Suggesting Query Completions for Xapian

Pallavi Gudipati ■ Tanmay Dhote ■ Parth Joshi ■ Siddhant Mutha ■ Mohammed Shamil

Intoduction

Xapian is a Open Source Search Engine Library, written in C++, which currently has basic search utility. In this project we have incorporated two new functionalities in the project - Query Auto-completion and Query Expansion. Auto-Complete saves the time of the user by giving possible solutions while he is typing the query, whereas Query Expansion improves user satisfaction by giving him more relevant searches to his query. Auto-complete has been achieved by the implementation of a prefix tree whereas for Query Expansion, we have used WordNet for thesaurus support. A user interface has been created in Qt for testing out the performance of our additions.
We have tried to incorporate different design patterns in the new modules that we have added to Xapian. These patterns are elaborated below.

Flyweight pattern

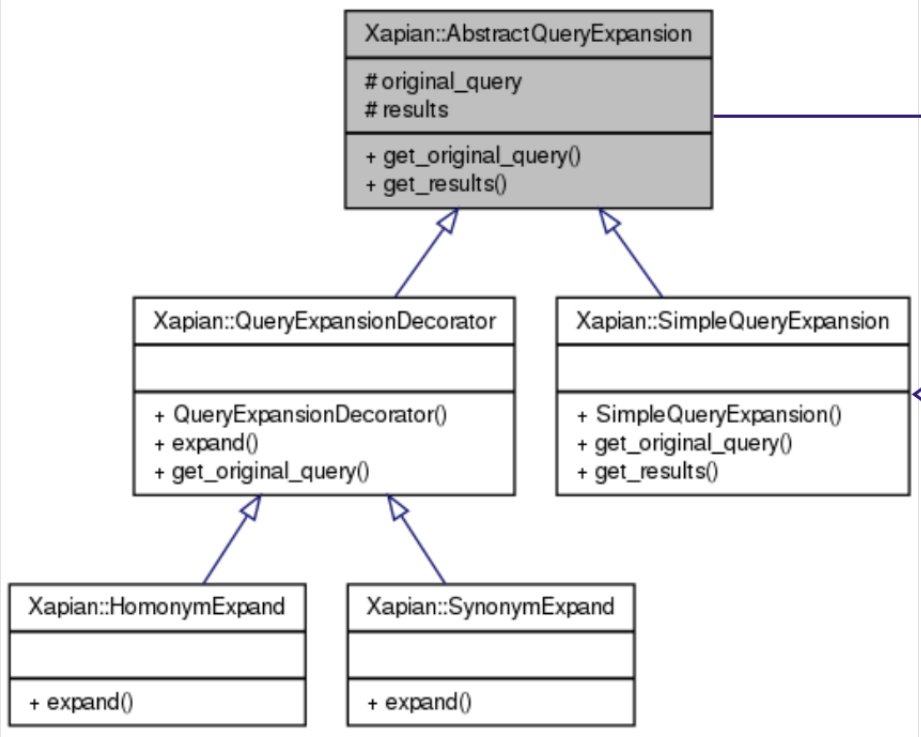
Flyweight pattern was used in designing our code to reduce memory usage. It is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. Often some parts of the object state can be shared, and it is common practice to hold them in external data structures and pass them to the flyweight objects temporarily when they are used.
We have added a new module for the implementation of auto-completion, $class\,Trie$. The whole trie data structure in our code is represented using a light weight pointer to its root node rather than all nodes of the tree, thus saving lot of memory. The data member $root$ is a pointer of type $struct\ trie\_node$ and points to the root of the trie tree.
If we have two threads trying to run queries on the same database, they can potentially share the data structure by having a pointer to the same root node.

Decorator pattern

Decorator pattern allows to add new functionality an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.
The original $AbstractQueryExpansion$ class is subclassed into a $QueryExpansionDecorator$ class and a $SimpleQueryExpansion$ class. $QueryExpansionDecorator$ is further subclassed

into two classes, with $SynonymExpand$ and $HomonymExpand$ being the two decorators. We haven't implemented $HomonymExpand$ but have created an empty class to give future developers an idea about the pattern. This pattern is designed so that multiple decorators can be stacked on top of each other, each time adding a new functionality to the overridden method(s). The decorators and the original class object share a common set of features.
The user will first have to create a $SimpleQueryExpansion$ object with the query as an argument and then add functionality by giving the current object as an argument to the constructor of the decorator and receiving an object with the added functionality in return. The user can get the results by calling $get\_results()$. We integrated WordNet with Xapian for thesaurus support to get synonyms. We also subclassed the existing $SimpleStopper$ class to $PopulatedSimpleStopper$ to remove stop-words from the given query.

Figure 1: Decorator pattern

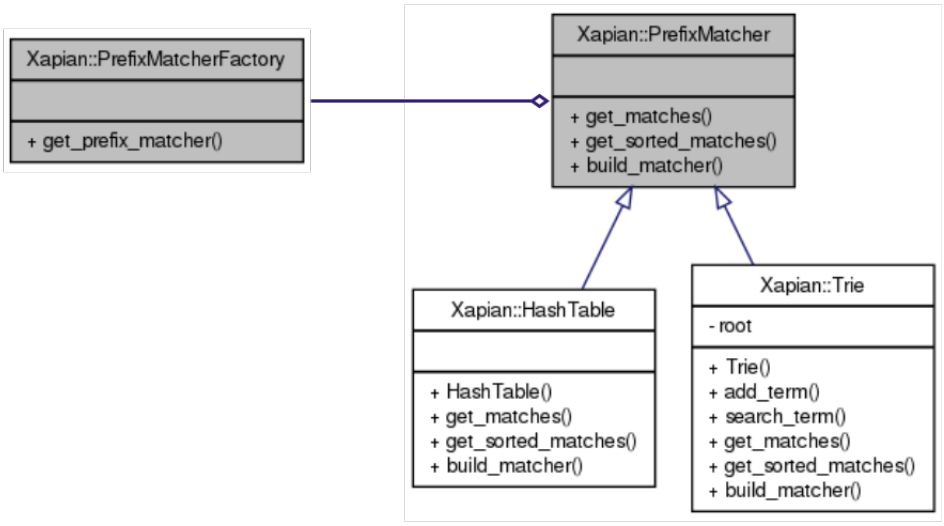


Factory pattern

Factory method pattern is a creational pattern which uses factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created. This is done by creating objects via a factory method, which is either specified in an interface (abstract class) and implemented in implementing classes (concrete classes); or implemented in a base class, which can be overridden when inherited in derived classes; rather than by a constructor.
We have created a $PrefixMatcher$ class that is subclassed into the classes $Trie$ and $HashTable$. We have also created a $PrefixMatcherFactory$ class that takes in the type of the prefix matcher needed and outputs a pointer to the appropriate matcher. Similar to the class $HomonymExpand$,

we haven't implemented the $HashTable$ class but have left it empty for future use.
To get the desired $prefixMatcher$ object, the user just needs to call $PrefixMatcherFactory.get\_prefix\_matcher()$ with the argument $TRIE$ or $HASHTABLE$. To build the prefix matcher, a user will need give a $Xapian :: Database$ object to the method $build\_matcher()$. The user can then get results sorted on the frequency with which they have occurred in history by giving the string for which they need auto-complete suggestions to $get\_sorted\_matches()$. They can also get unsorted results by calling $get\_matches()$ and save time.

Figure 2: Factory pattern



Access modifiers

We have used the $public$, $private$ and $protected$ access modifiers appropriately. For example:

- $root$ is private is $Trie$ class. No one can access the tree in general, but can only get the prefix matches.
- $original\_string$ and $results$ are protected in $AbstractQueryExpansion$ class as this restricts the access at the same time allowing it to be inherited.
- $stop\_words$ is protected in $PopulatedSimpleStopper$ class thus allowing it to be inherited.