

Something about toyEngine

WEI YAN

wyglauk@gmail.com

Contents

Introduction	1
Architecture Overview	2
Interesting Things	3
Double Layers Term Lock	3
Scanner and Plugins	4
Information_manager and Plugins	5
Posting List Lazily Loading and Automatic Deactivation.....	5
Cleaner and Lazily Deleting of Posting Units.....	6
Persistence	7
Plain Search, MaxScore and WAND	8
API and toyEngine_operator	8
References.....	10

Introduction

The toyEngine is an implementation of the search engine invert-index, along with basic and advanced functionalities. Specifically this program is currently providing the functionalities including: **a)** persisting and loading of lexicon / last posting unit id / term associated information; **b)** posting list independent persisting and lazily loading; **c)** double layers lock service for posting list; **d)** posting unit adding, lazily deleting and posting list cleaning; **e)** document adding and deleting; **f)** posting list accessing status recording and automatically deactivating; **g)** inverted index reloading for garbage collection and reallocating post unit IDs; **h)** posting list scanning and simple document scoring models; **i)** three searching algorithms including plain search, maxScore and WAND;

From the perspective of design, the program mainly consists of three parts, **a)** inverted-index and associated operations; **b)** entities supporting the implementations of the operations; **c)** helper classes like commonly used basic data structures and various utils. The design of entities are mostly applying the schema of “mainstay and plugins”, in which specific functionalities and data structures are provided and maintained by the plugins, this is for the convenience of developing additional functionalities based on the current backbone.

Architecture Overview

The following figure shows the overall architecture of the program.

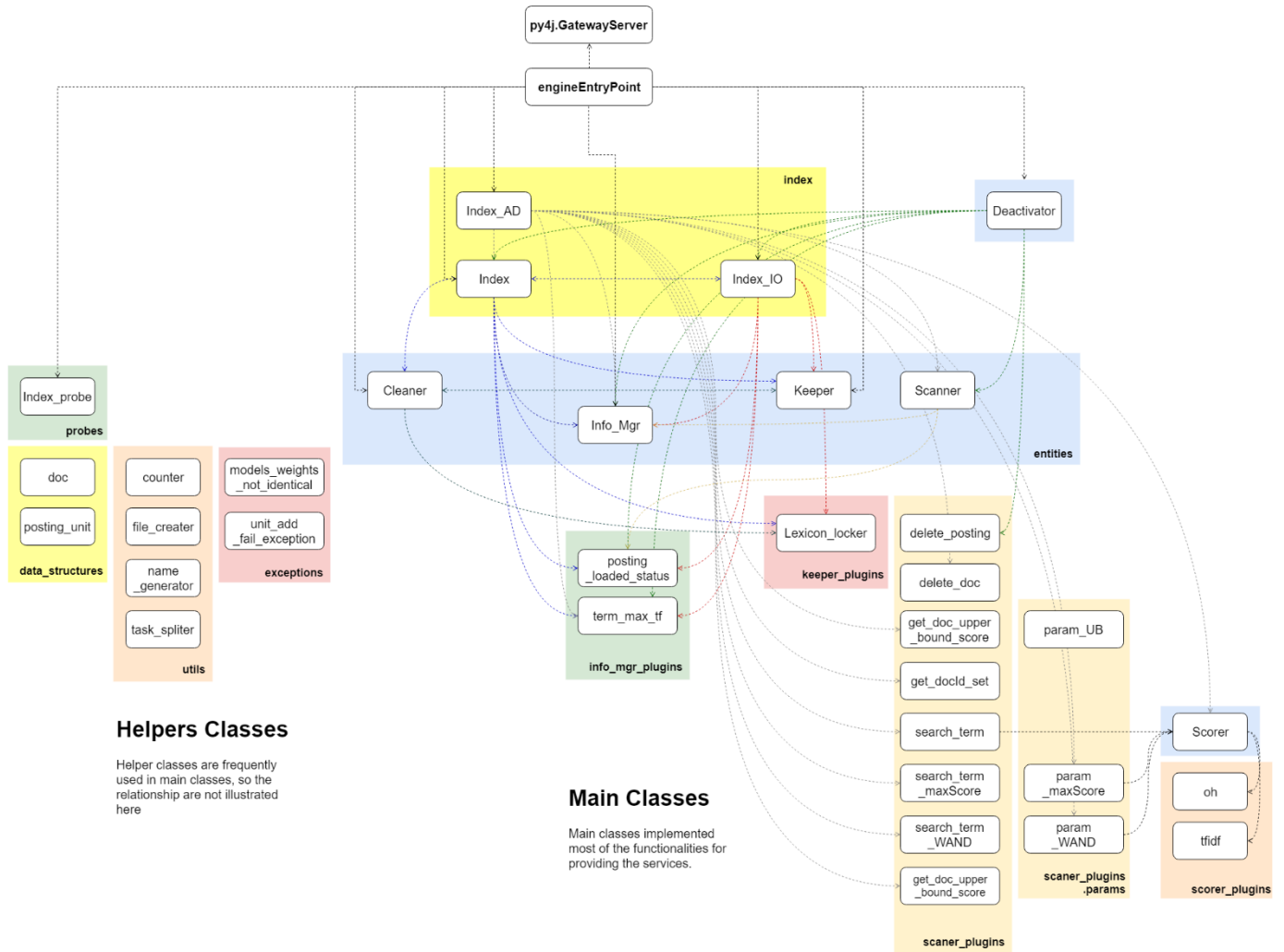


Fig 1. Overall architecture of toyEngine

On the left of figure 1, there are 9 helper classes belong to 4 packages, which are wildly used in other parts of the program; on the right, the rest classes belong to 6 packages (count to depth 2) consist the main classes which provide most of the functionalities.

To be more specific, on the right of figure 1, classes are in fact belong to 3 categories: **a)** classes in packages *index* (with yellow background); **b)** classes belong to *entities* package (with blue background); **c)** The rest classes which are plugins or parameters of plugins of corresponding entity classes.

To explain why the structure is designed as this, let us use the plain search functionality, which is implemented as *index_advanced_operations.search*, as an example:

```
// DAAT
for(String term : targetTerms) {
    counter documentScoreCounter = new counter();
    scanner.scan_term_thread st = new scanner.scan_term_thread(snr, search_term.class, documentScoreCounter, new String[] {term});
    st.run();
    threadList.add(st);
    counterList.add(documentScoreCounter);
}
```

Fig 2. Code snippet of *index_advanced_operations.search*.

```

public class search_term {
    public static counter docScoreCounter; // passed from the outside method

    public static void set_parameters (counter documentScoreCounter) {
        docScoreCounter = documentScoreCounter;
    }

    public static long conduct(posting_unit pUnit) { // conduct the operation
        long relatedUnitId = -1L;
        double score = scorer.getInstance().cal_score(pUnit);
        docScoreCounter.increase(pUnit.docId, score);
        return relatedUnitId; // affected post unit Ids
    }
}

```

Fig 3. Detail in scanner_plugins.search_term.

It can be seen from figure 2 that the function instantiated the *scanner.scan_term_thread* by pass the *scanner_plugins.search_term.class* into it, *documentScoreCounter* is a counter object worked as the parameter of *search_term* for collecting the score of documents contributed by posting units. What needs to be mentioned is, the *counter* data structure is highly inspired by the class *collection.Counter* [1] in python.

By defining the *search_term* as plugin of *scanner*, we can split the common functionality of scanning through the posting list with the specific functionality of calculating the contribution score from one posting. **This allows the developer does not need to care about the implementation details in scanner when need to develop some functionalities involve the scanning of posting list, as long as one follows the common structure of plugin classes – plugin classes needs to implement the *set_parameters* method and *conduct* method.**

Similarly, entities including *information_manager*, *keeper*, *scorer* and *scanner* are defined with plugins to provide specific functionalities.

Interesting Things

Double Layers Term Lock

When adding or deleting posting units during the serving of engine, we need to make sure the synchronization of modifications mainly on *index.lexicon* and *index.postUnitMap*, so we need to add locks on terms. In this program the locks are manipulated through the methods defined in *entities.keeper*, but the actual lock pool is provided by the plugin of keeper, *entities.keeper_plugins.lexicon_locker*:

```

private static HashMap<String, HashMap<String, Long>> lexiconLockInfoMap
private static HashMap<String, ReentrantLock> lexiconLockMap = new HashMa

```

Fig 4. Maps in lexicon_keeper.

As can be seen in figure 4, the *lexicon_locker* provides two maps, *lexiconLockInfoMap* and *lexiconLockMap*. The first one maps from term string to the information map which contains the *lockStatus* and *threadName*. *lockStatus* could set as 0 or a timestamp, and *threadName* denotes the thread successfully required the lock. *lexiconLockMap* contains the actual *ReentrantLocks* associated with the terms, which are created when new terms are added into the lexicon.

```

public int require_lock(Class lockerClass, String targetName, String threadNum) {
    int required; // 0 not required, 1 required

    HashMap<String, Long> infoMap = get_lockInfoMap(lockerClass).get(targetName);
    ReentrantLock targetLock = get_lockMap(lockerClass).get(targetName);

    // TODO: test
    // System.out.println(term + "!");
    if (infoMap.get("lockStatus") == 0) { // if one target is not being modifying, e.

        targetLock.lock(); // add the lock, here not using try.. as there will be a t
        infoMap.put("lockStatus", System.currentTimeMillis()); // change lock status,
        infoMap.put("threadNum", Long.parseLong(threadNum)); // record the thread tha
        required = 1;
    }
}

```

Fig 5. Snippet of entities.keeper.require_lock.

Figure 5 shows the snippet of *entities.keeper.require_lock*, it can be seen that the *lockStatus* is checked firstly, then the lock is added, and thirdly status and information are recorded, this is why it is called “double layer”. This design is useful when there are multiple threads try to add new documents into inverted-index, each thread needs to add many posting units, so it needs to require locks on the corresponding terms, if without the first layer checking *lockStatus*, all the threads try to add units of the same term will be holding up and wait for the lock, however, **with such *lockStatus* checking, only threads seeing status as 0 will be waiting, other threads will just return a failed lock requirement flag and skip this unit, as a result the overall efficiency is boosted.** To ensure all the units are eventually added, a retry logic is implemented in the *index.add_doc*.

Scanner and Plugins

In architecture overview section, the usage of scanner is discussed as an example of schema “mainstay and plugins”, here goes deeper into the implementation of scanner.

```
public void visit_next_unit (posting_unit pUnitCurrent, Class operationOnPostingList, ArrayList<Long> affectedUnits) throws Exception {
    if(pUnitCurrent != null) {
        Method conduct = operationOnPostingList.getMethod("conduct", posting_unit.class); // the class object already provide the necess
        long affectedUnitId = (long)conduct.invoke(operationOnPostingList, pUnitCurrent); // object -> long
        if(affectedUnitId != -1) { // -1 denotes the processed unit was not affected
            affectedUnits.add(affectedUnitId);
        }
        visit_next_unit(pUnitCurrent.nextUnit, operationOnPostingList, affectedUnits);
    }
}
```

Fig 6. entities.scanner.visit_next_unit.

Figure 6 shows the core method of scanner, which recursively accesses the *nextUnit* field of *posting_unit* object which points to the next posting unit, and invokes the *operationOnPostingList.conduct* method on each posting unit, for *index_advanced_operations.search*, the operation is *scanner_plugins.search_term*; for *index_advanced_operations.search_maxScore*, the operation is *scanner_plugins.search_term_maxScore*; for *index_advanced_operations.search_WAND*, it is *scanner_plugins.search_term_WAND*, etc. All these plugins implements two methods *set_parameters* and *conduct*. Use *search_term_WAND* as an example:

```
public class search_term_WAND {
    public static param_search_term_WAND param; // need to be shared by all threads, nc

    public static void set_parameters (param_search_term_WAND paramSearchTermWAND) {
        param = paramSearchTermWAND;
    }

    // operations are done by methods provided by param
    public static long conduct(posting_unit pUnit) {
        long relatedUnitId = -1L;
        relatedUnitId = param.try_to_score_add_doc(pUnit);
        return relatedUnitId;
    }
}
```

Fig 7. scanner_plugins.search_term_WAND.

As shown in figure 7, *search_term_WAND* implements the two common methods of plugins. However, what need to be noticed is, in *search_term_WAND* the parameter is not simple data structure like *search_term* has, instead it is a new param data structure *param_search_term_WAND*.

```
public class param_search_term_WAND {
    // use this class to pass multiple data structu
    scorer scr;
    Iterator<Map.Entry<String, Double>> tMaxSI;
    HashSet<String> vDocSet;
    counter curUB;
    HashMap<String, HashSet<String>> tDocSetMap;
    counter docSC;
    int tpK;
}
```

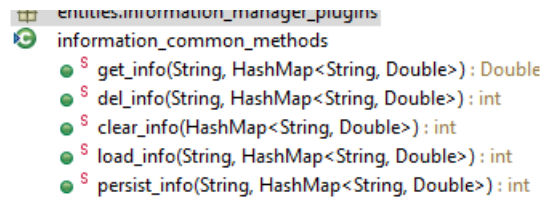
Fig 8. Snippet of scanner_plugins.parameters.param_search_term_WAND.

Figure 8 shows a snippet of *param_search_term_WAND*, which wraps up the necessary data structures passed from *index_advanced_operations*, also implements the logic of scoring the document in method *param_seawrch_term_WAND.try_to_score_add_doc*. **The reason of implementing the scoring process in the param instead of in plugin is for the convenience of making use of the wrapped data structures.**

Scanner provides two types of threads *scan_term_thread* and *scan_term_thread_with_lock* for making use of the scanner with multi-threads, the second one require term locks before the scanning, which is used in *entities.deactivator* for deleting posting units of the stale posting lists.

Information_manager and Plugins

It is very much like the idea of *keeper*, *information_manager* works as a unified interface to the information maps maintained in its plugins. Among *information_manager_plugins*, there is a class *information_common_methods* provides the shared common methods for plugins.



```

Entities.information_manager_plugins
information_common_methods
  • get_info(String, HashMap<String, Double>) : Double
  • del_info(String, HashMap<String, Double>) : int
  • clear_info(HashMap<String, Double>) : int
  • load_info(String, HashMap<String, Double>) : int
  • persist_info(String, HashMap<String, Double>) : int

```

Fig 9. Common methods of *information_manager_plugins*.

Figure 9 shows the methods provided by *information_common_methods*, in fact a more natural implementation approach is making a parent abstract class provides all the common methods and define several abstract methods, **however, currently for the convenience of using reflections, most of the data structures and methods provided by plugins are made static, in the next version of toyEngine, this could be changed.**

Additionally, the current two plugins are *posting_loaded_status* and *term_max_tf*. The first one **records information of when a term is loaded or accessed last time**, so that this information is set when posting units are created / loaded / scanned. The *deactivator* relies on the *posting_loaded_status* to check whether a posting list is expired, which means a posting list is not accessed for a pre-set period. The *term_max_tf* is for recording the **max term frequency of a term**, such that is frequently updated when new posting units are added, currently this information is persisted to path *persistence/information/term_max_tf*, whose content is shown in figure 20.

In next version, a new basic data structure *term* will be extract out like *posting_unit*, and information like max term frequency could be persisted along with the lexicon.

Posting List Lazily Loading and Automatic Deactivation

In order to save the memory consumed by the engine, the posting list are not persisted in memory all the time. In fact **during the starting process, the lexicon, last posting unit ID, term related information, added documents information are loaded directly, but not the posting lists, they are only loaded when the *index_io_operations.load_posting* method is invoked**, which is always invoked when a term is searched or the whole inverted-index is reloaded. As we just want to load the terms that are not already loaded, the method *index_io_operations.check_term_loaded* is used to check the load status.

```

// check if the last posting unit is existing in the posting list of term,
private boolean check_term_loaded(String term) {
    boolean loadedFlag = false;
    Double lf = infoManager.get_info(posting_loaded_status.class, term);
    if(lf != null) { // as long as the info is existing means is loaded
        loadedFlag = true;
    }
    return loadedFlag;
}

```

Fig 10. *check_term_loaded*.

Figure 10 shows the *check_term_loaded* method, within which the *information_manager* described in last paragraph is used to access the *posting_loaded_status.infoMap* and to get the load status of the posting list.

```
// lazily load the posting list of target terms
public long[] load_posting(String[] targetTerms) {
    long[] loaded_units = new long[] {};

    for(String term : targetTerms) {
        if(!check_term_loaded(term) && idx.lexicon.containsKey(term)) {
```

Fig 11. Snippet of load_posting.

```
private void persist_postings() {
    try {
        for(String term : idx.lexicon.keySet()) {

            if(check_term_loaded(term)) { // only try to persist
```

Fig 12. Snippet of persist_posting.

As figure 11 and figure 12 show, the *check_term_loaded* is used for each term in both the *load_posting* and *persist_posting* so as to realise the lazily and term independent loading, the persisting will be discussed more in Persistence section.

Also by using the *information_manager*, a method *entities.deactivator.check_expired* is used to tell whether a posting list is accessed within the expiring time constraint from the last time it was accessed. **When a the posting list of a term is checked as expired, the deactivator persists the posting units to local disk firstly, then makes use of scanner and the corresponding plugin to delete the units from the memory.**

```
for(String[] workload : workloads ) {
    scanner.scan_term_thread_with_lock st = new scanner.scan_term_thread_with_lock(snr, delete_posting.class, "", workload);
    st.run();
    threadList.add(st);
}
```

Fig 13. Snippet of entities.deactivator. deactivate.

Additionally, *deactivator* works as an independent thread, it periodically checks the expired (loaded) status of posting lists, and conduct the deactivations.

Cleaner and Lazily Deleting of Posting Units

Besides the directly posting units deleting of deactivation, for the reason of making the posting list cleaning and garbage collection mechanism simpler, **toyEngine refers to the lazily deleting mechanism in GFS[2] and HDFS [3], which does not really remove the targets from the system during the deleting process, instead it uses a periodically merging / compressing mechanism to eliminating the stale data lazily.**

```
public class delete_doc {
    public static String docName;

    public static void set_parameters (String targetDocName) { //
        docName = targetDocName;
    }

    public static long conduct(posting_unit pUnit) { // the input
        long affectedUnitId = -1L;
        if(pUnit.docId.matches(docName)) {
            pUnit.status = 0; // lazily delete
            affectedUnitId = pUnit.currentId;
        }
        return affectedUnitId; // affected post unit Ids
    }
}
```

Fig 14. entitie.scanner_plugins.delete_doc.

As an example, the *index_advanced_opertions.delete_doc* makes use of the scanner and its plugin *delete_doc*, which set the unit status as 0 instead of directly removing it from the *index.postUnitMap*.

The *entities.cleaner.clean_unit* scans through the posting list, isolates the zero-status units and link the previous and following units, then finally deletes the stale ones.

```
posting_unit curUnit = idx.postUnitMap.get(pUnitId);
int pStatus = curUnit.status;
if(pStatus == 0) {
    posting_unit prevUnit = (pUnitIndex != 0) ? idx.postUnitMap.get(postingUnitIds.get(pUnitIndex - 1)) : null; // skip the first
    posting_unit nextUnit = (pUnitIndex != postingUnitIds.size() - 1) ? idx.postUnitMap.get(postingUnitIds.get(pUnitIndex + 1)) :

    // relink
    if (prevUnit != null) { // when current unit is not the starter
        prevUnit.link_to_next(nextUnit);
    }
    if (nextUnit != null) { // when current unit is not the ender
        nextUnit.link_to_previous(prevUnit);
    }

    delPostUnitList.add(pUnitId);
}
}

// avoid the ArrayList is changing when iterating it
for (Long pUnitId : delPostUnitList) {
    // delete from lexicon and postUnitMap
    postingUnitIds.remove(pUnitId);
    idx.postUnitMap.remove(pUnitId);
}
```

Fig 15. Snippet of cleaner.clean_unit.

What needs to be mentioned is, currently the *cleaner* is not working independently as the *deactivator*, this is because the posting lists are lazily loaded, at most of the time the majority of the posting units should be on the hard disk instead of in the memory. Instead, the *cleaner* is now used by *index.reload_index* method.

```
index_io_operations.get_instance().load_all_posting();
clr.clean();

index_io_operations.get_instance().persist_index(); // at this time the ids are not corrected
clear_index();

try {
    // collect the pathes of posting files
    // ref: https://stackoverflow.com/cn/q/12079625
    List<String> postingPaths = Files.walk(Paths.get(configs.index_config.postingsPersistencePath), 2)
        .filter(path -> Files.isRegularFile(path))
        .map(path -> path.toString())
        .filter(path -> path.endsWith("posting"))
```

Fig 16. Snippet of index.reload_index.

Figure 16 shows the snippet of *index.reload_index*, which **firstly load all the posting units into memory, then makes use of *cleaner.clean* to eliminate all the stale units physically, then dumps the index to the hard disk, empties the memory, and re-adds all posting units back**. This process fulfil two objectives: a) physically delete the stale data; b) recycle and reallocate the posting unit IDs.

Persistence

In toyEngine, the posting lists, lexicon, term related information, id of last posting unit and added document information are persisted in the local hard disk.

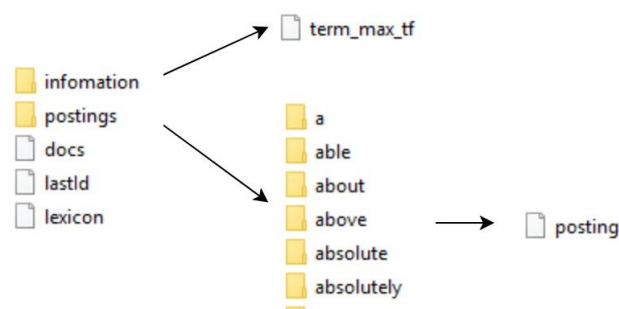


Fig 17. Persistence structure.

Figure 17 shows the persistence structure, information are persisted in isolated files under path *persistence/information*, currently there is only *term_max_tf*, which will be merged to lexicon in next version (this can be easily done by developing a new *information_manager* plugin). **Posting lists are persisted under different directories named with the target term**, the advantage of this structure is supporting the independent term loading and persistence easily, but the drawback is the terms are not case sensitive.

Specifically, the persisted files look like:

```
half 2471 2472 2473 2474
chips 1115 1116
numerous 3720 3721 3722
nachos 3537 3538
spacious 5125 5126
```

Fig 18. Persisted lexicon.

```
chips 1.0
numerous 1.0
nachos 1.0
spacious 1.0
your 1.0
these 1.0
tea 7.0
```

Fig 20. Persisted term_max_tf.

```
a 117 118 -1 {} -- 1
a 118 119 117 {"tf":3,"oh":1} /test_1/-Km-gkgaAJAx37yEHIERDg
a 119 120 118 {"tf":1,"oh":1} /test_1/loPgWJHJiKZShjdJxZHylA
a 120 121 119 {"tf":1,"oh":1} /test_1/3KQ3_xutS5R3mX7HsPwpgw
a 121 122 120 {"tf":1,"oh":1} /test_1/40sWchjx-4rQ273cb85T7A
a 122 123 121 {"tf":3,"oh":1} /test_1/51TLGhFncBnpaBN5vHlCw
```

Fig 19. Persisted posting list.

```
/test_1/bifEpGC-8TE19JPZwn-Rpw 22 {}
/test_2/PIsUSmvaUWB00qv5KTFIxA 43 {}
/test_2/y2lFom8a_SdAyC6I0v554w 46 {}
/test_2/zEDdYhDYYfvd8bSQqpc_ww 41 {}
/test_2/UXmBQNWMD1PBLDC_nbTsSw 28 {}
/test_1/gDBVb6Qdg5VAr2L95NEeFw 37 {}
/test_1/40sWchjx-4rQ273cb85T7A 80 {}
```

Fig 21. Persisted document information.

Figure 18 – 21 show the content of the persisted files. In figure 18 the sequence of number following the term is the corresponding list of posting unit IDs, in next version this will be change to only storing the starter posting unit ID along with term related information, i.e. *term_max_tf* shown in figure 20 will be merged in to the lexicon, this modification is for furtherly saving the memory.

Plain Search, MaxScore and WAND

The toyEngine currently provides three searching algorithms: plain search, maxScore and WAND. **a)** plain search demands scoring all the added documents of the target term then conduct ranking based on the scores, which is implemented by *index_advanced_operations.search*; **b)** maxScore [4] avoids scoring all the related documents by calculating the upper bound scores of the documents firstly, then compares them to the current topK threshold, only conducts the real scoring when the upper bound is over the threshold, this algorithm is implemented as *index_advanced_operations.search_maxScore*; **c)** WAND [5] is also trying to score as fewer documents as possible, but different from maxScore, it focus on find the combination of necessary terms in terms of the current threshold, then only scoring the documents related all the necessary terms, this algorithm is implemented as *index_advanced_operations.search_WAND*. What needs to be mentioned is, currently the *search_WAND* is only an imperfect version which demands the user to rank the terms in order of descending importance, which means contribution scores from term could be lost if it is not the most important one.

In the implementation of maxScore and WAND, the threshold and contribution score calculations are conducted in the parameter classes of corresponding scanner plugins, i.e.

entities.scanner_plugins.parameters.param_search_term_maxScore and *...param_search_term_WAND*, **because they are operating the shared document score counter to get the threshold, so that the synchronization needs to be considered**. For this reason the methods of *counter* are implemented as thread safe.

API and toyEngine_operator

The toyEngine provides services through the API defined in *engine_api.engineEngrypoint*, which currently makes use of *py4j*. *GatewayServer* to allow the accesses from python by using the *py4j* [6] library, the communication is in fact rely on http protocol. Base on the *py4j* library to communicate with the toyEngine, **the toyEngine_operator is developed with python for a) processing the documents, b) generating posting units in persisting format and c) operating the engine API to feed the data into inverted-index.**

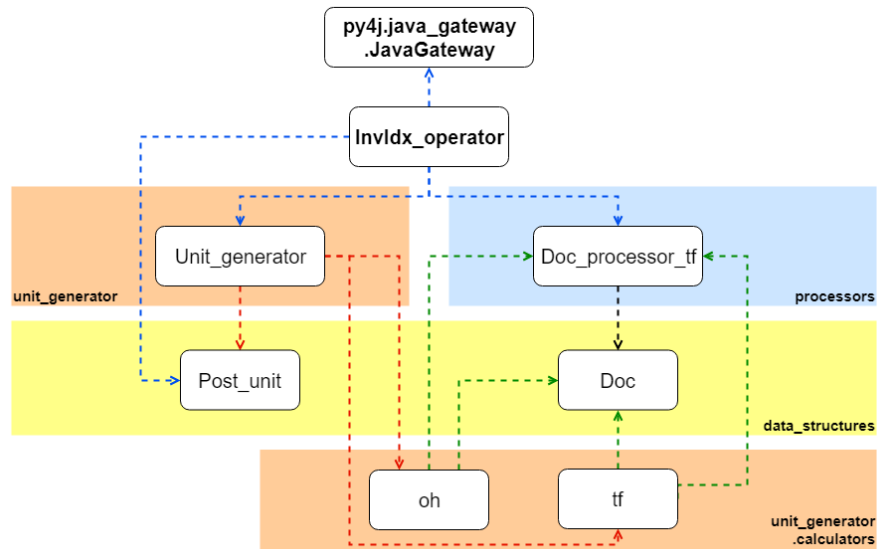


Fig 22. Architecture of `toyEngine_operator`.

The inverted-index operations are defined in class `operators.Invlidx_operator`, currently only including simple functionalities like adding / deleting documents, adding / deleting sources (batches of documents in directories corresponding to the sources), searching, showing statistics, etc.

References

- [1] Python Software Foundation. (n.d.). [collections.html#collections.Counter](https://docs.python.org/3/library/collections.html#collections.Counter). Retrieved from python: <https://docs.python.org/3/library/collections.html#collections.Counter>
- [2] Ghemawat, S., Gobioff, H., & Leung, S. T. (2003). The Google file system (Vol. 37, No. 5, pp. 29-43). ACM.
- [3] Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010, May). The hadoop distributed file system. In Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on (pp. 1-10). Ieee.
- [4] Turtle, H., & Flood, J. (1995). Query evaluation: strategies and optimizations. Information Processing & Management, 31(6), 831-850.
- [5] Broder, A. Z., Carmel, D., Herscovici, M., Soffer, A., & Zien, J. (2003, November). Efficient query evaluation using a two-level retrieval process. In Proceedings of the twelfth international conference on Information and knowledge management (pp. 426-434). ACM.
- [6] Dagenais, B. (2018). [getting_started.html#writing-the-java-program](https://www.py4j.org/getting_started.html#writing-the-java-program). Retrieved from Py4J: https://www.py4j.org/getting_started.html#writing-the-java-program
- [7] Iadh Ounis, C. M. (2018). Information Retrieval Slides. Information Retrieval H/M. Glasgow, UK: University of Glasgow.
- [8] Craig Macdonald, I. O. (2017). Information Retrieval Infrastructure. Information Retrieval H/M. Glasgow, UK: University of Glasgow.