Xapian developer guide1.4

Release 1.4.3

Xapian Documentation Team
Contributors

Jul 18, 2022

Contents

Xapian is an open source search engine library, which allows developers to add advanced indexing and search facilities to their own applications. This manual aims to explain how to work on and contribute to Xapian itself; if you want to use Xapian in your own project, you should look at our Xapian user manual.

Note: This is early days for this guide, so please let us know any issues you spot or how we can improve it in any way.

Since Xapian is an open source project, it is entirely dependent on contributions – many of them from people like you! These contributions come in many forms, from spotting when some documentation isn't clear and maybe improving it, through fixing bugs up to designing and implementing completely new features. We need all types of contribution for Xapian to continue to evolve and serve its users.

This guide is intended to:

- help you understand how to make and contribute changes to Xapian
- lay out some of our "rules of the road", which are there to make it easier for everyone to collaborate on Xapian

We recommend you at least skim through all of this guide, so you know what information is available to you. However, if you absolutely must jump in head first, you should at least read our *advice for new contributors*.

Todo: Talk about how to manage security issues. This article may be helpful.

CHAPTER 1

Getting in touch

The Xapian community typically works "in the open", via mailing lists and a chat channel:

- Our mailing lists are open for anyone to join, although (because we don't want to relay spam to everyone) if you aren't subscribed to the list someone will have to manually approve your message. Please be patient and don't resend a message just because it doesn't appear right away.
- We're on #xapian on `irc.libera.chat`, which is bridged to the matrix room #xapian:matrix.org. Note that because we're distributed around the world you may not get an instant response. That doesn't mean we're ignoring you, so either hang on for a reply, or you can use the mailing list instead.

CHAPTER 2

Contents

2.1 Getting started

Note: Currently this guide is written assuming you are either developing on something like Unix (probably either Linux or macOS). You can use software such as Virtual Box to run a 'virtual' Linux machine on another operating system; in this case we recommend using the most recent Ubuntu LTS release.

It should be possible to build and develop Xapian on Windows, but we currently don't have any documentation on doing so, or any active developers with suitable experience.

2.1.1 Getting the source code

First off, let's make sure you have a copy of the Xapian source and can build it and run the tests. This is generally a little different to if you're just installing Xapian to use it, because you'll be working with the entire source tree rather than individual pieces. The Xapian build system has some support for this, but let's get you a copy of everything first:

```
$ git clone https://git.xapian.org/xapian
$ cd xapian
```

This will 'clone' a complete copy of the Xapian source code, including not only the core library but also the various language bindings (for use from Python, Lua, Ruby and so on) and the self-contained web search system 'Omega'. It also contains all the tests for those various components.

2.1.2 Installing the dependencies

Debian / Ubuntu

For a recent version of Debian or Ubuntu, this command should ensure you have all the necessary tools and libraries:

```
$ apt-get install build-essential m4 perl python zlib1g-dev uuid-dev \
  wget bison tcl libpcre3-dev libmagic-dev valgrind ccache eatmydata \
  doxygen graphviz help2man python-docutils pngcrush python-sphinx \
  python3-sphinx mono-devel default-jdk lua5.3 liblua5.3-dev \
  php-dev php-cli python-dev python3-dev ruby-dev tcl-dev texinfo
```

macOS

You need to install Apple's XCode tools, which contain their compiler, debugger and various other tools. You can do that from within the AppStore.

We recommend using homebrew to install and manage additional libraries and tools on macOS. Once you've installed XCode and homebrew, you can get all the dependencies you need for Xapian using:

```
$ brew install libmagic pcre \
  lua mono perl php python python3 ruby tcl-tk \
  doxygen help2man graphviz pngcrush
# and some python-specific documentation tools
$ pip install sphinx docutils
$ pip3 install sphinx
```

(We install documentation tools for both python2 and python3, in the same way we build the bindings for both of them.)

Windows

Building using MSVC is supported by the autotools build system. You need to install a set of Unix-like tools first – we recommend MSYS2.

For details of how to specify MSVC to `configure` see the "INSTALL" document in `xapian-core`.

When building from git, by default you'll need some extra tools to generate Unicode tables (Tcl) and build documentation (doxygen, help2man, sphinx-doc). We don't currently have detailed advice on how to do this (if you can provide some then please send a patch).

You can avoid needing Tcl by copying `xapian-core/unicode/unicode-data.cc` from another platform or a release which uses the same Unicode version. You can avoid needing most of the documentation tools by running configure with the `--disable-documentation` option.

On other platforms

You will need the following tools installed to build from git:

- GNU m4 >= 4.6 (for autoconf)
- perl >= 5.6 (for automake; also for various maintainer scripts)
- python >= 2.3 (for generating the Python bindings)
- GNU make (or another make which support VPATH for explicit rules)
- GNU bison (for building SWIG, used for generating the bindings)
- Tcl (to generate unicode/unicode-data.cc)

There are also a number of libraries you'll need available, including development headers:

- zlib1g

- libuuid
- PCRE
- libmagic (which is part of the open source implementation of file(1))

If you're doing much development work, you'll probably also want the following tools installed:

- valgrind for better testsuite error finding
- ccache for faster rebuilds
- eatmydata for faster testsuite runs

If you want to be able to build distribution tarballs (with `make dist`) then you'll also need some further tools:

- doxygen (v1.8.8 is used for 1.3.x snapshots and releases; 1.7.6.1 fails to process git master after `PL2Weight` was added).
- dot (part of Graphviz. Doxygen's `DOT_MULTI_TARGETS` option apparently needs ">1.8.10")
- help2man
- rst2html or rst2html.py (`pip install docutils`)
- pngcrush (optional - used to reduce the size of PNG files in the HTML apidocs)
- sphinx-doc (`pip install sphinx` should do)

2.1.3 Building Xapian

Bootstrapping the code

The easiest way of building Xapian from git master is to use our bootstrap script. It takes care of a number of things which are otherwise fiddly to get right, including checking you have the right version of various tools we use, and setting up the build system for you.

Note: One things that bootstrap does is to set up a top-level `configure` script which ensures that the in-tree version of `xapian-core` is built first and then used for building everything else. You almost certainly want to build Xapian this way.

Not using this means you have to check by hand that you're building other subdirectories against the in-tree core library, as by default they will pick any installed copy. An installed copy of Xapian is likely to be a different version to the source tree you are building. Building the git master version of Xapian against an earlier released library will probably fail. If you're working on Xapian then you almost certainly want to build everything against the in-tree version, so you should use `bootstrap` and the `configure` script it creates.

The repository does not contain any automatically generated files (such as `configure`, `Makefile.in`, Snowball-generated stemmers, Lemon-generated parsers, SWIG-generated code, and so on) because experience shows it's best to keep these out of version control. To avoid requiring you to install the correct versions of the tools required, we either include the source to these tools in the repo directly (in the case of Snowball and Lemon), or the bootstrap script will download them as tarballs (autoconf, automake, libtool) or from git (SWIG), build them, and install them within the source tree.

The bootstrap script doesn't care what the current directory is, but you can easily run it in the `xapian` directory that was created earlier when you cloned the source code:

```
$ ./bootstrap
```

To download some tools, bootstrap will use `wget`, `curl` or `lwp-request` if installed. If not, it will give an error telling you the URL to download from by hand and where to copy the file to. You can control whether Xapian tries to download, patch and install autotools with the `--download-tools` option to `bootstrap`:

`--download-tools=always` Always download, patch and install autotools we rely on.

`--download-tools=ifneeded` (the default) Download, patch and install autotools only if your installed version isn't recent enough, or if we have to apply patches that haven't yet been accepted upstream.

`--download-tools=never` Never download and install autotools; always use your installed versions.

Note that in this case the build may fail if you have out of date versions of the tools, and you may also fall foul of behaviour fixed in our patches.

You can also ask the build system to delete the downloaded and installed versions by passing `--clean`.

Our bootstrap script will check which directories you have checked out, so you can bootstrap a partial tree. You can also `touch .nobootstrap` in a subdirectory to tell bootstrap to ignore it, or you can pass just the directories you want to build as arguments to `bootstrap`.

If you need to add any extra macro directories to the path searched by aclocal (which is part of automake), you can do this by specifying these in the `ACLOCAL_FLAGS` environment variable. For instance:

```
$ ACLOCAL_FLAGS=-I/extra/macro/directory ./bootstrap
```

Note: As well as installing some tools, bootstrap will also run `autoreconf` on each of the checked-out subdirectories, and generate a top-level `configure` script. This configure script allows you to configure xapian-core and any other modules you've checked out with a single simple command, such that the other modules link against the uninstalled xapian-core (which is very handy for development work and a bit fiddly to set up by hand). It automatically passes `--enable-maintainer-mode` to the subprojects so that the autotools will be rerun if `configure.ac`, `Makefile.am`, etc are modified.

Warning: If you are tracking development in git, there will sometimes be changes to the build system sources which require regeneration of the generated makefiles and associated machinery. We aim to make the build system automatically regenerate the necessary files, but in the event that a build fails after an update, it may be worth re-running the bootstrap script to regenerate the build system from scratch, before looking for the cause of the error elsewhere.

Configuring the code

Configuring the code is mostly about Xapian's build system automatically detecting where all its dependencies are on your computer, so it knows how to use them. However there are various options that allow you to either override the autodetection (for instance if you wanted to build python bindings against a particular version of python) or change some defaults. To find out about the configure options available, you can run `configure --help`. For now, however, we'll just run it accepting all its defaults:

```
$ ./configure
```

Note that on macOS you probably want to turn off the Perl and TCL8 bindings when developing, as there are some complexities when developing against the system versions, and the homebrew versions are slightly awkward:

```
$ ./configure --without-perl --without-tcl
```

Our configure script supports building in a separate directory to the sources. Simply create the directory you want to build in, and then run the configure script from inside that directory. For example, to build in a directory called "build" (starting in the top level source directory):

```
$ ./bootstrap
# output from bootstrap
$ mkdir build
$ cd build
$ ../configure
```

Building Xapian

Building Xapian is just a matter of typing:

```
$ make
```

First it will build xapian-core, the core library. Then it will build Omega and the language bindings, using the version of xapian-core you've just built, but not yet installed. (This is the bit that causes some problems on macOS if you use system versions of any of the languages.)

2.1.4 Running the tests

Xapian has a comprehensive test suite, and it's a good idea to get into the habit of running it. From the top of the clone, just run:

```
$ make check
```

Again, the tests for xapian-core are run first, then Omega and then the language bindings. If any test fails, the build system will stop there.

2.1.5 A quick note about the build system

Here, we've been working from a clone of the Xapian git repository, which means that the following options are on by default. However if you are ever building from a source tarball, the following may be of use.

`--enable-maintainer-mode` This tells configure to enable make dependencies for regenerating build system files (such as `configure`, `Makefile.in`, and `Makefile`) and other generated files when required. These are disabled by default as some make programs try to rebuild them when it's not appropriate (e.g. BSD make doesn't handle VPATH except for implicit rules). For this reason, we recommend GNU make if you enable maintainer mode.

For `xapian-core`, generated files include the stemmers and query parser; you'll need a non-cross-compiling C compiler for compiling the Lemon parser generator and the Snowball stemming algorithm compiler. The configure script will attempt to locate one, but you can override this autodetection by passing `CC_FOR_BUILD` on the command line like so:

```
./configure CC_FOR_BUILD=/opt/bin/gcc
```

For `xapian-bindings`, generated files include the bindings glue code, which requires SWIG. You'll need to have maintainer mode enabled if you're going to work on the bindings at all.

`--enable-documentation` This tells configure to enable make dependencies for regenerating documentation files. By default it uses the same setting as `--enable-maintainer-mode`. You can turn off documentation

rules in maintainer mode (which means that documentation won't be rebuilt on `make check`, which will save some time) by passing `--disable-documentation` to configure.

Note that `make dist` requires the documentation to have been built, and so won't work with a git checkout if you disable building the documentation. You can still configure and build the code itself.

Xapian's build system has a lot of other options you can use to control exactly what gets built and in what ways. Check out help information for the various tools for more information, such as `./bootstrap --help` and `./configure --help`.

2.1.6 Summary

Now you've got everything working, you probably want to look at *contributing to Xapian*, or if you're trying to fix a bug then you might want to learn about *debugging Xapian*.

2.2 Coding and other conventions in Xapian

We aim for Xapian to be:

- *Cross-platform*: this means it will compile and run on a range of different platforms. We have three sets of automated build systems to help us keep track of this: the Xapian buildbots (currently offline), our builds on Github Actions, and finally builds on AppVeyor. The last two will be triggered automatically if you submit changes using *a pull request on github*.
- *Able to build "cleanly"*, meaning without warnings, on a range of compilers. Note that the two main C++ compilers in use these days are from clang and GCC, and they have slightly different sets of warnings and behaviours. Our automated builds run against both compilers. When you build Xapian, the compiler configuration used by default will highlight warnings and refuse to complete the build if it finds any.

- *Documented* and *tested* throughout. Although not all of it is fully-documented or tested as it stands, if we add documentation and tests every time we add a feature, fix a bug, or work on existing code, then we will keep on improving.

  If you'd like to improve our test code coverage, our code coverage report may be helpful in choosing something to add tests for. Remember that code coverage isn't the same as having useful tests for the code, so even in parts of the codebase that have good coverage there may be new tests worth writing.

With that in mind, we have some conventions and standards that we try to adhere to. It's difficult to get away from big lists of things to do and not do, but we've tried to explain why as much as possible, and to group things in a way that makes it easier to find useful information. It's a good idea to at least skim through this material so you can go back for a detailed look when you need.

2.2.1 C++ conventions

Code layout

Editor configuration

vim

If you use the `vim` or `neovim` editor, then these settings will help you lay out code to match Xapian C++ coding conventions:

```
set sw=4
set sts=4
set noet
set cinoptions=l1,g0.5s,h0.5s,:0.5s,=0.5s,t0,(0
```

Indentation

Indent C++ code by 4 spaces for a new indentation level, and set your editor to tab-fill indentation (with a tab being 8 spaces wide).

As an exception, "public", "protected" and "private" declarations in classes and structs should be indented by 2 spaces, and the following code should be indented by 2 more spaces:

```
class Foo {
  public:
    method();
};
```

The rationale for this exception is that class definitions in header files often have fairly long lines, so losing an indent level to the access specifier tends to make class definitions less readable.

The default access for a class is always "private", so there's no need to specify that explicitly - in other words, write this:

```
class Foo {
    int internal_method();

  public:
    int external_method();
};
```

Don't write this:

```
class Foo {
  private:
    int internal_method();

  public:
    int external_method();
};
```

If a class only contains public methods and data, consider declaring it as a "struct" (the only difference in C++ is that the default access for a struct is "public").

Spaces and line breaks

Put a space before the `(` after control flow constructs like `for`, `if`, `while`, and so on. So write `if (strlen(p) > 10)` not `if(strlen (p) > 10)`. Don't put a space before the `(` in function calls.

When `if`, `else`, `for`, `while`, `do`, `switch`, `case`, `default`, `try`, or `catch` is followed by a block enclosed in braces, the opening brace should be on the same line, like so:

```
if (x > 12) {
    foo(x);
    x = 12;
} else {
    bar(x);
}
```

The rationale for this is that it conserves vertical space (allowing more code to fit on screen) without reducing readability.

C++ idioms in Xapian

- If you have an empty loop body, use `{ }` rather than `;` as the former stands out more clearly to the reader (but also consider if the code might be clearer written a different way).
- Prefer `++i;` to `i++;`, `i += 1;`, or `i = i + 1`. For simple integer variables these should generate equivalent (if not identical) code, but if `i` is an iterator object then the pre-increment form can be more efficient in some cases with some compilers. It's simpler and more consistent to always use the pre-increment form (unless you make use of the old value which the post-increment form returns). For the same reasons, prefer `--i;` to `i--;`, `i -= 1;`, or `i = i - 1;`.
- Prefer `container.empty()` to `container.size() == 0` (and `!container.empty()` to `container.size() != 0` or `container.size() > 0`).

  Some containers (e.g. `std::forward_list`) support `empty()` but not `size()`. Pre-C++11 finding the size of a container wasn't necessarily a constant time operation for some containers (e.g. `std::list` with GCC) - that's no longer the case for any STL containers since C++11, but it could still be true for non-STL containers.

  Also the `empty()` form makes the intent of the test more explicit.
- Prefer not to use `else` when the control flow is diverted elsewhere at the end of the `if` block (e.g. by `return`, `continue`, `break`, `throw`). This eliminates a level of indentation from the code in the `else` block, and typically makes the control flow logic clearer. For example:

  ```
  if (x == 0) {
      foo();
      return;
  }

  while (x--) {
      bar();
  }
  ```

  rather than:

  ```
  if (x == 0) {
      foo();
      return;
  } else {
      while (x--) {
          bar();
      }
  }
  ```

- For standard ISO C headers, prefer the C++ form for ISO C headers (e.g. `#include <cstdlib>` rather than `#include <stdlib.h>`) unless there's a good reason (e.g. portability) to do otherwise. Be sure to document such exceptions to avoid another developer changing them to the standard form. Global exceptions: <signal.h> (lots of POSIX stuff which e.g. Sun's compiler doesn't provide in <csignal>).
- For standard ISO C++ headers, *always* use the ISO C++ form `#include <list>` (pre-ISO compilers used `#include <list.h>`, but GCC has generated a warning for this form for years, and GCC 4.3 dropped support entirely).
- Prefer `new SomeClass` to `new SomeClass()`, since the latter tends to lead one to write `SomeClass foo();` which is a function prototype, and not equivalent to the variable definition `SomeClass foo`. However, note that `new SomePODType()` is *not* the same as `new SomePODType` (if SomePODType is a Plain Old Data type) - the former will zero-initialise scalar members of SomePODType.
- RTTI (`dynamic_cast<>`, `typeid`, `std::typeinfo`): Needing to use RTTI features in the library most likely indicates a design flaw, and you should avoid use of these features. Where necessary, you can use a technique similar to `Database::as_networkdatabase()` to replace `dynamic_cast<>`.
- `using namespace std;` and `using std::XXX;` - it's OK to use these in applications, library code, and internal library headers. But in externally visible headers (such as anything included by `#include <xapian.h>`) you *MUST* use explicit `std::` qualifiers - it's not acceptable to pull anything from namespace std into the namespace of an application which uses Xapian.
- Use C++ style casts (`static_cast<>`, `reinterpret_cast<>`, and `const_cast<>`) or constructor-syntax (e.g. `double(value)`) in preference to C style casts. The syntax of the C++ casts is ugly, but they do make the intent much clearer which is definitely a good thing, and they avoid issues such as casting away const when you only meant to cast the type of a pointer.
- `std::pair<>` with an STL class as one (or both) of the members can produce very long symbols (over 4KB!) after name mangling - long enough to overflow the size limits of some vendor compilers or toolchains (so this can affect GCC if it is using the system ld or as). Even where the compiler works, the symbol bloat in an unstripped build is probably best avoided, so it's preferable to use a simple two member struct instead. The code is probably more readable anyway, and easier to extend if more members are needed later.
- We try to avoid putting the full definition of virtual methods in header files. This is because current compilers can't (as far as we know) inline virtual methods, so putting the definition in the header file simply slows down compilation (and, because method definitions often require further header files to be included, this can result in many more files needing recompilation after a change to a header file than is really necessary). Just put the declaration in the header file, and put the definition in a .cc file with the same basename.

Efficient use of `std::string`

- When passing an empty string to a method expecting `const std::string &` prefer `std::string()` to `""` or `std::string("")` (it is more efficient with some compilers).
- To make a string object empty, `s.resize(0)` (if you want to keep the current reserved space) or `s = string()` (if you don't) seem the best options.
- Use `std::string::assign()` rather than building a temporary string object and assigning that. For example, `foo = std::string(ptr, len);` is better written as `foo.assign(ptr, len);`.
- It's generally better to build up strings using += rather than combining series of components with +. So `foo = a + " and " + c` is better written as `foo = a; foo += " and "; foo += c;`. It's possible for compilers to handle the former without a lot of temporary string objects by returning a proxy object to allow the concatenation to happen lazily, but not all compilers do this, and it's likely to still have some overhead. Note that GCC 4.1 seems to produce larger code in some cases for the latter approach, but it's a definite win with GCC 4.4.

- `std::string(1, '\0')` seems to be slightly more efficient than `std::string("", 1)` for constructing a `std::string` containing a single ASCII nul character.

Use of C++ Features

C++11

As of Xapian 1.3.3, a compiler with decent support for C++11 is required to build Xapian. We currently aim to allow users to use a non-C++11 compiler to build code which uses Xapian.

There are now several compilers with good C++11 support, but there are a few shortfalls in commonly deployed versions of most of them. Often we can work around this, and we should do where the effort is low compared to the gain (so a compiler version which is widely used is more worth supporting than one which is hardly used by anyone).

However, we shouldn't have to jump through hoops to cater for compilers where their authors aren't putting in the effort to keep up with the language standards.

Please avoid the following C++11 features for the time being:

- `std::to_string()` - this is completely missing on current versions of mingw and cygwin - in the library, you can `#include "str.h"` and then use the `str()` function instead for most cases. This is also usually faster than `std::to_string()`.

C++ features we assume

We assume that all compilers will correctly implement the following, so it's safe to rely on them when working on Xapian.

- We assume `<sstream>` is available. GCC < 2.95.3 didn't have it but GCC 2.95.3 includes a backported version. We aren't aware of any other compilers still in use which lack it.
- Non-".h" versions of standard ISO C++ headers (e.g. `#include <list>` rather than `#include <list.h>`). We aren't aware of any compiler still in use which lacks these, and GCC 4.3 no longer has the old versions. If there are any, we could add a directory full of forwarding headers to work around this issue.
- Standard header `<limits>` (for `numeric_limits<>`) - for GCC, this was added in GCC 3.0.
- Standard header `<streambuf>` (GCC < 3.0 only has `<streambuf.h>`).

Exceptions

When catching an exception which is an object, do it by const reference, so like this:

```
try {
    foo();
} catch (const ErrorClass& e) {
    bar(e);
}
```

Catching by value is bad because it "slices" the object if an object of a derived type is thrown. Even if derived types aren't a worry, it also causes the copy constructor to be called needlessly. More information is available in a Standard C++ FAQ entry.

A const reference is preferable to a non-const reference as it stops the object being inadvertently modified. In the rare cases when you want to modify the caught object, a non-const reference is OK.

Exceptions should be avoided except for truly exceptional situations, since throwing and handling them has a significant cost. It also generally makes the API easier to understand, and client code easier to read.

Include ordering for source files

To help us move towards a consistent ordering of `#include` lines in source files, please follow the following policy when ordering them:

- `#include <config.h>` should be first, and use <> not "" (as recommended by the autoconf manual). Always include `config.h` from C/C++ source files, but don't include it from header files - the autoconf manual recommends that it should be included first, so including it from headers is either redundant, or may hide a missing config.h include in the source file the header was included from (better to get an error in this case).
- The header corresponding to the source file should be next. This means that compilation of the library ensures that each header with a corresponding source file is "self supporting" (i.e. it implicitly or explicitly includes all of the headers it requires).
- External xapian-core headers, alphabetically. When included from other external headers, use <> to reduce problems with finding headers in the user's source tree by mistake. In sources and internal headers, use "" (?) - practically this makes no difference as we have `-I` for srcdir and builddir, but <> suggests installed header files so "" seems more natural).
- Internal headers, alphabetically (using "").
- "Safe" versions of library headers (include these first to avoid issues if other library headers include the ones we want to wrap). Use "" and order alphabetically.
- Library headers, alphabetically.
- Standard C++ headers, alphabetically. Use the modern (no .h suffix) names.

Branch Prediction Hints

For compilers which support `__builtin_expect()` (GCC >= 3.0 and some others) you can provide manual hints to assist branch prediction. We've wrapped these in macros which evaluate to just their argument for compilers which don't support `__builtin_expect()`.

Within the xapian-core library code, you can mark the expressions in `if` and `while` statements as `rare` (if the condition is rarely true) or `usual` (if the condition is usually true).

For example:

```
if (rare(something_unusual())) deal_with_it();

while (usual(!end_condition()) keep_going();
```

It's easy to make incorrect assumptions about where hotspots are and which branches are usually taken or not, so except for really obvious cases (such as `if (!consistency_check()) throw_exception();`) you should benchmark that new `rare` and `usual` hints help rather than hinder before committing them to the repository. It's also likely to be a waste of effort to add them outside of areas of code which are executed very frequently.

Don't expect miracles - the first 15 uses added saved approximately 1%.

If you know how to implement the `rare` and `usual` macros for other compilers, please let us know.

Use of Assert

Use Assert to perform internal consistency checks, and to check for invalid arguments to functions and methods (e.g. passing a NULL pointer when this isn't permitted). It should *NOT* be used to check for error conditions such as file read errors, memory allocation failing, etc (since we want to perform such checks in non-debug builds too).

File format errors should also not be tested with Assert - we want to catch a corrupted database or a malformed input file in a non-debug build too.

There are several variants of Assert:

- `Assert(P)` – asserts that expression P is true.
- `AssertRel(a,rel,b)` – asserts that (a rel b) is true - rel can be a boolean relational operator, i.e. one of ==, !=, >, >=, <, <=. The message given if the assertion fails reports the values of a and b, so `AssertRel(a,<,b);` is more helpful than `Assert(a < b);`
- `AssertEq(a,b)` – shorthand for `AssertRel(a,==,b)`.
- `AssertEqDouble(a,b)` – asserts a and b differ by less than `DBL_EPSILON`
- `AssertParanoid(P)` – a particularly expensive assertion. If you want a build with Asserts enabled, but without a great performance overhead, then passing –enable-assertions=partial to configure and AssertParanoids won't be checked, but Asserts will. You can also use `AssertRelParanoid` and `AssertEqParanoid`.

An earlier assert, `CompileTimeAssert(P)`, has now been removed, since we require C++11 support from the compiler, and C++11 added `static_assert`.

2.2.2 Portability

C++ Portability Issues

Web Resources

The C++ Super-FAQ covers many frequently asked C++ questions.

Header Portability Issues

`<fcntl.h>`

> Don't directly `#include <fcntl.h>` - instead `#include "safefcntl.h"`.
>
> The main reason for this is that when using certain compilers on certain versions of Solaris, fcntl.h does `#define open open64`. Sadly this breaks C++ code which has methods called open (as we do). There's a cunning workaround for this problem in `common/safefcntl.h`.
>
> Also, `safefcntl.h` ensures the `O_BINARY` is defined (to 0 if not required) so calls to `open()` and `creat()` can specify `O_BINARY` unconditionally for the benefit of platforms which discriminate between text and binary files.

`<windows.h>`

> Don't directly `#include <windows.h>` - instead `#include "safewindows.h"` which reduces the bloat of header files included and prevents some of the more egregious namespace pollution. It also defines any constants we need which might be missing in older versions of the mingw headers.

`<winsock2.h>`

Don't directly `#include <winsock2.h>` - instead `#include "safewinsock2.h"`. This ensure that safewindows.h is included before <winsock2.h> to avoid winsock2.h including windows.h without our namespace pollution reducing workarounds.

`<sys/select.h>`

Don't directly `#include <sys/select.h>` - instead `#include "safesysselect.h"` which supports older UNIX platforms which predate POSIX 1003.1-2001 and works around a problem on Solaris.

`<sys/socket.h>`

Don't directly `#include <sys/socket.h>` - instead `#include "safesyssocket.h"` which supports older UNIX platforms which predate POSIX 1003.1-2001 and works on Windows too.

`<sys/stat.h>`

Don't directly `#include <sys/stat.h>` - instead `#include "safesysstat.h"` which under MSVC enables stat to work on files > 2GB, defines the missing POSIX macros S_ISDIR and S_ISREG, pulls in <direct.h> for mkdir() (which is provided by sys/stat.h under UNIX) and provides a compatibility wrapper for mkdir() which takes 2 arguments (so code using mkdir can always just pass two arguments).

`<sys/wait.h>`

To get *WEXITSTATUS* or *WIFEXITED* defined, `#include "safesyswait.h"`. Note that this won't provide *waitpid()*, etc on Microsoft Windows, since these functions are only really useful to use when *fork()* is available.

`<unistd.h>`

Don't directly `#include <unistd.h>` - instead `#include "safeunistd.h"` – MSVC doesn't even HAVE unistd.h!

The various "safe" headers are maintained in `xapian-core/common`, but also used by Omega. Currently bootstrap sorts out setting up a copy of this subdirectory via a secondary git checkout.

Warning-Free Compilation

Compiling without warnings on every platform is our goal, though it's not always possible to achieve. For example, some GCC 3.x compilers produce the occasional bogus warning (e.g. warning that a variable may be used uninitialised, despite it being initialised at the point of declaration!)

You should consider configure-ing with:

```
./configure CXXFLAGS=-Werror
```

when doing development work on Xapian. This promotes warnings to errors, which should ensure you at least don't introduce new warnings for the compiler you're using.

If you configure with `--enable-maintainer-mode`, and are using GCC 4.1 or newer, this is done for you automatically. This is intended to be an aid rather than a form of automated punishment - it's all too easy to miss a new warning as once a file is compiled, you don't see it unless you modify that file or one of its dependencies.

With Intel's C++ compiler, `--enable-maintainer-mode` also enables `-Werror`. If you know the equivalent of `-Werror` for other compilers, please add a note here, or tell us so that we can add a note.

Miscellaneous Portability Issues

Make sure that the last line of any source file ends with a linefeed character since it's undefined behaviour if it doesn't (most compilers accept it, though at least GCC gives a warning).

Makefile Portability

We don't want to force those building Xapian from the source distribution to have to use GNU make. Requiring GNU make for "make dist" isn't such a problem but it's probably better to use portable constructs everywhere to avoid problems when people move or copy code between targets. If you do make use of non-portable constructs where it's OK, add a comment noting the special circumstances which justify doing so.

Here's an incomplete list of things to avoid:

- Don't use "$(RM)" - it's defined by GNU make, but using it actually harms portability as other makes don't define it. Use plain "rm" instead.
- Don't use "%" pattern rules - these are GNU make specific. Use an implicit rule (e.g. ".c.o:") if you can. Otherwise, write out each version explicitly.
- Don't use "$<" except in implicit rules. This is an annoying restriction, as using "$<" makes it much easier to make VPATH builds work. But it's only portable in implicit rules. Tips for rewriting - if it's a source file, write it as:

  ```
  $(srcdir)/foo.ext
  ```

  If it's a generated object file or similar, just write the name as is. The tricky case is a generated file which isn't in git but is shipped in the distribution tarball, as such a file could be in either the source or build tree. Use this trick to make sure it's found whichever directory it's in:

  ```
  `test -f foo.ext || echo '$(srcdir)/'`foo.ext
  ```

- Don't use "exit 0" to make a rule fail. Use "false" instead. BSD make doesn't like "exit 0" in a rule.
- Don't use make conditionals. Automake offers conditionals which may be of use, and these are implemented to work with any make. See the automake manual for details, and a few caveats.
- The list of portable utilities is:

  ```
  cat cmp cp diff echo egrep expr false grep install-info
  ln ls mkdir mv pwd rm rmdir sed sleep sort tar test touch true
  ```

  Note that versions of these (GNU versions in particular) support switches which aren't portable - notably, "test -r" isn't portable; neither is "cp -a". And note that "mkdir -p" isn't portable - the semantics vary. The autoconf manual has some useful information about writing portable shell code (most of it not specific to autoconf).
- Don't use "include" - it's not present in BSD make (at least some versions have ".include" instead, but that doesn't really seem to help...) Automake provides a configure-time include, which may provide a replacement for some uses of "include".
- It appears that BSD make only supports VPATH for implicit rules (e.g. ".c.o:") - there's certainly a restriction there which is not present in GNU make. We used to try to work around this, but now we use AM_MAINTAINER_MODE to disable rules which are only needed by those developing Xapian (these were the rules which caused problems). And we recommend those developing Xapian use GNU make to avoid problems.
- Rules with multiple targets can cause problems for parallel builds. These rules are really just a shorthand for multiple rules with the same prerequisites and commands, and it is fine to use them in this way. However, a common temptation is to use them when a single invocation of a command generates multiple output files,

by adding each of the output files as a target. Eg, if a swig language module generates xapian_wrap.cc and xapian_wrap.h, it is tempting to add a single rule something like:

```
# This rule has a problem
xapian_wrap.cc xapian_wrap.h: xapian.i
        SWIG_commands
```

This can result in SWIG_commands being run twice, in parallel. If SWIG_commands generates any temporary files, the two invocations can interfere causing one of them to fail.

Instead of this rule, one solution is to pick one of the output files as a primary target, and add a dependency for the second output file on the first output file:

```
# This rule also has a problem
xapian_wrap.h: xapian_wrap.cc
xapian_wrap.cc: xapian.i
        SWIG_commands
```

This ensures that make knows that only one invocation of SWIG_commands is necessary, but could result in problems if the invocation of SWIG_commands failed after creating xapian_wrap.cc, but before creating xapian_wrap.h. Instead, we recommend creating an intermediate target:

```
# This rule works in most cases
xapian_wrap.cc xapian_wrap.h: xapian_wrap.stamp
xapian_wrap.stamp: xapian.i
        SWIG_commands
        touch $@
```

Because the intermediate target is only touched after the commands have executed successfully, subsequent builds will always retry the commands if an error occurs. Note that the intermediate target cannot be a "phony" target because this would result in the commands being re-run for every build.

However, this rule still has a problem - if the xapian_wrap.cc and xapian_wrap.h files are removed, but the xapian_wrap.stamp file is not, the .cc and .h files will not be regenerated. There is no simple solution to this, but the following is a recipe taken from the automake manual which works. For details of *why* it works, see the section in the automake manual titled "Multiple Outputs":

```
# This rule works even if some of the output files were removed
xapian_wrap.cc xapian_wrap.h: xapian_wrap.stamp
## Recover from the removal of $@.  A full explanation of these rules is␣
↪in
## the automake manual under the heading "Multiple Outputs".
        @if test -f $@; then :; else \
          trap 'rm -rf xapian_wrap.lock xapian_wrap.stamp' 1 2 13 15; \
          if mkdir xapian_wrap.lock 2>/dev/null; then \
            rm -f xapian_wrap.stamp; \
            $(MAKE) $(AM_MAKEFLAGS) xapian_wrap.stamp; \
            rmdir xapian_wrap.lock; \
          else \
            while test -d xapian_wrap.lock; do sleep 1; done; \
            test -f xapian_wrap.stamp; exit $$?; \
          fi; \
        fi
xapian_wrap.stamp: xapian.i
        SWIG_commands
        touch $@
```

- This is actually a robustness point, not portability per se. Rules which generate files should be careful not

to leave a partial file in place if there's an error as it will have a timestamp which leads make to believe it's up-to-date. So this is bad:

```
foo.cc: script.pl
        $PERL script.pl > foo.cc
```

This is better:

```
foo.cc: script.pl
        $PERL script.pl > foo.tmp
        mv foo.tmp foo.cc
```

Alternatively, pass the output filename to the script and make sure you delete the output on error or a signal (although this approach can leave a partial file in place if the power fails). All used Makefile.am-s and scripts have been checked (and fixed if required) as of 2003-07-10 (didn't check xapian-bindings).

- Another robustness point - if you add a non-file target to a makefile, you should also list it in ".PHONY". Otherwise your target won't get remade reliably if someone creates a file with the same name in their tree. For example:

```
.PHONY: hello goodbye

hello:
      echo hello

goodbye:
      echo goodbye
```

And lastly a style point - using "@" to suppress echoing of commands being executed removes choice from the user - they may want to see what commands are being executed. And if they don't want to, many versions of make support the use "make -s" to suppress the echoing of commands.

Using `@echo` on a message sent to stdout or stderr is acceptable (since it avoids showing the message twice). Otherwise don't use `@` - it makes it harder to track down problems in the makefiles.

2.2.3 Other conventions

Configure Options

Especially for a library, compile-time options aren't a good solution for how to integrate a new feature. An increasingly large number of users install pre-built binary packages rather than building from source, and unless the package is capable of being split into modules, the packager has to choose a set of compile-time options to use. And they'll tend to choose either the standard ones, or perhaps a broader set to try to keep everyone happy. For a library, similar issues occur when installing from source as well - the sysadmin must choose the options which will keep all users happy.

Another problem with compile-time options is that it's hard to ensure that a change doesn't break compilation under some combination of options without actually building and running the test-suite on all combinations. The fewer compile-time options, the more likely the code will compile with every combination of them.

So please think carefully before adding more compile-time options. They're probably OK for experimental features (but should go away once a feature is no longer experimental). Options to instrument a build for special purposes (debug, profiling, etc) are also acceptable. Disabling whole features probably isn't (e.g. the `--disable-backend-XXX` options we already have are dubious, though being able to disable the remote backend can be useful when trying to get Xapian going on a platform).

Naming of Scripts

Scripts generally should *not* have an extension indicating the language they are currently implemented in (e.g. `runtest` rather than `runtest.sh` or `runtest.pl`). The problem with such an extension is that if we decide to reimplement the script in a different language, we either have to rename the script (which is annoying as people will be used to the name, and may have embedded it in their own scripts), or we have a script with a confusing name (e.g. a Python script with extension `.pl`).

The above reasoning doesn't apply to scripts which have to be in a particular language for some reason, though for consistency they probably shouldn't get an extension either, unless there's a good reason to have one.

2.2.4 Marking Features as Deprecated

In the API headers, a feature (a class, method, function, enum, typedef, etc) can be marked as deprecated by using the `XAPIAN_DEPRECATED()` or `XAPIAN_DEPRECATED_CLASS` macros. Note that you can't deprecate a preprocessor macro.

For compilers with a suitable mechanism (such as GCC, clang and MSVC) this causes compile-time warning messages to be emitted for any use of the deprecated feature. For compilers without support, the macro just expands to its argument.

Sometimes a deprecated feature will also be removed from the library itself (particularly something like a typedef), but if the feature is still used inside the library (for example, so we can define class methods), then use `XAPIAN_DEPRECATED_EX()` or `XAPIAN_DEPRECATED_CLASS_EX` instead, which will only issue a warning in user code (this relies on user code including xapian.h and library code including individual headers)

You must add this line to any API header which uses `XAPIAN_DEPRECATED()` or `XAPIAN_DEPRECATED_CLASS`:

```
#include <xapian/deprecated.h>
```

When marking a feature as deprecated, document the deprecation in `docs/deprecation.rst`. When actually removing deprecated features, please tidy up by removing the inclusion of `<xapian/deprecated.h>` from any file which no longer marks any features as deprecated.

The `XAPIAN_DEPRECATED()` macro should wrap the whole declaration except for the semicolon and any "definition" part, for example:

```
XAPIAN_DEPRECATED(int old_function(double arg));

class Foo {
  public:
    XAPIAN_DEPRECATED(int old_method());

    XAPIAN_DEPRECATED(int old_const_method() const);

    XAPIAN_DEPRECATED(virtual int old_virt_method()) = 0;

    XAPIAN_DEPRECATED(static int old_static_method());

    XAPIAN_DEPRECATED(static const int OLD_CONSTANT) = 42;
};
```

Mark a class as deprecated by inserting `XAPIAN_DEPRECATED_CLASS` after the class keyword like so:

```
class XAPIAN_DEPRECATED_CLASS Foo {
  public:
```

(continues on next page)

(continued from previous page)

```
    Foo() { }

    // ...
};
```

You can simply mark a method defined inline in a class with `XAPIAN_DEPRECATED()` like so:

```
class Foo {
  public:
    // This failed to compile with GCC 3.3.5.
    XAPIAN_DEPRECATED(int old_inline_method()) { return 42; }
};
```

Implementing Deprecation Warnings for the Bindings

Currently we don't have an equivalent of the C++ `XAPIAN_DEPRECATED()` macro for the bindings, but it would be good to have. Here are some notes on how this could be achieved for various languages we support:

- PHP now has a `E_USER_DEPRECATED` error level - in a deprecated method we could do:

```
trigger_error(
    'World::hi() is deprecated, use World::hello() instead',
    XAPIAN_DEPRECATED
);
```

- Python has `DeprecationWarning`, which we were using in 1.2.x a bit:

```
warnings.warn(
    'World::hi() is deprecated, use World::hello() instead',
    DeprecationWarning,
)
```

- Ruby - there are external libraries to handle deprecation warnings, but the simplest option without external dependencies seems to be:

```
warn "[DEPRECATION] World::hi() is deprecated, use World::hello() instead"
```

- Perl:

```
use warnings;
warnings::warnif('deprecated', 'World::hi() is deprecated, use World::hello()␣
↪instead');
```

- Java has an annotation, `@Deprecated`, which can be used to indicate deprecation.

It would be great (but probably hard) to reuse the `XAPIAN_DEPRECATION()` markers. Perhaps parsing the doxygen XML for `@deprecated` markers would be simpler?

Also, it would be good if the warnings could be turned off easily, as runtime deprecation warnings can be annoying for end users.

2.2.5 API Structure Notes

We use reference counted pointers for most API classes. These are implemented using `Xapian::Internal::intrusive_ptr`, the implementation of which is exposed for efficiency, and be-

cause it's unlikely we'll need to change it frequently, if at all.

For the reference counted classes, the API class (e.g. `Xapian::Enquire`) is really just a wrapper around a reference counted pointer. This points to an internal class (e.g. `Xapian::Enquire::Internal`). The reference counted pointer is a member variable of the API class called internal. Conceptually this member is private, though it typically isn't declared as private (this is to avoid littering the external headers with friend declarations for non-API classes).

There are a few exceptions to the reference counted structure, such as `MSetIterator` and `ESetIterator` which have an exposed implementation. Tests show this makes a substantial difference to speed (it's ~20% faster) in typical cases of iterator use.

The postfix `operator++` for iterators should be implemented inline in terms of the prefix form as described by Joe Buck on the gcc mailing list:

```
class some_iterator {
public:
    // ...
    some_iterator& operator++();
    some_iterator operator++(int) {
        some_iterator tmp = *this;
        operator++();
        return tmp;
    }
};
```

> The compiler is allowed to assume that the copy constructor only does a copy, and to optimize away unneeded copy operations. The result in this case should be that, for some_iterator above, using the postfix operator without using the result should give code equivalent to using the prefix operator.
>
> [With modern compilers], you should find that this style comes very close to eliminating any penalty from "incorrect" use of the postfix form.

Xapian's PostingIterator, TermIterator, PositionIterator, and ValueIterator all have only one data member which fits in a register.

2.3 Xapian documentation

2.3.1 Available documentation

There are a number of types of documentation available for Xapian.

1. User documentation that explains how people will use Xapian, typically either as a "full worked example", or as a deep dive into a particular feature.

   Most of this documentation lives in the *Xapian user guide*, which is available online (built with python examples) and as a git repo on Github which you can build for a range of different languages.

2. API documentation, which is built from specially-formatted comments in Xapian's C++ header files. This allows people writing code that uses Xapian to check on details of the classes and methods they are using, as well as to explore what features there are in Xapian.

3. Developer documentation, for people who want to contribute directly to Xapian. This guide is part of that (and is also available on github).

   There is also some developer documentation available elsewhere, particularly in the main Xapian source repository – look in particular in `xapian-core/docs`. Some of this is being migrated to the developer guide. (Some is also being migrated to the user guide.)

4. The Xapian wiki also contains information, including links to articles and talks that people have written about using Xapian in various ways. Although this isn't always official Xapian documentation, it may be helpful. Some of it may now be quite out of date, so if something you find there doesn't work, you may want to check the official documentation to see if you can figure out how to change the advice or code you find to work with a recent version of Xapian.
5. Omega, a standalone indexing and web search tool built using Xapian, has its own documentation bundled with its source code, and available on our website.
6. The language bindings, which allow you to use Xapian from programming languages others than C++, have their own documentation which mostly focusses on ways that the API is different from C++ in those languages.

Building the documentation

The user guide and developers guide can both be built using Sphinx; there is more information in a `README.md` in each repository.

If you're building from a clone of Xapian, then the API docs will be built automatically as part of building Xapian. (If you've downloaded a source tarball, then the API docs will already have been built for you.) The API docs of the latest release are also available on the Xapian website, along with some of the developer documentation that isn't in this developer guide.

Similarly, documentation for Omega and the bindings will be built when you build them from source, and is also available online.

2.3.2 Updating the documentation

As you work on changes to Xapian, it's important to write or update documentation so people will know how to use your work. Looking through the list of documentation above, you can see there are a number of places where you could add helpful information. The main ones are:

- The user guide, which is where most new users expect to learn how to use Xapian.
- The API documentation, in doxygen comments in the source code.
- The developer guide, if you make changes which future developers will need to know.

  Note that you can also provide information to future developers in code comments, and in the commit messages you write. For many types of information, they are more convenient for other developers to find when they need. Try to add information where you yourself would expect to find it.

Don't worry about getting the words exactly right, particularly if English isn't your first language. If you can get down the information that needs to be there, someone else can tidy up the language if necessary.

2.3.3 PDF versions of docs

If you want PDF versions of the API documentation, you'll need to install some more tools:

- gs (part of Ghostscript)
- pdflatex (in texlive-latex-base on Debian/Ubuntu)
- epstopdf (in texlive-extra-utils on Debian/Ubuntu)
- makeindex (in texlive-binaries on Debian/Ubuntu, or texlive-base-bin for older releases)

Note that pdflatex, epstopdf, gs, and makeindex must all currently be on your path (as specified by the environment variable PATH), since doxygen will look for them there.

For a recent version of Debian or Ubuntu, this command should work:

```
apt-get install ghostscript texlive-latex-base texlive-extra-utils \
    texlive-binaries texlive-fonts-extra texlive-fonts-recommended \
    texlive-latex-extra texlive-latex-recommended
```

On macOS, you can install MacTeX.

Once you've got the extra tools you need, you can build the PDF documentation:

```
(cd xapian-core && make -C docs apidoc.pdf)
```

2.4 Testing Xapian

Xapian contains three types of code, which require different kinds of tests. The C++ libraries (`xapian-core` and `xapian-letor`) have a common test harness, and have extensive test suites written in C++. Omega, our pre-packaged web search app, has its own tests. Finally, the language bindings have "smoke tests" to check some basic functionality, and may have further tests for language-specific code, such as iterators (which generally work differently in language bindings than they do in C++, to make them more familiar to existing users of that programming language).

Generally, tests for any part of Xapian can be run using `make check`.

2.4.1 Testing the libraries

Both `xapian-core` and `xapian-letor` have automated test suites covering their APIs and various internals of the API implementations. They are implemented using the same C++ test harness, and so have similar features and tests in both should feel familiar once you get used to one.

There are a few environment variables which the test harness checks for which you might find useful:

XAPIAN_TESTSUITE_SIG_DFL By default, the testsuite harness catches signals and handles them gracefully - the current test is failed, and the testsuite moves onto the next test. If you want to suppress this (some debugging tools may work better if the signal is not caught) set the environment variable `XAPIAN_TESTSUITE_SIG_DFL` to any value to prevent the testsuite harness from installing its own signal handling.

XAPIAN_TESTSUITE_OUTPUT By default, the testsuite harness uses ANSI escape sequences to give colour output if stdout is a tty. You can disable this feature by setting `XAPIAN_TESTSUITE_OUTPUT=plain`. Alternatively, piping the output, such as through `cat` or `more`, will have the same effect.

Auto-detection can be explicitly specified with `XAPIAN_TESTSUITE_OUTPUT=auto` (or empty). Any other value forces the use of colour.

Colour output is always disabled on Microsoft Windows, so `XAPIAN_TESTSUITE_OUTPUT` has no effect there.

XAPIAN_TESTSUITE_LD_PRELOAD The runtest script will add this to `LD_PRELOAD` if it is set, allowing you to easily load `LD_PRELOAD` libraries when running the testsuite. The original intended use was to allow use of libeatmydata which makes `fsync` and related calls no-ops, but configure now checks for the eatmydata wrapper script and this is used automatically.

However, there may be other `LD_PRELOAD` libraries which are useful, so we've left the machinery in place.

Running test programs

To run all tests, use `make check`. You can also run just the subset of tests which exercise the inmemory, remote progserver, remote TCP, multi-database, glass or honey backends using `make check-inmemory`, `make check-remoteprog`, `make check-remotetcp`, `make check-multi`, `make check-glass` or `make check-honey` respectively.

Also, `make check-remote` will run the tests on both variants of the remote backend, and `make check-none` will run those tests which don't use any backend. These are handy shortcuts when doing development work on a particular backend.

Running individual tests

The runtest script (in the tests subdirectory) takes care of the details of running the test programs (including setting up the environment so they work when srcdir != builddir and handling libtool dynamically linked binaries). To run a test program by hand (rather than via make) just use:

```
./runtest ./apitest
```

You can specify options and arguments. Individual test programs optionally take one or more test names as arguments, and you can also pass `-v` to get more verbose output from failing tests, e.g.:

```
./runtest ./apitest -v deldoc1
```

If the number of the test is omitted, all tests with that basename are run, so to run deldoc1, deldoc2, etc:

```
./runtest ./apitest deldoc
```

Running under a debugger or other tool

You can also use runtest to run a test program under gdb (or most other tools):

```
./runtest gdb ./apitest -v deldoc1
./runtest valgrind ./apitest -v deldoc1
```

Some test programs take special arguments - for example, you can restrict apitest to the glass backend using `-bglass`.

Speeding up the testsuite with eatmydata

The testsuite does a lot of small database operations, and the calls to fsync, fdatasync, etc which Xapian makes by default can slow down testsuite runs substantially. There's a handy `LD_PRELOAD` library called eatmydata, which can help here, by turning fsync and related calls into no-ops.

You need a version of eatmydata with the eatmydata wrapper script (version 37 or newer), and then configure should auto-detect it and it'll get used when running the testsuite (via runtest). If you wish to disable this auto-detection for some reason, you can run configure with:

```
./configure EATMYDATA=
```

Or you can disable use of eatmydata during a particular run of "make check" like so:

```
make check EATMYDATA=
```

Or disable it while running a test directly (under sh or bash):

```
EATMYDATA= ./runtest ./apitest
```

2.4.2 Writing library tests

Note: Generally we don't "unit test" the lower levels of Xapian

A common convention in a lot of software is to write unit tests, testing individual units of source code. In Xapian we generally do not test the lower levels separately. Instead, we write tests that use Xapian's public API.

So for instance, we don't write unit tests for our `::Internal` classes. So if you make a change to `Xapian::Weight::Internal` you wouldn't write a test for that change directly. However, your changes are motivated by a desire to add certain behaviour to the library. This generally will be new functionality via the API, which you can write a test for. Sometimes it will be what is sometimes called "non-functional behaviour", such as speed of operation or memory usage. These may be harder to write tests for, and are worth *discussing with other members of the community* to figure out the best way to test them.

This means that our tests are often testing a more complex bundle of functionality than you may be used to from unit tests. You can think of most of Xapian's test suite as automated functional tests.

Test programs live in `tests/`. They mostly use a standard test harness, in `tests/harness/`, which wraps each test, reports results, and generally packages things up nicely. The test harness counts how many testcases pass/fail/skip, catches signals and unhandled exceptions, and so forth. It can also also check for memory leaks and accesses to uninitialised values by making use of valgrind, for platforms which valgrind supports (configure automatically enables use of valgrind if a suitably recent version is detected).

A typical test program has three parts: the tests themselves (at the top), a table of tests (at the bottom), and a tiny main which sets the test harness in motion. It uses the table to figure out what the tests are called, and what function to call to run them.

Incidentally, when fixing bugs, it's often better to write the test before fixing the bug. Firstly, it's easier to assure yourself that the bug is (a) genuine, and (b) fixed, because you see the test go from fail to pass (though sometimes you don't get the testcase quite right, so this isn't doesn't always work as well as it should). Secondly you're more likely to write the test carefully, because once you've fixed something there's often a feeling that you should commit it for the good of the world, which tends to distract you.

The framework is done for you, so you don't need to worry about that much. You are responsible for doing two things:

- writing a minimal test or tests for the feature
- adding that test to the list of tests to be run

Adding the test is simple. There's a `test_desc` array in each file that comprises a set of tests (I'll come to that in a minute), and you just add another entry. The entry is an array consisting of a name for the test and a pointer to the function that is the test. Easy. The procedure is even simpler for apitest tests - there you just use `DEFINE_TESTCASE` to define your new testcase, and a script picks it up and makes sure it is run.

Look at the bottom of `tests/stemtest.cc` for the `test_desc` array. Now look up about 20 lines to where the test functions are defined. You need to write a function like these. There are a bunch of macros to help you perform standards testing tasks, such as `TEST_EQUAL`, which are all in `tests/harness/testsuite.h`. They're pretty simple to use.

API tests

The most important test system for most people will be `apitest`. This also uses the test harness, but has several tables of tests to be run depending what facilities each backend supports. A lot of the work is done by macros and

helper functions, which may make it hard to work out quite what is going on, but make life easier once you've grasped what's going on. The `main()` function and other bits are in `apitest.cc`, and tests themselves are in various other C++ files starting `api_`. Each one of these has its own tables for various different groups of tests (eg: `api_db.cc`, which performs tests on the API that require a database backend, has basic tests, a few specialised groups that only contain one or two tests, tests that require a writable database, tests that require a local database, and finally tests that require a remote database).

To add a new api test, figure out what the test will be dependent on and put it in the appropriate place (eg: if adding a test for a bug that occurs while writing to a database, you want a writable database, so you add a test to `api_db.cc` and reference it in the `writabledb_tests` table).

Currently, there's `api_nodb.cc` (no db required, largely testing query construction and boundary conditions), `api_posdb.cc` (db with positional information required) and `api_db.cc` (everything else, with lots of subgroups of tests). It's easiest to base a test on an existing one.

You'll notice in `apitest.cc` that it runs all appropriate test groups against each backend that is being built. The backends are inmemory, multi, glass, honey, remoteprog and remotetcp. If you need to create a new test group with different requirements to any current ones, put it in the appropriate `api_` file (or create a new one, and add it into Makefile.am) and remember to add the group to all pertinent backends in `apitest.cc`.

Test databases

Many of the automated tests work by building a small test database and then testing a particular library feature against it. To make things easier, the test harness provides a way of doing this without having to write indexing code for every test.

Typically you use this by starting your test case with:

```
Xapian::Database db(get_database("dataset"));
```

which would then index the data in `tests/testdata/dataset.txt` and return an open database object.

You can also get an empty writable database, giving it a name:

```
Xapian::WritableDatabase db(get_named_writable_database("testdbname"));
```

The actual database files are generally put into `tests/.<dbtype>` using either the name (for `get_named_writable_database()` or the dataset(s) used. So you can end up with database paths such as:

```
tests/.glass/db__apitest_allterms
tests/.honey/db__apitest_allterms
```

The various functions that support this are declared in the header `tests/apitest.h`, and `tests/harness/backendmanager.h` contains doc comments that will help. (The functions pass through to the particular backend manager being used.)

2.4.3 Debugging the libraries

Extra options to give to configure

Note: Non-developer configure options are described in `INSTALL` files in the source tree.

You will probably want to use some of these if you're going to be developing Xapian.

--enable-assertions This enables compiling of assertion code which will throw Xapian::AssertionError if the code detects violating of preconditions, postconditions, or fails other consistency checks.

--enable-assertions=partial This option enables a subset of the assertions enabled by "–enable-assertions", but not the most expensive. The intention is that it should be suitable for use in a real-world system for tracking down problems without imposing too much of an overhead (but note that we haven't yet performed timings to measure the overhead...)

--enable-log This enables compiling code into the library which generates verbose debugging messages. See *Debugging messages*.

--enable-log=profile In 1.2.0 and earlier, this used to use the debug logging macros to report to stderr how long each method takes to execute. This feature was removed in 1.2.1 - you are likely to get better results using dedicated profiling tools - for more information see: https://trac.xapian.org/wiki/ProfilingXapian

Debugging messages

If you configure with `--enable-log`, lots of places in the code generate debugging messages to tell us what they're up to - this information can be very useful for debugging both the Xapian library and code which uses it. But the quantity of information generated is potentially vast so there's a mechanism to allow you to select where to store the log and which types of message you're interested by setting environment variables. You can:

- set `XAPIAN_DEBUG_LOG` to be the path to a file that you would like debugging output to be appended to, or to the special value `-` to indicate that you would like debugging output to be sent to stderr. Unless `XAPIAN_DEBUG_LOG` is set, no debug logging will be performed. Occurrences of `%p` in `XAPIAN_DEBUG_LOG` will be replaced with the current process-id.

  If you're debugging a crash and want to avoid losing the most recent log messages then include `%!` in XAPIAN_DEBUG_LOG (which is replaced with the empty string). This will cause the log file to be opened with `O_DSYNC` or `O_SYNC` or similar if running on a platform that supports a suitable mechanism. In 1.4.10 and earlier this was on by default (and `%!` has no special meaning) but it can incur a significant performance overhead and in most cases isn't necessary.

- set `XAPIAN_DEBUG_FLAGS` to a string of capital letters indicating the types of debugging message you would like to display (the default is to log calls to API functions and methods). These letters are shown in the first column of the log output, and are also listed in `common/debuglog.h`. If the first character is `-`, then the letters indicate those categories of message *not* be shown instead. As a consequence of this, setting `XAPIAN_DEBUG_FLAGS=-` will give you all debugging messages.

These environment variables only have any effect if you ran configure with the `--enable-log` option.

The format is:

```
<message type> <pid> [<this>] <message>
```

For example:

```
A 16747 [0x57ad1e0] void Xapian::Query::Internal::validate_query()
```

Each nested call adds another space before the `[` so you can easily see which function call and return messages correspond.

Using various debugging, profiling, and leak-finding tools

Note: If you have runes for using other tools, please add them to this section, or send them to us so we can.

libstdc++ debug mode

GCC's `libstdc++` supports a debug mode, which checks for various misuses of the STL - to enable this, define _GLIBCXX_DEBUG when building Xapian:

```
./configure CPPFLAGS=-D_GLIBCXX_DEBUG
```

See their documentation of this option.

Note: All C++ code must be compiled with this defined or you'll get problems. Xapian's API headers include a check that the same setting is used when building code using Xapian as was used to build Xapian.

Using gdb

To use gdb, no special build options are required, but make sure you compile with debugging information (on by default for GCC). You'll probably find debugging easier if you compile without optimisation (with optimisation, line numbers in error messages can be confusing due to code inlining, etc, and the values of some variables can't be printed because they've been eliminated from the code completely):

```
./configure CXXFLAGS='-O0 -g'
```

Using gprof and other profiling tools

To enable profiling for gprof:

```
./configure CXXFLAGS=-pg LDFLAGS=-pg
```

To use Purify (a proprietary tool):

```
./configure CXXLD='purify c++' --disable-shared
```

To use Insure (another proprietary tool):

```
./configure CXX=insure
```

Using lcov

You can use lcov (at least version 1.10) to generate a test coverage report. You ideally want lcov 1.11 or later, since 1.11 includes patches to reduce memory usage significantly - lcov 1.10 would run out of memory in a 1GB VM.

See lcov.xapian.org for automatically generated reports for git master.

If you use ccache, you'll need ccache >= 3.2.2 for coverage builds to actually be cached. Since ccache 3.0 (released 2010-06-20) coverage builds are supported, but initially by disabling caching if the coverage options are used. See below for a workaround for ccache < 3.0.

There are three make targets (currently supported in the `xapian-core` and `xapian-letor` directories):

`make coverage-reconfigure` This reruns configure in the source tree with options to configure the build to generate coverage information. See `Makefile.am` for details of the configure options used and why they are needed.

You can specify additional options via `COVERAGE_CONFIGURE_ARGS` on the `make` command line. For example:

- To configure `xapian-letor` to use the in-tree `xapian-core` use:

```
make coverage-reconfigure COVERAGE_CONFIGURE_ARGS=XAPIAN_CONFIG="`pwd`/../
↪xapian-core/xapian-config
```

- If you're using ccache < 3.0 this doesn't support coverage builds. To work around this you can disable use of ccache with:

```
make coverage-reconfigure COVERAGE_CONFIGURE_ARGS=CCACHE_DISABLE=1
```

- On older systems, coverage reports don't seem to work with shared libraries. To work around this disable use of shared libraries with:

```
make coverage-reconfigure COVERAGE_CONFIGURE_ARGS=--disable-shared
```

`make coverage-reconfigure-maintainer-mode` This does the same thing, except the tree is configured in "maintainer mode", which is what you want if generating coverage reports while working on the code.

`make coverage-check` This runs `make check` and generates an HTML report in a directory called `lcov`.

You can specify extra arguments to pass to the `genhtml` tool using `GENHTML_ARGS`, so for example if you plan to serve the generated HTML coverage report from a webserver, you can tell `genhtml` to gzip all generated HTML files and add a suitable `.htaccess` file using:

```
make coverage-check GENHTML_ARGS=--html-gzip
```

Using valgrind

The testsuite can make use of valgrind 3.3.0 or newer to check for memory leaks, reads from uninitialised memory, and some other bugs during tests.

Valgrind doesn't support every platform, but Xapian contains very little platform specific code (and most of what there is is Microsoft Windows specific) so even just testing with valgrind on one platform gives good coverage.

If you have a new enough version of valgrind installed, it's automatically detected by configure and used when running the testsuite. No special build options are required, but make sure you compile with debugging information (on by default for GCC) and the valgrind documentation recommends disabling optimisation (with optimisation, line numbers in error messages can be confusing due to code inlining, etc):

```
./configure CXXFLAGS='-O0 -g'
```

The testsuite runs more slowly under valgrind, so if you wish to disable this auto-detection you can run configure with:

```
./configure VALGRIND=
```

Or you can disable use of valgrind during a particular run of "make check" like so:

```
make check VALGRIND=
```

Or disable it while running a test directly (under sh or bash):

```
VALGRIND= ./runtest ./apitest
```

Current versions of valgrind result in false positives on current versions of macOS, so on this platform configure only enables use of valgrind if it's specified explicitly, for example if valgrind is on your `PATH` you can just use:

```
./configure VALGRIND=valgrind
```

2.4.4 Testing Omega

Omega's test suite can be run in the normal fashion: `make check` within the `omega` directory. It runs a number of smaller tests, most of them unit testing code that Omega relies on.

`atomparsetest`, `htmlparsetest` Test the `AtomParser` and `MyHtmlParser` classes respectively, which parse Atom and HTML files as part of `omindex`.

`csvesctest`, `jsonesctest`, `urlenctest` Test CSV, JSON, and URL escaping, used by the `$csv{}`, `$json{}`, and `$url{}` OmegaScript commands.

`md5test` Test our MD5 hashing routines, used both by the `$hash{}` OmegaScript command and for keeping a hash of indexed file contents (in Xapian document value 1), which allows duplicate documents to be collapsed.

`utf8convertest` Test our UTF8 conversion routines.

Clickmodel tests

All the clickmodel code lives in `omega/clickmodel`, with tests for each model in `omega/clickmodel/tests` and sample data in `omega/clickmodel/testdata`.

Omegascript and functionality tests

We test most Omega functionality by running it against a specific database with a particular Omegascript template. The `omegatest` script provides a lightweight test harness (in shell) for writing test cases. Test databases are created using `scriptindex` (which is easier to drive programmatically than `omindex`).

It takes a bit of time to get used to the way these tests are written, but once you get the hang of it, writing tests for new Omegascript commands, or other Omega functionality, is usually quite easy.

Currently, we do not have any tests for `omindex`.

2.4.5 Testing the language bindings

We generally don't test the entire API with bindings; we rely on the existing tests for `xapian-core` to make sure that the library is working. So bindings tests should concentrate on three things:

1. A basic "smoketest" that all is well. This checks that the bindings can actually be loaded, and that a basic index and search is possible.
2. Anything specific to these bindings. For instance, when writing bindings using SWIG, you usually write a number of "typemaps" that convert between the bindings language's types and the C++ types used by the Xapian API. Testing the use of each typemap at least once is important, and more valuable than trying to test the entire API.

   Note that it's helpful to test use of SWIG's built-in typemaps for the binding language, so that any SWIG changes that cause problems cause test failures and can be identified and addressed.
3. Regression tests for bugs in the bindings.

It's usually practical to do this all in one test file, although for languages that have more sophisticated test frameworks built in it may be easier to split across multiple files.

2.5 Contributing to Xapian

Xapian is an open source project, depending on a community of volunteers. There are lots of ways of contributing to Xapian:

- Join our mailing lists or chat channel and help people out.
- Talk or write about Xapian. By explaining how you've used Xapian you not only help spread the word, but also might learn more about Xapian itself.
- Let us know (either via our bug tracker or by the mailing lists or IRC) if you run into any problems with our documentation, or with bugs you come across while using Xapian.
- Help improve our documentation, either by suggesting changes or by writing up something on our list of missing documentation.
- Contribute new features by working on one of our project ideas
- Tackle an existing bug or feature request, or something you yourself want to see in Xapian.
- Help others get their changes into shape for inclusion in Xapian.

One of the things this guide will do is take you through the process of getting comfortable with the Xapian codebase and submitting your first "patch". There's also lots of more detailed information if you want to get more deeply involved in writing code for Xapian.

2.5.1 Licensing your contributions

If you want a patch to be considered for inclusion in Xapian, you must own the copyright on your changes. Employers often claim copyright on code written by their employees (even if the code is written in their spare time), so please check with your employer if this applies. Be aware that even if you are a student your university may try and claim some rights on code which you write.

Patches which are submitted to Xapian will only be included if the copyright holder(s) dual-license them under each of the following licences:

- GPL version 2 and all later versions (see the file COPYING in `xapian-core` for details).
- MIT/X license:

```
Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to
deal in the Software without restriction, including without limitation the
rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN THE SOFTWARE.
```

The current distribution of Xapian contains many files which are only licensed under the GPL, but we are working towards being able to distribute Xapian under a more permissive license, and are not willing to accept patches which we will have to rewrite before this can happen.

2.5.2 Advice for new contributors

New to Xapian (or even open source)? Don't worry! Here we try to guide you through your first contribution, but if anything is unclear or you want to ask a question, please *get in touch*. This may look a bit daunting the first time, but we're here to help, and a lot of the details will become natural over time.

Checking out and building Xapian

A good way to start learning about Xapian is to check out the code, and get it to build. It's better to use the latest code from the repository rather than a release, as that's what we want the projects to be based on.

We recommend you use Linux or another UNIX-like system for development work, as we're better set up for development on such platforms. In particular we use them ourselves, so can more easily help with any set up issues you may encounter. If you want to run Linux (perhaps virtualised) for development and have no existing preference, we suggest Debian or Ubuntu as our documentation covers these well.

It can take a while to get the code if your network connection is slow, and it may take a while to build and run the testsuite if your computer is slow - while you are waiting, you might want to make a start on the next section.

If you haven't used git before, or want a refresher on "branches", "remotes" and so forth, then this article by James Aylett was written for our students a number of years ago, and may be helpful. There are also a number of free books and resources online, such as Pro Git.

Learn about Xapian's API

It's a good idea to get familiar with Xapian by going through the user guide. The online version has examples in Python, but you can also grab the source and build for other languages; most example code is also available in C++ and PHP.

For more details on individual classes, you may want to look at the automatically generated API documentation. If you're building from git, this will be built for you in `xapian-core/docs`; the API may have some changes between the stable release documented on the website and the latest version in git.

Get familiar with the code

If you're going to be writing code, it's a good idea to read some of Xapian's existing sources, particularly in the main library (`xapian-core`). When you come to write your own, you'll want to follow the style of how the current code works, both in terms of layout (where spaces go and so on) and *how we use various C++ language features*.

Picking something to start with

It can be difficult sometimes to find a place to start, so here are some suggestions:

- Start small.

  By picking a small contribution first, other people can help you with details of how Xapian's documentation, code and so on work. It's much easier to get feedback on a small change to start off with.

  If documentation is your thing, then you might like to take a look at our list of missing documentation.

On the features side, our bite-sized projects are intended to be suitable for someone new to Xapian to pick up. We've tried to have a range of things to work on across different parts of Xapian.

Note: No change is too small to consider! Some people's first contributions to an open source project are fixing a single stray letter in documentation, or adding a line break where one is needed.

- Pick something you care about, or which you already have some knowledge of.

  For instance, if you've been using Omega, you might want to pick up a small bug or feature for that. Or if you've studied (or are studying now!) different Information Retrieval weighting schemes, you might want to implement one of the ones we don't currently support.

- Don't be afraid to *ask for help*.

  Please pop onto the mailing list or IRC if you need any help in getting started or picking something to work on.

Do some work

Add or correct some documentation, fix a bug or implement a new feature! You'll probably find our *suggested workflow* helpful. We also have *detailed information* on contributing your changes back to Xapian for inclusion in future releases.

2.5.3 A helpful workflow

If you're working on a bug or new feature, you'll follow something like the following steps. If you haven't worked on Xapian before, it's a good idea to adopt this workflow.

Claim the ticket in trac

Trac is where we keep track of proposed features and known bugs. You'll have to sign up for an account, including verifying your email address, in order to make changes. (This is to prevent unwanted spam on the wiki and in tickets.)

To 'claim' the ticket so others know you're working on it, you can either reassign the ticket to yourself and then claim it, or you can just add a comment saying you're working on it. If you haven't already done so, now's a good time to drop a message either in IRC or to the mailing list saying what you'l be working on.

Note: If there isn't a ticket for what you want to work on, you should create one. (If you're creating one from something on our project list on the wiki then it's good practice to update the wiki to link to the new ticket, so other people can find it easily.)

Create a branch in your local git repository

You want somewhere where you can keep your changes until they're ready, and that won't get confused with anyone else's work. In git these are called branches.

Before you create your branch, you will generally want to make sure your local copy of the code is up to date with the central repository. Generally you can this as follows:

```
$ git checkout master
$ git pull origin/master
```

You can check create your new branch:

```
$ git checkout -b feature-x
```

The branch name doesn't really matter, but you'll probably find it easiest to name it something related to the work that you're doing.

Write a plan

Even the smallest contribution is worth thinking about before starting to type, and with larger changes it's all but essential. If you put your plan in the ticket on trac, it will help when someone else comes to review your patch. Also, if you ask for help, having a plan that someone else can refer to can let them see how you're thinking, so they can provide some useful advice or recommendations more easily.

For a larger piece of work, you ideally want to be able to break down the work into smaller "sub-projects" which can be completed, reviewed and released in turn. (If you're familiar with agile development, think of it as a series of development sprints.)

When planning work on a bug or new feature, you should bear in mind that there are a number of *standards we work to* when accepting changes into Xapian, and the more of them you cover yourself the easier it will be to get your changes into a future release. In particular, it's worth thinking about documentation and tests in advance.

About the only time you don't need to write a plan is when you're making a small change to some documentation, correcting a spelling mistake or making something clearer.

Make your changes!

Now you can start making changes. There's a host of *other information that can help you* if you're writing code. As usual, there are *other people in the community who can help* if you need.

If you discover partway through that your plan isn't working, it's a good idea to stop and write a new plan. You'll have learned something about the problem that you didn't know when you wrote the initial plan, so taking the time to think things through from the beginning can often unblock you and let you start moving again. Of course, if it doesn't clear things up, that's a good time to ask the community for help.

Make commits out of your changes

This is where you probably want to know a little more about git. A very quick introduction is that you first "stage" changes, then you "commit" those changes. You don't have to stage all your changes at once, which means you can keep small notes or parts of future work lying around while you're creating your commits, without them creeping into your commits and confusing matters.

To stage changes for your next commit:

```
$ git add -p
```

The `-p` tells git that you want it to find all the changes, then one by one ask you if you want each staged. Just type `y` to stage a change (it calls them "hunks"), or `n` to skip it this time round. If the file is completely new, you can run `git add <path>` to stage the whole file. (There are lots of other options available in `git add -p`; if you type `?` then it will explain what they all do.)

Then to make a commit:

```
$ git commit -v
```

git will open your editor for you to write a commit message. The `-v` means that your changes will be shown at the bottom of the editor (although they won't be included in the commit message), which helps you do a final check that you're committing only what you want, and everything that is needed.

A good commit in git relies on getting two things right: changes that do a single thing, and a commit message that describes the thing clearly. We have some quick tips on each.

Good commits

Structuring your changes into commits can take a bit of getting used to, but makes it a lot easier for other people to review, both before we merge into Xapian and in the future when someone – which might be you! – needs to understand why a change was made in the past, to help them do whatever work they need to do. There's a good article by Anna Shipman that may help you think about structuring your changes into a set of commits that are easy for others to read.

- Only make *one change* per commit, and make the *whole change* in that commit – you don't want to end up with essential bits of code in a different commit.

  Many people struggle with this at first, and it can be difficult to get into the habit of thinking in terms of the distinct changes to the system rather than in terms of how you did the work. *A plan* here can help structure your commits once you've finished working.

  One of the reasons we suggest using `git add -p` is that it enables you to review every single change that goes into a commit, which can help you put only the right things into it.

- Avoid committing code that has been commented out. If we need it again, it's in the git history.

Why the "size" of commits matters

We said above that you should only make one change per commit. This is for a few reasons.

- The most important is that if a change later turns out to be unhelpful for whatever reason, git provides a tool (called "revert") for creating a new commit that "undoes" a previous one. If the original commit did more than one thing, then those changes needs to be untangled in order to revert. That takes more time, and increases the chance that someone will make mistakes when undoing the previous changes.

  Note: You can use `git revert` with squash and fixup commits

  The trick is to run `git revert -n`. `-n` means that git will revert the changes in your working tree and index, but not make a commit out of them. You can then use `git commit --fixup` or similar as usual.

  You can also revert several different commits as one commit this way.

- Where it makes sense, we like to apply fixes and smaller improvements to the current release series of Xapian. Some larger improvements, including completely new features, are generally not suitable for this, so if you make a commit which say fixes a bug and adds a new method to one of our core classes, it makes it harder to "backport" just the bug fix. If you separate that into two commits, we can use a git tool called "cherry picking" to pull just the bugfix commit in for the next stable release.

  In fact, with separate commits, it's easier to spot candidates for backporting in the first place.

- In a similar vein, some changes that aren't pulled into a release series may be of interest to someone else who doesn't want to, or cannot, use the "bleeding edge" git master. With separate commits, they can use the patches for just the functionality they are looking for. This can be particularly useful for people packaging Xapian for various operating system distributions.

Good commit messages

Writing a great commit message is important both for people reviewing your code now to help get it ready for a future Xapian release, and for when someone needs to understand how and why a particular change was made, months or years in the future – when that someone might be you!

- Start with a short (50 characters) summary line.

  git (and github) are designed to work better this way. The summary should be in the imperative ("Fix bug on macOS" rather than "Fixed bug on macOS"). This matches git's automatic messages around merges, reverts and so on.

- Follow that with more detail as needed, wrapping long lines at 72 characters (one exception is that long URLs are best not wrapped).
- Describe the effect, not the code. The important thing is for people to be able to read the commit message and understand what you were trying to achieve when you made those changes. That way, if someone needs to work on that part of the code in future, they can understand the purpose of it, and not accidentally remove some useful functionality. (Tests help here, but the commit message is very important.)

There are a few articles around on writing good commit messages; Thoughtbot's "5 Useful Tips For A Better Commit Message" has some good advice.

> **Warning:** Lots of online git tutorials will tell you to write commit messages on the command line, using `git commit -m <message>`. If you do that, you'll never write really good commit messages.

For more details on using git, there are free books and resources online, such as Pro Git.

Contribute your changes

We have *detailed information* to help you here.

2.5.4 Contributing changes

If you have a patch to fix a problem in Xapian, or to add a new feature, please send it to us for inclusion. Any major changes should be discussed on the xapian-devel mailing list first.

The rest of this section gives information on how to get your changes adopted quickly into Xapian. We have noted some things we aim for in our code and our changes, as well as how to get changes to us and some other details.

License grant

We ask everyone contributing changes to Xapian to *dual-license* under the GPL (which Xapian currently uses) and the MIT/X license (which we would like to move to in future). The simplest way to do this is to drop an email to the xapian-devel mailing list stating that you own the copyright on your changes and are happy to dual-license accordingly.

Some things that we look for

Beyond looking for changes that improve Xapian, code that works and so forth, there are a number of things that we aim for when accepting changes. This then is a list of good practices when contributing changes.

Code that compiles cleanly and looks like existing code

We like Xapian to compile without any warnings. In "maintainer mode", which will be how you're building Xapian if you're working from a git clone, all warnings will actually become errors. You should fix the problems rather than change the compilation settings to ignore these warnings.

Please configure your editor to:

- indent each block of code 4 columns from the containing block
- display the tab character by advancing to the next column that is a multiple of 8
- "tab fill" indents: all indents should start with as many tab characters as possible followed by as few spaces as possible
- use Unix line endings (so each line ends with just LF, rather than CR+LF)

We don't currently have a formal coding standards document, so you should try to follow the style of the existing code. In particular, it's a good idea to pay close attention to code alignment and where we have spaces.

We have a small tool that can help spot common formatting problems. It's run on all *pull requests*, so it's a good idea to run it on your changes. From a clone of the Xapian source tree, the following will tell you if there are any problems in the changes you've made since you branched from master:

```
git diff master..HEAD | xapian-maintainer-tools/xapian-check-patch
```

Updated documentation

If you add a new feature, please ensure that you've documented it. Don't worry too much about the language you use, or if English isn't your first language. Others can help get the documentation into shape, but having a first draft from the person who wrote the feature is usually the best way to get started.

- API classes, methods, functions, and types should be documented by documentation comments alongside the declaration in `include/xapian/*.h`.

  These are collated by doxygen – see doxygen's documentation for details of the supported syntax. We've decided to prefer to use `@` rather than `\` to introduce doxygen commands (the choice is essentially arbitrary, but `\` introduces C/C++ escape sequences so `@` is likely to make for easier to read mark up for C/C++ coders).

- The documentation comments don't give users a good overview, so we also need documentation which gives a good overview of how to achieve particular tasks.

  If there's relevant documentation already in the user guide, then you should update that. For completely new features, you should create either a "how to" or an "advanced feature" document in the user manual, so that people can get started without having to start with the API documentation.

- Internal classes, etc should also be documented by documentation comments where they are declared.

Automated tests

If you're fixing a bug, you should first write a regression test. The test will fail on the existing code, then when you fix the bug it will pass. In the future, the test will make sure no one accidentally re-introduces the same bug.

If you're adding a new feature, you'll want to write tests that it behaves correctly. Thinking about the tests you need to write can often help you plan how to implement the feature; it can also help when thinking about what API any new classes or methods should expose. A good set of tests will both ensure that the feature isn't broken to start with and detect if later changes stop it working as intended.

And of course you should check that the existing tests all continue to pass. It's good practice to get into the habit of running tests locally as part of your development process, and certainly before you share changes with others, such as by opening a Pull Request or sending us a patch.

If you don't know how to write tests using the Xapian test rig, then please ask. It's reasonably simple once you've done it once. There is a brief introduction to the Xapian test system in `docs/tests.html`, as well as some helpful information in the *Testing Xapian* section of this guide.

Note: If you're adding a new testcase to demonstrate an existing bug, and not checking a fix in at the same time, mark the testcase as a known failure (by calling `XFAIL("explanatory message")` at the start of your testcase (if necessary this can be conditional on backend or other factors - the backend case has explicit support via `XFAIL_FOR_BACKEND("backend", "message")`).

This will mean that this testcase failing will be reported as "XFAIL" which won't cause the test run to fail. If such a testcase in fact passes, that gets reported as "XPASS" and *will* cause the test run to fail. A testcase should not be flagged as "XFAIL" for a long time, but it can be useful to be able to add such testcases during development. It also allows a patch series which fixes a bug to first demonstrate the bug via a new testcase marked as "XFAIL", then fix the bug and remove the "XFAIL" – this makes it clear that the regression test actually failed before the fix.

Note that failures which are due to valgrind errors or leaked fds are not affected by this macro – such errors are inherently not suitable for "XFAIL" as they go away when the testsuite is run without valgrind or on a platform where our fd leak detector code isn't supported.

Updated attributions

If necessary, modify the copyright statement at the top of any files you've altered. If there is no copyright statement, you may add one (there are a couple of Makefile.am's and similar that don't have copyright statements; anything that small doesn't really need one anyway, so it's a judgement call). If you've added files which you've written from scratch, they should include the GPL boilerplate with your name only.

If you're not in there already, add yourself to the `xapian-core/AUTHORS` file. If you forget, as well as if we have used patches from you, or received helpful reports or advice, we will add you to this file, unless you specifically request us not to. If you see we have forgotten to do this, please draw it to our attention so that we can address the omission.

Update trac

If there's a related trac ticket or other reference for a bug or feature, update it (if the issue is completely addressed by the changes you've made, then close it). It's also good to mention it in the commit message – it's a great help to future developers trying to work out why a change was made.

If you've fixed a bug, it's helpful to update the release notes for the most recent release with a copy of the patch. If the commit from git applies cleanly, you can just link to it. If it fails to apply, please attach an adjusted patch which does. If there are conflicts in test cases which aren't easy to resolve, it is fine to just drop those changes from the patch if we can still be confident that the issue is actually fixed by the patch.

Consider backporting bug fixes

If there's an active release branch, please check if the bug is present in that branch, and if the fix is appropriate to backport - if the fix breaks ABI compatibility or is very invasive, you may need to fix it in a different way for the release branch, or decide not to backport the fix.

Submit your patch

There are two ways of working, depending on whether you want to use Github or not. In both cases, review and acceptance of the changes will generally go more easily if you've included tests, updated documentation and so on *as discussed earlier*.

Pull requests via Github will have the tests run automatically on a variety of platforms. This means that you should run the tests before creating a pull request (since it's not worth reviewing something where the tests are failing – you may have to make significant changes to get the tests to pass, so reviewing too early could be a waste of everyone's time).

Of course, if you have difficulty getting the tests to pass on your local machine, or if locally they do pass but fail on the automated systems connected to Github but you can't figure out why, then *get in touch* and someone should be able to help.

Attach a patch directly to the trac ticket

We find patches in unified diff format easiest to work with. `git diff` produces the right output for a single commit (or `git format-patch` for a series of commits).

If you're working from a tarball, you can unpack a second clean copy of the files and compare the two versions with `diff -pruN` (`-p` reports the function name for each chunk, `-r` acts recursively, `-u` does a unified diff, and `-N` shows new files in the diff). Alternatively `ptardiff` (which comes with perl, at least on Debian and Ubuntu) can diff against the original tarball, unpacking it on the fly.

Someone from the community will then be able to review the patch and decide if it needs further work before integrating. If so, they'll leave comments on the trac ticket (trac will generally email you if you're marked as the owner, or you can explicitly add yourself to the "cc" list for a ticket).

Open a Pull Request on github

Github pull requests provide a web-based interface for review and discussion of changes before they are accepted into Xapian. Github's documentation explains how you can go about opening them.

If your patch is a sub-project in a larger piece of work, then it's important not to assume the patch is fine as it stands and to immediately start the next sub-project. Instead you should concentrate on completing the sub-project before moving on. Since you'll almost always have to wait at least a little time to get feedback on any changes, you may want to put the code and tests up while still working on documentation.

You should add further changes to pull requests by creating additional commits locally, typically by using `git commit --fixup`, and then pushing the branch up to Github. Only once everything's been approved should you squash your commits together to keep the history clean.

Note: Once you've opened a pull request, you shouldn't have to close it until it's merged (in which case we'll generally close it for you). Even if you need to redo some work, you can either add fixup commits or (with agreement from whoever is reviewing the PR) unwind your work and create completely new commits, force pushing to replace the previous commits in the pull request.

It makes it much harder to review if you close a pull request in the middle of a review only to open another with similar code.

2.5.5 Adding support for other programming languages

Ambition

We'd like Xapian to be usable from as many languages as possible. However, this means more than just having bindings code. To truly support a language, we need to have:

- working bindings that can be built against a recent release of Xapian or git master, as well as against the latest version of the target language
- documentation to help people familiar with the target language to start using the bindings
- example code – ideally translations of the code in the Xapian user guide. You can start from the user guide source code, which already supports a few languages.
- automated tests for the bindings; at least a "smoketest" hooked into the tests for the build system, so we'll notice if something significant breaks. See *the section on testing bindings*.

Some things you need to know

Many languages can be done using SWIG, and it's probably easier to do so even though some languages may have better tools available just because it's less overall work. SWIG makes it particularly easy to wrap a new method for all the supported languages at once - in many cases just adding the method to the C++ API is enough, since SWIG parses the C++ API headers.

To give an idea of how much work a set of bindings might be, the author of the Ruby bindings estimated they took about 25 hours, starting from not knowing SWIG. However, the time taken could vary substantially depending on the language, how well you know it, and how well SWIG supports it.

What's really needed is someone interested in bindings for a particular language who knows that language well and will be using them actively. We can help with the Xapian and SWIG side, but without somebody who knows the language well it's hard to produce a solid, well tested set of bindings rather than just a token implementation.

Experiments and partially-completed bindings

XS bindings for Perl have been contributed for Perl, and these are available on CPAN as `Search::Xapian`. We also have SWIG-based Perl bindings which are in the `perl` subdirectory here, as a `Xapian` module. These are replacing the XS-based `Search::Xapian`, and will eventually be on CPAN.

These are languages which SWIG supports and which people have done some work on producing Xapian bindings for:

Go There are some basic Go bindings written by Marius Tibeica in the "golang" branch, which is currently based on the `RELEASE/1.4` branch.

Ocaml Dan Colish did some initial work on Ocaml support.

Pike Bill Welliver has written some Pike bindings for Xapian covering some of the API.

These bindings appear to be hand-coded rather than generated using SWIG. SWIG 4.0.0 has disabled Pike support due to lack of recent maintenance, so SWIG is probably not a good option here.

There are a number of other languages which SWIG supports, but which nobody has yet (to our knowledge!) worked on producing Xapian bindings for – see the SWIG documentation for a list of supported languages. It may be possible to support a language which isn't listed above, but it's likely to be harder unless you're already familiar with the tools available for wrapping a C++ library for use with that language.

2.5.6 Handy tips for aiding development

Disabling documentation builds

If you are find you are repeatedly changing the API headers (in include/) during development, then you may become annoyed that the docs/ subdirectory will rebuild the doxygen documentation every time you run "make" since this takes a while. You can disable this temporarily (if you're using GNU make), by creating a file "docs/GNUmakefile" containing these two lines:

```
%:
    @echo "Skipping 'make $@' in docs"
```

Note that the whitespace at the start of the second line needs to be a single "tab" character!

Don't forget to remove (or rename) this and check the documentation builds before committing or generating a patch though!

Integration syntax checking with your editor

If you are using an editor or other tool capable of running syntax checks as you work there you can use the *make* target 'check-syntax'. For 'emacs' users this works well with 'flymake'. Usage from a shell:

```
make check-syntax check_sources=api/omdatabase.cc
```

2.6 Mentoring new contributors

Xapian frequently participates in Google Summer of Code (GSoC), which encourages student developers to contribute to open source projects. We also welcome new contributors at any time.

This section contains advice and information for anyone within the community helping newcomers come up to speed as members of our community. Especially during GSoC, we welcome anyone to act as a mentor who is prepared to commit enough time. Many of our GSoC students have gone on to mentor in subsequent years.

Note: If you're looking for help in getting started, then we have a guide for potential GSoC students (which is worth reading through even if you aren't participating in GSoC). The *Contributing to Xapian* section of this guide also has useful pointers.

2.6.1 There's plenty of help

You may be daunted by the idea of mentoring someone else, particularly if you only have a limited amount of expeirence with Xapian itself. Please don't let this put you off! Those who have been around in the community for longer are generally happy to provide advice and assistance, and there's also a mentor guide written for GSoC (including contributions from Olly Betts from Xapian).

As with almost everything in life, **it's good to ask for help**. Xapian as a community has years of experience with Summer of Code, and some individuals have mentored five or more times. If we can't figure out a problem together as a community, we can also ask for help from other projects and mentors, and from the Google Summer of Code team themselves.

2.6.2 Helping newcomers step by step

Building Xapian

The first step should always to ensure that a new contributor can build Xapian on a machine they have access to. Our *getting started information* is a good guide to follow.

Most Xapian developers use one or more of Debian, Ubuntu, and macOS. While it's possible to develop for Xapian on a wide range of operating systems, if someone runs into problems with something else it's often easier for them to work with a linux virtual machine running on their computer. As the getting started information says, we recommend using Virtual Box and the latest LTS (long-term support) release of Ubuntu.

Note: If a new contributor runs into problems, it's worth getting them to explicitly confirm exactly what steps they've taken. It's easy to skip a step, or to try something else when you're working through issues – but when it comes to helping someone else, you need to be sure you know exactly what they've done.

A lot of our project communication happens on IRC, and it's difficult to read long outputs of commands such as `make` there. It's helpful to have people copy anything substantial into something like Pastebin or Gist and then provide a link in IRC. This can be used for command output, or for the contents of files such as `config.log`.

Getting familiar with the API

While the urge to jump right in and start fixing bugs or adding features is often strong, it's a good idea for a new contributor to become familiar with Xapian's API early on. This is particularly true for Summer of Code students, who often won't have used Xapian previously.

The Xapian user manual covers the concepts behind Xapian, works through a practical example of indexing and searching documents using Xapian, and also covers a range of more advanced features. The online version uses python, but you can grab the source code and build for a range of languages, including C++.

Note: Almost every contributor should get familiar with Xapian's C++ API. Anyone who's adding new APIs to Xapian should also think about how that API will be used from bindings languages, so it's often helpful for them to have at least used Xapian via python or one of the other languages we support.

Completing a small task

We ask all Summer of Code students to complete at least one small task, effectively as part of their application. This could be fixing a bug, tidying up some code, improving test coverage, or completing a small feature.

A main part of the reason for this is to help new contributors get used to *the way we manage changes* to Xapian. It's good to pick something small, and go through the entire process of planning and doing the work, creating a pull request, and getting everything merged. Firstly, that means that they've become a contributor to Xapian before Summer of Code even begins. Secondly, it means that subsequent contributions as part of the GSoC project should be smoother. Finally, it helps someone new to open source and collaborative development to start thinking in terms of small changes, merged quickly.

Most students will have an idea of what project they are interested in, from our list of project ideas. Some of those will have a small first step, or sometimes a bug or similar in the general area of the codebase of the project. For everyone else, we keep a list of bite sized projects, and there are also some bugs in our tracker marked as suitable for newcomers to tackle.

Helping them through their first contribution

Just as with getting Xapian built for the first time, new contributors may need support in getting through our contribution flow. There are some common things to watch out for.

Pull requests where the automated tests aren't passing

> This includes a special test run which checks that the code diff follows some of our conventions, such as around spaces and blank lines. Generally, the error messages should help track down the problem.
>
> It's possible (and a good idea) to run all the tests locally before opening a pull request. The code diff checks can be run by piping the output of `git diff` through the `xapian-maintainer-tools/xapian-check-patch` script. Something like this is often what you want:
>
> ```
> git diff master..HEAD | xapian-maintainer-tools/xapian-check-patch
> ```
>
> (The `git diff` command there will output the changes in your local commits compared to the "master" branch.)

Not following our *coding conventions*

> We can't automate checks for all of these, but we also don't expect anyone to be able to spot all possible problems. One of the reasons pull request reviews are open is so that several different people can help spot and straighten out issues, and get a contribution over the line.

Not following our conventions for *pull request flow*

> In particular, first-time contributors often need reminding not to force-push branches once a PR is open. It feels tidier to have a tidy list of commits. However, it makes it harder for reviewers to check that earlier comments have been addressed. Contributors should use "fixup" commits, as described in our documentation, and only tidy up commits right before a pull request is merged.
>
> A similar problem is a pull request with lots of commits without good commit messages. Each commit in a pull request should make a single, well-described change, including any necessary tests and documentation.

Some of these take a long time to get used to, and even experienced developers will make mistakes. That means that it's worth checking for the basics on every pull request.

2.6.3 Expectations of mentors

We operate a "group mentoring" approach, which means you can – and should! – help any students you can during the summer. Where possible, we expect mentors to find time every week to engage with Xapian and our Summer of Code students. Here are some ways to do that.

Answer questions on the mailing list and IRC

> It's demotivating to ask a question and get no reply. Sometimes even just a response that says you don't know can help reassure a new contributor that they aren't on their own.
>
> Particularly during the early phases of Summer of Code, there are a lot of questions that come up repeatedly. New contributors regularly need help getting Xapian built and installed on their computers. People often need pointing at our guidance for potential students (the GSoC site sends people straight to our ideas list, and it's easy to miss the links we provide to furhter information). So something as simple as chipping in to point people to existing information and documentation can be incredibly valuable.

Help review pull requests

The core of a contribution to Xapian is often a pull request. Before it's merged by one of the Xapian team, we want to make sure it's in *good shape*. Anyone can check over our guidelines on what we're looking for, and provide feedback to a contributor on how to improve their pull request.

Provide feedback on design ideas

Most Summer of Code projects have a knotty or interesting problem at the heart of them. That's what makes them appealing to work on over a period of months. However, that means that there's often one or more points during the project where some decisions have to be made. APIs need designing, data structures need choosing, and sometimes different competing algorithms need assessing.

While we expect our students to do most of the work here, getting timely feedback and input from the rest of the community is often important in keeping a project on track. As with reviewing pull requests, anyone can look over a proposal and provide their thoughts.

Note: API design is a particularly difficult problem, and we generally do not expect any one person (student or not!) to design a great API on their own. It's not always obvious the best approach until you've written code that uses an API in a range of different situations.

We generally recommend that projects that require a new API start by implementing a very simple one. Ideally this will leave time later in the project to revise the initial version based on feedback and experience of actually using it. (Summer of Code students also include a section on possible future improvements in their project write-up. If there isn't enough time to improve an API based on feedback, that can always become a future project!)

Encourage small, regular contributions that can be merged

We recommend structuring any project as a series of small sub-projects, each of which can be submitted as a pull request, reviewed, and merged. It's usually possible to start work on the next sub-project while the previous one is going through review.

As well as encouraging contributors to submit small changes, it's also important that they address review comments quickly. It's all too easy to move on to the next sub-project, but never actually get the previous one merged. It's far better to spend time getting two or three sub-projects merged than have pull requests for four or five none of which is in a good enough state to be merged.

Our best Summer of Code students have often had early contributions released during their project. Our experience shows that contributors are more likely to become longer-term members of the Xapian community if their early work can be merged and released.

As well as group mentoring, every student has a specific mentor assigned, who is there to make sure there is always someone looking out for them. You should keep in regular contact with the student you're assigned to, making sure that they're getting the support they need from the community.

Note: Mentors, as well as students, can have something unexpected come up during the summer. Plans change, work becomes busier, or any one of a hundred things can mean you suddenly have less time than you anticipated.

If something comes up, please let Xapian's "org admins" for Summer of Code know as soon as possible. Our group mentoring approach makes it easier to cope with people who have to step away from Summer of Code, but we need to know to ensure we can support all our students as best we can.

2.7 Making a release of Xapian

This is a (hopefully complete) list of the jobs which need doing:

- If there are changes which are likely to affect the RPM packaging, email Fabrice Colin and Tim Brody so they can check it.
- Check if *config/config.guess* and *config/config.sub* need updating to more recent versions from https://git.savannah.gnu.org/gitweb/?p=config.git
- Check the revision currently specified in the bootstrap for the common subdirectory. Unless there's a good reason, we should release xapian-core and omega with synchronised versions of the shared files.
- Make sure that any new/changed/removed API methods in xapian-core have been wrapped/updated/removed in xapian-bindings.
- Update the lists of deprecated/removed API methods in docs/deprecation.rst
- Update the NEWS files using information from the git logs
- Update the version in configure.ac for each module (xapian-core, omega, and xapian-bindings), and the library version info in xapian-core's configure.ac
- Make sure the submitters of fixed bugs are mentioned in the "thanks" list in xapian-core/AUTHORS. Check the list for the appropriate milestone:

```
https://trac.xapian.org/query?col=id&col=summary&col=reporter&milestone=1.4.12
```

- Check for any unfixed bugs on the milestone for the new release, and if they aren't blockers, retarget them:

```
https://trac.xapian.org/roadmap
```

- Tag the source trees for the new revision - use the git-tag-release script, running it with the new version number, for example:

```
xapian-maintainer-tools/git-tag-release 1.4.12
```

  This script also generates tarballs for the new release and copies them across to the website.

- Update the wiki:

  Fill in https://trac.xapian.org/wiki/ReleaseOverview/1.4.12

  If the next release isn't likely to be in 2 months time, update the roadmap at https://trac.xapian.org/wiki/RoadMap with a revised estimated date.

- Update the website: `generate` in the `www.xapian.org` git repo contains the latest version and the date it was released.
- Run `./update_website` in the cloned repo on the webserver.
- Announce the new version on xapian-discuss
- Update the version numbers in this checklist ready for the next release
- Have a nice cup of tea!

2.7.1 How to make Debian packages for a new release

Debian control files are stored in separate git repositories:

- https://anonscm.debian.org/cgit/collab-maint/xapian-bindings.git
- https://anonscm.debian.org/cgit/collab-maint/xapian-core.git
- https://anonscm.debian.org/cgit/collab-maint/xapian-omega.git

To package a new upstream release, these should be updated as follows:

- If there are any patch files in "debian/patches", check if these have been incorporated into the new release, and if so remove them and update "debian/patches/series".
- Update the `debian/changelog` file, being sure to keep it in the standard Debian format (the easiest way is to use the dch utility like so: `dch -v 1.2.19-1`. The new version number should be the version number of the release followed by "-1" (i.e., a debian patch number of 1). The changelog message should indicate that there is a new upstream release, and should mention any significant changes in the new release.
- Tag using: `git tag -s -m 1.2.19-1 1.2.19-1`
- FIXME: Document how to make source packages, or update `make-source-packages`.
- FIXME: Document how to build binary packages, or update `build-packages`.
- Test the packages.
- Run `debsign build/*_amd64.changes` to GPG sign the packages.
- Run `dput build/*_amd64.changes` to upload them to Debian.
- For the Ubuntu backports:

```
./backport-source-packages xapian-core 1.2.19-1 ubuntu
./backport-source-packages xapian-omega 1.2.19-1 ubuntu
./backport-source-packages xapian-bindings 1.2.19-1 ubuntu
```

And once libsearch-xapian-perl is uploaded to Debian unstable:

```
./backport-source-packages libsearch-xapian-perl 1.2.19.0-1 ubuntu
```

Then sign:

```
debsign build/*99*_source.changes
```

Upload:

```
dput xapian-backports build/xapian-core*99*_source.changes
```

Wait for that to have a chance to build, and then:

```
dput xapian-backports build/xapian-[bo]*99*_source.changes
dput xapian-backports build/libsearch-xapian-perl*_source.changes
```

2.8 License

This license applies to all documentation and example code in this book.

copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Todo: There needs to be a section about macOS and bindings, somewhere.
