Thread Search Engine

Introduction:

The prime motivation for development of the Thread Search mechanism was the realization that large amount of the information that can be productively shared online is in fact not available in any stable format on the net. For example, let us consider a simple logical link: asynchronous calls for web browser can be achieved with JavaScript XMLHttpRequest object (Ajax). Yet a person who is not familiar with key words such as XMLHttpRequest or Ajax will need to complete additional filtering of results retrieved with generic keywords to disregard any multi-threading approaches in languages such as C++ or Java. Further, it can be reasonably assumed that similar types of questions can and do appear for many users – thus once a user establishes a logical link between asynchronous calls for web browser and XMLHttpRequest object, this link can be indexed and later accessed for similar search query. Such well ordered logical links are precisely what we call Search Threads. Each such logical thread is created as the user proceeds with search queries. Additionally, for each query within a thread we establish a list of terms that resulted in a click on this query within a thread. Based on relevance of each query within a thread, length of thread, and utility of each search query determined by the amount of clicks, we derive a heuristic to assign appropriate score for a thread. The Thread Search mechanism is implemented as Java web application running on Apache Tomcat container with Lucene used as base index engine implementation.

Functionality Overview:

The Thread Search engine is meta-data driven search engine which attempts to increase relevance of retrieved result by organizing the order of queries and providing these ordered list as search results. The Thread Search mechanism on majority of requests retrieves relevant threads and updates the index with extension to current thread. Thus, we create a mechanism where user does not need to take any specific action to increase the utility of the index for the user base. Each query that is submitted to the SearchServlet by the user can be stored in the index with several key attributes. Here we will discuss the necessary attributes of query that is stored in the index. First, we use unique SESSIONID of the user to link all the queries submitted within the same session into Search Thread. Thus, a thread in this context is synonymous to search SESSIONID – that is we can establish a one-to-one correspondence between these entities. Next, we use the enumerator attached to each indexed query to order the queries within a thread. This is important architectural decision in that we gain a lot of flexibility in scoring of the threads by storing each query separately and reconstructing the thread during the runtime. On the other hand, we also suffer certain reduction in efficiency as more runtime processing is needed before thread is returned to the user. Although this is not implemented in the current version, we can try to avoid the problem by double storing the results in an index. We can use currently implemented query based index to assign accurate and flexible scoring to thread and thread based index to quickly retrieve threads (using SESSIONID) once the ranking of threads is determined. The following is the high level overview of retrieval procedure for a Search Thread. First, we take user’s query and use Lucene index to do a simple search within index on a query fields. Each returned hit for user’s query is retrieved with a SESSIONID and enumerator within a session. Using SESSIONID as a common key we reconstruct the thread and order queries using the enumerator. The base score for each returned query is used in the compound calculation for a thread score. The precise heuristic that we use for thread score will be discussed in “Scoring of a thread” section in more detail. However, it should be noted that although there are other factors in the ranking of threads such its length and the amount of clicks that it triggered, it was noted that it is very important that the base score for each retrieved query is not substituted by simpler, but perhaps more efficient, Boolean flag indicating a match. Although
each query within a Search Thread is very simple and short document, the presence of TF-IDF weighting due Lucene’s scoring have showed in the experiments that more relevant threads are ranked higher accordingly. Especially, it should was noted that it is the IDF part that plays very important role with query documents that are very short. Further, for each query that is displayed within a thread we also maintain separate lists of the terms that generated a click on that query. However, to save on bandwidth and make the retrieval of results more efficient we only create Ajax trigger for asynchronous call to retrieve the actual list of terms that resulted in a click. The trigger is attached to the onclick JavaScript event for a tag displaying the length of that list. Please see sample screenshot in “Sample Results” section for example of such triggers. Similarly this list is updated asynchronously when user clicks on a relevant thread. Thus, there are two events that are attached to each clickable term within a thread. First, the click submits this term to Google GET based API (other base search engines’ API can be used). Second, a search term that was used to pull these relevant threads is added to the list of “search queries that clicked” for clicked query. The list maintains only unique search queries, thus if the search query that resulted in a click is already in the list – it is not added. Similarly, each Search Thread contains only unique search queries – with this restriction we assume that the first occurrence of the search query determines its position within a thread. For example, if the user searches for “One”, followed by a search for “Two”, followed by search for “Three” the thread will look like this: One ➔ Two ➔ Three. Subsequent search for term “Two” again does not change the order of the terms – that is the thread is not reorder as One ➔ Three ➔ Two. Overall, the meta-data driven search mechanism is focused on the retrieval of the Search Threads created by user base to assist and predict the set of relevant queries. Below we will discuss some parts of the engine’s functionality in more detail.

Query-Driven Index:

This mechanism is very dependent on the quality of index and since the index is driven by user queries, considerable effort should be completed to ensure that index maintains its utility. First, it should be quite clear that not every search query that is entered is part of the useful semantic thread of queries – thus, we provide functionality to the user to avoid building new thread and simply search within existing index. More precisely, there are two ways to search within thread meta-data index. First, user can submit a query for both result retrieval and index insertion by using “Build Thread” button. Second, a user can submit a query to only retrieve results without inserting it into index by using “Simple Search” button. The second option is provided for those cases when user is aware of the fact that ordered list of queries does not carry any semantic meaning. Although, the system does not benefit from those users that do no expand its index, this option should reduce the number of low quality Search Threads. Further, a threshold of minimum length is established for all the threads. This threshold prevents insertion of thread into index before the minimum length is obtained. The Search Threads that are below minimum length are cached in SESSIONID key driven synchronized static HashMap. Once the thread is long enough it is removed from the HashMap and inserted into the Lucene index. The threads that are in HashMap are not searchable by the user base but are available to the owner the thread. More precisely, the interface of the Thread Search engine, independent of the length of the thread, displays the thread that is being created by the user. At this point this threshold is set to be 3 but can be altered easily. Current setting for threshold was motivated by the observation that single query thread can not create any logical connections and a thread of length two quite often adds noise to the results and thus is also excluded from the index insertion. Finally, in order to periodically remove incomplete threads from cached HashMap those threads that that were never inserted into index and for which corresponding session has expired), asynchronous java thread runs once a day. When Search Thread is cached the timestamp of last modification is also recorded; thus, a cleaning java thread only verifies that timestamp of cached Search Thread is outside of scope of session that can still be active. All inactive Search Threads are removed as no user can increase the length of Search Thread once its session expires. Currently, the session is set to expire within 30 minutes of user inactivity. Overall, one of the most important functionality of Thread Search is its index – thus, in this section we discussed couple of mechanisms that are implemented to improve index’s quality. These are only very preliminary mechanisms that need be extended to more complicated algorithms. We can also note that the target user base for this engine is composed of users involved in complicated research where typical search engine might not be very helpful. On the other hand, it is expected that for simple search terms Thread Search
engine can only introduce unnecessary overhead. Thus, using this knowledge we can proceed to implement aggressive filter that blocks common words that are not likely to be used in research thread. This filter should not only ignore such common terms in scoring (similarly how stop words are ignored) but prevent the insertion of simple search threads completely. Ultimately, it might be possible to implement sophisticated logic that without informing the user makes a decision to insert into index, disregard a thread, split a thread that is too long, or split thread that may contain several distinct semantic fields. These features; however, are not added in the current version.

Scoring of a thread:

In this section a more detailed description of Thread Search scoring is provided. The code below runs when all the information is already retrieved from the Lucene’s index and SESSIONID and enumerator attributes have been used to construct a thread. At that point a score is assigned to a Search Thread. Let us first introduce some data structures that will be helpful in code review. First, each thread essentially is an ordered list of queries – we store this list in ArrayList<String> threadLinks, further certain nonempty subset of this list is a set of queries that matched user input. The raw Lucene based scores for queries that matched are stored in HashMap<String, Float> matchedQueries, where each score is accessed by the query key in the threadLinks. If a particular query does not have a value in matchedQueries then it is not a matched query. Finally, for each query in a thread we are interested in a number of distinct search queries that resulted in a click on that query. This information is stored in HashMap<String, Integer> clickCounts where click counts are accessed by the search query key in the threadLinks. In this implementation absence of value for particular query in clickCounts HashMap indicates that no clicks have been made on this query. With this information we arrive at the following heuristic score determination:

```java
for (int ind = 0; ind < this.threadLinks.size(); ind++) {
    String query = this.threadLinks.get(ind);
    Integer numClicks = this.clickCounts.get(query);
    if (numClicks == null) {
        numClicks = 0;
    }
    Float termScore = this.matchedQueries.get(query);
    if (termScore != null) {
        termScore = 1.0f + termScore * 9;
    } else {
        termScore = 0.7f;
    }
    termScore *= 1.0 / (Math.pow(ind + 2.0, 1.2));
    termScore *= Math.log(Math.E + Math.pow(numClicks, 2.0));
    this.score += termScore;
}
```

Perhaps the first part we can note here is that even the queries within a thread that are not matched are given certain positive weight. This is important since we would like to boost the score of the terms that have been clicked by multiplying the base positional score by the factor of Math.log(Math.E + Math.pow(numClicks, 2.0)). Thus, if no base weight is given to unmatched queries this approach would not work. Further, since the base score to unmatched queries is below that of the matched queries – only queries that have been clicked on multiple times (and thus, in theory, are high utility queries) can substantially change the scoring of the Search Thread. This behavior proved to be useful during scoring heuristic tuning. Further, we can also note that base Lucene query score is converted into base positional score – a score where the search queries appearing earlier in the Search Thread are given more weight than those appearing later. The motivation behind this approach has two important parts. First, by introducing
this model we limit the advantage longer threads can have over the shorter threads. In fact, relevant and moderately long threads should have the best scoring in this model. This approach was derived by noting that although longer threads may contain more information, quite likely they are also not as focused on particular semantic field. Thus, a user might be overwhelmed with information in very long thread. Short threads might suffer from the opposite problem where the lack of number of connections between the thread components may fail to establish complete semantic field. Threads that are moderate in their length, in theory, should be both focused and relatively complete in their coverage of particular field. Second, we note that that a term that appears earlier in the Search Thread might be more useful to the user in that there is more information available that might correctly predict target search field. On the other hand, a match towards the end of Search Thread might only provide user with information that appears in the beginning of the thread and may already be known to the user. Overall, while more experiments is needed to establish more precise coefficients and approaches in thread scoring, it appears that current components are quite useful in ranking of Search Threads.

Similar implementations:

In order to analyze which systems implement functionality similar to this system, let us first underline the target usage pattern of Thread Search engine. As explained above this system is unlikely to be more useful than base search engine on simple or navigational queries. On these types of queries it is expected that base search engine will be able to retrieve target documents with high precision. On the other hand, when user is involved in a research the precise set of target documents is not well defined even for the user. In a research user is interested in discovering semantic links between topics and fields. Well defined Search Thread provides exactly that. Thus, Thread Search allows research work of a particular user to be reused by a research community focused on the same topic or field. With this definition of functionality we can arrive at several similar tools; however, none of these tools implement approach of Thread Search engine directly. Perhaps, the first engine that may be considered to be similar is Amazon.com suggestion mechanism. It is expected that based on the entities viewed on the Amazon.com pages, user will be offered some related entities. Usually these suggestions are based on the fact that users that view similar topics eventually purchase products within certain domain. This semantic linking mechanism, in fact, is very similar to links established between search queries within a Search Thread. Additionally both engines are improving their functionality without significant user’s effort. In case of Amazon.com user is simply browsing the items of interest where in case of Thread Search, user is simply searching meta-data. However, there are some important differences as well. First, in case of Amazon’s product selection there is a finite set of items that need to be considered — this should allow for reliable and efficient user pattern tracking. On the other hand, Thread Search engine does not have a finite set of entities with which Search Threads can be constructed. Further, for product selection each product thread needs to only contain two items: the browsing history that user has accumulated and the item (or set of items) that majority of users have selected with similar browsing history. In case of Thread Search the length of thread can be arbitrary and any query within a thread can be important. Another engine that in many ways resembles the functionality of Thread Search is Wikipedia. We can first note one large difference between Wikipedia and Thread Search: maintaining content in Wikipedia required targeted effort from users — while, ideally Thread Search mechanism does not need any specific effort from user to improve the quality of its index. Yet, there are very important similarities in that in both engines the utility of the tool is dependent entirely on the actions of its user base. Further, both engines attempt to provide semantic links within similar topics. Thus, in a way we can look at Thread Search as Wikipedia engine without description field for each topic. While absence of description field is obviously a disadvantage in terms of usability of content, it is an advantage in that no effort is needed to create it — thus in theory larger set of topics can be covered. Overall, while there are tools that may present similar concepts to the users, it appears that Amazon.com suggestions mechanism and Wikipedia are the most relevant technologies to Thread Search mechanism.
**Sample Results:**

The ability to accurately analyze the performance of this tool as a whole and as a scoring mechanism depends on existence of extensive index. Since the index is created by users who utilize the search functionality and very limited number of search queries have been made on the live version of this implementation, the analysis may be subject to insufficient data. Nonetheless, the interface may not suffer as much from this problem. Thus, we proceed with analysis of search engine’s interface. The following is a screenshot for typical result list returned by *Thread Search*:

![Screenshot of Thread Search interface]

In overview of this sample, we first can note that user’s first query was “ajax technology” from which user proceeded to search for “ajax issues”. Thus, **ajax technology → ajax issues** thread is now being created by the user. We can also note that since this thread’s length is less than **threshold=3**, at this point it is not inserted into index and is maintained within a synchronized shared HashMap. Further, in the main section of the interface we can observe the list of returned **Search Threads**. In each **Search Thread** a query that matched user’s input is highlighted in green color. Each query within a **Search Thread** is clickable link that submits the query to base search engine. For each query within a **Search Thread** a number in parenthesis indicates the number of clicks that have been made on this query. The thread score is thus adjusted accordingly as described in the scoring of a thread section. Finally, a right sidebar in this screenshot displays the exact terms that clicked on the query “*ActiveXObject in ajax*” appearing in the third retrieved thread. For each thread a compound score can also be displayed – this can be indicator of relevance of particular **Search Thread** compared to the other threads. Finally, if more than 10 **Search Threads** are retrieved for particular query a paging mechanism is used to limit each page to 10 results. As can be seen in this example the threads appear to be ranked appropriately with more relevant threads being ranked higher. Overall, the efficiency of scoring mechanism has been tested on quite a few queries – with some of them returning more than 20 relevant threads and as a result high precision has been obtained. In fact, current scoring heuristic is optimally tuned for existing index. However, it is difficult to predict how existing ranking system will perform when more user queries are inserted into the index.
Further improvements:

Throughout the description of the functionality of Thread Search we have noted several possible ways how this tool can be improved. Here we would like to build on this analysis. First, the main component of this mechanism is maintenance of very useful index. The usefulness of the index can only be perceived by usefulness of retrieved results – thus, we have two ways to improve on this component. We can either establish aggressive filtering on insertion of a search thread or we can add filtering to the retrieval procedure. Although more precisely this can be determined only after design of the filter, it appears that it is much more efficient to prevent low utility threads from being inserted. Further, it is possible to pre-populate the index based on crawling some extremely reliable sources. This approach might especially be useful upon introduction of this mechanism to larger user base. Additionally very little work has been done in current implementation to make sure that it scales with larger number of requests. For example, since every search query can potentially update index, IndexSearcher needs to be often refreshed to retrieve newly inserted threads. This is helpful in that the user always searches the most recent index but it is also inefficient in that more than one IndexSearcher needs to be opened in JVM. One possible solution to this problem given that the index is large enough, is to provide a buffer of newly created search threads. The buffer will be inserted into index within certain time interval – once the index is updated we can re-open a new IndexSearcher. This should decrease the number of times that IndexSearcher needs to be refreshed and given large existing index should not have considerable impact on quality of search threads returned. Further, by buffering the threads before insertion into index we can apply sophisticated filters. Not only we will be able to filter threads based on the content within particular Search Thread but we will also be able to create algorithms to perform analysis of entire set of threads in the buffer. For example, we can eliminate threads that are very similar to other threads in the buffer. Using such buffer should therefore improve efficiency of the engine and help maintain usefulness of index. Overall, above is a description of several improvements that can be added to existing implementation; however, many more unlisted modifications remain as current version is only a concept implementation.

During implementation of Thread Search engine quite a few interesting mechanisms have been added and tested. Current implementation is far from complete but does have some interesting features that might be useful to the user base. Thus, it might be interesting to consider adding the features described above to search engine implementation.