C++ Language Tools Team, Google

Dex: efficient symbol index for Clangd

Summary

This document describes the proposed efficient symbol index implementation for clangd.

Author: **Kirill Bobyrev (kirillbobyrev@gmail.com)**
Reviewers: **Eric Liu, Sam McCall**
Contributors: **Alexander Neubeck, Matei-Stefan Chiperi**

Created: **2018-07-06**
Last updated: **2018-07-17**
Link to this document on Google Drive - "Dex: efficient symbol index for Clangd"

Objective

The goal of this project is to build an efficient symbol index for Clangd (Clang-based C++ Language Server Protocol implementation), which would reduce the latency of requests such as code completion, symbol lookups, and refactorings. Introduced search index would replace the existing inefficient implementation which Clangd currently relies on yielding performance boost. Designed system index must satisfy few important properties:

- Symbol lookups should be efficient both in terms of memory consumption and computational complexity
- The index should scale well for projects of a medium size such as LLVM (over 2M LOC) and Chromium (over 18M LOC)
- Live changes should be reflected in the index without noticeable performance overhead

Non-goals

There are certain limitations which are reflected in the design choices described in this document:

- The source code of the examined project is located on the workstation hard drive, the build also happens on the workstation, both index and clangd instance do not operate on a distributed environment
- The core implementation should replace existing Clangd symbol index by the end of September, the primary focus right now is on the extensible core functionality which would support efficient code completion requests handling. Many interesting features are potential extensions which should be considered later.

Background

One of the most useful Clangd features is real-time code completion. As user types text in the editor and sends a completion request, Clangd queries known symbols to identify potential matches while utilizing a number of useful code completion signals such as symbol name, symbol definition location and scopes in which the symbol is defined. Symbol names are scored using fuzzy matching[1] which calculates the similarity between matched symbol and what the user actually typed in the editor (query string). The presented design tries to address the problems of efficient fuzzy search on symbol name and symbol ranking. Symbol index design aims to reduce the latency of these operations while preserving code search quality.
Another useful feature of LSP is querying workspace symbols which is very similar to the Google Code Search problem described in Regular Expression Matching with a Trigram Index or How Google Code Search Worked" overview by Russ Cox. The difference is that the query is not based on regular expression search. Google Code Search can also rank symbols based on file proximity, scopes and other useful information similarly to what is expected from Clangd search queries. However, the

[1] "Note on fuzzy languages" by Lee and Zadeh, "Construction of fuzzy automata from fuzzy regular expressions" (contains the majority of relevant paper references)

Code Search engine operates on a codebase of a significantly larger size, uses distributed storage and uses sharding for more efficient operations which introduces different challenges.

Overview

This section provides a high-level overview of the proposed design without going too much into the details. Detailed Design goes into more detail about each of the ideas described below.

Static and incremental indices

During the development process, the majority of the codebase remains unchanged while small incremental changes are applied. Therefore there is no need to rebuild index for the most parts of the project, which naturally leads to an idea of building a static and incremental indices to utilize that idea. *Static index* contains most of the symbols in the codebase, has quite large size (and therefore can be stored on hard drive) and is not rebuilt very often, on the other hand the *incremental index* keeps track of the changes happening in real-time (as user types symbols in the active files in the editor of choice) and is often rebuilt but has relatively small size and can be stored in memory. Such separation would require additional memory consumption, but the overhead is neglectable and this would allow latency reduction which is more important. Querying would involve submitting the request to both static and incremental index and merging the results afterward.

Hierarchical incremental index

As the user types text in the editor there is a continuous stream of immediate changes which should be reflected in the symbol index. Rebuilding incremental index after each small-scale change would be slow and merging such changes into the index would require additional design decisions and impose more complexity (although is a reasonable approach and is discussed in Mutable index architecture subsection). A viable solution for preserving reasonable performance and reducing the complexity would be introducing two layers of the incremental index: a stable layer for most of the changes which are not reflected in static symbol index and an instant layer which will keep track of the live changes in open files.
Similarly to the higher-level request, a query should go through both of the layers and then the results are merged to produce a response. The instant layer should be much

smaller than the stable layer, therefore as soon as the instant layer grows so that it is no longer much smaller than the stable layer, the whole incremental index should be rebuilt emptying the instant layer and merging all existing changes into the stable layer. This allows core incremental index structure to be immutable. The layer merge trigger should be handled by Clangd and is not a subject for the symbol index design.

Retrieval and scoring

During the code completion, the user is interested in the symbols having the highest probability of being the completion items. Clangd uses code completion signals to score the symbols and rank them accordingly. These signals include fuzzy matching score, number of references, file proximity, type and so on. Some of these signals are query specific and computing all of them would require too much computational effort if every symbol in the index had to be processed.
There are symbols which might have a high final score but very low initial score. Such symbols are likely not to be referenced many times and hence would be initial ranked poorly. The proposed solution is to support supplemental retrieval: processing query would not only involve text-based search, but it would also boost the symbols based on factors such as file proximity.

Trigram generation[2]

The preprocessing in the proposed design is primarily based on Trigram Search Index which was used in the original Google Code Search implementation to implement regular expression-based search. Clangd uses a fuzzy-matching scheme rather than regular expressions as queries, but the implementation is similar.
Each unqualified symbol name from the index is split into trigrams which characterize that symbol. The idea is that trigrams can also be generated from the query string, and matching these sets of trigrams approximates the fuzzy-matching rules.
These symbol-specific lists are used to produce the *posting lists* (also known as “inverted index”) which are used for the search queries. It is possible to sort the posting lists by the first-layer search score in order to allow efficient truncation of top **k** symbols which will be re-ranked using fuzzy match score and other criteria later to produce the final result. A very similar design which is a background of the presented one was validated for the purposes of code completion in internal Google services.

[2] Trigram generation techniques are substantially different from those used in the “Google Code Search overview”. The reason for that is that the current design does not imply regular expression-based search.

Posting list iterators

When the symbol index receives a lookup request with the query string it splits given string in the trigrams, just as it did with the index symbols before. After that, the query would process each posting list associated with each trigram of the given string while merging these posting lists. Since all of the posting lists are sorted it is possible to efficiently merge them in linear time and use early stopping as soon as top **k** symbols are identified. Merging can be viewed as greedily iterating through a number of posting lists and producing the result in the process.

Detailed design

Index build

The index building process can be separated into the three steps, which are performed for the following toy example. Let's assume the project has the following symbols:

- `class clang::Expr` (2000 references)
- `class clang::Decl` (5000 references)
- `SourceLocation clang::Decl::getLocEnd()` (1000 references)
- `class std::unique_ptr` (4000 references)
- `int Symbols` (500 references)
- … (<100 references)

The symbols collection stage is managed by Clangd.

Ranking received symbols

Symbols are ranked using given criteria (number of references in this example):

Rank	Symbol Name
0	`Decl`
1	`unique_ptr`
2	`Expr`
3	`getLocEnd`
4	`Symbols`

…	…

Generating trigrams from the symbol names

The next stage processes each symbol name and yields a list of trigram-based tokens. The trigram generation techniques are covered in-depth later. What is important here is that the result of this step is a mapping from symbol rank to the list of trigrams:

Rank	Trigrams
0	[“dec”, “ecl”]
1	[“uni”, “niq”, “iqu”, “que”, “ptr”, “unp”, “upt”, …]
2	[“exp”, “xpr”]
3	[“get”, “loc”, “end”, “gle”, “glo”, ...]
4	[“sym”, ...]
…	…

Given that Clangd can utilize multiple threads and the fact that trigrams generation is not data-dependent between different symbol names, this can be done in parallel.

Building symbol index

The final step produces an inverted index. The symbol index maps trigram to the corresponding posting list, which is a sorted sequence of pointers (which are mapped to rank in this toy example) to the symbols which contain this trigram. The posting lists construction and population can be done after each trigram and token generation and therefore there is no need to actually store the result of the previous step anywhere. The resulting inverted index would look like this:

Trigram	Posting list
”dec”	[0, …]
”exp”	[2, …]
…	…

Retrieval and scoring

As discussed before, there are likely to be symbols with a high final score but low initial rank. This problem is solved by BOOST iterators which can substantially increase the score of relevant symbols. The complete pipeline of query processing involves the following steps:

- Retrieval based on filtering (intersection and union) and boosting (covered in later subsection) iterators to increase the score of relevant symbols
- Truncation of top `N` results, typically `N` would be quite large here (`N ~ 10-100k`)
- Sorting truncated symbols based on the boosted score
- Truncating top `M` symbols (`M << N`) smaller `k`
- Scoring each symbol from the truncated list using fuzzy matching score, file proximity score in more computationally expensive manner
- Sorting symbols based on the final score
- Returning top `k` results (`k << M << N`, this would be typically `<100` results for code completions as showing too many results in the completion window is irrelevant)

Fuzzy search

A fuzzy lookup request operates on the lookup string (among other criteria), which is given as a field of `FuzzyFindRequest`. This string is split into the trigrams in the same manner symbol preprocessing happened during the index build stage.
Let's assume that the generated trigrams for the query strings are: `["dec", "ecl"]` (e.g. if the fuzzy lookup request was `"Decl"`).

Text search filtering

To filter symbols using text search in the first stage index would intersect posting lists of each trigram generated given the query string. Using the toy example introduced before, the algorithm would look at the inverted index, which can look like this:

Trigram	Posting list
`"dec"`	`[0, 7, 10, 40, 55, ...]`
`"exp"`	`[2, ...]`
`"ecl"`	`[0, 7, 15, 30, 55, ...]`
...	...

Now, the algorithm would create an AND Iterator for those posting lists, which correspond to each trigram found in the query string producing two iterators for the following posting lists: `[0, 7, 10, 40, 55, ...]` and `[0, 7, 15, 30, 55, ...]`. Assuming the Code Completion Request is to return up to 3 relevant results, these lists can be merged in an efficient manner using the Iterator interface in an efficient manner. The result would be `[0, 7, 55]` since these are the three first common symbols in both posting lists. Note that these are sorted by rank which makes them the most valuable candidates for a generic query.

Trigram generation

Trigram generation algorithm is crucial for the lookup quality. For the first iteration, it might be enough to use relatively simple trigram generation algorithm and improve the search quality later.

Step 1: Splitting symbol into chunks

During the first step, the query string is split into chunks. The `FuzzyFindRequest` API already provides an unqualified query string and stores scopes in a different field. Hence what's left to do here is to split the unqualified query string into chunks using the following rules:

- `_` is a separator (i.e. `unique_ptr` should be split into `["unique", "ptr"]`)
- Lowercase followed by an uppercase is a separator (i.e. `MyVariable` -> `["My", "Variable"]`, but `MAX_CANDIDATE_COUNT`->`["MAX", "CANDIDATE", "COUNT"]`)
- Sequences of consecutive uppercase letters followed by a lowercase letter: the last uppercase letter is treated as the beginning of a next chunk. Example: `MySUPERVariable` -> `["My", "SUPER", "Variable"]`

Digits are treated as lowercase letters.

Step 2: Normalizing text

The next step is normalizing the text by casting all chunks into lowercase. This should be done after the first step is complete, otherwise the second rule wouldn't be applied correctly.

Step 3: Trigram generation

The final step produces actual trigrams extracting several classes of trigrams out of the collected chunks. The rules are based on the observation that these trigrams are 3-char suffixes of paths through the fuzzy automaton.

- Each chunk is processed using the sliding window of three characters and the resulting views are returned as trigrams. Example: `“translation”` -> `[“tra”, “ran”, “ans”, “nsl”, ...]`.
- The next class of trigrams consists of front chunk letters (skipping more than 1 chunk is not allowed). Example: `[“translation”, ”unit”, ”decl”]` -> `”tud”`.
- Another class of chunks consists of a character from the chunk and two starting characters of any chunk which comes later (same as in two previous classes, the next chunk should be either next or the one after next). Example: `[“dec”, “hex”, “oct”]` -> `[“dhe”, “ehe”, “che”, “doc”, “eoc”, “coc”]`
- The last class of chunks consists of two consecutive characters from a chunk and the first character of the next chunk or the chunk after next. Example: `[“dec”, “hex”, “oct”]` -> `[“deh”, “ech”, “deo”, “eco”]`

Corner cases: handling short requests

Some queries are shorter than 3 symbols (e.g. when triggering code completion after the first/second symbol) and these should be correctly addressed, too. There are few possible solutions to this problem and they are briefly described below.
In this section, the algorithm is dealing with an incomplete trigram query, e.g. `?` or `??` where `?` represents any character from the alphabet of possible identifier symbols at that position.

Fuzzy matching each entry

Probably the most straightforward approach is just scoring the symbols via applying fuzzy matching by iterating through the symbols sorted by priority .

Generating incomplete trigrams

This corner-case can also be addressed by adding unigrams and bigrams to the symbol index.

Posting list Iterators

AND iterator

AND iterator manages intersection of posting lists. `min(scores)` is applied to produce the final score in the merged list.
An example use case where AND iterator should be used was covered in the subsection about symbol filtering.
In general, both AND and OR iterators would act in a similar manner. AND iterator should maintain the value of a lowest rank which is pointed by any of the processed iterators.

Algorithm description

Before the algorithm starts, each iterator is assigned to the `begin()` of each posting list and the lowest rank is chosen among the front ranks of each posting list. The algorithm consecutively advances each iterator to the lowest rank item. If it is missing from the corresponding posting list then lowest rank is updated and each iterator from the merging list is being advanced to the new lowest rank again. If all of the posting lists happen to contain that item, it should be added to the result. The process resumes after one iterator is calling `advance()` to move to the next item. As soon as any iterator reaches the end or resulting posting list is populated with enough items, the execution stops.
This would be trivial to maintain and update lowest rank in `O(1)` time.

Example

It would be easier to understand how the algorithm works using a small example to illustrate the description given before.

	Item ranks			
Posting list #0	0	3	7	END
Posting list #1	2	7	10	42
Posting list #2	1	4	7	42

The initial set of iterators points to the first item of each posting list and the highest is chosen among these initial ranks (here it happens to be `2`).

Lowest rank: 2	Item ranks			
Posting list #0	0	3	7	END
Posting list #1	2	7	10	42
Posting list #2	1	4	7	42

Each element which is pointed to by the corresponding posting list iterator, is highlighted in blue. The first step advances the iterator for posting list #0 to `2` by applying binary search to the range `[3, END]`:

Lowest rank: 3	Item ranks

Posting list #0	0	3	7	END
Posting list #1	2	7	10	42
Posting list #2	1	4	7	42

Since there is no `2` present in the posting list #0, the iterator advances to the closest higher value which is `3` in this case. Lowest rank is also updated since last iterator advanced past the previous lowest rank. Now, each iterator is being advanced to `3` (AND iterator uses `advanceTo(3)` on all of its children).
Posting list #1 iterator is advanced to `3` and happens to get to `7`:

Lowest rank: 7	Item ranks			
Posting list #0	0	3	7	END
Posting list #1	2	7	10	42
Posting list #2	1	4	7	42

The lowest rank is updated once again. The process of moving each iterator to the lowest rank should start from scratch: iterators for posting lists #0 and #2 are moved to `7` without updating the lowest rank value:

Lowest rank: 7	Item ranks			
Posting list #0	0	3	7	END
Posting list #1	2	7	10	42
Posting list #2	1	4	7	42

All of the iterators are pointing to the item with the same rank. Hence, it is present in all of the lists and added to the result. After that the posting list iterator #0 calls `advance()` which moves it to the `END`. Hence, there are no more items present in all posting lists and the process of merging lists is finished.

	Item ranks			
Posting list #0	0	3	7	END

Posting list #1	2	7	10	42
Posting list #2	1	4	7	42

OR iterator

OR iterator manages the union of the tokens in multiple posting lists and assigns a score of `max(scores)` where `scores` belong to the matched token in each processed posting list.
This iterator can be used to filter the scope of a symbol. For example, if the code completion query is triggered after typing `clang::clangd::Ind` the query would return symbols from `clang::clangd` **OR** `clang::` **OR** `::` (global) scopes. Similar to proximity paths, closest scopes can be upranked using boosting iterator.
Unlike AND iterator in the previous example, the inverted index used by OR iterator in such query maps scope (instead of trigrams) to the corresponding symbols.
Each step of OR iterator union process consists of

- Picking an iterator of the highest rank
- Adding the item it points to the resulting list
- Using `advance()` to proceed to the next posting list item

The choice of a data structure which would allow efficient updates of the highest rank with its iterator is obviously a priority queue with the Iteration Indices being keys and iterators as the values. As soon as the iterator reaches the end of its posting list, it is removed from the priority queue.

Boosting iterator

Boosting iterator operates on a single posting list and multiplies the score of each matched item by a given factor. It is useful for upranking symbols matching certain criteria, e.g. when symbol is defined in the same directory with opened files its score might be boosted by a factor of 3, if it is defined one directory below or above it can be boosted by a factor of 2 and so on.

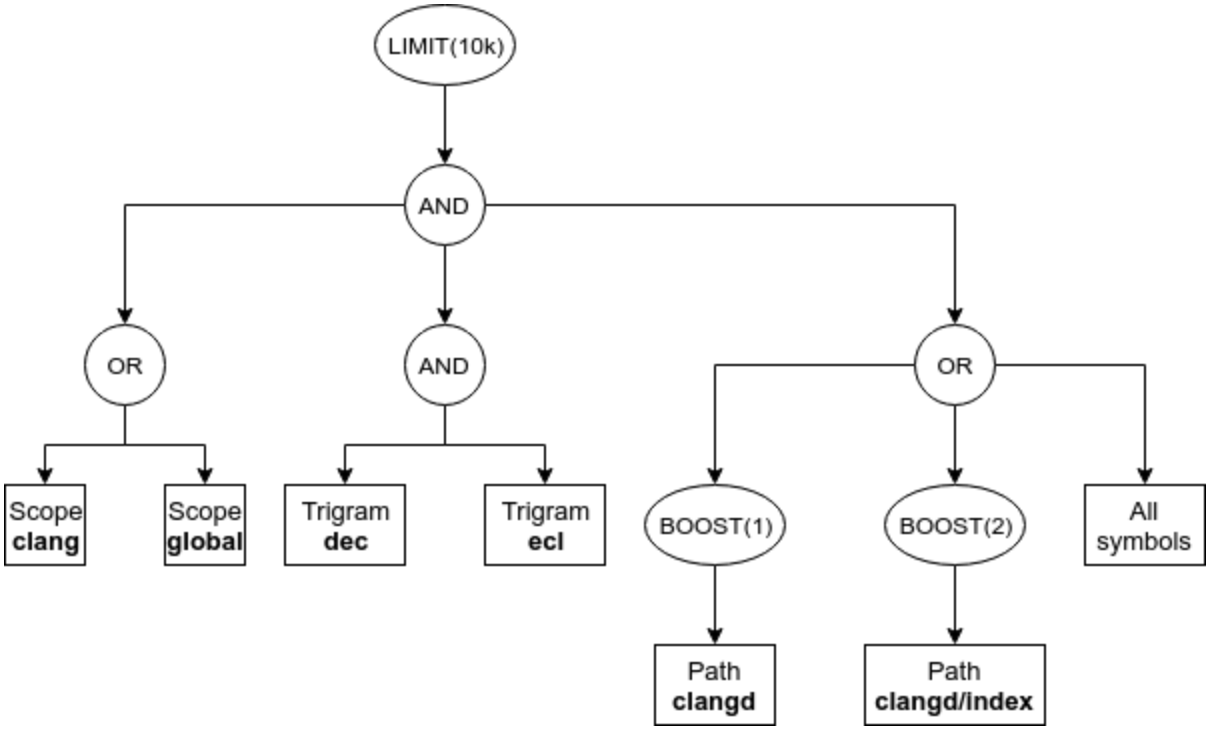
Supplemental retrieval iterator

This kind of iterators is used to limit the number of returned postings. As soon as the limit is reached the iterator advances to the end. Only those tokens which matched the full iterator trees are counted.

Query trees

A natural way of representing multi-level search queries is to build an iterator tree which

is to be “evaluated” bottom-up. Here’s an example of symbol lookup based on supplemental retrieval which boosts symbols based on file proximity. Similar techniques can be applied to retrieve symbols from nearby scopes in case user mistakenly specified a wrong one.
Supposing that this is a code completion query triggered after user typed `clang::Decl` and the current file is in `clangd/Index` directory, this is what a retrieval query might look like:



Abstraction interfaces

The following code snippet contains interfaces for the most substantial pieces of the symbol index core.

```
// Posting list contains a vector of (symbol, score) pairs sorted by a pre-computed
// metrics such as references count. It allows efficient operations via iterator
// interface which operates on a number of posting lists and produces a result of
// the query.
//
// NOTE: Static and incremental indices can have different storage type and
// different implementation.
class PostingList {
public:
  // Posting list entries contain the information about a specific item: its rank
```

```
  // and the score which will be used later in the filtering stage.
  struct Entry {
    const unsigned Rank;
    double Score;
  };

private:
  Container Entries;
};

// Posting list iterator implements the iterator interface over the PostingList
// instance.
class Iterator {
public:
  void advance();
  void advanceTo(unsigned Rank);
};

// Interface for the query tree evaluation. Given a specific query, it produces the
// final posting list.
class RetrievalSession {
public:
  PostingList retrieveSymbols();

  // Number of items truncated after the first retrieval stage (filtering via
  // iterator tree).
  Unsigned getBoostedSymbolCount();
  // Number of items which are scored using fuzzy matching and other techniques.
  unsigned getScoredSymbolsCount();
  // Final symbols count, i.e. the PostingList.size() returned by retrieveSymbols()
  unsigned getReturnedSymbolCount();
};

// Hashable clangd::Symbol Token, which represents a searching query criteria
// primitive. The following items are examples of tokens:
//
// * Symbol name for trigram-based search.
// * Proximity path primitives, e.g. "symbol is defined in directory
//   $HOME/dev/llvm or its prefix".
// * Scope primitives, e.g. "symbol belongs to namespace foo::bar or its prefix".
// * If the symbol represents a variable, token can be its type such as int,
//   clang::Decl, ...
// * For a symbol representing a function, this can be the return type.
//
// Tokens can be used to perform more sophisticated search queries by constructing
// complex iterator trees.
class Token {
```

```
public:
  // Returns precomputed hash.
  size_t operator()(const Token &T) const;
};

// Specifies what should be searched (e.g. path, scope, symbol name) and how (using
// prefix/fuzzy search or exact match).
class QueryAtom {
public:
  enum class MatchType : char {
    ExactMatch,
    FuzzyMatchSearch,
    PrefixMatch,
  };

  Type getType();

  llvm::Option<llvm::StringRef> getSymbolName();
  llvm::Option<llvm::StringRef> getScope();
  llvm::Option<llvm::StringRef> getPath();
};

class InvertedIndex {
private:
  llvm::DenseMap<Token, PostingList> PostingLists;
};

struct FuzzyFindRequest {
  std::string UnqualifiedName;
  // A mechanism for upranking certain symbols: e.g. based on user history it is
  // likely that symbol is defined in a specific file.
  std::vector<std::pair<Token, double>> BoostTokens;
};
```

Caveats and/or [rejected] alternatives

Mutable index architecture

The designed approach requires the symbol index to be an immutable structure. Because of that, merging two Indices would basically mean rebuilding an index from scratch given symbols from each Index. The alternative approach would be to allow Index structure mutability which would give the opportunity to merge, delete and modify symbols in the given index. While this is a tempting perspective, this would most likely

lead to inefficiency and higher latency in the lookup queries since the data structures used before would not be feasible for the efficient mutable operations. Our priority, in this case would, be faster lookups and hence the mutable architecture is a rejected alternative. If it is, in fact, possible to allow mutability without sacrificing core functionality performance this alternative would be reconsidered.

Current implementation

Existing symbol index implementation stores symbols in `std::vector` which is iterated whenever a lookup request is received and applies fuzzy matching to the candidates. Both static and incremental indices have the same interface, the results of queries to both of them are merged after each request. The incremental index is rebuilt each time opened file is changed.
Such implementation is not inefficient for several reasons, one which is that rebuilding incremental index too often is costly. Instead, the proposed solution would address minimal changes in the front layer of the incremental index, only rebuilding the index (merging both layers) once in a while.

Alternative approaches

There are several data structures used to store indices such as the suffix tree, inverted index, citation index, **n**-gram index and document-term matrix. To satisfy the desired properties of the design requirements, the proposed solution utilizes **n**-gram index and posting lists as none of the mentioned alternatives seem to improve the performance of the designed system.

Testing plan

Given the incremental nature of the proposed project implementation, it would be great to track the performance of the index implementation by examining benchmarks of the core symbol index functionality. While it might be very time-consuming to maintain a comprehensive benchmark setup covering all range of possible caveats and corner cases, it is still worth introducing a relatively simple benchmark infrastructure to get enough information about the performance evolution to make sure it is improving over time. Benchmarking in general is rather difficult and it is probably most important to track the relative difference between different revisions/versions. LLVM test-suite repository contains a set of benchmarks, but these are meant for the LLVM/Clang internals and hence this might not be a suitable place for Clangd benchmarks, which

might be located in clang-tools-extra repository in the end.

Work estimates

The main goal of this project is to complete the basic functionality covered in this design document and push it to the upstream while replacing current symbol index implementation by the end of September, 2018.

Potential extensions

This section introduces a set of features, which are substantial for reliable symbol index functionality but are not a priority yet. With that in mind, the design decisions should allow these extensions, but the initial implementation is unlikely to have any of them.

Misspelled queries

Having relevant code completion results despite having misspelled the beginning of symbol name would significantly improve the user experience. This is likely to have a positive influence on some of the refactorings (e.g. variable name suggestions) which are expected to be eventually available in Clangd.
A possible approach would be to swap each pair of characters while generating additional queries and to merge results of all resulting quires. That would potentially generate much noise and affect latency, but it is probably a viable solution.

Index compression

Symbol index can potentially occupy quite a lot of memory and therefore it might be beneficial to use any kind of compression to reduce that. The “Information Retrieval” book has a whole section dedicated to the index compression, it might be worth exploring the ideas presented there. An example of a rather simple idea is that posting lists can be very dense and in order to reduce memory usage it might be worth using delta encoding or any other compression scheme.
However, it might be not worth the effort if the index proves to be relatively compact. Right now, the experimental global symbol index tool produces a 300 Mb YAML file for the LLVM and Clang.

Generate and update symbol index during build process

Static Index generation is quite long (takes approximately an hour to build LLVM and Clang index on a rather fast machine with 11 cores and 64 Gb RAM). Apple recently uploaded an index-while-building patch which generates static symbol index as a part of the project build. This is used in XCode 9 but is the patch did not land yet and is under an ongoing review. Since static index generation takes so long, it would be nice to both trigger its generation upon build runs and to reuse information collected by Clang compiler in the build process to effectively build static symbol index. One of the difficulties is that symbols have to be sorted by some priority (such as the number of references) and not having them in one go makes such process more complicated. It is also important to mention that the discovery of new content via index-while-build is similar to that for the incremental index but with a different rate. The update mechanism can be similar in both cases.
This is potentially a collaborative effort with the index-while-building developers.

Appendix

More reading

- Regular Expression Matching with a Trigram Index or How Google Code Search Worked, 2012 Russ Cox
- google/codesearch - original Go implementation of `gsearch` on Github
- Search Engine Indexing page on Wikipedia
- Information Retrieval Book by Chris Manning (Chapters 4 covers Search Index Construction and explains Dynamic Indices in Section 4.5, Chapter 5 introduces Index Compression)
- Clangd documentation
- Microsoft Language Server Protocol (LSP) website with specification
- index-while-building patch in LLVM by Apple folks and the corresponding design document for index-while-building in XCode 9
- “Construction of fuzzy automata from fuzzy regular expressions”, 2011 Aleksandar Stamenković and Miroslav Ćirić