *terms* and *values*. We'll discuss terms and data first - values are useful for some more advanced search types.

Document Data

The *document data* is an arbitrary binary blob of data associated with the document. Xapian treats this as completely opaque, and does nothing with this data other than storing it in the database (compressed with zlib if it is compressible) and returning it when requested.

It can be used to hold a reference to the document elsewhere (such as the primary key in an external database table), or could be used to store the full text of the document.

Generally, the best thing to do with the document data is to store any information you need in order to display the resulting document to the user (or to whatever process consumes the results of searches). Xapian doesn't enforce a serialisation scheme for putting structured information into the document data: depending on your application, you might want to implement a simple scheme using newlines to separate values, use JSON or XML serialisation, or use some other method of pickling data.

2.2.3 Terms

Terms are the basis of most searches in Xapian. At its simplest, a search is a process of comparing the terms specified in a query against the terms in each document, and returning a list of the documents which match the best.

A term will often be generated for each word in a piece of text, usually by applying some form of normalisation (it's usual to at least change all the characters to be lowercase). There are many useful strategies for producing terms.

Often the same word will occur multiple times in a piece of text. Xapian calls this number the *within document frequency* and stores it in the database. It is often useful when searching to give documents in which a term occurs more often a higher weight.

It is also possible to store a set of positions along with each term; this allows the positions at which words occur to be used when searching, e.g., in a phrase query. These positions are usually stored as word counts (rather than character or byte counts).

The database also keeps track of a couple of statistics about the frequency of terms in the database; the *term frequency* is the number of documents that a particular term occurs in, and the *collection frequency* is the total number of times that term occurs (i.e., the sum of the within document frequencies for the term).

Stemmers

A common form of normalisation is *stemming*. This process converts various different forms of words to a single form: for example, converting a plural (e.g., "birds") and a singular form of a word ("bird") to the same thing (in this case, both are converted to "bird").

Note that the output of a stemmer is not necessarily a valid word; what is important is that words with closely related meaning are converted to the same form, allowing a search to find them. For example, both the word "happy" and the word "happiness" are converted to the form "happi", so if a document contained "happiness", a search for "happy" would find that document.

The rules applied by a stemmer are dependent on the language of the text; Xapian includes stemmers for more than a dozen languages (and for some languages there is a choice of stemmers), built using the Snowball language. We'd like to add stemmers for more languages - see the Snowball site for information on how to contribute.

2.2.4 Term Generator

Rather than force all users to write their own code to process text into terms for indexing, Xapian provides a `xapian.TermGenerator` class. This parses chunks of text, producing appropriate terms, and adds them to a document.

The `xapian.TermGenerator` can be configured to perform stemming (and stopwording) when generating terms. It can optionally store information about the positions of words within the text, and can apply field-specific prefixes to the generated terms to allow searches to be restricted to specific fields. It can also add additional information to the database for use when performing spelling correction.

If you're using the `xapian.TermGenerator` to process text in this way, you will probably want to use the *QueryParser* (described later) when performing searches.

2.2.5 Using identifiers with Xapian

Every document stored in a Xapian database has a unique positive integer id, either assigned automatically or manually.

Often the documents which you're indexing with Xapian will already have unique ids, and you'll want to be able to use these to reindex an updated version of an existing document, or delete an expired document from the Xapian index. There are two ways of approaching this.

One is to use a one-to-one mapping between your identifiers and Xapian docids. This will work if your identifiers are positive integers and they all fit within 32 bits (under about 4 billion).

The other is to use a special term containing your identifier, which will work for any type of identifier. Typically you will prefix this (by convention with 'Q') to avoid collisions with other terms. Terms have a limited length (245 bytes in chert), so if your unique identifiers are really long you'll need to do something more complicated.

For more information on both techniques, see our FAQ on this.

2.2.6 Values

Values are in a sense a more flexible version of terms. Each document can have a set of values associated with it, which hold pieces of data which can be useful during a search. These pieces of data could be things such as keys which you want to be able to sort the results on, numeric values used for range searches, or numbers to be used to affect the weight calculated for documents during the search.

Each value is stored in a numbered *slot*; so for example, a document might have a value indicating a category in slot 0, a value indicating a price in slot 1, and a value indicating some measure of the importance of the document in slot 10. It's fine to use widely separated slot numbers - the data isn't stored in a simple array. Slot numbers can be any 32 bit unsigned integer, except for `0xffffffff` which has a special meaning (it's `xapian.BAD_VALUENO` which is used to indicate things like "not sorting on any value slot").

The core of Xapian treats the contents of value slots as opaque binary strings, rather than having support for numeric value types. This becomes significant if you want to perform range searches based on a value stored in a slot, or to sort results based on the value stored in a slot. To allow this type of use of slots, Xapian provides a utility function, *sortable_serialise*, which serialises a numeric value into a binary string in such a way that the sort order of the resulting binary string matches the numeric sort order of the unserialised values.

In chert and later backends, the values for each slot are stored as a separate stream, so the cost of accessing values doesn't depend on how many slots are in use (unlike with the older flint backend, where all the values for a particular document were stored together).

This stream is stored as a series of chunks; the chunks are indexed primarily by the value slot number, and then by the document id of the first entry in the chunk - this means that the data for a particular slot will be stored together and it also provides the ability to efficiently skip ahead in a stream. So access to many values from a particular slot in ascending docid order is fairly efficient, which is the access pattern that you will generally get when values are used during the match.

Within a chunk, any common prefix between a value and the previous value in that chunk is compressed away by simply storing how much of the previous value to reuse, which typically saves a lot of space. Finding an entry within a chunk will require decoding the chunk up to that point, but this decoding is fairly cheap.

For performance it is important to keep the amount of data stored in the values to a minimum, since the values for a large number of documents may be read during the search - the more data that has to be read, the slower the search will be.

Developers are sometimes tempted to use the value slots to hold information needed to display a result. This means that loading that information will have to read values from several different slots - if you have ten fields stored in this way, that will mean approximately ten times as many blocks will need to be read for each result shown. So resist this temptation - information needed to display a result should be stored in the document's data area.

2.2.7 Index limitations

There are a few limitations on the data which can be stored in an index; these rarely get in the way, and are generally easy to work around if they do, but it's worth knowing about them.

Todo

add more and fill out those already here a little more

Term length

Terms are limited to 245 bytes in length (at least with the "chert" backend), but each zero byte in a term is currently internally encoded as two bytes, so the limit is less for a term which contains zero bytes.

Todo

actually, reference the FAQ instead of saying something here.

It's rarely useful to have longer terms, but one situation where it can be is if you're using something like a URL as an ID term. A workaround for the length limit is to use a hash of the URL instead of the ID - if a cryptographic hash (like SHA1) is used, the likelihood of a collision is low enough that it can be ignored.

Document data length

The document data has a length limit which depends on the blocksize and some other factors, but with the default block size of 8KB, the document data length limit will be somewhere over 100MB.

Document value length

Document values are limited in length to a similar length to document data, but for performance reasons you probably wouldn't want to store document values longer than a few tens of bytes, as reading multiple 100MB+ values during the match would be rather slow.

Document ID

Document IDs are (currently) 32-bit which limits you to 2^{32}-1 (nearly 4.3 billion) documents in a database. Document IDs for deleted documents aren't reused for when automatically assigning a new document ID, so this limit also includes documents you've deleted. You can effectively reclaim such no-longer-used document IDs by compacting the database.

B-tree block number

The B-trees use a 32-bit unsigned block count. The default blocksize is 8KB which limits you to 32TB tables. You can increase the blocksize if this is a problem, but it's best to do it before you create the database as otherwise you need to use xapian-compact to make a compacted copy of the database with the new blocksize, and that will take a while for such a large database. The maximum blocksize currently allowed is 64K, which limits you to 256TB tables.

OS file size

Any operating or filing system limit on file size obviously applies to Xapian. On modern platforms, you're unlikely to hit these limits (e.g. on Linux, ext4 allows files up to 16TB and filesystems up to 1EB, while btrfs allows files and filesystems up to 16EB (figures from Wikipedia).

Total document length limit

Chert stores the total length (i.e. number of terms) of all the documents in a database so it can calculate the average document length. This is currently stored as an unsigned 64-bit quantity so you're almost certain to hit another limit first.

2.3 Search concepts

2.3.1 Queries

Queries within Xapian are the mechanism by which documents are searched for within a database. They can be a simple search for text-based terms or a search based on the values assigned to documents, which can be combined using a number of different methods to produce more complex queries.

Simple Queries

The most basic query is a search for a single textual term. This will find all documents in the database which have that term assigned to them. For example, a search might be for the term “wood”.

Queries can also be used to match values assigned to documents by applying a *value operator* to a particular value slot.

When a query is executed, the result is a list of documents that match the query, together with a weight for each which indicates how good a match for the query that particular document is.

Logical Operators

Each query produces a list of documents with a weight according to how good a match each document is for that query. These queries can then be combined to produce a more complex tree-like query structure, with the operators acting as branches within the tree.

The most basic operators are the logical operators: OR, AND and AND_NOT - these match documents in the following way:

- `OP_OR` - matches documents which match query A *or* B (or both)
- `OP_AND` - matches documents which match both query A *and* B
- `OP_AND_NOT` - matches documents which match query A but *not* B

Each operator produces a weight for each document it matches, which depends on the weight of one or both subqueries in the following way:

- `OP_OR` - matches documents with the sum of weights from A and B
- `OP_AND` - matches documents with the sum of weights from A and B
- `OP_AND_NOT` - matches documents with the weight from A only

Maybe

In addition to the basic logical operators, there is an additional logical operator `OP_AND_MAYBE` which matches any document which matches A (whether or not B matches). If only B matches, then `OP_AND_MAYBE` doesn’t match. For this operator, the weight is the sum of the matching subqueries, so:

1. Documents which match A and B match with the weight of A+B
2. Documents which match A only match with weight of A

This allows you to state that you require some terms (A) and that other terms (B) are useful but not required.

Filtering

A query can be filtered by another query. There are two ways to apply a filter to a query depending whether you want to include or exclude documents:

- `OP_FILTER` - matches documents which match both subqueries, but the weight is only taken from the left subquery
- `OP_AND_NOT` - matches documents which match the left subquery but don't match the right hand one (with weights coming from the left subquery)

Todo

Note on difference between OP_FILTER and OP_AND?

Value ranges

When using document values, there are three relevant operators:

- `OP_VALUE_LE` - matches documents where the given value is less than or equal a fixed value
- `OP_VALUE_GE` - matches documents where the given value is greater than or equal to a fixed value
- `OP_VALUE_RANGE` - matches documents where the given value is within the given fixed range (including both endpoints)

Note that when using these operators, they decide whether to include or exclude documents only and do not affect the weight of a document.

Near and Phrase

Two additional operators that are commonly used are *NEAR*, which finds terms within 10 words of each other in the current document, behaving like `OP_AND` with regard to weights, so that:

- Documents which match A within 10 words of B are matched, with weight of A+B

The phrase operator allows for searching for a specific phrase and returns only matches where all terms appear in the document, in the correct order, giving a weight of the sum of each term. For example:

- Documents which match A followed by B followed by C gives a weight of A+B+C

Additional operators

Xapian also provides additional operators which can be used to provide more flexibility than the operators above. For more details of these, see the Xapian API documentation.

Todo

MatchAll MatchNothing

2.3.2 Query Parser

To make searching databases simpler, Xapian provides a *QueryParser* class which converts a human readable query string into a Xapian Query object, for example:

```
apple AND a NEAR word OR "a phrase" NOT (too difficult) +eh
```

The above example shows how some of the basic modifiers are interpreted by the QueryParser; the operators it supports follow the operators described earlier, for example:

- `apple AND pear` matches documents where both terms are present
- `apple OR pear` matches documents where either term (or both) are present
- `apple NOT pear` matches documents where `apple` is present and `pear` is not

Term Generation

The QueryParser uses an internal process to convert the query string into terms. This is similar to the process used by the *TermGenerator*, which can be used at index time to convert a string into terms. It is often easiest to use QueryParser and *TermGenerator* on the same database.

Operators

As well as the basic logical operators, QueryParser supports the additional operators discussed earlier and introduces some new ones, for example:

```
apple NEAR dessert
president "united states"
"race condition" -horse
+recipe +apple pie cake dessert
```

The NEAR and phrase support behaves in the same way as described earlier; the new features are the + and - operators, which select documents based on the presence or absence of specified terms, for example:

```
"race condition" -horse
```

Matches all documents with the phrase `"race condition"` but not `horse`; and:

```
+recipe +apple pie cake desert
```

Which matches all documents which have the terms `recipe` and `apple`; then all documents with these terms are weighted according to the weight of the additional terms.

One thing to note is that the behaviour of the +/- operators vary depending on the default operator used and the above examples assume that `OP_OR` is used.

Bracketed Expressions

When queries contain both OR and AND operators, AND takes precedence. To change the precedence of parts of the query, brackets can be used. For example, with the query:

```
apple OR pear AND dessert
```

The query parser will interpret this query as:

```
apple OR (pear AND dessert)
```

So to change the precedence and make the dessert a requirement, you would write the query initially as:

```
(apple OR pear) AND dessert
```

Stop words

Xapian also supports a *stop word* list, which allows you to specify words which should be removed from a query before processing. This list can be overridden within user search, so stop words can still be searched for if desired, for example if a stop word list contained 'the' and a search was for:

```
+the +document
```

Then the search would find relevant documents which contained both 'the' and 'document'. Also, when searching for phrases, stop words do not apply, for example here we will retrieve documents with the exact phrase including 'the':

```
"the green space"
```

Searching with Prefixes

When a database is populated using prefixed terms (for example, title, author) it is possible to tell the QueryParser that these fields can be searched for using a human-readable prefix; for example:

```
author:"william shakespeare" title:juliet
```

Ranges

The QueryParser also supports range searches on document values, matching documents which have values within a given range. There are several types of range processors available, but the two discussed here are Date and Number, which require that values are serialised as they are indexed.

To use a range, additional programming is required to tell the QueryParser what format a range is specified in and which value is to be searched for matches within that range. This then gives rise to the ability to specify ranges as:

```
$10..50
5..10kg
01/01/1970..01/03/1970
size:3..7
```

When date ranges are configured (as a DateValueRangeProcessor), you can configure which format dates are to be interpreted as (i.e. month-day-year) or otherwise.

Wildcards

It is also possible to use wildcards to match any number of trailing characters within a term; for example:

> `wild*` matches `wild, wildcard, wildcat, wilderness`

This feature is disabled by default; to enable it, see 'Parser Flags' below. It also requires a database to be set on the QueryParser (so that it can find the list of terms to expand the wildcard to).

By default the wildcard will expand to as many terms as there are with the specified prefix. This can cause performance problems, so you can limit the number of terms a wildcard will expand to by calling `xapian.QueryParser.set_max_wildcard_expansion()`. If this limit would be exceeded then an exception will be thrown. The exception may be thrown by the QueryParser, or later when Enquire handles the query.

Searching for Partially Entered Queries

The QueryParser also supports performing a search with a query which has only been partially entered. This is intended for use with "incremental search" systems, which don't wait for the user to finish typing their search before displaying an initial set of results. For example, in such a system a user would enter a search, and the system would display a new set of results after each letter, or whenever the user pauses for a short period of time (or some other similar strategy).

The problem with this kind of search is that the last word in a partially entered query often has no semantic relation to the completed word. For example, a search for “dynamic cat” would return a quite different set of results to a search for “dynamic categorisation”. This results in the set of results displayed flicking rapidly as each new character is entered. A much smoother result can be obtained if the final word is treated as having an implicit terminating wildcard, so that it matches all words starting with the entered characters - thus, as each letter is entered, the set of results displayed narrows down to the desired subject.

A similar effect could be obtained simply by enabling the wildcard matching option, and appending a “*” character to each query string. However, this would be confused by searches which ended with punctuation or other characters.

This feature is disabled by default - pass `FLAG_PARTIAL` flag in the flags argument of `xapian.QueryParser.parse_query(query_string, flags)` to enable it, and tell the QueryParser which database to expand wildcards from using the `xapian.QueryParser.set_database(database)` method.

Default Operator

When the QueryParser receives a query, it joins together its component queries using a default operator which defaults to `OP_OR` but can be modified at run time.

Parser Flags

The operation of the QueryParser can be altered through the use of flags, combined with the bitwise OR operator; these flags include:

- `FLAG_BOOLEAN`: enables support for AND, OR, etc and bracketed expressions
- `FLAG_PHRASE`: enables support for phrase expressions
- `FLAG_LOVEHATE`: enables support for + and - operators
- `FLAG_BOOLEAN_ANY_CASE`: enables support for lower/mixed case boolean operators
- `FLAG_WILDCARD`: enables support for wildcards

You can find more information about the available flags in the API documentation.

By default, the QueryParser enables `FLAG_BOOLEAN`, `FLAG_PHRASE` and `FLAG_LOVEHATE`.

2.3.3 Ranked matches

When you run a Query using Xapian, what you get is a list of *ranked matches*.

Each match is a Xapian Document which satisfies the Query, with a *weight*, and the list is ordered by decreasing weight, the weight being an indicator of how good a match that Document is for the query that was run: a higher weight means a better match. The *rank* of each match is simply the position in the list of all matches, starting from 0. Some other search systems use the word “score” instead of weight.

The actual weight is calculated by a *weighting scheme*; Xapian comes with a few different ones or you can write your own, although often the default is fine. (It uses a scheme called BM25, which takes into account things like how common a matching term is in a matching document compared to in the entire database, and the lengths of different matching documents.)

Rather than having to run through the entire list of matches from the beginning, you actually ask for a sub-range of the entire list of matches, from an offset and extending for a given number of matches. Many search applications will provide the user with a way of "paging" through the matches, so the first page might be starting at 0 for 10 matches, the second page starting at 10 for 10 matches, and so on.

A page of matches in Xapian is called an MSet (for "match set").

Alternative sort orders

Sometimes, rather than getting results sorted by *weight*, it would be more useful to get them in some other order. For example, it might be desirable to get results in order of the values stored in a date field.

To do this, you first need to store the information used for the sort in a value slot, as described in the indexing documentation. You can then tell Xapian at search time to sort by the value in that slot. It is also possible to sort by the values in several slots (e.g., to sort items which have the same value in a particular slot by the value in a secondary slot).

Finally, it is possible to ask Xapian to return the documents in order of the Xapian document ID numbers.

2.3.4 Search-time Limitations

Document ID Range

When searching over multiple database at once, the document ids from each database are interleaved to produce the document ids in the combined database, so the 32-bit document id size can hit well before you reach four billion documents - for example, if you're searching one large database and 31 small ones, you'll reach the limit of document id size on the combined database when the largest database reaches document id 2^{27} (about 134 million).

Also, there may be gaps in the range of used document ids, either because you've deleted documents, or because you're explicitly setting them to match an external system.

Positional Queries

Currently `OP_PHRASE` and `OP_NEAR` don't really support non-term subqueries, though simple cases get rearranged (so for example, `A PHRASE (B OR C)` becomes `(A PHRASE B) OR (A PHRASE C)`).

Queries which use positional information (`OP_PHRASE` and `OP_NEAR`) can be significantly slower to process. The way these are implemented is to find documents which have all the necessary terms in, and then to check if the terms fulfil the positional requirements, so if a lot of documents contain the required terms but not in the right places, a lot more work is required than for just doing an AND query. This will be improved in a future release.

Collapsing

You can't perform more than one collapse operation during a search.

Concurrently Open Databases

If you try to search many databases concurrently, you may hit the per-process file-descriptor limit - each chert database uses between 3 and 7 fds depending which tables are present, and a process can only open a certain number (on Linux, the default is usually 1024, so that limits you to a few hundred concurrently open databases). You can raise the per-process limit on some Unix-like platforms, though you may need to be root to do so.

Todo

add or link to some details of how to do this

You can address this issue (and spread the search load) by using the remote backend to search databases on a cluster of machines - the remote backend only uses one fd per database on the client machine.

Todo

(as howto?) pagination (ask for pagesize*#pages + 1)

Todo

get_matches_estimated() and check_at_least

2.4 Introduction to IR

Information retrieval system retrieves relevent documents based on information need from a document collection. Generally document is a piece of text, described as collection of terms. Information need can often be described as collection of few terms. If you are searching for “golf”, Aim of an IR system would be to return all the document containing golf.

2.4.1 Indexing: Offline processing

In order to retrieve all the document with query terms, there are two methods. * Linear scan of all the documents to see if term is in the document. (Inefficient) * Offline process all the document in a special format to get list of all document with term for all terms. (Used in IR)

Linear scan is highly inefficient due to scanning of all the docuemnt in collection to serve the information need. It would take huge time to return document.

In modern information retrieval systems, documents are processed offline to arrange it in special format called index to avoid linear scan. Special format is index.

Indexing

Indexing is a process of converting the text documents into a special format(index) for efficient retrieval of document. It’s a list of documents(referenced by document id) containing particular term, it is know as posting list.

Documents:

D[0] = {This is a book}
D[1] = {Books are worth reading}
D[2] = {This book is worth buying}

Index:

This - {0,2}
book - {0,1,2}
is - {0,2}

a - {0}
are - {1}
worth - {1,2}
reading - {1}
buying - {2}

Stemming

Some words are used in there inflected form in the document, for example connect can be used as connecting, connected , connection . Since all the terms(connecting, connected) are similar, but as per design of index, we will not get documents containing connected with query connect using posting list of connect. Therefore before we store the term in index, term is stemmed to reduce the inflected word to their word stem form or root form. For example buying will be stored as buy in the index.

2.4.2 Retrieval: Online processing

Given a information need, generally given as search query (terms), we try to retrieve relevant documents from the collection using the index formed in the offline processing stage. There are two major paradigm to retrieve document. * Boolean Retrieval

Boolean retrieval, retrieve the documents by doing union, intersection or difference on the posting list of terms in the query.

Posting list of query term's:

book -> {1,4,10}
work -> {1,2,4,9}

Boolean Retrieval

book AND work -> {1,4}
book OR work -> {1,2,4,9,10}
book AND_NOT work -> {10}
work AND_NOT book -> {2,9}

Probabilistic Retrieval

Probabilistic retrieval model is based on the Probability Ranking Principle, which states that an information retrieval system is supposed to rank the documents based on their probability of relevance to the query, given all the evidence available [Belkin and Croft 1992]. The principle takes into account that there is uncertainty in the representation of the information need and the documents. There can be a variety of sources of evidence that are used by the probabilistic retrieval methods, and the most common one is the statistical distribution of the terms in both the relevant and non-relevant documents.

Posting list stores the frequency of occurance of word in document. Posting list for work:

work -> {1:3,2:1,4:2,9:10}

document 1 contains work 3 times, where as document id 2 constains 1 time.

In probabilistic model we give score to the documents based on frequency of the query words in document.

Query Expansion:

Query expansion is a process where query provided by the user is expanded with extra terms to improve the search results. Aim of query expansion is to generate alternative or expanded queries for the user. There are two broad approaches to do this: * Relevance feedback - Users give additional input on documents (by marking documents in the results set as relevant or not), and this input is used to reweight the terms or add new term in the query for documents. * Pseudo Relevance feeback - Top ranking document of the search are considered to be relevant and used to reweight the terms or add new term.

References:

- "Information Retrieval" by C. J. van Rijsbergen is well worth reading. It's out of print, but is available for free from the author's website (in HTML or PDF).
- "Readings in Information Retrieval" (published by Morgan Kaufmann, edited by Karen Sparck Jones and Peter Willett) is a collection of published papers covering many aspects of the subject.
- "Managing Gigabytes" (also published by Morgan Kaufmann, written by Ian H. Witten, Alistair Moffat and Timothy C. Bell) describes information retrieval and compression techniques.
- "Modern Information Retrieval" (published by Addison Wesley, written by Ricardo Baeza-Yates and Berthier Ribeiro-Neto) gives a good overview of the field. It was published more recently than the books above, and so covers some more recent developments.
- Introduction to Information Retrieval (published by Cambridge University Press, written by Christopher D. Manning, Prabhakar Raghavan and Hinrich)

CHAPTER 3

A practical example

3.1 Indexing

3.1.1 Building a museum catalogue

We're going to build a simple search system based on museum catalogue data released under the Creative Commons Attribution-NonCommercial-ShareAlike license by the Science Museum in London, UK.

Preparing to run the examples

You should download both the two sample datasets and example code as described in the *overview*, and also check that you've installed Xapian as detailed there.

3.1.2 What data is there?

Each row in the CSV file is an object from the catalogue, and has a number of fields. There are:

id_NUMBER: a unique identifier

ITEM_NAME: a simple name, often from an established thesaurus

TITLE: a short caption

MAKER: the name of who made the object

DATE_MADE: when the object was made, which may be a range, approximate date or unknown

PLACE_MADE: where the object was made

MATERIALS: what the object is made of

MEASUREMENTS: the dimensions of the object

DESCRIPTION a description of the object

COLLECTION: the collection the object came from (eg: Science Museum - Space Technology)

There are obviously a number of different types of data here: free text, identifiers, dates, places (which could be geocoded to geo coordinates), and dimensions. Additionally, COLLECTION and MAKER both come from a list of possible values.

3.1.3 What do people want to search for?

We can think of a large number of different things that people might want to find from our catalogue. For instance, they may want to find objects that were created in Nantes, or after 1812, or by Hurd-Brown Ltd. They may want to find everything made of brass, or not containing wood, or more than a metre in length. They may care only about objects in the National Railway Museum, or in their Railway Heraldry collection, or everything not in the Railway Heraldry collection. And of course they may want to look up objects that have certain words or phrases in the title or description - “free text search”, one of the most common uses of search today.

In order to support all of this we’ll need to use many of the features of Xapian, but to get started we’ll just look at one: free text search of the title and description.

In later sections of this guide (if we ever write them), and in guides to do specific things (if we ever convert them) we’ll use the same data and build on the system we create here.

3.1.4 The index plan

In order to index the CSV, we want to take two fields from each row, title and description, and turn them into suitable terms. For straightforward textual search we don’t need document values.

Because we’re dealing with free text, and because we know the whole dataset is in English, we can use stemming so that for instance searching for “sundial” and “sundials” will both match the same documents. This way people don’t need to worry too much about exactly which words to use in their query.

Finally, we want a way of separating the two fields. In Xapian this is done using *term prefixes*, basically by putting short strings at the beginning of terms to indicate which field the term indexes. As well as prefixed terms, we also want to generate unprefixed terms, so that as well as searching within fields you can also search for text in any field.

There are some conventional prefixes used, which is helpful if you ever need to interoperate with omega (a web-based search engine) or other compatible systems. From this, we’ll use ‘S’ to prefix title (it stands for ‘subject’), and for description we’ll use ‘XD’. A full list of conventional prefixes is given at the top of the omega documentation on termprefixes.

When you’re indexing multiple fields like this, the term positions used for each field when indexed unprefixed need to be kept apart. Say you have a title of “The Saints”, and description “Don’t like rabbits? Keep reading.” If you index those fields without a gap, the phrase search “Saints don’t like rabbits” will match, where it really shouldn’t. Usually a gap of 100 between each field is enough.

To write to a database, we use the WritableDatabase class, which allows us to create, update or overwrite a database.

To create terms, we use Xapian’s TermGenerator, a built-in class to make turning free text into terms easier. It will split into words, apply stemming, and then add term prefixes as needed. It can also take care of term positions, including the gap between different fields.

3.1.5 Let’s write some code

Here’s the significant part of some example code to implement this index plan.

```python
def index(datapath, dbpath):
    # Create or open the database we're going to be writing to.
    db = xapian.WritableDatabase(dbpath, xapian.DB_CREATE_OR_OPEN)

    # Set up a TermGenerator that we'll use in indexing.
    termgenerator = xapian.TermGenerator()
    termgenerator.set_stemmer(xapian.Stem("en"))

    for fields in parse_csv_file(datapath):
        # 'fields' is a dictionary mapping from field name to value.
        # Pick out the fields we're going to index.
        description = fields.get('DESCRIPTION', u'')
        title = fields.get('TITLE', u'')
        identifier = fields.get('id_NUMBER', u'')

        # We make a document and tell the term generator to use this.
        doc = xapian.Document()
        termgenerator.set_document(doc)

        # Index each field with a suitable prefix.
        termgenerator.index_text(title, 1, 'S')
        termgenerator.index_text(description, 1, 'XD')

        # Index fields without prefixes for general search.
        termgenerator.index_text(title)
        termgenerator.increase_termpos()
        termgenerator.index_text(description)

        # Store all the fields for display purposes.
        doc.set_data(json.dumps(fields, encoding='utf8'))

        # We use the identifier to ensure each object ends up in the
        # database only once no matter how many times we run the
        # indexer.
        idterm = u"Q" + identifier
        doc.add_boolean_term(idterm)
        db.replace_document(idterm, doc)
```

A full copy of this code is available in code/python/index1.py.

You can run this code to index a sample data file (held in data/100-objects-v1.csv) to a database at path `db` as follows:

```
$ python code/python/index1.py data/100-objects-v1.csv db
```

3.1.6 Verifying the index using delve

Xapian comes with a handy utility called *delve* which can be used to inspect a database, so let's look at the one you just built. If you just run `delve db`, you'll get an overview: how many documents, average term length, and some other statistics:

```
$ delve db
UUID = 1820ef0a-055b-4946-ae73-67aa4ef5c226
number of documents = 100
average document length = 100.58
document length lower bound = 33
document length upper bound = 251
highest document id ever used = 100
has positional information = true
```

You can also look at an individual document, using Xapian's docid (`-d` means output document data as well):

```
$ delve -r 1 -d db       # output has been reformatted
Data for record #1:
{
 "MEASUREMENTS": "",
 "DESCRIPTION": "Ansonia Sunwatch (pocket compas dial)",
 "PLACE_MADE": "New York county, New York state, United States",
 "id_NUMBER": "1974-100",
 "WHOLE_PART": "WHOLE",
 "TITLE": "Ansonia Sunwatch (pocket compas dial)",
 "DATE_MADE": "1922-1939",
 "COLLECTION": "SCM - Time Measurement",
 "ITEM_NAME": "Pocket horizontal sundial",
 "MATERIALS": "",
 "MAKER": "Ansonia Clock Co."
}
Term List for record #1: Q1974-100 Sansonia Scompas Sdial Spocket
Ssunwatch XDansonia XDcompass XDdial XDpocket XDsunwatch ZSansonia
ZScompas ZSdial ZSpocket ZSsunwatch ZXDansonia ZXDcompas ZXDdial
ZXDpocket ZXDsunwatch Zansonia Zcompass Zdial Zpocket Zsunwatch
ansonia compass dial pocket sunwatch
```

You can also go the other way, starting with a term and finding both statistics and which documents it indexes:

```
$ delve -t Stime db
Posting List for term 'Stime' (termfreq 4, collfreq 4, wdf_max 4):
41 56 58 65
```

This means you can look documents up by identifier:

```
$ delve -t Q1974-100 db
Posting List for term 'Q1974-100' (termfreq 1, collfreq 1, wdf_max 1):
1
```

`delve` is frequently useful if you aren't getting the behaviour you expect from a search system, to check that the database contains the documents and terms you expect.

3.1.7 Updating the database

If you look back at the verifying step of the database, you may notice that the first item we have indexed has the word 'compass' spelled incorrectly, which means that we will need to either update just that document, or to re-index the entire database.

Reindexing the database can be done immediately using the index1.py script we used for the initial indexing; this is because we are using an external ID for each document we add to the database, taken from the *id_NUMBER* field from the original data set. We then pass this to the `xapian.Database.replace_document` method, which updates if there's already a document under that external ID, or adds a document to the database otherwise.

In fact, because of this, index1.py can update just part of the database. Just give it a file with only the rows that correspond to documents that need updating. Everything else in the database will be left untouched.

Deleting documents

It is also possible to delete documents from the index using the `xapian.Database.delete_document` method on a `xapian.WritableDatabase` object. This can be done either by Xapian docid or using unique ID terms, as with `xapian.Database.replace_document`.

```python
def delete(dbpath, identifiers):
    # Open the database we're going to be deleting from.
    db = xapian.WritableDatabase(dbpath, xapian.DB_CREATE_OR_OPEN)

    for identifier in identifiers:
        idterm = u'Q' + identifier
        db.delete_document(idterm)
```

A copy of this code is available in code/python/delete1.py.

Then we just run our deletion tool, giving it identifiers taken from the *id_NUMBER* field in the data set:

```
$ python code/python/delete1.py db 1953-448 1985-438
```

After that, we expect to see two fewer documents in our database using delve:

```
$ delve db
UUID = 1820ef0a-055b-4946-ae73-67aa4ef5c226
number of documents = 98
average document length = 100.041
document length lower bound = 33
document length upper bound = 251
highest document id ever used = 100
has positional information = true
```

3.2 Searching

3.2.1 Building the search

Now we have our database populated with some values, it is time for the code to search the database and display some results.

We want to take some text from the user, and search for it in the database; to do that we need to convert it into a Xapian Query, which you will recall is a tree made up of terms (which in this case will be the stemmed forms of words in the text from the user), and operations such as AND, OR and so forth.

There are many ways to go from the user's text to a Query, but the most simple of these is to use the QueryParser. We then pass the Query to an Enquire object, which also needs setting up with a database, and is where you'd set various other options that affect how the query is run (such as sorting, for instance) which we won't address here.

```python
def search(dbpath, querystring, offset=0, pagesize=10):
    # offset - defines starting point within result set
    # pagesize - defines number of records to retrieve

    # Open the database we're going to search.
    db = xapian.Database(dbpath)

    # Set up a QueryParser with a stemmer and suitable prefixes
    queryparser = xapian.QueryParser()
    queryparser.set_stemmer(xapian.Stem("en"))
    queryparser.set_stemming_strategy(queryparser.STEM_SOME)
    queryparser.add_prefix("title", "S")
    queryparser.add_prefix("description", "XD")

    # And parse the query
    query = queryparser.parse_query(querystring)

    # Use an Enquire object on the database to run the query
    enquire = xapian.Enquire(db)
    enquire.set_query(query)

    # And print out something about each match
    matches = []
    for match in enquire.get_mset(offset, pagesize):
        fields = json.loads(match.document.get_data())
        print u"%(rank)i: #%(docid)3.3i %(title)s" % {
            'rank': match.rank + 1,
            'docid': match.docid,
            'title': fields.get('TITLE', u''),
            }
        matches.append(match.docid)

    # Finally, make sure we log the query and displayed results
    support.log_matches(querystring, offset, pagesize, matches)
```

A full copy of this code is available in code/python/search1.py.

3.2.2 Running a Search

To search the database we're built, you just run our simple search engine:

```
$ python code/python/search1.py db watch
1: #004 Watch with Chinese duplex escapement
2: #018 Solar/Sidereal verge watch with epicyclic maintaining power
3: #013 Watch timer by P
4: #033 A device by Favag of Neuchatel which enables a stop watch to
5: #015 Ingersoll "Dan Dare" automaton pocket watch with pin-pallet
6: #036 Universal 'Tri-Compax' chronographic wrist watch
7: #046 Model by Dent of mechanism for setting hands and winding up
INFO:xapian.search:'watch'[0:10] = 4 18 13 33 15 36 46
```

These results show that 7 documents match our search for the term 'watch', providing the document IDs (e.g. #004) and title for each. If you want to search for multiple words, just chain them together on the command line:

```
$ python code/python/search1.py db Dent watch
1: #046 Model by Dent of mechanism for setting hands and winding up
2: #004 Watch with Chinese duplex escapement
3: #018 Solar/Sidereal verge watch with epicyclic maintaining power
4: #013 Watch timer by P
5: #094 Model of a Lever Escapement , 1850-1883
6: #093 Model of Graham's Cylinder Escapement, 1850-1883
7: #033 A device by Favag of Neuchatel which enables a stop watch to
8: #015 Ingersoll "Dan Dare" automaton pocket watch with pin-pallet
9: #086 Model representing Earnshaw's detent chronometer escapement, 1950-1883
10: #036 Universal 'Tri-Compax' chronographic wrist watch
INFO:xapian.search:'Dent watch'[0:10] = 46 4 18 13 94 93 33 15 86 36
```

You'll notice that all of the results from the first time come back the second time also, with additional ones (the match 'Dent' but not 'watch'), because by default QueryParser will use the OR operator to combine the different search terms. Also, because #046 contains both 'Dent' and 'watch', it now ranks highest of all the matches.

3.2.3 Searching separate fields

When we built our index, we used prefixes to separate the terms generated from the title and description fields. This allows us to perform searches which are restricted to the text in just one of those fields, by searching only terms with the desired prefix.

When using the Query Parser, it is possible to restrict your search to certain prefixed terms (e.g. title, or description). These can be searched for either by using a search prefix (which can correlate to an indexing prefix) or as a general text document.

To set up a search prefix, the QueryParser needs to be told which prefixes in the search query relate to those in the index:

Todo

This example should really be pulled directly from the code. There seems to be a bug with line number limiting somewhere in the literal include directive.

```
queryparser.add_prefix("title", "S")
queryparser.add_prefix("description", "XD")
```

This allows us to perform a search based on either field, for example:

```
$ python code/python/search1.py db title:sunwatch
1: #001 Ansonia Sunwatch (pocket compas dial)
INFO:xapian.search:'title:sunwatch'[0:10] = 1
```

We can also combine prefixes with the logical operators to perform more complex queries (note that we need to escape quotes or else the shell will eat them):

```
$ python code/python/search1.py db description:\"leather case\" AND title:sundial
1: #055 Silver altitude sundial in leather case
INFO:xapian.search:'description:"leather case" AND title:sundial'[0:10] = 55
```

3.2.4 Database Modified

If you're updating the same database you search from (rather than updating a separate database and then "flipping" between them, using a stub database), you may run into *DatabaseModifiedError*, and exception that can be raised while reading from the database. What this means is that the database has changed too much since you opened it for Xapian to be able to continue supplying you with information. The solution here is to re-open the database with its `reopen` method.

CHAPTER 4

How To...

4.1 How to filter search results

Todo

point out that lowercasing by TermGenerator or similar will prevent unexpected matching of prefixes terms by "real" words in the source data

In our earlier discussion of building an index of museum catalog data, we showed how to index text from the title and description fields with separate prefixes, allowing searches to be performed across just one of those fields. This is a simple type of fielded search, but often some fields in a document won't contain unconstrained text; for example, they may contain only a few specific values or identifiers. We often wish to use such fields to restrict the results to only those matching a particular value, rather than treating them as unstructured "free text".

In the museum catalog, the `MATERIALS` field is an example of a field which contains text from a restricted vocabulary. This text can be thought of as an identifier, rather than text which needs to be parsed. In fact, for many records this field contains several identifiers for materials in the object, separated by semicolons.

4.1.1 Indexing

When indexing such fields, we don't want to perform stemming, though we may well want to convert the identifiers to lowercase if case is not significant. We also don't expect the number of times a term from these fields occurs in a document to be significant (we only expect it to occur 0 or 1 times), so we don't need to store "within document frequency" information. A field like this, which we're using to restrict the results returned from a search rather than as part of the weighted search, is referred to as a *boolean term*.

We can therefore just add the identifiers to the `xapian.Document` directly, after splitting on semicolons, using the `add_boolean_term()` method.

```
        # Index the MATERIALS field, splitting on semicolons.
        for material in fields.get('MATERIALS', u'').split(';'):
            material = material.strip().lower()
            if material != '':
                doc.add_boolean_term('XM' + material)
```

A full copy of the indexer with this updated code is available in code/python/index_filters.py.

We run this like so:

```
$ python code/python/index_filters.py data/100-objects-v1.csv db
```

If we check the resulting index with delve, we will see that documents for which there was a value in the `MATERIALS` field now contain terms with the `XM` prefix (output snipped to show the relevant lines):

```
$ delve -r 3 -1 db
Term List for record #3:
...
XDwooden
XMglass
XMmounted
XMsand
XMtimer
XMwood
ZSabbot
...
```

4.1.2 Searching

Suppose that the interface we want to provide allows users to type a free text search into one form input, but also has a set of checkboxes for different possible materials. We want to return documents which match the text search entered, but only if they also contain one of the materials for which the checkbox is selected.

To build a query which performs this task, we can take the Query object returned by the query parser, and combine it with a manually built Query representing the checkboxes which are selected, using the `OP_FILTER` operator. If multiple checkboxes are selected, we need to combine the Query objects for each checkbox with an `OP_OR` operator.

An arbitrarily complex Query tree can be built using queries returned from the QueryParser and manually constructed Query objects, which allows very flexible filtering of the results from parsed queries.

```
    # Set up a QueryParser with a stemmer and suitable prefixes
    queryparser = xapian.QueryParser()
    queryparser.set_stemmer(xapian.Stem("en"))
    queryparser.set_stemming_strategy(queryparser.STEM_SOME)
    queryparser.add_prefix("title", "S")
    queryparser.add_prefix("description", "XD")

    # And parse the query
    query = queryparser.parse_query(querystring)

    if len(materials) > 0:
        # Filter the results to ones which contain at least one of the
        # materials.

        # Build a query for each material value
        material_queries = [
            xapian.Query('XM' + material.lower())
            for material in materials
        ]

        # Combine these queries with an OR operator
        material_query = xapian.Query(xapian.Query.OP_OR, material_queries)

        # Use the material query to filter the main query
        query = xapian.Query(xapian.Query.OP_FILTER, query, material_query)
```

A full copy of the this updated search code is available in search_filters.py. With this, we could perform a search for documents matching “clock”, and filter the results to return only those with a value of `"steel (metal)"` as one of the semicolon separated values in the materials field:

```
$ python code/python/search_filters.py db clock 'steel (metal)'
1: #012 Assembled and unassembled EXA electric clock kit
2: #098 'Pond' electric clock movement (no dial)
3: #052 Reconstruction of Dondi's Astronomical Clock, 1974
4: #059 Electrically operated clock controller
5: #024 Regulator Clock with Gravity Escapement
6: #097 Bain's subsidiary electric clock
7: #009 Copy  of a Dwerrihouse skeleton clock with coup-perdu escape
8: #091 Pendulum clock designed by Galileo in 1642 and made by his son in 1649, model.
INFO:xapian.search:'clock'[0:10] = 12 98 52 59 24 97 9 91
```

4.1.3 Using the query parser

The previous section shows how to write code to filter the results of a query programmatically. This can be very flexible, but sometimes you want users to be able to specify filters themselves, within the text query that they enter.

You can do this using the `QueryParser.add_boolean_prefix()` method. This lets you tell the query parser about a field to use for filtering, and the prefix that terms have been stored in for that term. For our materials search, we just need to a add a single line to the search code:

```
# Set up a QueryParser with a stemmer and suitable prefixes
queryparser = xapian.QueryParser()
queryparser.set_stemmer(xapian.Stem("en"))
queryparser.set_stemming_strategy(queryparser.STEM_SOME)
queryparser.add_prefix("title", "S")
queryparser.add_prefix("description", "XD")
queryparser.add_boolean_prefix("material", "XM")

# And parse the query
query = queryparser.parse_query(querystring)
```

Users can then perform a filtered search by preceding a word or phrase with "material:", similar to the syntax supported for this sort of thing by many web search engines:

```
$ python code/python/search_filters2.py db 'clock material:"steel (metal)"'
1: #012 Assembled and unassembled EXA electric clock kit
2: #098 'Pond' electric clock movement (no dial)
3: #052 Reconstruction of Dondi's Astronomical Clock, 1974
4: #059 Electrically operated clock controller
5: #024 Regulator Clock with Gravity Escapement
6: #097 Bain's subsidiary electric clock
7: #009 Copy  of a Dwerrihouse skeleton clock with coup-perdu escape
8: #091 Pendulum clock designed by Galileo in 1642 and made by his son in 1649, model.
INFO:xapian.search:'clock material:"steel (metal)"'[0:10] = 12 98 52 59 24 97 9 91
```

4.1.4 What to supply to the query parser

Often, developers seem to be tempted to apply filters to a query by modifying the query supplied by a user (eg, by adding things like `material:steel` to the end of it). This is generally a bad idea, because the query parser contains various heuristics to handle input from users; it is very hard to modify the input to a query parser to reliably add a filter to the parsed query.

The rule is that the query parser should be supplied with direct user input, and if you want to apply extra filters to the query, you should apply them to the output of the query parser.

In later sections, we'll see how to tell the query parser about other types of searches that users might enter (for example, range searches). In each of these cases, it is also possible to perform such searches and restrictions without using the query parser; the query parser just allows the user of the search system to perform such restrictions in the query string.

4.2 Range queries

Todo

check valueranges.rst to see if anything else needs moving across

4.2.1 I'm only interested in the 1980s

In the museums dataset we used in our earlier examples, there is a field *DATE_MADE* that tells us when the object in question was made, so one of the natural things people might want to do is to only search for objects made in a particular time period. Suppose we want to extend our original system to allow that, we're going to have to do a number of things.

1. Parse the field from the data set to turn it into something consistent; at the moment it includes years, year ranges ("1671-1700"), approximate years ("c. 1936") and commentary ("patented 1885", or "1642-1649 (original); 1883 (model)"). Additionally, some records have no information about when the object was made.
2. Store that information in the Xapian database.
3. Provide a way during search of specifying a date range to constrain to.

If we look through the other fields in the data set, there are more that could be useful for range queries: we could extract the longest dimension from *MEASUREMENTS* to enable people to restrict to only very large or very small objects, for instance.

We'll see how to perform range searches across both those dimensions, and then we'll look at how to cope with full dates rather than just years.

4.2.2 How Xapian supports range queries

If you think back to when we introduced the query concepts behind Xapian, you'll remember that one group of query operators is responsible for handling *value ranges*: `OP_VALUE_LE`, `OP_VALUE_GE` and `OP_VALUE_RANGE`. So we'll be tackling range queries by using document values, and constructing queries using these operators to restrict matches suitably.

Since we want to expose this functionality generally to users, we want them to be able to type in a query that will include one or more range restrictions; the QueryParser contains support for doing this, using *value range processors*, subclasses of `xapian.ValueRangeProcessor`. Xapian comes with some standard ones itself, or you can write your own.

Since document values are stored as strings in Xapian, and the operators provided perform string comparisons, we need a way of converting numbers to strings to store them. For this, Xapian provides a pair of utility functions: *sortable_serialise* and *sortable_unserialise*, which convert between floating point numbers (strictly, each works with a *double*) and a string that will sort in the same way and so can be compared easily.

4.2.3 Creating the document values

We need a new version of our indexer. This one is index_ranges.py, and creates document values from both *MEASUREMENTS* and *DATE_MADE*. We'll put the largest dimension in value slot 0 (fortunately the data is stored in millimetres and kilograms, so we can cheat a little and assume that dimensions will always be larger than weights), and a year taken from *DATE_MADE* into value slot 1 (we choose the first year we can parse, since it can contain such a variety of date formats).

```
        # parse the two values we need
        measurements = fields.get('MEASUREMENTS', u'')
        if measurements != u'':
            numbers = numbers_from_string(measurements)
            if len(numbers) > 0:
                doc.add_value(0, xapian.sortable_serialise(max(numbers)))

        date_made = fields.get('DATE_MADE', u'')
        years = numbers_from_string(date_made)
        if len(years) > 0:
            doc.add_value(1, xapian.sortable_serialise(years[0]))
```

We run this like so:

```
$ python code/python/index_ranges.py data/100-objects-v1.csv db
```

We can check this has created document values using *delve*:

```
$ delve -V0 db | cat -v
Value 0 for each document: 5:M-@M-@ 8:M-HV 9:M-EM-p 10:M-MF 11:M-AM-0 12:M-AP 15:M-8^P 19:M-Dt 20:M-(
```

All the odd characters are because *delve* doesn't know to run *sortable_unserialise* to turn the strings back into numbers.

4.2.4 Searching with ranges

All we need to do once we've got the document values in place is to tell the QueryParser about them. The simplest value range processor is `xapian.StringValueRangeProcessor`, but here we need two `xapian.NumberValueRangeProcessor` instances.

To distinguish between the two different ranges, we'll require that dimensions must be specified with the suffix 'mm', but years are just numbers. For this to work, we have to tell QueryParser about the value range with a suffix first:

```
    queryparser.add_valuerangeprocessor(
        xapian.NumberValueRangeProcessor(0, 'mm', False)
    )
    queryparser.add_valuerangeprocessor(
        xapian.NumberValueRangeProcessor(1, '')
    )
```

The first call has a final parameter of *False* to say that 'mm' is a suffix (the default is for it to be a prefix). When using the empty string, as in the second call, it doesn't matter whether you say it's a suffix or prefix, so it's convenient to skip that parameter.

This is implemented in search_ranges.py, which also modifies the output to show the measurements and date made fields as well as the title.

We can now restrict across dimensions using queries like '..50mm' (everything at most 50mm in its longest dimension), and across years using '1980..1989':

```
$ python code/python/search_ranges.py db ..50mm
1: #031 (1588) overall diameter: 50 mm
        Portable universal equinoctial sundial, in brass, signed "A
2: #073 (1701-1721) overall: 15 mm x 44.45 mm, weight: 0.055kg
        Universal pocket sundial
3: #074 (1596) overall: 13 mm x 44.45 mm x 44.45 mm, weight: 0.095kg
```

```
        Sundial, made as a locket, gilt metal, part silver
INFO:xapian.search:'..50mm'[0:10] = 31 73 74

$ python code/python/search_ranges.py db 1980..1989
1: #050 (1984) overall: 105 mm x 75 mm x 57 mm,
        Quartz Analogue "no battery" wristwatch by Pulsar Quartz (CA
2: #051 (1984) overall: 85 mm x 65 mm x 38 mm,
        Analogue quartz clock with voice controlled alarm by Braun,
INFO:xapian.search:'1980..1989'[0:10] = 50 51
```

You can of course combine this with 'normal' search terms, such as all clocks made from 1960 onwards:

```
$ python code/python/search_ranges.py db clock 1960..
1: #052 (1974) clock: 1185 x 780 mm, 122 kg; rewind unit: 460 x 640 x 350 mm
        Reconstruction of Dondi's Astronomical Clock, 1974
2: #051 (1984) overall: 85 mm x 65 mm x 38 mm,
        Analogue quartz clock with voice controlled alarm by Braun,
3: #009 (1973) overall: 380 mm x 300 mm x 192 mm, weight: 6.45kg
        Copy  of a Dwerrihouse skeleton clock with coup-perdu escape
INFO:xapian.search:'clock 1960..'[0:10] = 52 51 9
```

and even combining both ranges at once, such as all large objects from the 19th century:

```
$ python code/python/search_ranges.py db 1000..mm 1800..1899
1: #024 (1845-1855) overall: 1850 mm x 350 mm x 250 mm
        Regulator Clock with Gravity Escapement
INFO:xapian.search:'1000..mm 1800..1899'[0:10] = 24
```

Note the slightly awkward syntax *1000..mm*. The suffix must always go on the end of the entire range; it may also go on the beginning (so you can do *1000mm..mm*). Similarly, you can have *100mm..200mm* or *100..200mm* but not *100mm..200*. These rules are reversed for prefixes.

If you get the rules wrong, the QueryParser will raise a *QueryParserError*, which in production code you could catch and either signal to the user or perhaps try the query again without the *ValueRangeProcessor* that tripped up:

```
$ python code/python/search_ranges.py db 1000mm..
Traceback (most recent call last):
  File "code/python/search_ranges.py", line 54, in <module>
    search(dbpath = sys.argv[1], querystring = " ".join(sys.argv[2:]))
  File "code/python/search_ranges.py", line 31, in search
    query = queryparser.parse_query(querystring)
xapian.QueryParserError: Unknown range operation
```

4.2.5 Handling dates

To restrict to a date range, we need to decide how to both store the date in a document value, and how we want users to input the date range in their query. `xapian.DateValueRangeProcessor`, which is part of Xapian, works by storing the date as a string in the form 'YYYYMMDD', and can take dates in either US style (month/day/year) or European style (day/month/year).

To show how this works, we're going to need to use a different dataset, because the museums data only gives years the objects were made in; we've built one using data on the fifty US states, taken from Wikipedia infoboxes on 5th November 2011 and then tidied up a small amount. The CSV file is states.csv, and the code that did most of the work is from_wikipedia.py, using a list of Wikipedia page titles in us_states_on_wikipedia. The CSV is licensed as Creative Commons Attribution-Share Alike 3.0, as per Wikipedia.

We need a new indexer for this as well, which is index_ranges2.py. It stores two numbers using *sortable_serialise*: year of admission in value slot 1 and population in slot 3. It also stores the date of admission as 'YYYYMMDD' in slot 2. Here's the code which does this:

```
        # index each field with a suitable prefix
        termgenerator.index_text(name, 1, 'S')
        termgenerator.index_text(description, 1, 'XD')
        termgenerator.index_text(motto, 1, 'XM')

        # Index fields without prefixes for general search.
        termgenerator.index_text(name)
        termgenerator.increase_termpos()
        termgenerator.index_text(description)
        termgenerator.increase_termpos()
        termgenerator.index_text(motto)

        # Add document values.
        if admitted is not None:
            doc.add_value(1, xapian.sortable_serialise(int(admitted[:4])))
            doc.add_value(2, admitted) # YYYYMMDD
        if population is not None:
            doc.add_value(3, xapian.sortable_serialise(population))
```

We'll look at just the date ones for now, and come back to the others in a minute.

We use the indexer in the same way as previous ones:

```
$ python code/python/index_ranges2.py data/states.csv statesdb
```

With this done, we can change the set of value range processors we give to the QueryParser.

```
    queryparser.add_valuerangeprocessor(
        xapian.DateValueRangeProcessor(2, True, 1860)
    )
    queryparser.add_valuerangeprocessor(
        xapian.NumberValueRangeProcessor(1, '')
    )
```

The `xapian.DateValueRangeProcessor` is working on value slot 2, with an "epoch" of 1860 (so two digit years will be considered as starting at 1860 and going forward as far 1959). The second parameter is whether it should prefer US style dates or not; since we're looking at US states, we've gone for US dates. The `xapian.NumberValueRangeProcessor` is as we saw before.

This enables us to search for any state that talks about the Spanish in its description:

```
$ python code/python/search_ranges2.py statesdb spanish
1: #004 State of Montana November 8, 1889
        Population 989,415
2: #019 State of Texas December 29, 1845
        Population 25,145,561
INFO:xapian.search:'spanish'[0:10] = 4 19
```

or for all states admitted in the 19th century:

```
$ python code/python/search_ranges2.py statesdb 1800..1899
1: #001 State of Washington November 11, 1889
        Population 6,744,496
2: #002 State of Arkansas June 15, 1836
        Population 2,915,918
3: #003 State of Oregon February 14, 1859
        Population 3,831,074
4: #004 State of Montana November 8, 1889
```

```
        Population 989,415
5: #005 Idaho July 3, 1890
        Population 1,567,582
6: #006 State of Nevada October 31, 1864
        Population 2,700,551
7: #007 State of California September 9, 1850
        Population 37,253,956
8: #009 State of Utah January 4, 1896
        Population 2,763,885
9: #010 State of Wyoming July 10, 1890
        Population 563,626
10: #011 State of Colorado August 1, 1876
        Population 5,029,196
INFO:xapian.search:'1800..1899'[0:10] = 1 2 3 4 5 6 7 9 10 11
```

That uses the `xapian.NumberValueRangeProcessor` on value slot 1, as in our previous example. Let's be more specific and ask for only those between November 8th 1889, when Montana became part of the Union, and July 10th 1890, when Wyoming joined:

```
$ python code/python/search_ranges2.py statesdb 11/08/1889..07/10/1890
1: #001 State of Washington November 11, 1889
        Population 6,744,496
2: #004 State of Montana November 8, 1889
        Population 989,415
3: #005 Idaho July 3, 1890
        Population 1,567,582
4: #010 State of Wyoming July 10, 1890
        Population 563,626
INFO:xapian.search:'11/08/1889..07/10/1890'[0:10] = 1 4 5 10
```

That uses the `xapian.DateValueRangeProcessor` on value slot 2; it can't cope with year ranges, which is why we indexed to both slots 1 and 2.

4.2.6 Writing your own ValueRangeProcessor

We haven't yet done anything with population. What we want is something that behaves like `xapian.NumberValueRangeProcessor`, but knows what reason possible values are. If we insert it *before* the `xapian.NumberValueRangeProcessor` on slot 1 (year), it can pick up anything that should be treated as a population, and let everything else be treated as a year range.

To do this, we need to know how a `xapian.ValueRangeProcessor` gets called by the QueryParser. What happens is that each processor in turn is passed the start and end of the range. If it doesn't understand the range, it should return `xapian.BAD_VALUENO`. If it *does* understand the range, it should return the value number to use with `xapian.Query.OP_VALUE_RANGE` and if it wants to, it can modify the start and end values (to convert them to the correct format for the string comparison which `xapian.OP_VALUE_RANGE` uses).

What we're going to do is to write a custom `xapian.ValueRangeProcessor` that accepts numbers in the range 500,000 to 50,000,000; these can't possibly be years in our data set, and encompass the full range of populations. If either number is outside that range, we will return `xapian.BAD_VALUENO` and the QueryParser will move on.

```python
class PopulationValueRangeProcessor(xapian.ValueRangeProcessor):
    def __init__(self, value, low, high):
        super(PopulationValueRangeProcessor, self).__init__()
        self.nvrp = xapian.NumberValueRangeProcessor(value)
        self.low = low
        self.high = high
    def __call__(self, begin, end):
        if begin != u"":
            try:
                _begin = int(begin)
                if _begin < self.low or _begin > self.high:
                    raise ValueError()
            except:
                return (xapian.BAD_VALUENO, begin, end)
        if end != u"":
            try:
                _end = int(end)
                if _end < self.low or _end > self.high:
                    raise ValueError()
            except:
                return (xapian.BAD_VALUENO, begin, end)
        return self.nvrp(begin, end)
queryparser.add_valuerangeprocessor(
    PopulationValueRangeProcessor(3, 500000, 50000000)
    )
```

Most of the work is in *__call__* (python's equivalent of *operator()* in C++), which gets called with the two strings at either end of the range in the query string; either but not both can be the empty string, which indicates an open-ended range. In python this method should return a tuple of the value slot and the two strings modified so they can be used for `OP_VALUE_RANGE`. Rather than re-implement `xapian.NumberValueRangeProcessor`, we wrap it to do the serialisation (due to the way python interacts with the API it's currently not possible to subclass it successfully here).

Value range processors are called in the order they're added, so our custom one gets a chance to look at all ranges, but will only 'claim' ranges which use integer numbers within the 500 thousand to 50 million range.

We can then search for states by population, such as all over 10 million:

```
$ python code/python/search_ranges2.py statesdb 10000000..
1: #007 State of California September 9, 1850
        Population 37,253,956
2: #019 State of Texas December 29, 1845
        Population 25,145,561
3: #027 State of Illinois December 3, 1818
        Population 12,830,632
4: #030 State of Ohio March 1, 1803
        Population 11,536,504
5: #035 State of Florida March 3, 1845
        Population 18,801,310
6: #040 Commonwealth of Pennsylvania December 12, 1787
        Population 12,702,379
7: #041 State of New York July 26, 1788
        Population 19,378,102
INFO:xapian.search:'10000000..'[0:10] = 7 19 27 30 35 40 41
```

Or all that joined the union in the 1780s and have a population now over 10 million:

```
$ python code/python/search_ranges2.py statesdb 1780..1789 10000000..
1: #040 Commonwealth of Pennsylvania December 12, 1787
        Population 12,702,379
2: #041 State of New York July 26, 1788
        Population 19,378,102
INFO:xapian.search:'1780..1789 10000000..'[0:10] = 40 41
```

With a little more work, we could support ranges such as '..5m' to mean up to 5 million, or '..750k' for up to 750 thousand.

4.2.7 Performance limitations

Without other terms in a query, a `xapian.ValueRangeProcessor` will cause a value operation to be performed across the whole database, which means loading all the values in a given slot. On a small database, this isn't a problem, but for a large one it can have performance implications: you may end up with very slow queries.

Todo

the above paragraph isn't entirely inaccurate; the processor is unweighted, so if there's no other query, and the docid ordering is don't care or ascending, then the search can terminate early. If the VRP isn't matching many documents, that could still be slow, but might not be. If it's not matching any documents, it might be fast because the bounds on stored values may show that it can't match anything. Oh, it's all quite complicated really. It would be nice to explain how this is done somewhere, but probably not here.

If combined with a suitable term-based query (such as an *OP_AND* query over one or more terms), this performance impact will be less because the range operation will only have to run over the potential matches, which are reduced from the entire database by the term-based query.

If, as well as using document values, you also convert groups of those values into terms, you can provide those term-based queries even when your users are only interested in a pure range search. For instance, consider the population information. If you divide the range of populations into a number of subranges, you can allocate a term to describe each. We'll use a prefix of *XP* (for "population") here.

Population range	Term
0 - 10 million	XP0
10 - 20 million	XP1
20 - 30 million	XP2
30 - 40 million	XP3

Then you can use a custom `xapian.ValueRangeProcessor` to both generate the relevant information for QueryParser to construct an `OP_VALUE_RANGE` query and to record which subranges we're interested in. For instance, if the user asks for '..15000000', your processor can remember that and later spit out an additional `OP_AND` query with terms *XP0* and *XP1*, that can be combined with the query generated by the QueryParser using `OP_FILTER`.

Todo

actually, you can't safely combine the query with an external filter, because other bits of the query might be higher level. For example, a query of '1790..1799 OR york' couldn't have the filter applied to the generated query because it shouldn't be applied to the "york" part.

Todo

possibly implementing this example would help make it more clear.

4.3 Facets

Xapian provides functionality which allows you to dynamically generate complete lists of values which feature in matching documents. For example, colour, manufacturer, size values are good candidates for faceting.

There are numerous potential uses this can be put to, but a common one is to offer the user the ability to narrow down their search by filtering it to only include documents with a particular value of a particular category. This is often referred to as *faceted search*.

4.3.1 Implementation

Faceting works against information stored in document value slots [link to value slots] and, when executed, provides a list of the unique values for that slot together with a count of the number of times each value occurs.

Todo

add the link above

Indexing

No additional work is needed to implement faceted searching, except to ensure that the values you wish to use in facets are stored as document values.

```
def index(datapath, dbpath):
    # Create or open the database we're going to be writing to.
    db = xapian.WritableDatabase(dbpath, xapian.DB_CREATE_OR_OPEN)

    # Set up a TermGenerator that we'll use in indexing.
    termgenerator = xapian.TermGenerator()
    termgenerator.set_stemmer(xapian.Stem("en"))

    for fields in parse_csv_file(datapath):
        # 'fields' is a dictionary mapping from field name to value.
        # Pick out the fields we're going to index.
        description = fields.get('DESCRIPTION', u'')
        title = fields.get('TITLE', u'')
        identifier = fields.get('id_NUMBER', u'')
        collection = fields.get('COLLECTION', u'')
        maker = fields.get('MAKER', u'')

        # We make a document and tell the term generator to use this.
        doc = xapian.Document()
        termgenerator.set_document(doc)

        # Index each field with a suitable prefix.
        termgenerator.index_text(title, 1, 'S')
        termgenerator.index_text(description, 1, 'XD')

        # Index fields without prefixes for general search.
        termgenerator.index_text(title)
        termgenerator.increase_termpos()
        termgenerator.index_text(description)

        # Add the collection as a value in slot 0.
        doc.add_value(0, collection)

        # Add the maker as a value in slot 1.
        doc.add_value(1, maker)

        # Store all the fields for display purposes.
        doc.set_data(json.dumps(fields, encoding='utf8'))

        # We use the identifier to ensure each object ends up in the
        # database only once no matter how many times we run the
        # indexer.
        idterm = u"Q" + identifier
        doc.add_boolean_term(idterm)
        db.replace_document(idterm, doc)
```

Here we're using two value slots: 0 contains the collection, and 1 contains the name of whoever made the object. We know from the documentation of the dataset that both from fixed and curated lists, so we don't have to worry about normalising the values before using them as facets. Let's run that to build a dataset with document values suitable for faceting:

```
$ python code/python/index_facets.py data/100-objects-v1.csv db
```

Querying

To query, Xapian uses the concept of spies to observe slots of matched documents during a search.

The procedure works in three steps: first, you create a spy (instance of `xapian.ValueCountMatchSpy`) for each slot you want the facets; second, you bind each spy to the `xapian.Enquire` using `add_matchspy(spy)`; third, after the search was performed, you retrieve the results that each spy observed. This is an example of how this is done:

```
    # Set up a spy to inspect the MAKER value at slot 1
    spy = xapian.ValueCountMatchSpy(1)
    enquire.add_matchspy(spy)

    for match in enquire.get_mset(offset, pagesize, 100):
        fields = json.loads(match.document.get_data())
        print u"%(rank)i: #%(docid)3.3i %(title)s" % {
            'rank': match.rank + 1,
            'docid': match.docid,
            'title': fields.get('TITLE', u''),
            }
        matches.append(match.docid)

    # Parse and display the spy values
    for facet in spy.values():
        print "Facet: %(term)s; count: %(count)i" % {
            'term' : facet.term,
            'count' : facet.termfreq
        }

    # Finally, make sure we log the query and displayed results
    support.log_matches(querystring, offset, pagesize, matches)
```

Here we're faceting on value slot 1, which is the object maker. After you get the MSet, you can ask the spy for the facets it found, including the frequency. Note that although we're generally only showing ten matches, we use a parameter to `get_mset()` called *checkatleast*, so that the entire dataset is considered and the facet frequencies are correct. See Limitations for some discussion of the implications of this. Here's the output:

```
$ python code/python/search_facets.py db clock
1: #044 Two-dial clock by the Self-Winding Clock Co; as used on the
2: #096 Clock with Hipp pendulum (an electric driven clock with Hipp
3: #012 Assembled and unassembled EXA electric clock kit
4: #098 'Pond' electric clock movement (no dial)
5: #083 Harrison's eight-day wooden clock movement, 1715.
6: #005 "Ever Ready" ceiling clock
7: #039 Electric clock of the Bain type
8: #061 Van der Plancke master clock
9: #064 Morse electrical clock, dial mechanism
10: #052 Reconstruction of Dondi's Astronomical Clock, 1974
```

```
Facet: Bain, Alexander; count: 3
Facet: Bloxam, J. M.; count: 1
Facet: Braun (maker); count: 1
Facet: British Horo-Electric Ltd. (maker); count: 1
Facet: British Vacuum Cleaner and Engineering Co. Ltd., Magneto Time division (maker); count: 1
Facet: EXA; count: 1
Facet: Ever Ready Co. (maker); count: 2
Facet: Ferranti Ltd.; count: 1
Facet: Galilei, Galileo, 1564-1642; Galilei, Vincenzio, 1606-1649; count: 1
Facet: Harrison, John (maker); count: 1
Facet: Hipp, M.; count: 1
Facet: La Précision Cie; count: 1
Facet: Lund, J.; count: 1
Facet: Morse, J. S.; count: 1
Facet: Self Winding Clock Company; count: 1
Facet: Self-Winding Clock Co. (maker); count: 1
Facet: Synchronome Co. Ltd. (maker); count: 2
Facet: Thwaites and Reed Ltd.; count: 1
Facet: Thwaites and Reed Ltd. (maker); count: 1
Facet: Viviani, Vincenzo; count: 1
Facet: Vulliamy, Benjamin, 1747-1811; count: 1
Facet: Whitefriars Glass Ltd. (maker); count: 1
INFO:xapian.search:'clock'[0:10] = 44 96 12 98 83 5 39 61 64 52
```

Note that the spy will give you facets in alphabetical order, not in order of frequency; if you want to show the most frequent first you should use the *top_values* iterator (`begin_top_values()` in C++ and some other languages).

If you want to work with multiple facets, you can register multiple `xapian.ValueCountMatchSpy` objects before running `get_mset()`, although each additional one will have some performance impact.

Restricting by Facets

If you're using the facets to offer the user choices for narrowing down their search results, you then need to be able to apply a suitable filter.

For a single value, you could use `xapian.Query.OP_VALUE_RANGE` with the same start and end, or `xapian.MatchDecider`, but it's probably most efficient to also index the categories as suitably prefixed boolean terms and use those for filtering.

4.3.2 Limitations

The accuracy of Xapian's faceting capability is determined by the number of records that are examined by Xapian whilst it is searching. You can control this number by specifying the *checkatleast* value of `get_mset`; however it is important to be aware that increasing this number may have an effect on overall query performance.

Todo

It is probably worth saying that a typical database will not hit these performance issues

4.3.3 In Development

Some additional features currently in development may benefit users of facets. These are:

- Multiple values in slots: this will allow you to have a single value slot (e.g. colour) which contains multiple values (e.g. red, blue). This will also allow you to create a facet by colour which is aware of these multiple values, giving counts for both red and blue.
- Bucketing: this provides a means to group together numeric facets, so that a single facet can contain a range of values (e.g. price ranges).

4.4 Sorting

By default, Xapian orders search results by decreasing relevance score. However, it also allows results to be ordered by other criteria, or a combination of other criteria and relevance score.

If two or more results compare equal by the sorting criteria, then their order is decided by their document ids. By default, the document ids sort in ascending order (so a lower document id is "better"), but descending order can be chosen using `enquire.set_docid_order(enquire.DESCENDING);`. If you have no preference, you can tell Xapian to use whatever order is most efficient using `enquire.set_docid_order(enquire.DONT_CARE);`.

It is also possible to change the way that the relevance scores are calculated - for details, see the *document on weighting schemes and scoring* for details.

4.4.1 Sorting by Value

You can order documents by comparing a specified document value. Note that the comparison used compares the byte values in the value (i.e. it's a string sort ignoring locale), so `1` < `10` < `2`. If you want to encode the value such that it sorts numerically, use `Xapian::sortable_serialise()` to encode values at index time - this works equally well on integers and floating point values:

```
doc.add_value(0, xapian.sortable_serialise(price));
```

There are three methods which are used to specify how the value is used to sort, depending if/how you want relevance used in the ordering:

- `Enquire::set_sort_by_value()` specifies the relevance doesn't affect the ordering at all.
- `Enquire::set_sort_by_value_then_relevance()` specifies that relevance is used for ordering any groups of documents for which the value is the same.
- `Enquire::set_sort_by_relevance_then_value()` specifies that documents are ordered by relevance, and the value is only used to order groups of documents with identical relevance values (note: the weight has to be exactly the same for values to determine the order, so this method isn't very useful when using BM25 with the default parameters, as that will rarely give identical scores to different documents).

We'll use the states dataset to demonstrate this, and the code from dealing with dates in the *range queries* HOWTO:

```
$ python code/python/index_ranges2.py data/states.csv statesdb
```

This has three document values: slot 1 has the year of admission to the union, slot 2 the full date (as "YYYYMMDD"), and slot 3 the latest population estimate. So if we want to sort by year of entry to the union and then within that by relevance, we want to add the following before we call *get_mset*:

```
    enquire.set_sort_by_value_then_relevance(1, False)
```

The final parameter is *False* for ascending order, *True* for descending. We can then run sorted searches like this:

```
$ python code/python/search_sorting.py statesdb spanish
1: #019 State of Texas December 29, 1845
        Population 25,145,561
```

```
2: #004 State of Montana November 8, 1889
        Population 989,415
INFO:xapian.search:'spanish'[0:10] = 19 4
```

4.4.2 Generated Sort Keys

To allow more elaborate sorting schemes, Xapian allows you to provide a functor object subclassed from xapian-class:*KeyMaker* which generates a sort key for each matching document which is under consideration. This is called at most once for each document, and then the generated sort keys are ordered by comparing byte values (i.e. with a string sort ignoring locale).

Sorting by Multiple Values

There's a standard subclass xapian-class:*MultiValueKeyMaker*` which allows sorting on more than one document value (so the first document value specified determines the order; amongst groups of documents where that's the same, the second document value determines the order, and so on).

We'll use this to change our sorted search above to order by year of entry to the union and then by decreasing population.

```
keymaker = xapian.MultiValueKeyMaker()
keymaker.add_value(1, False)
keymaker.add_value(3, True)
enquire.set_sort_by_key_then_relevance(keymaker, False)
```

As with the *Enquire* methods, *add_value* has a second parameter that controls whether it uses an ascending or descending sort. So now we can run a search with a more complex sort:

```
$ python code/python/search_sorting2.py statesdb State
1: #040 Commonwealth of Pennsylvania December 12, 1787
        Population 12,702,379
2: #043 State of New Jersey December 18, 1787
        Population 8,791,894
3: #049 State of Delaware December 7, 1787
        Population 897,934
4: #041 State of New York July 26, 1788
        Population 19,378,102
5: #034 State of Georgia January 2, 1788
        Population 9,687,653
6: #038 Commonwealth of Virginia June 25, 1788
        Population 8,001,024
7: #046 Commonwealth of Massachusetts February 6, 1788
        Population 6,547,629
8: #050 State of Maryland April 28, 1788
        Population 5,773,552
9: #036 State of South Carolina May 23, 1788
        Population 4,625,384
10: #048 State of Connecticut January 9, 1788
        Population 3,574,097
INFO:xapian.search:'State'[0:10] = 40 43 49 41 34 38 46 50 36 48
```

Other Uses for Generated Keys

xapian-class:*KeyMaker*` can also be subclassed to sort based on a calculation. For example, "sort by geographical distance", where a subclass could take the latitude and longitude of the user's location, and coordinates of the document from a value slot, and sort results so that those closest to the user are ranked highest.

For this, we're going to want the geographical coordinates of each state stored in a value. We can use the approximate middle of the state for this purpose, which is calculated for us when parsing the *states.csv* file:

```
        midlat = fields['midlat']
        midlon = fields['midlon']
        if midlat and midlon:
            doc.add_value(4, "%f,%f" % (midlat, midlon))
```

We don't have to sort on these, so we've just put them both into one slot that we can easily read them out from again:

```
$ python code/python/index_values_with_geo.py data/states.csv statesdb
```

Now we need a KeyMaker; let's have it return a key that sorts by distance from Washington, DC.

```
    class DistanceKeyMaker(xapian.KeyMaker):
        def __call__(self, doc):
            # we want to return a sortable string which represents
            # the distance from Washington, DC to the middle of this
            # state.
            coords = map(float, doc.get_value(4).split(','))
            washington = (38.012, -77.037)
            return xapian.sortable_serialise(
                distance_between_coords(coords, washington)
                )
    enquire.set_sort_by_key_then_relevance(DistanceKeyMaker(), False)
```

And running it is as simple as before:

```
$ python code/python/search_sorting3.py statesdb State
1: #050 State of Maryland 17880428
        Population 5773552
2: #049 State of Delaware 17871207
        Population 897934
3: #040 Commonwealth of Pennsylvania 17871212
        Population 12702379
4: #043 State of New Jersey 17871218
        Population 8791894
5: #039 State of West Virginia 18630620
        Population 1859815
6: #037 State of North Carolina 17891121
        Population 9535483
7: #041 State of New York 17880726
        Population 19378102
8: #038 Commonwealth of Virginia 17880625
        Population 8001024
9: #048 State of Connecticut 17880109
        Population 3574097
10: #036 State of South Carolina 17880523
        Population 4625384
INFO:xapian.search:'State'[0:10] = 50 49 40 43 39 37 41 38 48 36
```

4.5 Collapsing of Search Results

Todo

Write about how to use collapsing with searches. Discuss collapsing to return only a unique item for each value, and collapsing to return multiple. Make clear that it simply excludes some items from the result set; doesn't reorder results, or guarantee to return the top N from each category. Also add a code example.

Xapian provides the ability to eliminate "duplicate" documents from the MSet. This feature is known as "collapsing" - think of a pile of duplicates being collapsed down to leave a single result (or a small number of results).

The collapsing always removes the worse ranked documents (if ranking by relevance, those with the lowest weight; if ranking by sorting, those which sort lowest).

Whether two documents count as duplicates of one another is determined by their "collapse key". If a document has an empty collapse key, it will never be collapsed, but otherwise documents with the same collapse key will be collapsed together.

Currently the collapse key is taken from a value slot you specify (via the method `xapian.Enquire.set_collapse_key()`), but in the future you should be able to build collapse keys dynamically using `xapian.KeyMaker` as you already can for sort keys.

4.5.1 Performance

The collapsing is performed during the match process, so it is pretty efficient. In particular, this approach is much better than generating a larger MSet and post-processing it.

However, if the collapsing eliminates a lot of documents then the collapsed search will typically take rather longer than the uncollapsed search because the matcher has to consider many more potential matches.

4.5.2 API

To enable collapsing, call the method `xapian.Enquire.set_collapse_key` with the value slot, and optionally the number of matches with each collapse key to keep (this defaults to 1 if not specified), e.g.:

```
// Collapse on value slot 4, leaving at most 2 documents with each
// collapse key.
enquire.set_collapse_key(4, 2);
```

Once you have the `xapian.MSet` object, you can read the collapse key for each match with `xapian.MSetIterator.get_collapse_key()`, and also the "collapse count" with `xapian.MSetIterator.get_collapse_count()`. The latter is a lower bound on the number of documents with the same collapse key which collapsing eliminated.

Beware that if you have a percentage cutoff active, then the collapse count will always be either 0 or 1 as it is hard to tell if the collapsed documents would have failed the cutoff.

4.5.3 Statistics

As well as the usual bounds and estimate of the "full" MSet size (i.e. the size if you'd asked for enough matches to get them all), the matcher also calculates bounds and an estimate for what the MSet size would be if collapsing had not been used - you can obtain these using these methods:

```
Xapian::doccount get_uncollapsed_matches_lower_bound() const;
Xapian::doccount get_uncollapsed_matches_estimated() const;
Xapian::doccount get_uncollapsed_matches_upper_bound() const;
```

4.5.4 Examples

Here are some ways this feature can be used:

Duplicate Elimination

If your document collection includes some identical documents, it's unhelpful when these show up in the search results. Sometimes it is possible to eliminate them at index time, but this isn't always feasible.

If you store a checksum (e.g. SHA1 or MD5) of the document contents and store this in a document value then you can collapse on this to eliminate such duplicates.

If the document files will be identical, then the checksum can just be of the file, but sometimes it makes sense to extract and normalise the text, then calculate the checksum of this.

Restricting the Number of Matches per Source

It's sometimes desirable to avoid one source dominating the results. For example, in a web search application, you might want to show at most three matches from any website, in which case you could collapse on the hostname with collapse_max set to 3.

When displaying the results, you can use the collapse count of each match to inform the user that there are at least that many other matches for this host (unless you are also using a percentage cutoff - see above). If it is non-zero it means you can usefully provide a "show all documents for host <COLLAPSE KEY>" button which reruns the search without collapsing and with a boolean filter for a prefixed term containing the hostname (though note that this may not always give a button when there are collapsed documents because the collapse count is a lower bound and may be zero when there are collapsed matches with the same key).

This approach isn't just useful for web search - the "source" can be defined usefully in many applications. For example, a forum or mailing list search could collapse on a topic or thread identifier, an index at the chapter level could collapse on a book identifier (such as an ISBN), etc.

4.6 Spelling Correction

4.6.1 Introduction

Xapian provides functionality which can suggest corrections for misspelled words in queries, or in other situations where it might be useful. The suggestions can be generated dynamically from the data that has been indexed, so the correction facility isn't tied to particular languages, and can suggest proper nouns or specialist technical terms.

4.6.2 Indexing

The spelling dictionary can be built with words from indexed text, or by adding words from a static word list, or a combination of the two.

Static spelling data

If `db` is a `xapian.WritableDatabase`, you can add to the spelling dictionary using:

```
db.add_spelling(word, frequency_inc);
```

The `frequency_inc` parameter is optional, and defaults to 1.

And the corresponding way to remove from the spelling dictionary is:

```
db.remove_spelling(word, frequency_dec);
```

The `frequency_dec` parameter is optional and defaults to 1. If you try to decrement the frequency of a word by more than its current value, it's just removed.

Dynamic spelling data

`xapian.TermGenerator` can be configured to automatically add words from indexed documents to the spelling dictionary:

```
Xapian::TermGenerator indexer;
indexer.set_database(db);
indexer.set_flags(indexer.FLAG_SPELLING);
```

Note that you must call the `xapian.TermGenerator.set_database()` method as well as setting `xapian.TermGenerator.FLAG_SPELLING` so that Xapian knows where to add the spelling dictionary entries.

If a document is removed or replaced, any spelling dictionary entries that were added when it was originally indexed won't be automatically removed. This might seem like a flaw, but in practice it rarely causes problems, and spellings in documents which were in the database, or in older versions of documents, are still interesting. You can think of this as using the history of the document collection as a source of spelling data.

If you really want these entries removed, you can run through the termlist of each document you are about to remove or replace (if you indexed terms unstemmed) and call `xapian.TermGenerator.remove_spelling()` for each word.

4.6.3 Searching

Todo

Getting a correction for a single word "by hand"

QueryParser Integration

If `xapian.QueryParser.FLAG_SPELLING_CORRECTION` is passed to `xapian.QueryParser.parse_query()` and `xapian.QueryParser.set_database()` has been called, the QueryParser will look for corrections for words in the query. In Xapian 1.2.2 and earlier, it only did this for terms which aren't found in the database.

If a correction is found, then a modified version of the query string will be generated which can be obtained by calling `xapian.QueryParser.get_corrected_query_string()`. However, the original query string will still be parsed, since you'll often want to ask the user "Did you mean: [...] ?" - if you want to automatically use the corrected eform, just call `xapian.QueryParser.parse_query()` on it.

4.6.4 Omega

As of Omega 1.1.1, omindex and scriptindex support indexing spelling correction data and omega supports suggesting corrected spellings at search time. See the Omega documentation for more details.

4.6.5 Algorithm

A list of candidate words is generated by matching trigrams (groups of 3 adjacent characters) in the candidates against those in the misspelled word. As well as groups of adjacent characters, "starts" and "ends" are generated with the first two and last two characters respectively (e.g. "FISH" generates: "<start>FI", "FIS", "ISH", and "SH<end>").

This technique alone would missing many single-edit errors in two and three character words, so we handle these specially as follows:

For a three character word (e.g. "ABC"), we generate trigrams for the two transposed forms too ("BAC" and "ACB"), in addition to "<start>AB", "ABC", and "BC<end>".

For a two character word (e.g. "AB"), we generate the special start and end trigrams for the reversed form (i.e. "BA"), so the trigrams are "<start>AB", "AB<end>", "<start>BA", and "BA<end>".

And for two, three, and four character words, we generate "bookend" bigrams consisting of the prefix 'B' followed by the first and last letters. This allows us to handle transposition of the middle two characters of a four letter word, substitution or deletion of the middle character of a three letter word, or insertion in the middle of a two letter word.

Note that we don't attempt to suggest corrections for single character words at all, since the suggestions are unlikely to be of good quality (we'd always suggest the same correction for a given database, probably "a" for English). We also don't currently attempt to suggest substitution corrections for two character words, though this would perhaps be useful in some cases.

Those candidates with the better trigram matches are compared to the misspelled word by calculating the "edit distance" - that's the smallest number of operations required to turn one word into another. The allowed operations are: insert a character; delete a character; change a character to another; transpose two adjacent characters. The candidate with the smallest edit distance is found, and if more than one word has the smallest edit distance, that which occurs the most times is chosen. If there's a tie of this too, it's essentially arbitrary which is chosen.

If the word passed in is in the spelling dictionary, then a candidate will still be returned if one is found with the same or greater frequency.

The maximum edit distance to consider can be specified as an optional parameter to `xapian.Database.get_spelling_suggestion()`. If not specified, the default is 2, which generally does a good job. 3 is also a reasonable choice in many cases. For most uses, 1 is probably too low, and 4 or more probably too high.

Unicode Support

Trigrams are generated at the byte level, but the edit distance calculation currently works with Unicode characters, so `xapian.Database.get_spelling_suggestion()` should suggest suitable spelling corrections respecting the specified (or default) edit distance threshold.

4.6.6 Current Limitations

Exactness

Because Xapian only tests the edit distance for terms which match well (or at all!) on trigrams, it may not always suggest the same answer that would be found if all possible words were checked using the edit distance algorithm. However, the best answer will usually be found, and an exhaustive search would be prohibitively expensive for many uses.

Backend Support

Currently spelling correction is supported for chert, and brass databases. It works with a single database or multiple databases (use `xapian.Database.add_database()` as usual). We've no plans to support it for the InMemory backend, but we do intend to support it for the remote backend in the future.

Prefixed Terms

Currently spelling correction ignores prefixed terms.

QueryParser changed word locations

The QueryParser doesn't currently report the locations of changed words in the query string, so it's a bit fiddly to mark up the altered words specially in HTML output, for example.

API limitations

Only a single possible correction can currently be returned. It would be nice to be able to get a ranked list.

Spelling data from deleted documents

If you're adding spelling data using `xapian.TermGenerator`, then spelling data from deleted (or updated) documents doesn't automatically get removed from the spelling dictionary when documents are deleted. This is usually not a problem, can be if the topic area covered by a database moves significantly with time.

4.6.7 References

The algorithm used to calculate the edit distance is based on that described in the paper "An extension of Ukkonen's enhanced dynamic programming ASM algorithm" by Hal Berghel, University of Arkansas, and David Roach, Acxiom Corporation. It's available online at: http://berghel.net/publications/asm/asm.php

4.7 Synonyms

4.7.1 Introduction

Xapian provides support for storing a synonym dictionary, or thesaurus. This can be used by the xapian-class:*QueryParser* class to expand terms in user query strings, either automatically, or when requested by the user with an explicit synonym operator (~).

Note: Xapian doesn't offer automated generation of the synonym dictionary.

4.7.2 Model

The model for the synonym dictionary is that a term or group of consecutive terms can have one or more synonym terms. A group of consecutive terms is specified in the dictionary by simply joining them with a single space between each one.

Todo

Discuss interactions with stemming (ie, should the input and/or output values in the synonym table be stemmed).

4.7.3 Adding Synonyms

Todo

Document this!

4.7.4 QueryParser Integration

In order for any of the synonym features of the QueryParser to work, you must call `xapian.QueryParser.set_database()` to specify the database to use.

If `FLAG_SYNONYM` is passed to `xapian.QueryParser.parse_query()` then the QueryParser will recognise ~ in front of a term as indicating a request for synonym expansion. If `FLAG_LOVEHATE` is also specified, you can use + and − before the ~ to indicate that you love or hate the synonym expanded expression.

A synonym-expanded term becomes the term itself OR-ed with any listed synonyms, so `~truck` might expand to `truck OR lorry OR van`. A group of terms is handled in much the same way.

If a term to be synonym expanded will be stemmed by the QueryParser, then synonyms will be checked for the unstemmed form first, and then for the stemmed form, so you can provide different synonyms for particular unstemmed forms if you want to.

If `FLAG_AUTO_SYNONYMS` is passed to `xapian.QueryParser.parse_query()` then the QueryParser will automatically expand any term which has synonyms, unless the term is in a phrase or similar.

If `FLAG_AUTO_MULTIWORD_SYNONYMS` is passed to `xapian.QueryParser.parse_query()` then the QueryParser will look at groups of terms separated only by whitespace and try to expand them as term groups. This is done in a “greedy” fashion, so the first term which can start a group is expanded first, and the longest group starting with that term is expanded. After expansion, the QueryParser will look for further possible expansions starting with the term after the last term in the expanded group.

4.7.5 OP_SYNONYM

Todo

Query.OP_SYNONYM, and how that relates to synonym expansion.

4.7.6 Current Limitations

Explicit multi-word synonyms

There ought to be a way to explicitly request expansion of multi-term synonyms, probably with the syntax `~"stock market"`. This hasn’t been implemented yet though.

Backend Support

Currently synonyms are supported by the chert and brass databases. They work with a single database or multiple databases (use `xapian.Database.add_database()` as usual). We've no plans to support them for the InMemory backend, but we do intend to support them for the remote backend in the future.

4.8 How to change how documents are scored

The document weighting schemes which Xapian includes by default are `BM25Weight`, `TradWeight` and `BoolWeight`. The default is BM25Weight.

4.8.1 BoolWeight

BoolWeight assigns a weight of 0 to all documents, so the ordering is determined solely by other factors.

4.8.2 TradWeight

TradWeight implements the original probabilistic weighting formula, which is essentially a special case of BM25 (it's BM25 with k2 = 0, k3 = 0, b = 1, and min_normlen = 0, except that all the weights are scaled by a constant factor).

4.8.3 BM25Weight

The BM25 weighting formula which Xapian uses by default has a number of parameters. We have picked some default parameter values which do a good job in general. The optimal values of these parameters depend on the data being indexed and the type of queries being run, so you may be able to improve the effectiveness of your search system by adjusting these values, but it's a fiddly process to tune them so people tend not to bother.

Todo

Say something more useful about tuning the parameters!

4.8.4 LMWeight (Unigram language modelling)

An important aspect of language model-based weighting is that, since not all terms appear in all documents (and hence the wdf of some terms is zero with respect to a given document), we have to employ smoothing to avoid problems.

Xapian provides *four different smoothing types*, which take further parameters to control the effects of smoothing; we have picked some default parameter values which do a good job, using two stage smoothing.

The UnigramLM weighting formula is based on an original approach by Bruce Coft. It uses statistical language modelling; 'unigram' in this case means that words are considered to occur independently.

4.8.5 Custom Weighting Schemes

You can also implement your own weighting scheme, provided it can be expressed in the form of a sum over the matching terms, plus an extra term which depends on term-independent statistics (such as the normalised document length).

For example, here's an implementation of "coordinate matching" - each matching term scores one point:

```
class CoordinateWeight : public Xapian::Weight {
  public:
    CoordinateWeight * clone() const { return new CoordinateWeight; }
    CoordinateWeight() { }
    ~CoordinateWeight() { }

    std::string name() const { return "Coord"; }
    std::string serialise() const { return ""; }
    CoordinateWeight * unserialise(const std::string &) const {
        return new CoordinateWeight;
    }

    Xapian::weight get_sumpart(Xapian::termcount, Xapian::doclength) const {
        return 1;
    }
    Xapian::weight get_maxpart() const { return 1; }

    Xapian::weight get_sumextra(Xapian::doclength) const { return 0; }
    Xapian::weight get_maxextra() const { return 0; }

    bool get_sumpart_needs_doclength() const { return false; }
};
```

4.8.6 Implement a custom weighting scheme that requires various statistics

The Coordinate scheme given above does not require any statistics. However, custom weighting schemes that require various statistics such as average document length in the database, the query length, total number of documents in the collection etc. can also be implemented.

For that, the weighting scheme subclassed from xapian-class:*Weight* simply needs to "tell" xapian-class:*Weight* which statistics it will be needing. This is done by calling the need_stat(STATISTIC REQUIRED) function in the constructor of the subclassed weighting scheme. Note however, that only those statistics which are absolutely required must be asked for as collecting statistics is expensive. For a full list of statistics currently available from xapian-class:*Weight* and the enumerators required to access them, please refer to: http://xapian.org/docs/sourcedoc/html/classXapian_1_1Weight.html#e3c11f1d2d96a18e0eb9b9b31c5c5479

The statistics can then be obtained by the subclass by simply calling the corresponding function of the xapian-class:*Weight* class. For eg:- The document frequency (Term frequency) of the term can be obtained by calling get_termfreq(). For a full list of functions required to obtain various statistics, refer: http://xapian.org/docs/sourcedoc/html/weight_8h_source.html#l00277

Example:- Consider a simple weighting scheme such as a pseudo Tf-Idf weighting scheme which returns the document weight as the product of the within document frequency of the term and the inverse of the document frequency of the term (Inverse of the number of documents the term appears in).

The implementation will be as follows:

```
class TfIdfWeight : public Xapian::Weight {
  public:
    TfIdfWeight * clone() const { return new TfIdfWeight; }
    TfIdfWeight() {
        need_stat(WDF);
        need_stat(TERMFREQ);
        need_stat(WDF_MAX);
    }
    ~TfIdfWeight() { }
    std::string name() const { return "TfIdf"; }
    std::string serialise() const { return ""; }
    TfIdfWeight * unserialise(const std::string &) const {
        return new TfIdfWeight;
    }

    Xapian::weight get_sumpart(Xapian::termcount wdf, Xapian::doclength) const {
        Xapian::doccount df = get_termfreq();
        double wdf_double(wdf);
        Xapian::weight wt = wdf_double / df;
        return wt;
    }

    Xapian::weight get_maxpart() const {
        Xapian::doccount df = get_termfreq();
        double max_wdf(get_wdf_upper_bound());
        Xapian::weight max_weight = max_wdf / df;
        return max_weight;
    }
    Xapian::weight get_sumextra(Xapian::doclength) const { return 0; }
    Xapian::weight get_maxextra() const { return 0; }
};
```

Note: The get_maxpart() function returns an upper bound on the weight returned by get_sumpart(). In order to do that, it requires the WDF_MAX statistic (the maximum wdf of the term among all documents).

4.9 Iterate through all documents

Sometimes you want to access all the documents in a Xapian database. This can actually be done in two ways:

4.9.1 MatchAll Queries

The *Xapian::Query::MatchAll* query is a special static query which will match all documents in the database. If you run this query on its own, with appropriate start and end parameters, you could retrieve all the documents. However be aware that even if you paged through the result sets, when you try to access a page deep in the result set a lot of processing and memory will be used even if the page is small, so running a plain *MatchAll* query is rarely a good idea.

However, this method *is* appropriate if you're constructing a complicated query, and one of the components of that query should be all the documents. In particular, since Xapian doesn't support a unary *NOT* operator, if you want to run a "pure NOT" query to retrieve all documents which do not contain a given term, this can be only be done using a *MatchAll* query and the binary *NOT* operator.

4.9.2 Iterating through all documents

If you do need access to all the documents in the database, it is better to use a "posting list iterator". Such an iterator, which returns all documents in the database, can be created using:

```
Xapian::Database::postlist_begin("")
```

In Xapian, a postlist is a list of the documents in which a term exists. Here, we're again using the special "empty" term, which implicitly matches all documents, to get an iterator over all documents.

The iterator can be dereferenced to get the document IDs; to get the actual documents, the xapian-method:*Database::get_document()* method should be used.

Todo

relevance feedback and query expansion

CHAPTER 5

Advanced features

5.1 Posting sources

Table of contents

5.1.1 Introduction

Xapian::PostingSource is an API class which you can subclass to feed data to Xapian's matcher. This feature can be made use of in a number of ways - for example:

As a filter - a subclass could return a stream of document ids to filter a query against.

As a weight boost - a subclass could return every document, but with a varying weight so that certain documents receive a weight boost. This could be used to prefer documents based on some external factor, such as age, price, proximity to a physical location, link analysis score, etc.

As an alternative way of ranking documents - if the weighting scheme is set to Xapian::BoolWeight, then the ranking will be entirely by the weight returned by Xapian::PostingSource.

5.1.2 Anatomy

When first constructed, a PostingSource is not tied to a particular database. Before Xapian can get any postings (or statistics) from the source, it needs to be supplied with a database. This is performed by the init() method, which is passed a single parameter holding the database to use. This method will always be called before asking for any information about the postings in the list. If a posting source is used for multiple searches, the init() method will be called before each search; implementations must cope with init() being called multiple times, and should always use the database provided in the most recent call:

```
virtual void init(const Xapian::Database & db) = 0;
```

Three methods return statistics independent of the iteration position. These are upper and lower bounds for the number of documents which can be returned, and an estimate of this number:

```
virtual Xapian::doccount get_termfreq_min() const = 0;
virtual Xapian::doccount get_termfreq_max() const = 0;
virtual Xapian::doccount get_termfreq_est() const = 0;
```

These methods are pure-virtual in the base class, so you have to define them when deriving your subclass.

It must always be true that:

```
get_termfreq_min() <= get_termfreq_est() <= get_termfreq_max()
```

PostingSources must always return documents in increasing document ID order.

After construction, a PostingSource points to a position *before* the first document id - so before a docid can be read, the position must be advanced by calling `next()`, `skip_to()` or `check()`.

The `get_weight()` method returns the weight that you want to contribute to the current document. This weight must always be >= 0:

```
virtual Xapian::weight get_weight() const;
```

The default implementation of `get_weight()` returns 0, for convenience when deriving "weight-less" subclasses.

You also need to specify an upper bound on the value which `get_weight()` can return, which is used by the matcher to perform various optimisations. You should try hard to find a bound for efficiency, but if there really isn't one then you can set `DBL_MAX`:

```
void get_maxweight(Xapian::weight max_weight);
```

This method specifies an upper bound on what `get_weight()` will return *from now on* (until the next call to `init()`). So if you know that the upper bound has decreased, you should call `set_maxweight()` with the new reduced bound.

One thing to be aware of is that currently calling `set_maxweight()` during the match triggers an recursion through the postlist tree to recalculate the new overall maxweight, which takes a comparable amount of time to calculating the weight for a matching document. If your maxweight reduces for nearly every document, you may want to profile to see if it's beneficial to notify every single change. Experiments with a modified `FixedWeightPostingSource` which forces a pointless recalculation for every document suggest a worst case overhead in search times of about 37%, but reports of profiling results for real world examples are most welcome. In real cases, this overhead could easily be offset by the extra scope for matcher optimisations which a tighter maxweight bound allows.

A simple approach to reducing the number of calculations is only to do it every N documents. If it's cheap to calculate the maxweight in your posting source, a more sophisticated strategy might be to decide an absolute maximum number of times to update the maxweight (say 100) and then to call it whenever:

```
last_notified_maxweight - new_maxweight >= original_maxweight / 100.0
```

This ensures that only reasonably significant drops result in a recalculation of the maxweight.

Since `get_weight()` must always return >= 0, the upper bound must clearly also always be >= 0 too. If you don't call `get_maxweight()` then the bound defaults to 0, to match the default implementation of `get_weight()`.

If you want to read the currently set upper bound, you can call:

```
Xapian::weight get_maxweight() const;
```

This is just a getter method for a member variable in the `Xapian::PostingSource` class, and is inlined from the API headers, so there's no point storing this yourself in your subclass - it should be just as efficient to call `get_maxweight()` whenever you want to use it.

The `at_end()` method checks if the current iteration position is past the last entry:

```
virtual bool at_end() const = 0;
```

The `get_docid()` method returns the document id at the current iteration position:

```
virtual Xapian::docid get_docid() const = 0;
```

There are three methods which advance the current position. All of these take a Xapian::Weight parameter `min_wt`, which indicates the minimum weight contribution which the matcher is interested in. The matcher still checks the weight of documents so it's OK to ignore this parameter completely, or to use it to discard only some documents. But it can be useful for optimising in some cases.

The simplest of these three methods is `next()`, which simply advances the iteration position to the next document (possibly skipping documents with weight contribution < min_wt):

```
virtual void next(Xapian::weight min_wt) = 0;
```

Then there's `skip_to()`. This advances the iteration position to the next document with document id >= that specified (possibly also skipping documents with weight contribution < min_wt):

```
virtual void skip_to(Xapian::docid did, Xapian::weight min_wt);
```

A default implementation of `skip_to()` is provided which just calls `next()` repeatedly. This works but `skip_to()` can often be implemented much more efficiently.

The final method of this group is `check()`. In some cases, it's fairly cheap to check if a given document matches, but the requirement that `skip_to()` must leave the iteration position on the next document is rather costly to implement (for example, it might require linear scanning of document ids). To avoid this where possible, the `check()` method allows the matcher to just check if a given document matches:

```
virtual bool check(Xapian::docid did, Xapian::weight min_wt);
```

The return value is `true` if the method leaves the iteration position valid, and `false` if it doesn't. In the latter case, `next()` will advance to the first matching position after document id `did`, and `skip_to()` will act as it would if the iteration position was the first matching position after `did`.

The default implementation of `check()` is just a thin wrapper around `skip_to()` which returns true - you should use this if `skip_to()` incurs only a small extra cost.

There's also a method to return a string describing this object:

```
virtual std::string get_description() const;
```

The default implementation returns a generic answer. This default is provided to avoid forcing you to provide an implementation if you don't really care what `get_description()` gives for your sub-class.

5.1.3 Examples

Here is an example of a Python PostingSource which contributes additional weight from some external source (note that in Python, you call `next()` on an iterator to get each item, including the first, which is exactly the semantics we need to implement here):

```
class ExternalWeightPostingSource(xapian.PostingSource):
    """
    A Xapian posting source returning weights from an external source.
    """
    def __init__(self, db, wtsource):
        xapian.PostingSource.__init__(self)
        self.db = db
        self.wtsource = wtsource
    def init(self, db):
        self.alldocs = db.postlist('')

    def get_termfreq_min(self): return 0
    def get_termfreq_est(self): return self.db.get_doccount()
    def get_termfreq_max(self): return self.db.get_doccount()

    def next(self, minweight):
        try:
            self.current = self.alldocs.next()
        except StopIteration:
            self.current = None

    def skip_to(self, docid, minweight):
        try:
            self.current = self.alldocs.skip_to(docid)
        except StopIteration:
            self.current = None

    def at_end(self):
        return self.current is None

    def get_docid(self):
        return self.current.docid

    def get_maxweight(self):
        return self.wtsource.get_maxweight()

    def get_weight(self):
        doc = self.db.get_document(self.current.docid)
        return self.wtsource.get_weight(doc)
```

ExternalWeightPostingSource doesn't restrict which documents match - it's intended to be combined with an existing query using *OP_AND_MAYBE* like so:

```
extwtps = xapian.ExternalWeightPostingSource(db, wtsource)
query = xapian.Query(query.OP_AND_MAYBE, query, xapian.Query(extwtps))
```

The wtsource would be a class like this one:

```
class WeightSource(object):
    def get_maxweight(self):
        return 12.34;

    def get_weight(self, doc):
        return some_func(doc.get_docid())
```

5.1.4 Multiple databases, and remote databases

In order to work with searches across multiple databases, or in remote databases, some additional methods need to be implemented in your Xapian::PostingSource subclass. The first of these is `clone()`, which is used for multi database searches. This method should just return a newly allocated instance of the same posting source class, initialised in the same way as the source that clone() was called on. The returned source will be deallocated by the caller (using "delete" - so you should allocate it with "new").

If you don't care about supporting searches across multiple databases, you can simply return NULL from this method. In fact, the default implementation does this, so you can just leave the default implementation in place. If `clone()`

returns NULL, an attempt to perform a search with multiple databases will raise an exception:

```
virtual PostingSource * clone() const;
```

To work with searches across remote databases, you need to implement a few more methods. Firstly, you need to implement the `name()` method. This simply returns the name of your posting source (fully qualified with any namespace):

```
virtual std::string name() const;
```

Next, you need to implement the serialise and unserialise methods. The `serialise()` method converts all the settings of the PostingSource to a string, and the `unserialise()` method converts one of these strings back into a PostingSource. Note that the serialised string doesn't need to include any information about the current iteration position of the PostingSource:

```
virtual std::string serialise() const;
virtual PostingSource * unserialise(const std::string &s) const;
```

Finally, you need to make a remote server which knows about your PostingSource. Currently, the only way to do this is to modify the source slightly, and compile your own xapian-tcpsrv. To do this, you need to edit `xapian-core/bin/xapian-tcpsrv.cc` and find the `register_user_weighting_schemes()` function. If `MyPostingSource` is your posting source, at the end of this function, add these lines:

```
Xapian::Registry registry;
registry.register_postingsource(MyPostingSource());
server.set_registry(registry);
```

5.2 Tuning the Unigram Language Model: LMWeight

Table of Contents

Unigram language modeling weighing scheme ranks document based on ability to generate query from document language model. Unigram language model is intutive for user as they can think of term possible in document and add them to query which will increase performace of weighing scheme in this setting.

5.2.1 Clamping Negative value

Since unigram language model differs from xapian way of weighing scheme as xapian only support sum of various individual parts. Unigram language model have accomodated product of probabilities by summing log of individual parts. Due to introduction of log in probabilities a clamping factor to clamp negative value of log to positive is also introduced.

Default value for clamping parameter is document length upper bound. whereas API user can adjust this value using param_log parameter of LMWeight constructor.

5.2.2 Smoothing

Unigram Language model foundation is document language model but due to length of document document language model are usually sparse and affect the weight calculation for the documents hence smoothing with collection frequency and document length is done. Xapian Implements following Smoothing techniques:-

Dirichlet Prior Smoothing:

Smoothing based on document size, because longer document require less smoothing as they more accurately estimate language model. Dirichlet Prior Smoothing is better at Estimation Role.

DP Smoothing technique is better for title or smaller queries as it is better in estimation role.

Optimal Smoothing parameter

param_smoothing1 - Small,Long Query - 2000

Jelinek Mercer Smoothing:

Combine relative frequency of query term with relative frequency in collection. Address small sample problem and explain unobserved words in document. JM Smoothing is better at explaing common and noisy words in query. JM smoothing outperforms other smoothing scheme in Query Modelling.

This smoothing work better in case of noisy and long query as it DP smoothing is better in Query Modeling.

Optimal Smoothing parameter

param_smoothing1 - Parameter range (0-1)

Small Query - 0.1 {Conjuctive interpolation of Query Term} Longer Query - 0.7 {Disjunctive interpolation of Query Term}

Absolute Discounting Smoothing:

Absolute Discounting Smoothing is larger for flatter distribution of words. More Smoothing for documents with relatively large count of unique terms.

Optimal Smoothing parameter

param_smoothing1 - Parameter range (0-1){Small,Long query - 0.7}

Two Stage Smoothing(Default):

Two Stage smoothing is combination of Dirichlet Prior Smoothing and Jelinek Mercer Smoothing. Two Stage smoothing is application of Jelinek-Mercer followed by Dirichlet Prior smoothing. Jelinek-Mercer will first model the query and followed by Dirichlet Prior will account for missing and unseen terms.

Optimal Smoothing parameter

param_smoothing1

Parameter range (0-1) Small Query - 0.1 {Conjuctive interpolation of Query Term} Longer Query - 0.7 {Disjunctive interpolation of Query Term}

param_smoothing2

Small,Long Query - 2000

5.2.3 Constructor Parameters

User can select parameters to clamp negetive value and select smoothing scheme using. Xapian manages a enum for selection of smoothing technique:Following values need to be assigned to select_smoothing parameter to select smoothing type:

Jelinek Mercer Smoothing - JELINEK_MERCER_SMOOTHING

Dirichlet Prior Smoothing - DIRICHLET_SMOOTHING

Absolute Discounting Smoothing - ABSOLUTE_DISCOUNT_SMOOTHING

Two Stage Smoothing - TWO_STAGE_SMOOTHING

Following are Constructor provided by UnigramLM Weighting class. User can select constructor based on there requirement and number of paramter they want to provide. Refer generated documentation for constructor.

5.2.4 Selecting Weighting scheme:

Add following line in your code to select Unigram Language Model Weighting scheme:

```
enquire.set_weighting_scheme(Xapian::LMWeight(700.0,Xapian::Weight::JELINEK_MERCER_SMOOTHING,0.4,200
```

5.3 Xapian Administrator's Guide

Table of contents

5.3.1 Introduction

This document is intended to provide general hints, tips and advice to administrators of Xapian systems. It assumes that you have installed Xapian on your system, and are familiar with the basics of creating and searching Xapian databases.

The intended audience is system administrators who need to be able to perform general management of a Xapian database, including tasks such as taking backups and optimising performance. It may also be useful introductory reading for Xapian application developers.

The document is up-to-date for Xapian version 1.2.5.

5.3.2 Databases

Xapian databases hold all the information needed to perform searches in a set of tables. The following tables always exist:

- A posting list table, which holds a list of all the documents indexed by each term in the database.
- A record table, which holds the document data associated with each document in the database.
- A termlist table, which holds a list of all the terms which index each document.

And the following optional tables exist only when there is data to store in them (in 1.0.1 and earlier, the position and value tables were always created even if empty; spelling and synonym tables were new in 1.0.2):

- A position list table, which holds a list of all the word positions in each document which each term occurs at.
- A value table, which holds the "values" (used for sorting, collapsing, and other match-time calculations) associated with each document in the database (only for flint - newer backends store values in the postlist and termlist tables).
- A spelling table, which holds data for suggesting spelling corrections.
- A synonym table, which holds a synonym dictionary.

Each of the tables is held in a separate file, allowing an administrator to see how much data is being used for each of the above purposes. It is not always necessary to fully populate these tables: for example, if phrase searches are never going to be performed on the database, it is not necessary to store any positionlist information.

If you look at a Xapian database, you will see that each of these tables actually uses 2 or 3 files. For example, for a "chert" format database the termlist table is stored in the files "termlist.baseA", "termlist.baseB" and "termlist.DB".

The ".DB" file actually stores the data, and is structured as a tree of blocks, which have a default size of 8KB (though this can be set, either through the Xapian API, or with some of the tools detailed later in this document).

The ".baseA" and ".baseB" files are used to keep track of where to start looking for data in the ".DB" file (the root of the tree), and which blocks are in use. Often only one of the ".baseA" and ".baseB" files will be present; each of these files refers to a revision of the database, and there may be more than one valid revision of the database stored in the ".DB" file at once.

Changing the blocksize may have performance implications, but it is hard to tell whether these will be positive or negative for a particular combination of hardware and software without doing some profiling.

Atomic modifications

Xapian ensures that all modifications to its database are performed atomically. This means that:

- From the point of view of a separate process (or a separate database object in the same process) reading the database, all modifications made to a database are invisible until the modifications is committed.

- The database on disk is always in a consistent state.
- If the system is interrupted during a modification, the database should always be left in a valid state. This applies even if the power is cut unexpectedly, as long as the disk does not become corrupted due to hardware failure.

Committing a modification requires several calls to the operating system to make it flush any cached modifications to the database to disk. This is to ensure that if the system fails at any point, the database is left in a consistent state. Of course, this is a fairly slow process (since the system has to wait for the disk to physically write the data), so grouping many changes together will speed up the throughput considerably.

Many modifications can be explicitly grouped into a single transaction, so that lots of changes are applied at once. Even if an application doesn't explicitly protect modifications to the database using transactions, Xapian will group modifications into transactions, applying the modifications in batches.

Note that it is not currently possible to extend Xapian's transactions to cover multiple databases, or to link them with transactions in external systems, such as an RDBMS.

Finally, note that it is possible to compile Xapian such that it doesn't make modifications in an atomic manner, in order to build very large databases more quickly (search the Xapian mailing list archives for "DANGEROUS" mode for more details). This isn't yet integrated into standard builds of Xapian, but may be in future, if appropriate protections can be incorporated.

Single writer, multiple reader

Xapian implements a "single writer, multiple reader" model. This means that, at any given instant, there is only permitted to be a single object modifying a database, but there may (simultaneously) be many objects reading the database at once.

Xapian enforces this restriction using by having a writer lock the database. Each Xapian database directory contains a lock file named `flintlock` (we've kept the same name as flint used, since the locking technique is the same).

This lock-file will always exist, but will be locked using `fcntl()` when the database is open for writing. Because of the semantics of `fcntl()` locking, for each WritableDatabase opened we spawn a child process to hold the lock, which then exec-s `cat`, so you will see a `cat` subprocess of any writer process in the output of `ps`, `top`, etc.

If a writer exits without being given a chance to clean up (for example, if the application holding the writer is killed), the `fcntl()` lock will be automatically released by the operating system. Under Microsoft Windows, we use a different locking technique which doesn't require a child process, but also means the lock is released automatically when the writing process exits.

Revision numbers

Xapian databases contain a revision number. This is essentially a count of the number of modifications since the database was created, and is needed to implement the atomic modification functionality. It is stored as a 32 bit integer, so there is a chance that a very frequently updated database could cause this to overflow. The consequence of such an overflow would be to throw an exception reporting that the database has run out of revision numbers.

This isn't likely to be a practical problem, since it would take nearly a year for a database to reach this limit if 100 modifications were committed every second, and no normal Xapian system will commit more than once every few seconds. However, if you are concerned, you can use the `xapian-compact` tool to make a fresh copy of the database with the revision number set to 1.

The revision number of each table can be displayed by the `xapian-check` tool.

Network file systems

Xapian should work correctly over a network file system. However, there are various potential issues with such file systems, so we recommend extensive testing of your particular network file system before deployment.

Be warned that Xapian is heavily I/O dependent, and therefore performance over a network file system is likely to be slow unless you've got a very well tuned setup.

Xapian needs to be able to create a lock file in a database directory when modifications are being performed. On some network files systems (e.g., NFS) this requires a lock daemon to be running.

Which database format to use?

As of release 1.2.0, you should generally use the chert format (which is now the default).

Support for the pre-1.0 quartz format (deprecated in 1.0) was removed in 1.1.0. See below for how to convert a quartz database to a flint one.

The flint backend (the default for 1.0) is still supported by 1.2.x, but deprecated - only use it if you already have flint databases; and plan to migrate away.

There's also a development backend called brass. The main distinguishing feature of this is that the format may change incompatibly from time to time. It passes Xapian's extensive testsuite, but has seen less real world use than chert.

Can I put other files in the database directory?

If you wish to store meta-data or other information relating to the Xapian database, it is reasonable to wish to put this in files inside the Xapian database directory, for neatness. For example, you might wish to store a list of the prefixes you've applied to terms for specific fields in the database.

Current Xapian backends don't do anything which will break this technique, so as long as you don't choose a filename that Xapian uses itself, there should be no problems. However, be aware that new versions of Xapian may use new files in the database directory, and it is also possible that new backend formats may not be compatible with the technique (e.g., it is possible that a future backend could store its entire database in a single file, not in a directory).

5.3.3 Backup Strategies

Summary

- The simplest way to perform a backup is to temporarily halt modifications, take a copy of all files in the database directory, and then allow modifications to resume. Read access can continue while a backup is being taken.
- If you have a filesystem which allows atomic snapshots to be taken of directories (such as an LVM filesystem), an alternative strategy is to take a snapshot and simply copy all the files in the database directory to the backup medium. Such a copy will always be a valid database.
- Progressive backups are not easily possible; modifications are typically spread throughout the database files.

Detail

Even though Xapian databases are often automatically generated from source data which is stored in a reliable manner, it is usually desirable to keep backups of Xapian databases being run in production environments. This is particularly important in systems with high-availability requirements, since re-building a Xapian database from scratch can take many hours. It is also important in the case where the data stored in the database cannot easily be recovered from external sources.

Xapian databases are managed such that at any instant in time, there is at least one valid revision of the database written to disk (and if there are multiple valid revisions, Xapian will always open the most recent). Therefore, if it is possible to take an instantaneous snapshot of all the database files (for example, on an LVM filesystem), this snapshot is suitable for copying to a backup medium. Note that it is not sufficient to take a snapshot of each database file in turn - the snapshot must be across all database files simultaneously. Otherwise, there is a risk that the snapshot could contain database files from different revisions.

If it is not possible to take an instantaneous snapshot, the best backup strategy is simply to ensure that no modifications are committed during the backup procedure. While the simplest way to implement this may be to stop whatever processes are used to modify the database, and ensure that they close the database, it is not actually necessary to ensure that no writers are open on the database; it is enough to ensure that no writer makes any modification to the database.

Because a Xapian database can contain more than one valid revision of the database, it is actually possible to allow a limited number of modifications to be performed while a backup copy is being made, but this is tricky and we do not recommend relying on it. Future versions of Xapian are likely to support this better, by allowing the current revision of a database to be preserved while modifications continue.

Progressive backups are not recommended for Xapian databases: Xapian database files are block-structured, and modifications are spread throughout the /database file. Therefore, a progressive backup tool will not be able to take a backup by storing only the new parts of the database. Modifications will normally be so extensive that most parts of the database have been modified, however, if only a small number of modifications have been made, a binary diff algorithm might make a usable progressive backup tool.

5.3.4 Inspecting a database

When designing an indexing strategy, it is often useful to be able to check the contents of the database. Xapian includes a simple command-line program, “delve”, to allow this.

For example, to display the list of terms in document “1” of the database “foo”, use:

```
delve foo -r 1
```

It is also possible to perform simple searches of a database. Xapian includes another simple command-line program, “quest”, to support this. “quest” is only able to search for un-prefixed terms, the query string must be quoted to protect it from the shell. To search the database “foo” for the phrase “hello world”, use:

```
quest -d foo '"hello world"'
```

If you have installed the “Omega” CGI application built on Xapian, this can also be used with the built-in “godmode” template to provide a web-based interface for browsing a database. See Omega’s documentation for more details on this.

5.3.5 Database maintenance

Compacting a database

Xapian databases normally have some spare space in each block to allow new information to be efficiently slotted into the database. However, the smaller a database is, the faster it can be searched, so if there aren’t expected to be many further modifications, it can be desirable to compact the database.

Xapian includes a tool called “xapian-compact” for compacting databases. This tool makes a copy of a database, and takes advantage of the sorted nature of the source Xapian database to write the database out without leaving spare space for future modifications. This can result in a large space saving.

The downside of compaction is that future modifications may take a little longer, due to needing to reorganise the database to make space for them. However, modifications are still possible, and if many modifications are made, the database will gradually develop spare space.

There's an option ("-F") to perform a "fuller" compaction. This option compacts the database as much as possible, but it violates the design of the Btree format slightly to achieve this, so it is not recommended if further modifications are at all likely in future. If you do need to modify a "fuller" compacted database, we recommend you run xapian-compact on it without "-F" first.

While taking a copy of the database, it is also possible to change the blocksize. If you wish to profile search speed with different blocksizes, this is the recommended way to generate the different databases (but remember to compact the original database as well, for a fair comparison).

Merging databases

When building an index for a very large amount of data, it can be desirable to index the data in smaller chunks (perhaps on separate machines), and then merge the chunks together into a single database. This can be performed using the "xapian-compact" tool, simply by supplying several source database paths.

Normally, merging works by reading the source databases in parallel, and writing the contents in sorted order to the destination database. This will work most efficiently if excessive disk seeking can be avoided; if you have several disks, it may be worth placing the source databases and the destination database on separate disks to obtain maximum speed.

The `xapian-compact` tool supports an additional option, `--multipass`, which is useful when merging more than three databases. This will cause the postlist tables to be grouped and merged into temporary tables, which are then grouped and merged, and so on until a single postlist table is created, which is usually faster, but requires more disk space for the temporary files.

Checking database integrity

Xapian includes a command-line tool to check that a database is self-consistent. This tool, "xapian-check", runs through the entire database, checking that all the internal nodes are correctly connected. It can also be used on a single table, for example, this command will check the termlist table of database "foo":

```
xapian-check foo/termlist.DB
```

Converting a flint database to a chert database

It is possible to convert a flint database to a chert database by using the "copydatabase" example program included with Xapian. This is a lot slower to run than "xapian-compact", since it has to perform the sorting of the term occurrence data from scratch, but should be faster than a re-index from source data since it doesn't need to perform the tokenisation step. It is also useful if you no longer have the source data available.

The following command will copy a database from "SOURCE" to "DESTINATION", creating the new database at "DESTINATION" as a chert database:

```
copydatabase SOURCE DESTINATION
```

By default copydatabase will renumber your documents starting with docid 1. If the docids are stored in or come from some external system, you should preserve them by using the –no-renumber option (new in Xapian 1.2.5):

```
copydatabase --no-renumber SOURCE DESTINATION
```

Converting a quartz database to a flint database

It is possible to convert a quartz database to a flint database by installing Xapian 1.0.x (since this has support for both quartz and flint) and using the "copydatabase" example program included with Xapian. This is a lot slower to run than "xapian-compact", since it has to perform the sorting of the term occurrence data from scratch, but should be faster than a re-index from source data since it doesn't need to perform the tokenisation step. It is also useful if you no longer have the source data available.

The following command will copy a database from "SOURCE" to "DESTINATION", creating the new database at "DESTINATION" as a flint database:

```
copydatabase SOURCE DESTINATION
```

Converting a 0.9.x flint database to work with 1.0.y

In 0.9.x, flint was the development backend.

Due to a bug in the flint position list encoding in 0.9.x which made flint databases non-portable between platforms, we had to make an incompatible change in the flint format. It's not easy to write an upgrader, but you can convert a database using the following procedure (although it might be better to rebuild from scratch if you want to use the new UTF-8 support in Xapian::QueryParser, Xapian::Stem, and Xapian::TermGenerator).

Run the following command in your Xapian 0.9.x installation to copy your 0.9.x flint database "SOURCE" to a new quartz database "INTERMEDIATE":

```
copydatabase SOURCE INTERMEDIATE
```

Then run the following command in your Xapian 1.0.y installation to copy your quartz database to a 1.0.y flint database "DESTINATION":

```
copydatabase INTERMEDIATE DESTINATION
```

5.4 Scalability

Table of contents

5.4.1 Introduction

People often want to know how Xapian will scale. The short answer is "very well" - an early version of the software powered the (now defunct) Webtop search engine, which offered a search over around 500 million web pages (around 1.5 terabytes of database files). Searches took less than a second.

In terms of current deployments, gmane indexes and searches nearly 100 million mail messages on a single server at the time of writing (2012), and we've had user reports of systems with more than 250 million documents.

If you've questions about scalability not covered in this document, ask on the mailing lists - people using Xapian to search large databases may be able to make further suggestions.

5.4.2 Benchmarking

One effect to be aware of when designing benchmarks is that queries will be a lot slower when nothing is cached. So the first few queries on a database which hasn't been searched recently will be unrepresentatively slow compared to the typical case.

In real use, pretty much all the non-leaf blocks from the B-trees being used for the search will be cached pretty quickly, as well as many commonly used leaf blocks.

5.4.3 General Scalability Considerations

In a large search application, I/O will end up being the limiting factor. So you want a RAID setup optimised for fast reading, lots of RAM in the box so the OS can cache lots of disk blocks (the access patterns typically mean that you only need to cache a few percent of the database to eliminate most disk cache misses).

It also means that reducing the database size is usually a win. The Chert backend compresses the information in the tables in ways which work well given the nature of the data but aren't too expensive to unpack (e.g. lists of sorted docids are stored as differences with smaller values encoded in fewer bytes). There is further potential for improving the encodings used.

Another way to reduce disk I/O is to run databases through xapian-compact. The Btree manager usually leaves some spare space in each block so that updates are more efficient (though there are heuristics which will fill blocks fuller when they detect a long sequence of sequential insertions, which means adding documents to the end of an empty database will produce fairly compact tables, apart from the postlist table). Compacting makes all blocks as full as possible, and so reduces the size of the database. It also produces a database with revision 1 which is inherently faster to search. The penalty is that updates will be slow for a while, as they'll result in a lot of block splitting when all blocks are full.

Splitting the data over several databases is generally a good strategy. Once each has finished being updated, compact it to make it small and faster to search.

A multiple-database scheme works particularly well if you want a rolling web index where the contents of the oldest database can be rechecked and live links put back into a new database which, once built, replaces the oldest database. It's also good for a news-type application where older documents should expire from the index.

5.4.4 Size Limits in Xapian

The chert backend (which is currently the default and recommended backend) stores the indexes in several files containing Btree tables. If you're indexing with positional information (for phrase searching) the term positions table is usually the largest.

You should also consider *Index limitations* and *Search-time Limitations*.

5.5 Replication

It is often desirable to maintain multiple copies of a Xapian database, having a "master" database which modifications are made on, and a set of secondary (read-only, "slave") databases which these modifications propagate to. For example, to support a high query load there may be many search servers, each with a local copy of the database, and a single indexing server. In order to allow scaling to a large number of search servers, with large databases and frequent updates, we need an database replication implementation to have the following characteristics:

- Data transfer is (at most) proportional to the size of the updates, rather than the size of the database, to allow frequent small updates to large databases to be replicated efficiently.

- Searching (on the slave databases) and indexing (on the master database) can continue during synchronisation.
- Data cached (in memory) on the slave databases is not discarded (unless it's actually out of date) as updates arrive, to ensure that searches continue to be performed quickly during and after updates.
- Synchronising each slave database involves low overhead (both IO and CPU) on the server holding the master database, so that many slaves can be updated from a single master without overloading it.
- Database synchronisation can be recovered after network outages or server failures without manual intervention and without excessive data transfer.

The database replication protocol is intended to support replicating a single writable database to multiple (read-only) search servers, while satisfying all of the above properties. It is not intended to support replication of multiple writable databases - there must always be a single master database to which all modifications are made.

5.5.1 Backend Support

Replication is supported by the chert, and brass database backends, and can cleanly handle the master switching database type (a full copy is sent in this situation). It doesn't make a lot of sense to support replication for the remote backend. Replication of inmemory databases isn't currently available.

5.5.2 Setting up replicated databases

Todo

expand this section.

To replicate a database efficiently from one master machine to other machines, there is one configuration step to be performed on the master machine, and two servers to run.

Todo

XAPIAN_MAX_CHANGESETS is not irrelevant in brass. We should talk about that here but only if we are clear what version it was added in.

Firstly, on the master machine, the indexer must be run with the environment variable *XAPIAN_MAX_CHANGESETS* set to a non-zero value. (Currently, the actual value it is set to is irrelevant, but I suggest using a value of 10). This will cause changeset files to be created whenever a transaction is performed, which allow the transaction to be replayed efficiently on a replica of the database.

Secondly, also on the master machine, run the *xapian-replicate-server* server to serve the databases which are to be replicated. This takes various parameters to control the directory that databases are found in, and the network interface to serve on. The *–help* option will cause usage information to be displayed. For example, if */var/search/dbs‘* contains a set of Xapian databases to be replicated:

```
xapian-replicate-server /var/search/dbs -p 7010
```

would run a server allowing access to these databases, on port 7010.

Finally, on the client machine, run the *xapian-replicate* server to keep an individual database up-to-date. This will contact the server on the specified host and port, and copy the database with the name (on the master) specified in the *-m* option to the client. One non-option argument is required - this is the name that the database should be stored in on the slave machine. For example, contacting the above server from the same machine:

```
xapian-replicate -h 127.0.0.1 -p 7010 -m foo foo2
```

would produce a database "foo2" containing a replica of the database "/var/search/dbs/foo". Note that the first time you run this, this command will create the foo2 directory and populate it with appropriate files; you should not create this directory yourself.

As of 1.2.5, if you don't specify the master name, the same name is used remotely and locally, so this will replicate remote database "foo2" to local database "foo2":

```
xapian-replicate -h 127.0.0.1 -p 7010 foo2
```

Both the server and client can be run in "one-shot" mode, by passing *-o*. This may be particularly useful for the client, to allow a shell script to be used to cycle through a set of databases, updating each in turn (and then probably sleeping for a period).

5.5.3 Limitations

It is possible to confuse the replication system in some cases, such that an invalid database will be produced on the client. However, this is easy to avoid in practice.

To confuse the replication system, the following needs to happen:

- Start with two databases, A and B.
- Start a replication of database A.
- While the replication is in progress, swap B in place of A (ie, by moving the files around, such that B is now at the path of A).
- While the replication is still in progress, swap A back in place of B.

If this happens, the replication process will not detect the change in database, and you are likely to end up with a database on the client which contains parts of A and B mixed together. You will need to delete the damaged database on the client, and re-run the replication.

To avoid this, simply avoid swapping a database back in place of another one. Or at least, if you must do this, wait until any replications in progress when you were using the original database have finished.

Calling reopen

`xapian.Database.reopen()` is usually an efficient way to ensure that a database is up-to-date with the latest changes. Unfortunately, it does not currently work as you might expect with databases which are being updated by the replication client. The workaround is simple; don't use the reopen() method on such databases: instead, you should close the database and open it again from scratch.

Briefly, the issue is that the databases created by the replication client are created in a subdirectory of the target path supplied to the client, rather than at that path. A "stub database" file is then created in that directory, pointing to the database. This allows the database which readers open to be switched atomically after a database copy has occurred. The reopen() method doesn't re-read the stub database file in this situation, so ends up attempting to read the old database which has been deleted.

We intend to fix this issue in the Brass backend (currently under development by eliminating this hidden use of a stub database file).

5.5.4 Alternative approaches

Without using the database replication protocol, there are various ways in which the "single master, multiple slaves" setup could be implemented.

- Copy database from master to all slaves after each update, then swap the new database for the old.

- Synchronise databases from the master to the slaves using rsync.
- Keep copy of database on master from before each update, and use a binary diff algorithm (e.g., xdelta) to calculate the changes, and then apply these same changes to the databases on each slave.
- Serve database from master to slaves over NFS (or other remote file system).
- Use the “remote database backend” facility of Xapian to allow slave servers to search the database directly on the master.

All of these could be made to work but have various drawbacks, and fail to satisfy all the desired characteristics. Let’s examine them in detail:

Copying database after each update

Databases could be pushed to the slaves after each update simply by copying the entire database from the master (using scp, ftp, http or one of the many other transfer options). After the copy is completed, the new database would be made live by indirecting access through a stub database and switching what it points to.

After a sufficient interval to allow searches in progress on the old database to complete, the old database would be removed. (On UNIX filesystems, the old database could be unlinked immediately, and the resources used by it would be automatically freed as soon as the current searches using it complete.)

This approach has the advantage of simplicity, and also ensures that the databases can be correctly re-synchronised after network outages or hardware failure.

However, this approach would involve copying a large amount of data for each update, however small the update was. Also, because the search server would have to switch to access new files each time an update was pushed, the search server will be likely to experience poor performance due to commonly accessed pages falling out of the disk cache during the update. In particular, although some of the newly pushed data would be likely to be in the cache immediately after the update, if the combination of the old and new database sizes exceeds the size of the memory available on the search servers for caching, either some of the live database will be dropped from the cache resulting in poor performance during the update, or some of the new database will not initially be present in the cache after update.

Synchronise database using rsync

Rsync works by calculating hashes for the content on the client and the server, sending the hashes from the client to the server, and then calculating (on the server) which pieces of the file need to be sent to update the client. This results in a fairly low amount of network traffic, but puts a fairly high CPU load on the server. This would result in a large load being placed on the master server if a large number of slaves tried to synchronise with it.

Also, rsync will not reliably update the database in a manner which allows the database on a slave to be searched while being updated - therefore, a copy or snapshot of the database would need to be taken first to allow searches to continue (accessing the copy) while the database is being synchronised.

If a copy is used, the caching problems discussed in the previous section would apply again. If a snapshotting filesystem is used, it may be possible to take a read-only snapshot copy cheaply (and without encountering poor caching behaviour), but filesystems with support for this are not always available, and may require considerable effort to set up even if they are available.

Use a binary diff algorithm

If a copy of the database on the master before the update was kept, a binary diff algorithm (such as “xdelta”) could be used to compare the old and new versions of the database. This would produce a patch file which could be transferred to the slaves, and then applied - avoiding the need for specific calculations to be performed for each slave.

However, this requires a copy or snapshot to be taken on the master - which has the same problems as previously discussed. A copy or snapshot would also need to be taken on the slave, since a patch from xdelta couldn't safely be applied to a live database.

Serve database from master to slaves over NFS

NFS allows a section of a filesystem to be exported to a remote host. Xapian is quite capable of searching a database which is exported in such a manner, and thus NFS can be used to quickly and easily share a database from the master to multiple slaves.

A reasonable setup might be to use a powerful machine with a fast disk as the master, and use that same machine as an NFS server. Then, multiple slaves can connect to that NFS server for searching the database. This setup is quite convenient, because it separates the indexing workload from the search workload to a reasonable extent, but may lead to performance problems.

There are two main problems which are likely to be encountered. Firstly, in order to work efficiently, NFS clients (or the OS filesystem layer above NFS) cache information read from the remote file system in memory. If there is insufficient memory available to cache the whole database in memory, searches will occasionally need to access parts of the database which are held only on the master server. Such searches will take a long time to complete, because the round-trip time for an access to a disk block on the master is typically a lot slower than the round-trip time for access to a local disk. Additionally, if the local network experiences problems, or the master server fails (or gets overloaded due to all the search requests), the searches will be unable to be completed.

Also, when a file is modified, the NFS protocol has no way of indicating that only a small set of blocks in the file have been modified. The caching is all implemented by NFS clients, which can do little other than check the file modification time periodically, and invalidate all cached blocks for the file if the modification time has changed. For the Linux client, the time between checks can be configured by setting the acregmin and acregmax mount options, but whatever these are set to, the whole file will be dropped from the cache when any modification is found.

This means that, after every update to the database on the master, searches on the slaves will have to fetch all the blocks required for their search across the network, which will likely result in extremely slow search times until the cache on the slaves gets populated properly again.

Use the "remote database backend" facility

Xapian has supported a "remote" database backend since the very early days of the project. This allows a search to be run against a database on a remote machine, which may seem to be exactly what we want. However, the "remote" database backend works by performing most of the work for a search on the remote end - in the situation we're concerned with, this would mean that most of the work was performed on the master, while slaves remain largely idle.

The "remote" database backend is intended to allow a large database to be split, at the document level, between multiple hosts. This allows systems to be built which search a very large database with some degree of parallelism (and thus provide faster individual searches than a system searching a single database locally). In contrast, the database replication protocol is intended to allow a database to be copied to multiple machines to support a high concurrent search load (and thus to allow a higher throughput of searches).

In some cases (i.e., a very large database and a high concurrent search load) it may be perfectly reasonable to use both the database replication protocol in conjunction with the "remote" database backend to get both of these advantages - the two systems solve different problems.

5.6 Working with remote databases

Table of contents

Todo

This needs writing from scratch; the previous document wasn't very good and is license entangled.

5.7 Serialisation of Queries and Documents

Table of contents

5.7.1 Introduction

In order to pass Query and Document objects to or from remote databases, Xapian includes support for serialising these objects to strings, and then converting these strings back into objects. This support may be accessed directly, and used for storing persistent representations of Query and Document objects.

Be aware that the serialised representation may occasionally change between releases. This will be clearly noted in the release notes.

5.7.2 Serialising Documents

To get a serialised document, simply call the `Document::serialise()` method on the instance of the document:

```
std::string serialise() const;
```

Documents are often lazily fetched from databases: this method will first force the full document contents to be fetched from the database, in order to serialise them. The serialised document will have identical contents (data, terms, positions, values) to the original document.

To get a document from a serialised form, call the static `Document::unserialise()` method, passing it the string returned from `serialise()`:

```
static Document unserialise(const std::string &s);
```

5.7.3 Serialising Queries

Serialisation of queries is very similar to serialisation of documents: there is a `Query::serialise()` method to produce a serialised Query, and a corresponding `Query::unserialise()` method to produce a Query from a serialised representation:

```
std::string serialise() const;
static Query unserialise(const std::string &s);
```

However, there is a wrinkle. Queries can contain arbitrary user-defined PostingSource subqueries. In order to serialise and unserialise such queries, all the PostingSource subclasses used in the query must implement the `name()`, `serialise()` and `unserialise()` methods (see *Posting sources* for details). In addition, a special form of unserialise must be used:

```
static Query unserialise(const std::string & s, const Registry & registry);
```

The `Registry` passed to this method must know about all the custom posting sources used in the query. You can tell a Registry about a custom posting source using the `Registry::register_posting_source` method:

```
void register_posting_source(const Xapian::PostingSource &source);
```

Note that Registry objects always know about built-in posting sources (such as `ValueWeightPostingSource`), so you don't need to call `register_posting_source()` for them.

5.8 Geospatial searches

Xapian's geospatial support is currently still a work in progress, and until it is generally available this documentation will either be empty or may be out of date.

Todo

Write about how to index geolocation information, and how to use the geo posting sources and keymaker to sort by distance, bias results by distance, and limit results by distance. Discuss storing geo bounding boxes terms for accelerating distance limited searches.

5.9 Query authorisation

Say you are building a system that allows people to write private diary entries, and only share them with specific people. You wouldn't want search to expose those entries to everyone, so you need to build understanding of your authorisation scheme into the search system.

Todo

list up front the various methods

Todo

Check if there is existing documentation which can be used as a skeleton for this.

5.9.1 Filtering results

Todo

Discuss filtering results coming back from a query, and the problems with just doing that.

5.9.2 Putting authorisation data into the search index

Todo

Discuss implementing auth schemes by indexing appropriate data.

5.9.3 Hybrid schemes

Todo

Discuss hybrid schemes (implementing auth using indexed terms, and also filtering results).

5.9.4 Timeliness of index authorisation

Todo

Discuss issues relating to updates (in particular, how fast does something need to be hidden if it is changed to being private).

CHAPTER 6

Deprecation of features

6.1 Deprecation

Table of contents

6.1.1 Introduction

Xapian's API is fairly stable and has been polished piece by piece over time, but it still occasionally needs to be changed. This may be because a new feature has been implemented and the interface needs to allow access to it, but it may also be required in order to polish a rough edge which has been missed in earlier versions of Xapian, or simply to reflect an internal change which requires a modification to the external interface.

We aim to make such changes in a way that allows developers to work against a stable API, while avoiding the need for the Xapian developers to maintain too many old historical interface artefacts. This document describes the process we use to deprecate old pieces of the API, lists parts of the API which are currently marked as deprecated, and also describes parts of the API which have been deprecated for some time, and are now removed from the Xapian library.

It is possible for functions, methods, constants, types or even whole classes to be deprecated, but to save words this document will often use the term "features" to refer collectively to any of these types of interface items.

6.1.2 Deprecation Procedure

Deprecation markers

At any particular point, some parts of the C++ API will be marked as "deprecated". Deprecated features are annotated in the API headers with macros such as `XAPIAN_DEPRECATED()`, which will cause compilers with appropriate support (such as GCC 3.1 or later, and MSVC 7.0 or later) to emit compile-time warnings if these features are used.

If a feature is marked with one of these markers, you should avoid using it in new code, and should migrate your code to use a replacement when possible. The documentation comments for the feature, or the list at the end of this file, will describe possible alternatives to the deprecated feature.

If you want to disable deprecation warnings temporarily, you can do so by passing `"-DXAPIAN_DEPRECATED(X)=X"` to the compiler (the quotes are needed to protect the brackets from the shell). If your build system uses make, you might do this like so:

```
make 'CPPFLAGS="-DXAPIAN_DEPRECATED(X)=X"'
```

API and ABI compatibility

Releases are given three-part version numbers (e.g. 1.2.9), the three parts being termed "major" (1), "minor" (2), and "revision" (9). Releases with the same major and minor version are termed a "release series".

For Xapian releases 1.0.0 and higher, an even minor version indicates a stable release series, while an odd minor version indicates a development release series.

Within a stable release series, we strive to maintain API and ABI forwards compatibility. This means that an application written and compiled against version *X.Y.a* of Xapian should work, without any source changes or need to recompile, with a later version *X.Y.b*, for all *b* >= *a*. Stable releases which increase the minor or major version number will usually change the ABI incompatibly (so that code will need to be recompiled against the newer release series. They may also make incompatible API changes, though we will attempt to do this in a way which makes it reasonably easy to migrate applications, and document how to do so in this document.

It is possible that a feature may be marked as deprecated within a minor release series - that is from version *X.Y.c* onwards, where *c* is not zero. The API and ABI will not be changed by this deprecation, since the feature will still be available in the API (though the change may cause the compiler to emit new warnings when rebuilding code which uses the now-deprecated feature).

Users should generally be able to expect working code which uses Xapian not to stop working without reason. We attempt to codify this in the following policy, but we reserve the right not to slavishly follow this. The spirit of the rule should kept in mind - for example if we discovered a feature which didn't actually work, making an incompatible API change at the next ABI bump would be reasonable.

Normally a feature will be supported after being deprecated for an entire stable release series. For example, if a feature is deprecated in release 1.2.0, it will be supported for the entire 1.2.x release series, and removed in development release 1.3.0. If a feature is deprecated in release 1.2.1, it will be supported for the 1.2.x *and* 1.4.x stable release series (and of course the 1.3.x release series in between), and won't be removed until 1.5.0.

Experimental features

During a development release series (such as the 1.1.x series), some features may be marked as "experimental". Such features are liable to change without going through the normal deprecation procedure. This includes changing on-disk formats for data stored by the feature, and breaking API and ABI compatibility between releases for the feature. Such features are included in releases to get wider use and corresponding feedback about them.

Deprecation in the bindings

When the Xapian API changes, the interface provided by the Xapian bindings will usually change in step. In addition, it is sometimes necessary to change the way in which Xapian is wrapped by bindings - for example, to provide a better convenience wrapper for iterators in Python. Again, we aim to ensure that an application written (and compiled, if the language being bound is a compiled language) for version *X.Y.a* of Xapian should work without any changes or need to recompile, with a later version *X.Y.b*, for all *a* <= *b*.

However, the bindings are a little less mature than the core C++ API, so we don't intend to give the same guarantee that a feature present and not deprecated in version *X.Y.a* will work in all versions *X+1.Y.b*. In other words, we may remove features which have been deprecated without waiting for an entire release series to pass.

Any planned deprecations will be documented in the list of deprecations and removed features at the end of this file.

6.1.3 Support for Other Software

Support for other software doesn't follow the same deprecation rules as for API features.

Our guiding principle for supporting version of other software is that we don't aim to actively support versions which are no longer supported "upstream".

So Xapian 1.1.0 doesn't support PHP4 because the PHP team no longer did when it was released. By the API deprecation rules we should have announced this when Xapian 1.0.0 was released, but we don't have control over when and to what timescales other software providers discontinue support for older versions.

Sometimes we can support such versions without extra effort (e.g. Tcl's stubs mechanism means Tcl 8.1 probably still works, even though the last 8.1.x release was over a decade ago), and in some cases Linux distros continue to support software after upstream stops.

But in most cases keeping support around is a maintenance overhead and we'd rather spend our time on more useful things.

Note that there's no guarantee that we will support and continue to support versions just because upstream still does. For example, we ceased providing backported packages for Ubuntu dapper with Xapian 1.1.0 - in this case, it's because we felt that if you're conservative enough to run dapper, you'd probably prefer to stick with 1.0.x until you upgrade to hardy (the next Ubuntu LTS release). But we may decide not to support versions for other reasons too.

6.1.4 How to avoid using deprecated features

We recommend taking the following steps to avoid depending on deprecated features when writing your applications:

- If at all possible, test compile your project using a compiler which supports warnings about deprecated features (such as GCC 3.1 or later), and check for such warnings. Use the -Werror flag to GCC to ensure that you don't miss any of them.
- Check the NEWS file for each new release for details of any new features which are deprecated in the release.
- Check the documentation comments, or the automatically extracted API documentation, for each feature you use in your application. This documentation will indicate features which are deprecated, or planned for deprecation.
- For applications which are not written in C++, there is currently no equivalent of the `XAPIAN_DEPRECATED` macro for the bindings, and thus there is no way for the bindings to give a warning if a deprecated feature is used. This would be a nice addition for those languages in which there is a reasonable way to give such warnings. Until such a feature is implemented, all application writers using the bindings can do is to check the list of deprecated features in each new release, or lookup the features they are using in the list at the end of this file.

6.2 Features currently marked for deprecation

Note: We have a separate *list of all features that have been fully deprecated*, with the version they were removed.

6.2.1 Native C++ API

Deprecated	Remove	Feature name	Upgrade suggestion and comments
1.1.0	?	Xapian::WritableDatabase::flush()	Xapian::WritableDatabase::commit() should be used instead.

6.2.2 Bindings

Deprecated	Remove	Language	Feature name	Upgrade suggestion and comments
1.2.5	1.5.0	Python	MSet.items	Iterate the MSet object itself instead.
1.2.5	1.5.0	Python	ESet.items	Iterate the ESet object itself instead.

6.2.3 Omega

Deprecated	Remove	Feature name	Upgrade suggestion and comments
1.2.5	1.5.0	$set{spelling,true}	Use $set{flag_spelling_suggestion,true} instead.

6.3 Features removed from Xapian

Table of contents

6.3.1 Native C++ API

Removed	Feature name	Upgrade suggestion and comments
1.0.0	QueryParser::set_stemming_options()	Use `set_stemmer()`, `set_stemming_strategy()` and/or `set_stopper()` instead: • `set_stemming_options("")` becomes `set_stemming_strategy(Xapian::Qu` • `set_stemming_options("none")` becomes `set_stemming_strategy(Xapian::Qu` • `set_stemming_options(LANG)` becomes `set_stemmer(Xapian::Stem(LANG)` and `set_stemming_strategy(Xapian::Qu` • `set_stemming_options(LANG, false)` becomes `set_stemmer(Xapian::Stem(LANG)` and `set_stemming_strategy(Xapian::Qu` • `set_stemming_options(LANG, true)` becomes `set_stemmer(Xapian::Stem(LANG)` and `set_stemming_strategy(Xapian::Qu` If a third parameter is passed, `set_stopper(PARAM3)` and treat the first two parameters as above.
1.0.0	Enquire::set_sort_forward()	Use `Enquire::set_docid_order()` instead: • `set_sort_forward(true)` becomes `set_docid_order(ASCENDING)` • `set_sort_forward(false)` becomes `set_docid_order(DESCENDING)`

Continued on next page

Table 6.1 – continued from previous page

Removed	Feature name	Upgrade suggestion and comments
1.0.0	Enquire::set_sorting()	Use `Enquire::set_sort_by_relevance()`, `Enquire::set_sort_by_value()`, or `Enquire::set_sort_by_value_then_re` instead. • `set_sorting(KEY, 1)` becomes `set_sort_by_value(KEY)` • `set_sorting(KEY, 1, false)` becomes `set_sort_by_value(KEY)` • `set_sorting(KEY, 1, true)` becomes `set_sort_by_value_then_relevance` • `set_sorting(ANYTHING, 0)` becomes `set_sort_by_relevance()` • `set_sorting(Xapian::BAD_VALUENO, ANYTHING)` becomes `set_sort_by_relevance()`
1.0.0	Stem::stem_word(word)	Use `Stem::operator()(word)` instead.
1.0.0	Auto::open(path)	Use the `Database(path)` constructor instead.
1.0.0	Auto::open(path, action)	Use the `WritableDatabase(path, action)` constructor instead.
1.0.0	Query::is_empty()	Use `Query::empty()` instead.
1.0.0	Document::add_term_nopos()	Use `Document::add_term()` instead.
1.0.0	Enquire::set_bias()	Use `PostingSource` instead (new in 1.2).
1.0.0	ExpandDecider::operator()	Return type is now `bool` not `int`.
1.0.0	MatchDecider::operator()	Return type is now `bool` not `int`.
1.0.0	Error::get_type()	Return type is now `const char *` not `std::string`. Most existing code won't need changes, but if it does the simplest fix is to write `std::string(e.get_type())` instead of `e.get_type()`.
1.0.0	<xapian/output.h>	Use `cout << obj.get_description();` instead of `cout << obj;`

Continued on next page

Table 6.1 – continued from previous page

Removed	Feature name	Upgrade suggestion and comments
1.0.0	Several constructors marked as explicit.	Explicitly create the object type required, for example use `Xapian::Enquire enq(Xapian::Database(path));` instead of `Xapian::Enquire enq(path);`
1.0.0	QueryParser::parse_query() throwing `const char *` exception.	Catch `Xapian::QueryParserError` instead of `const char *`, and call `get_msg()` on the caught object. If you need to build with either version, catch both (you'll need to compile the part which catches `QueryParserError` conditionally, since this exception isn't present in the 0.9 release series).
1.1.0	xapian_version_string()	Use `version_string()` instead.
1.1.0	xapian_major_version()	Use `major_version()` instead.
1.1.0	xapian_minor_version()	Use `minor_version()` instead.
1.1.0	xapian_revision()	Use `revision()` instead.
1.1.0	Enquire::include_query_terms	Use `Enquire::INCLUDE_QUERY_TERMS` instead.
1.1.0	Enquire::use_exact_termfreq	Use `Enquire::USE_EXACT_TERMFREQ` instead.
1.1.0	Error::get_errno()	Use `Error::get_error_string()` instead.
1.1.0	Enquire::register_match_decider()	This method didn't do anything, so just remove calls to it!
1.1.0	Query::Query(Query::op, Query)	This constructor isn't useful for any currently implemented `Query::op`.
1.1.0	The Quartz backend	Use the Chert backend instead.
1.1.0	Quartz::open()	Use `Chert::open()` instead.
1.1.0	quartzcheck	Use `xapian-check` instead.
1.1.0	quartzcompact	Use `xapian-compact` instead.
1.1.0	quartzdump	Use `xapian-inspect` instead.
1.1.0	configure –enable-debug	configure –enable-assertions
1.1.0	configure –enable-debug=full	configure –enable-assertions –enable-log
1.1.0	configure –enable-debug=partial	configure –enable-assertions=partial
1.1.0	configure –enable-debug=profile	configure –enable-log=profile
1.1.0	configure –enable-debug-verbose	configure –enable-log
1.1.0	`Database::positionlist_begin()` throwing `RangeError` if the term specified doesn't index the document specified.	This check is quite expensive, and often you don't care. If you do it's easy to check - just open a `TermListIterator` for the document and use `skip_to()` to check if the term is there.

Continued on next page

Table 6.1 – continued from previous page

Removed	Feature name	Upgrade suggestion and comments
1.1.0	`Database::positionlist_begin()` throwing `DocNotFoundError` if the document specified doesn't exist.	This check is quite expensive, and often you don't care. If you do, it's easy to check - just call `Database::get_document()` with the specified document ID.
1.1.5	delve -k	Accepted as an undocumented alias for -V since 0.9.10 for compatibility with 0.9.9 and earlier. Just use -V instead.
1.3.0	The Flint backend	Use the Chert backend instead.
1.3.0	Flint::open()	Use `Chert::open()` instead.
1.3.0	xapian-chert-update	Install Xapian 1.2.x (where x >= 5) to update chert databases from 1.1.3 and earlier.
1.3.0	Default second parameter to `Enquire` sorting functions.	The parameter name was `ascending` and defaulted to `true`. However ascending=false gave what you'd expect the default sort order to be (and probably think of as ascending) while ascending=true gave the reverse (descending) order. For sanity, we renamed the parameter to `reverse` and deprecated the default value. In the more distant future, we'll probably add a default again, but of `false` instead. The methods affected are: `Enquire::set_sort_by_value(Xapian::v sort_key)` `Enquire::set_sort_by_key(Xapian::Sor * sorter)` `Enquire::set_sort_by_value_then_rele sort_key)` `Enquire::set_sort_by_key_then_releva * sorter)` `Enquire::set_sort_by_relevance_then_ sort_key)` `Enquire::set_sort_by_relevance_then_ * sorter)` To update them, just add a second parameter with value `true` to each of the above calls. For the methods which take a `Xapian::Sorter` object, you'll also need to migrate to `Xapian::KeyMaker` (see below).

Continued on next page

Table 6.1 – continued from previous page

Removed	Feature name	Upgrade suggestion and comments
1.3.0	`Sorter` abstract base class.	Use `KeyMaker` class instead, which has the same semantics, but has been renamed to indicate that the keys produced may be used for purposes other than sorting (we plan to allow collapsing on generated keys in the future).

1.3.0	`MultiValueSorter` class.	Use `MultiValueKeyMaker` class instead. Note that `MultiValueSorter::add()` becomes `MultiValueKeyMaker::add_value()`, but the sense of the direction flag is reversed (to be consistent with `Enquire::set_sort_by_value()`), so:
		`MultiValueSorter sorter;`
		`// Primary ordering is forwards on v`
		`sorter.add(4);`
		`// Secondary ordering is reverse on`
		`sorter.add(5, false);`
		becomes:
		`MultiValueKeyMaker sorter;`
		`// Primary ordering is forwards on v`
		`sorter.add_value(4);`
		`// Secondary ordering is reverse on`
		`sorter.add_value(5, true);`

1.3.0	`matchspy` parameter to `Enquire::get_mset()`	Use the newer `MatchSpy` class and `Enquire::add_matchspy()` method instead.
1.3.0	`Xapian::timeout` typedef	Use POSIX `useconds_t` (from `<sys/types.h>`) instead, which should also work with older Xapian releases.
1.3.0	`Xapian::percent` typedef	Use `int` instead, which should also work with older Xapian releases.
1.3.0	`Xapian::weight` typedef	Use `double` instead, which should also work with older Xapian releases.
1.3.0	`Xapian::Query::unserialise()` throws `Xapian::SerialisationError` not `Xapian::InvalidArgumentError` for errors in serialised data	To be compatible with older and newer Xapian, you can catch both exceptions.

Continued on next page

Table 6.1 – continued from previous page

Removed	Feature name	Upgrade suggestion and comments
1.3.1	`Xapian::ErrorHandler`	We feel the current ErrorHandler API doesn't work at the right level (it only works in Enquire, whereas you should be able to handle errors at the Database level too) and we can't find any evidence that people are actually using it. So we are deprecating it and will replace it with something better thought out, probably during the 1.3.x development series. There's so further thoughts at http://trac.xapian.org/ticket/3#comment:8
1.3.2	`Xapian::Auto::open_stub()`	Use the constructor with `Xapian::DB_BACKEND_STUB` flag (new in 1.3.2) instead.
1.3.2	`Xapian::Brass::open()`	Use the constructor with `Xapian::DB_BACKEND_BRASS` flag (new in 1.3.2) instead.
1.3.2	`Xapian::Chert::open()`	Use the constructor with `Xapian::DB_BACKEND_CHERT` flag (new in 1.3.2) instead.

6.3.2 Bindings

Removed	Language	Feature name	Upgrade suggestion and comments
1.0.0	SWIG [1]	Enquire::set_sort_forward()	Use `Enquire::set_docid_order()` instead. • `set_sort_forward(true)` becomes `set_docid_order(ASCENDI` • `set_sort_forward(false)` becomes `set_docid_order(DESCEND`
1.0.0	SWIG [1]	Enquire::set_sorting()	Use `Enquire::set_sort_by_relevan` `Enquire::set_sort_by_value()` or `Enquire::set_sort_by_value_t` instead. • `set_sorting(KEY, 1)` becomes `set_sort_by_value(KEY)` • `set_sorting(KEY, 1, false) becomes ``set_sort_by_value(KEY` • `set_sorting(KEY, 1, true)` becomes `set_sort_by_value_then_` • `set_sorting(ANYTHING, 0) becomes set_sort_by_relevance()` • `set_sorting(Xapian::BAD_ ANYTHING)` becomes `set_sort_by_relevance()`
1.0.0	SWIG [1]	Auto::open(path)	Use the `Database(path)` constructor instead.
1.0.0	SWIG [1]	Auto::open(path, action)	Use the `WritableDatabase(path, action)` constructor instead.
1.0.0	SWIG [2]	MSet::is_empty()	Use `MSet::empty()` instead.
1.0.0	SWIG [3]	ESet::is_empty()	Use `ESet::empty()` instead.
1.0.0	SWIG [3]	RSet::is_empty()	Use `RSet::empty()` instead.
1.0.0	SWIG [3]	Query::is_empty()	Use `Query::empty()` instead.

6.3.3 Omega

Removed	Feature name	Upgrade suggestion and comments
1.0.0	$freqs	Use `$map{$queryterms,$_:&nbsp;$nice{$freq{$_}}}` instead.
1.0.0	scriptindex -u	`-u` was ignored for compatibility with 0.7.5 and earlier, so just remove it.
1.0.0	scriptindex -q	`-q` was ignored for compatibility with 0.6.1 and earlier, so just remove it.
1.1.0	scriptindex index=nopos	Use `indexnopos` instead.
1.3.0	`OLDP` CGI parameter	Use `xP` CGI parameter instead (direct replacement), which has been supported since at least 0.5.0.

CHAPTER 7

Glossary

This glossary defines specialized terminology you may encounter while using Xapian. Some of the entries are standard in the field of Information Retrieval, while others have a specific meaning in the context of Xapian.

BM25 The weighting scheme which Xapian uses by default. BM25 is a refinement on the original probabilistic weighting scheme, and recent TREC tests have shown BM25 to be the best of the known probabilistic weighting schemes. It's sometimes known as "Okapi BM25" since it was first implemented in an academic IR system called Okapi.

Boolean Retrieval Retrieving the set of documents that match a boolean query (e.g. a list of terms joined with a combination of operators such as AND, OR, AND_NOT). In many systems, these documents are not ranked according to their relevance. In Xapian, a pure Boolean query may be used, or alternatively a Boolean style query can filter the retrieved documents, which are then ordered using a probabilistic ranking.

Brass Brass is the current "under development" database format used in Xapian. We try to keep it as stable as chert - the main difference is that incompatible changes may be made which will require you to reindex, even between stable point release. In release 1.4.0, brass will be declared stable and made the default backend, and chert will then be deprecated.

Chert Chert is the stable database format used in Xapian 1.2.x. It is similar to Flint in many ways, but generally faster, and uses significantly less disk space. Chert is very efficient and highly scalable. It supports incremental modifications, and concurrent single-writer and multiple-reader access to a database.

Collection Frequency The collection frequency of a term is the total number of times is occurs in the database. This is equal to the sum of the within-document frequency for the term in all the documents it occurs in.

Database In Xapian (as opposed to a relational database system) a database consists of little more than indexed documents: this reflects the purpose of Xapian as an information retrieval system, rather than an information storage system. These may also occasionally be called Indexes. Chert is the backend used from Xapian 1.2 onwards; Flint was used from Xapian 1.0 onwards; Quartz was used in older versions.

Document ID A unique positive integer identifying a document in a Xapian database.

Document data The document data is one of several types of information that can be associated with each document, the contents can be set to be anything in any format, examples include fields such as URL, document title, and an excerpt of text from the document. If you wish to interoperate with Omega, it should contain name=value pairs, one per line (recent versions of Omega also support one field value per line, and can assign names to line numbers in the query template).

Document These are the items that are being retrieved. Often they will be text documents (e.g. web pages, email messages, word processor documents) but they could be sections within such a document, or photos, video, music, user profiles, or anything else you want to index.

Edit distance A measure of how many "edits" are required to turn one text string into another, used to suggest spelling corrections. The algorithm Xapian uses counts an edit as any of inserting a character, deleting a character, changing a character, or transposing two adjacent characters.

ESet (Expand Set) The Expand Set (ESet) is a ranked list of terms that could be used to expand the original query. These terms are those which are statistically good differentiators between relevant and non-relevant documents.

Flint Flint was the default database format used in Xapian 1.0.x. It was deprecated in 1.2.x and removed in 1.3.0.

Index If a document is described by a term, this term is said to index the document. Also, the database in Xapian and other IR systems is sometimes called an index (by analogy with the index in the back of a book).

Indexer The indexer takes documents (in various formats) and processes them so that they can be searched efficiently, they are then stored in the database.

Information Need The information need is what the user is looking for. They will usually attempt to express this as a query string.

Information Retrieval (IR) Information Retrieval is the “science of search”. It’s the name used to refer to the study of search and related topics in academia.

MSet (Match Set) The Match Set (MSet) is a ranked list of documents resulting from a query. The list is ranked according to document weighting, so the top document has the highest probability of relevance, the second document the second highest, and so on. The number of documents in the MSet can be controlled, so it does not usually contain all of the matching documents.

Normalised document length (ndl) The normalised document length (ndl) is the length of a document (the number of terms it contains) divided by the average length of the documents within the system. So an average length document would have ndl equal to 1, while shorter documents have ndl less than 1, and longer documents greater than 1.

Omega Omega comprises two indexers and a CGI search application built using the Xapian library.

Posting List A posting list is a list of the documents which a specific term indexes. This can be thought of as a list of numbers - the document IDs.

Posting An instance of a particular term indexing a particular document.

Precision Precision is the density of relevant documents amongst those retrieved: the number of relevant documents returned divided by the total number of documents returned.

Probabilistic IR Probabilistic IR is retrieval based on probability theory, this can produce a ranked list of documents based upon relevance. Xapian uses probabilistic methods (the only exception is when a pure Boolean query is chosen)

Quartz Quartz was the database format used by Xapian prior to version 1.0. Support was dropped completely as of Xapian 1.1.0.

Query A query is the information need expressed in a form that an IR system can read. It is usually a text string containing terms, and may include Boolean operators such as AND or OR, etc.

Query Expansion Modifying a query in an attempt to broaden the search results.

RSet (Relevance Set) The Relevance Set (RSet) is the set of documents which have been marked by the user as relevant. They can be used to suggest terms that the user may want to add to the query (these terms form an ESet), and also to adjust term weights to reorder query results.

Recall Recall is the proportion of relevant documents retrieved - the number of relevant documents retrieved divided by the total number of relevant documents.

Relevance Essentially, a document is relevant if it is what the user wanted. Ideally, the retrieved documents will all be relevant, and the non-retrieved ones all non-relevant.

Searcher The searcher is a part of the IR system, it takes queries and reads the database to return a list of relevant documents.

Stemming A stemming algorithm performs linguistic normalisation by reducing variant forms of a word to a common form. In English, this mainly involves removing suffixes - such as converting any of the words “talking”, “talks”, or “talked” to the stem form “talk”.

Stop word A word which is ignored during indexing and/or searching, usually because it is very common or doesn’t convey meaning. For example, “the”, “a”, “to”.

Synonyms Xapian can store synonyms for terms, and use these to implement one approach to query expansion.

Term List A term list is the list of terms that index a specific document. In some systems this may be a list of numbers (with each term represented by a number internally), in Xapian it is a list of strings (the terms).

Term frequency The term frequency of a specific term is the number of documents in the system that are indexed by that term.

Term A term is a string of bytes (often a word or word stem) which describes a document. Terms are similar to the index entries found in the back of a book and each document may be described by many terms. A query is composed from a list of terms (perhaps linked by Boolean operators).

Term Prefix By convention, terms in Xapian can be prefixed to indicate a field in the document which they come from, or some other form of type information. The term prefix is usually a single capital letter.

Test Collection A test collection consists of a set of documents and a set of queries each of which has a complete set of relevance assignments - this is used to test how well different IR methods perform.

UTF-8 A standard variable-length byte-oriented encoding for Unicode.

Value A discrete meta-data attribute attached to a document. Each document can have many values, each stored in a different numbered slot. Values are designed to be fast to access during the matching process, and can be used for sorting, collapsing redundant documents, implementing ranges, and other uses. If you’re just wanting to store “fields” for displaying results, it’s better to store them in the document data.

Within-document frequency (wdf) The within-document frequency (wdf) of a term in a specific document is the number of times it is pulled out of the document in the indexing process. Usually this is the size of the wdp vector, but in Xapian it can exceed it, since we can apply extra wdf to some parts of the document text.

Within-document positions (wdp) In the case where a term derives from words actually in the document, the within-document positions (wdp) are the positions at which that word occurs within the document. So if the term derives from a word that occurs three times in the document as the fifth, 22nd and 131st word, the wdps will be 5, 22 and 131.

Within-query frequency (wqf) The within-query frequency (wqf) is the number of times a term occurs in the query. This statistic is used in the BM25 weighing scheme.

CHAPTER 8

Contributing

The source for this documentation is being kept on http://github.com/jaylett/xapian-docsprint. The best way to contribute is to add issues, comments and pull requests there.

Assuming you're on Debian or Ubuntu or another Debian-derived distro, you'll need to install either the *python-sphinx* or *python3-sphinx* package in order to be able to generate this documentation from a git checkout. Once you've done that, you can generate HTML output with *make html*.

We're monitoring IRC during the sprint sessions (and in general) so you can also contact us on channel #xapian on irc.freenode.net (webchat link: http://webchat.freenode.net/?channels=%23xapian).

CHAPTER 9

Indices and tables

- *search*

CHAPTER 10

Todo list

Todo

This example should really be pulled directly from the code. There seems to be a bug with line number limiting somewhere in the literal include directive.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/practical_example/searching/prefix.rst, line 18.)

Todo

split into sections

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/indexing/databases.rst, line 18.)

Todo

list backends

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/indexing/databases.rst, line 20.)

Todo

disk-based brass/chert/(flint quartz how to upgrade); autodetecton

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/indexing/databases.rst, line 22.)

Todo

inmemory; use ram disk for serious use

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/indexing/databases.rst, line 24.)

Todo

remotes

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/indexing/databases.rst, line 26.)

Todo

multiple databases

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/indexing/databases.rst, line 28.)

Todo

stubs

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/indexing/databases.rst, line 30.)

Todo

Xapian::Error hierarchy

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/language_specific/c++/index.rst, line 26.)

Todo

write me

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/language_specific/c++/index.rst, line 38.)

Todo

write me

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/language_specific/c++/index.rst, line 43.)

Todo

(as howto?) pagination (ask for pagesize*#pages + 1)

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/search/index.rst, line 12.)

Todo

get_matches_estimated() and check_at_least

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/search/index.rst, line 13.)

Todo

Note on difference between OP_FILTER and OP_AND?

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/search/queries.rst, line 79.)

Todo

MatchAll MatchNothing

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/search/queries.rst, line 121.)

Todo

This needs writing from scratch; the previous document wasn't very good and is license entangled.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/advanced/remote.rst, line 7.)

Todo

add more and fill out those already here a little more

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/indexing/limitations.rst, line 11.)

Todo

actually, reference the FAQ instead of saying something here.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/indexing/limitations.rst, line 20.)

Todo

add or link to some details of how to do this

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/concepts/search/search_limitations.rst, line 54.)

Todo

Write about how to index geolocation information, and how to use the geo posting sources and keymaker to sort by distance, bias results by distance, and limit results by distance. Discuss storing geo bounding boxes terms for accelerating distance limited searches.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/advanced/geospatial.rst, line 8.)

Todo

list up front the various methods

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/advanced/query_authorisation.rst, line 9.)

Todo

Check if there is existing documentation which can be used as a skeleton for this.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/advanced/query_authorisation.rst, line 11.)

Todo

Discuss filtering results coming back from a query, and the problems with just doing that.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/advanced/query_authorisation.rst, line 17.)

Todo

Discuss implementing auth schemes by indexing appropriate data.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/advanced/query_authorisation.rst, line 23.)

Todo

Discuss hybrid schemes (implementing auth using indexed terms, and also filtering results).

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/advanced/query_authorisation.rst, line 29.)

Todo

Discuss issues relating to updates (in particular, how fast does something need to be hidden if it is changed to being private).

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/advanced/query_authorisation.rst, line 35.)

Todo

relevance feedback and query expansion

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/index.rst, line 17.)

Todo

point out that lowercasing by TermGenerator or similar will prevent unexpected matching of prefixes terms by “real” words in the source data

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/boolean_filters.rst, line 4.)

Todo

Write about how to use collapsing with searches. Discuss collapsing to return only a unique item for each value, and collapsing to return multiple. Make clear that it simply excludes some items from the result set; doesn’t reorder results, or guarantee to return the top N from each category. Also add a code example.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/collapsing.rst, line 9.)

Todo

Getting a correction for a single word "by hand"

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/spelling.rst, line 71.)

Todo

Discuss interactions with stemming (ie, should the input and/or output values in the synonym table be stemmed).

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/synonyms.rst, line 28.)

Todo

Document this!

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/synonyms.rst, line 34.)

Todo

Query.OP_SYNONYM, and how that relates to synonym expansion.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/synonyms.rst, line 74.)

Todo

Say something more useful about tuning the parameters!

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/weighting_scheme.rst, line 36.)

Todo

Mention Omega here and link to its documentation

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/overview.rst, line 18.)

Todo

finalise datasets and code and link to them from here

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/overview.rst, line 92.)

Todo

link to here from every howto and everything that needs the data files and example code

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/overview.rst, line 115.)

Todo

expand this section.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/advanced/replication.rst, line 56.)

Todo

XAPIAN_MAX_CHANGESETS is not irrelevant in brass. We should talk about that here but only if we are clear what version it was added in.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/advanced/replication.rst, line 62.)

Todo

add the link above

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/facets.rst, line 27.)

Todo

It is probably worth saying that a typical database will not hit these performance issues

---

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/facets.rst, line 105.)

Todo

check valueranges.rst to see if anything else needs moving across

---

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/range_queries.rst, line 4.)

Todo

the above paragraph isn’t entirely inaccurate; the processor is unweighted, so if there’s no other query, and the docid ordering is don’t care or ascending, then the search can terminate early. If the VRP isn’t matching many documents, that could still be slow, but might not be. If it’s not matching any documents, it might be fast because the bounds on stored values may show that it can’t match anything. Oh, it’s all quite complicated really. It would be nice to explain how this is done somewhere, but probably not here.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/range_queries.rst, line 285.)

Todo

actually, you can’t safely combine the query with an external filter, because other bits of the query might be higher level. For example, a query of ‘1790..1799 OR york’ couldn’t have the filter applied to the generated query because it shouldn’t be applied to the “york” part.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/range_queries.rst, line 328.)

Todo

possibly implementing this example would help make it more clear.

(The *original entry* is located in /var/build/user_builds/getting-started-with-xapian/checkouts/latest/howtos/range_queries.rst, line 334.)