Optimizing Subscription Processing in Luwak

THESIS

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

MASTER OF SCIENCE (Computer Science)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Lina Qiu

May 2019

Optimizing Subscription Processing in Luwak

THESIS

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

MASTER OF SCIENCE (Computer Science)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Lina Qiu

May 2019

Approved:

Advisor Signature

Date

Department Chair Signature

Date

University ID:	**N15244150**
Net ID:	**lq437**

VITA

Lina Qiu was born in China on October 6th 1995. She joined New York University Tandon School of Engineering, where she pursued a Master's degree in Computer Science, in September 2017. Before she joined NYU, she got her Bachelor's degree from University of Edinburgh and South China University of Technology. She would like to sincerely thank Prof. Torsten Suel for a year-long guidance on the thesis project, which enlightened her a lot about the methodology of conducting research.

ABSTRACT

Optimizing Subscription Processing in Luwak

by

Lina Qiu

Advisor: Prof. Torsten Suel, Ph.D.

Submitted in Partial Fulfillment of the Requirements for

the Degree of Master of Science (Computer Science)

May 2019

Publish-subscribe is an information-seeking paradigm in which users submit their interests (subscriptions), and any documents published by producers that might match these interests are sent to the users. Luwak is a popular open-source stored-query engine that is used as a basis for publish-subscribe systems. Though Luwak has a comprehensive coverage of various functions required by publish-subscribe and text search systems, it is not competitive in terms of runtime performance. This thesis project describes an attempt to optimize the runtime performance of Luwak by adding one more layer of filtering in the two-level matching process of Luwak, at a cost of some extra memory. We evaluate our solution on three metrics: system throughput, memory utilization, and scalability. Our system can process document notifications 8 to 9 times faster than the original approach while keeping memory usage within an acceptable limit for modern machines. In terms of the scalability in subscription workloads, our algorithm has a better performance as well.

Table of Contents

List of Figures

1. Introduction

At the end of 2017, over 3.8 billion people (50%) out of the 7.6 billion population of the world are connected to the internet [14]. This number is likely to grow rapidly in the following years. While the volume of searchable data is constantly increasing, users' expectations for response time and the accuracy of search results are also growing up. The growth of data has increased the difficulty for web users to gather interesting information. For this reason, systems that can capture the rapidly changing information flow by notifying users of events (recorded in documents) that they are interested in draw many researchers' attention. Such systems parse and store interests indicated by users, and send them notifications when new events matching those interests are generated.

Publish-subscribe (hereafter referred to as pub-sub) systems [9] has a wide range of applications in diverse domains. In news alert systems [12], subscribers may subscribe to get alerts of breaking news of the topics and the regions they are interested in. Each subscription can be modeled as a Boolean expression, e.g., [(Field ∈ {Business} ∧ Company ∈ {Google} ∧ Region ∈ {USA}) ∨ (Field ∈ {Games} ∧ Company ∈ {Riot Games})]. When a piece of news comes to the system, it can be modeled as an assignment of values to attributes, e.g., [Field = Games ∧ Company = Riot Games ∧ Topic = League of Legends]. The goal is to rapidly return the set of news that satisfies the subscriptions. As another example, consider a bargain hunt application, where customers subscribe to know the fluctuation in the price of interesting items, e.g., [Item = laptop ∧ Discount > 10%]. While only a few attributes are shown here for illustration purposes, usually, the number of attributes in a subscription, or in an event, is in the order of tens or hundreds. A promotion event could be expressed as [Item = laptop ∧ Discount = 15% ∧ Expire in 10 days]. The application is supposed to timely send valid satisfying events to customers. As we can see, both subscriptions and events could include range queries and some other special operators, and get very complex.

In a pub-sub system, users express interests, intending to receive all events published by producers that match their interests. Subscriptions and events are usually specific and can be represented by Boolean expressions. In addition to attribute-value pairs, subscriptions and events can have other patterns, such as a combination of terms. Suppose a user wants to get notifications about future discounts of plus-size wool coats of some brand. The Boolean expression that conveys the interest can be *{+plus-size +wool +coats +brand +sale}*. The "+" before a term implies that the term must be satisfied for the whole expression to be satisfied. Any published events that meet the requirements, like *{brand mid-season sale, plus-size wool coats 30% off}*, would be sent to the user.

To handle complex subscriptions and a large volume of continuous data, the system must be efficient and scalable. In practical applications, a pub-sub tool should be able to handle millions of user demands, and thousands or more of incoming events per second, within an acceptable latency. Besides, the system must notify users of all events that match their subscriptions, since missed events may result in business losses. For example, a company may subscribe to know every piece of news that might affect their stock price. Untimely management of negative news can be a disaster for the company. In other words, pub-sub systems should not allow false-negative errors unless this is specified by users.

In this project, we try to optimize the document match search performance of Luwak, an open-source pub-sub system, given pre-processed queries consisting of logical conjunctions of terms. Hereafter, when talking about Luwak and our system, we refer to subscriptions as queries, and events as documents, to make term usage uniform. The reason we focus on conjunctive queries is: conjunctive predicate is a common composition of Boolean expressions. Improving match search performance for conjunctive queries should to some extent also improve search performance for queries in more complicated forms. For example, queries in DNF form consist of a disjunction of conjunctive predicates.

The rest of the report is organized as follows. In Section 2, we discuss related work. Some background necessary for understanding our work is provided in Section 3, which includes a description of the targeted problem. In Section 4, we explain what we want to achieve in this project in details, and in Section 5, we describe our solution. We then present our experimental results in Section 6. Finally, we draw conclusions in Section 7.

2. Related Work

Before us, a variety of content-based pub-sub systems have been heavily studied [1, 3, 5, 6, 7, 10, 15, 16, 17, 18]. There are two main categories of matching algorithms in pub-sub systems: tree-based and counting-based. Tree-based methods [1, 19] aim to recursively divide the search space by removing queries on encountering unsatisfiable predicates. Gryphon [1] is one of the earliest tree-based algorithms. The goal of counting-based approaches is to reduce the number of predicate evaluations. The simplest, and slowest counting-based method is [21, 22], where inverted lists and some arrays are used to find satisfying conjunctive expressions for a document. The method counts the number of distinct words in the incoming document that are present in the conjunctive expressions. If the number is equal to the total number of distinct words in a conjunctive expression, a satisfying expression is found. The counting-based algorithm was further enhanced by Carzaniga et al. [6]. Their work can find candidate conjunctions for DNF expressions in an efficient way by doing short-circuit evaluation. A way based on counting to handle arbitrary AND/OR trees has been proposed in [3]. It can be considered as a complementary extension to the counting-based algorithm.

Other than proposing efficient matching algorithms, some previous work addressed the challenge of evaluating Boolean expressions of different forms. Le Subscribe [10, 17] is an efficient algorithm for conjunctions evaluation. K-index [20] is a method based on the WAND algorithm [4]. It efficiently evaluates CNF and DNF expressions. Normalizing complex Boolean expressions to CNF or DNF form, and then using existing methods for evaluation might be ineffective, because of the exponential blow-up in the size of expressions. Fontoura et al. [11] attempted to resolve the issue, and proposed a single pass bottom-up evaluation algorithm over alternating AND-OR trees that can represent arbitrarily complex Boolean expressions. However, all these algorithms restrict the predicate expressiveness. They focus on solving matching problems of Boolean expressions consisting of attribute-value pairs, and the dimension of attribute names is comparably more limited (32 for [10, 17], and in the order of hundreds for [20] and [11]) than the working environment of Luwak. Unlike the works mentioned above that restrict attribute names within a limited space, any term is considered as valid in Luwak. In other words, Luwak does not have restrictions on the predicate expressiveness. Indexing in high-dimensional space has also been extensively studied [2, 13, 19].

Operator is also a part of the predicate expressiveness. K-index [20] shows a more efficient approach for DNF expressions, which additionally supports multi-valued attributes with NOTs, in comparison to Carzaniga's work [6]. BE-Tree [19] distinguishes itself from others by supporting richer predicate operators (e.g. range operators) and dynamic insertion and deletion of queries. The subscription languages in [5, 11] support matching complex Boolean expressions that involve a rich set of operators as well.

Recently, there is a paradigm shift in pub-sub systems. The new paradigm is referred to as symmetric publish-subscribe, in which event producers can impose filtering conditions on subscribers as well [15, 18]. Another recent trend in the study of pub-sub systems is focused on queries that contain spatial requirements. Chen et al. [7] proposed an efficient approach to evaluate temporal spatial-keyword subscriptions over a stream of geo-textual objects.

3. Background

In this section, we first define the problem that our project is trying to solve, and then provide some information regarding Luwak and its techniques.

3.1 Matching Problem Description

A query in our system is a conjunction of terms. For example, *{+plus +size +women +clothing +catalogs}* and *{+food +carts +brooklyn}* are valid queries. The "+" symbol before each term indicates that this term must occur in result documents. A document in our system is a collection of terms such as *{The top 10 food carts near Brooklyn Heights}*. Conjunctive subscriptions and events consisting of single-valued attribute-value pairs can easily be transformed to the form applicable in our system. For example, a conjunctive subscription [Field ∈ {Games} ∧ Company ∈ {Riot Games})] can be transformed to [+Field +Games +Company +Rio +Games], and a conjunctive event [Field = Games ∧ Company = Riot Games ∧ Topic = League of Legends] can be transformed into a document [Field Games Company Riot Games Topic League of Legends]. The transformation does not affect the match results. The simple form of conjunctions that our system is targeting can also be used as building blocks for more complex Boolean expressions that include conjunctive expressions.

A document *d* is a match for a query *q* if every term in *q* exists in *d*, ignoring case and order. For instance, document {The top 10 food carts near Brooklyn Heights} is a match for query {+brooklyn +food +carts}. The matching problem in our system is defined as: Given a set of pre-processed queries Q, find all matches {*q*, *d*} when a batch of documents D is submitted to the system.

3.2 Inverted Index

An inverted index is a data structure storing a mapping from terms to a set of integer document identifiers associated with the term. It is widely used in many ranked information retrieval systems, including Luwak. The inverted index data structure has the ability to manage a large number of dimensions (keywords). Its scalability is also impressive. Figure 2 shows an example of an inverted index. Each key term is mapped to a list of integer identifiers, whose corresponding documents (in our case, the identifiers are for queries, each of which is assigned an id by the system) contain the key term.

3.3 Workflow of Luwak

Luwak[1] is a stored-query engine based on the open-source Lucene search library[2]. It stores a set of user-defined queries and then evaluates a stream of documents to find matches for the subscribed queries. Typically, it is used by companies who monitor high volumes of news using often extremely complex Boolean expressions. The features of this library cover basically all functional demands of pub-sub systems. Luwak has been used in companies including Infomedia, Bloomberg, and Booz Allen Hamilton for their pub-sub systems.

The workflow of Luwak can be partitioned into three steps. First, the system reads all queries, and parses and stores them. Each query is assigned an integer identifier according to its input order. As a new batch of documents arrives, the system conducts a fast and approximate pre-search. During the pre-search, queries that are unlikely to have matches in the batch are filtered out. In the last step, the remaining candidate queries are

[1] https://github.com/flaxsearch/luwak
[2] https://lucene.apache.org

run against the batch of documents using normal information retrieval algorithms. More details about each step are given in the next few paragraphs.

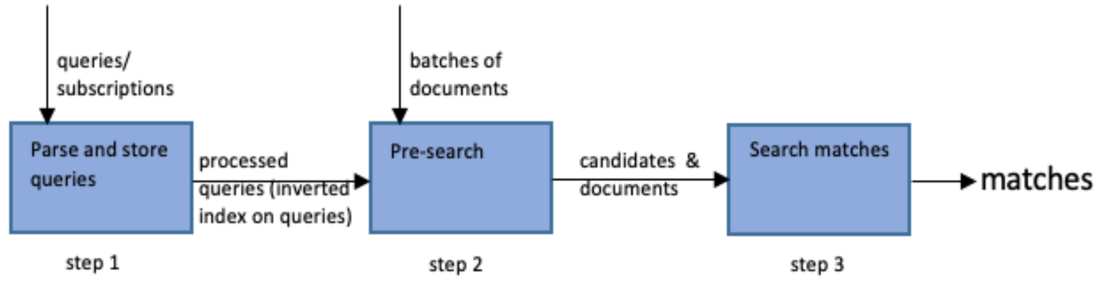

Figure 1: Workflow of Luwak

For every query, Luwak sorts its terms by descending weight, according to a user-defined scheme. Usually a rarer term has a higher rank. The rarest term, with the highest rank in a query, is called the representative term of the query. Luwak builds an inverted index on subscribed queries using only their representative terms.

In the pre-search phase, an intersection of the set of query representative terms, and the set of terms that exist in the current batch of documents, is computed. Then Luwak constructs a disjunction of the representative terms that are also in the documents (terms in the computed intersection), called a PresearcherQuery. Luwak searches the PresearcherQuery on all subscribed queries to filter out non-candidates whose representative terms are not in any of the documents. The remaining queries are called candidate queries. As soon as a candidate is found, it is pushed to the next step for final match search. The output order of candidate queries depends on their assigned id. A candidate with a smaller id is pushed to the next step before a candidate with a higher id.

We provide an example that can help understand the construction process of the inverted index on queries and the filtering procedure of pre-search.

Query 1: *{+graphic +tee +medium}*

Query 2: *{+tissue +facial +pack +eight}*

Query 3: *{+Disney +world +ticket +day}*

Query 4: *{+tissue +graphic +wet}*

Document: *{Sarah wants one Disney graphic tee}*

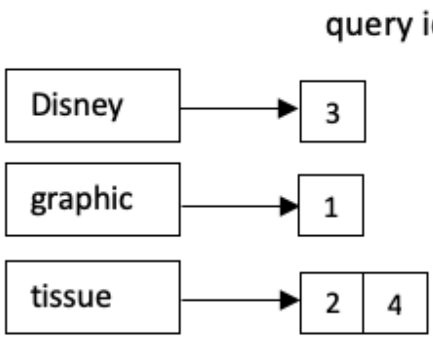

Figure 2: Inverted index of the four queries

Suppose we have four queries whose terms are already sorted by weight, and a document; the representative terms of the four queries are {**graphic**, **tissue**, **Disney**, **tissue**}. The inverted index of the queries is illustrated as Figure 2. The term **tissue** is not in the document. Therefore, the PresearcherQuery should be {**Disney** ∨ **graphic**}. Among the four queries, the representative terms of Query 1 and 3 satisfy the disjunctive PresearcherQuery, so the two queries are considered as candidates. Query 2 and 4 would be filtered out, because their representative term **tissue** does not satisfy the PresearcherQuery, even though Query 4 contains the term **graphic**.

Luwak builds an inverted index on incoming batches of documents in step 3. In the last step, candidate queries are run against the incoming documents one by one to get the subset of documents that match the current query. In the example above, the document does not match any query. The way that Luwak conducts conjunctive query evaluation is illustrated with an example as follows[3].

Suppose we have a candidate query *{+lucene +revolution +conference +the}*, which has been sorted by increasing document frequency df(t). Document frequency df(t) is the number of documents that contain the term t. The related inverted index of documents is shown in Figure 3. The rarest term **lucene** is selected as the lead for the search process, and *posting[0]* refers to its posting list (*posting[1]*, *posting[2]* and *posting[3]* refer to the posting lists of **revolution**, **conference** and **the** respectively). Posting lists are sorted by increasing document id.

```
Function advance(doc_id):
    for each id in the posting list
        if (id >= doc_id)
            return id
        else
            continue
    return NoMoreDocs
```

[3] https://www.slideshare.net/lucenerevolution/query-latency-optimization-with-lucene

Parameter DID is the current document id of the posting list. Initially, *posting[0].DID* = *posting[0].advance(0)* = *18*. The DID of the posting list of the second rarest term is set to a number not smaller than *posting[0].DID*: *posting[1].DID* = *posting[1].advance(18)* = *22*. Since *posting[1].DID* is not equal to *posting[0].DID*, the examination starts all over again with the lead, but this time *posting[0].DID* = *posting[0].advance(22)* = *31*. Then Lucene tries to apply *advance(31*) on all other posting lists, and a satisfying document whose id is 31 is found. After recording this match, the next examination starts from *lead.advance(31)*. Such process is repeated until the lead reaches its end (*lead.advance()* returns a constant NoMoreDocs). At this point we know that all matches of the query have been found.

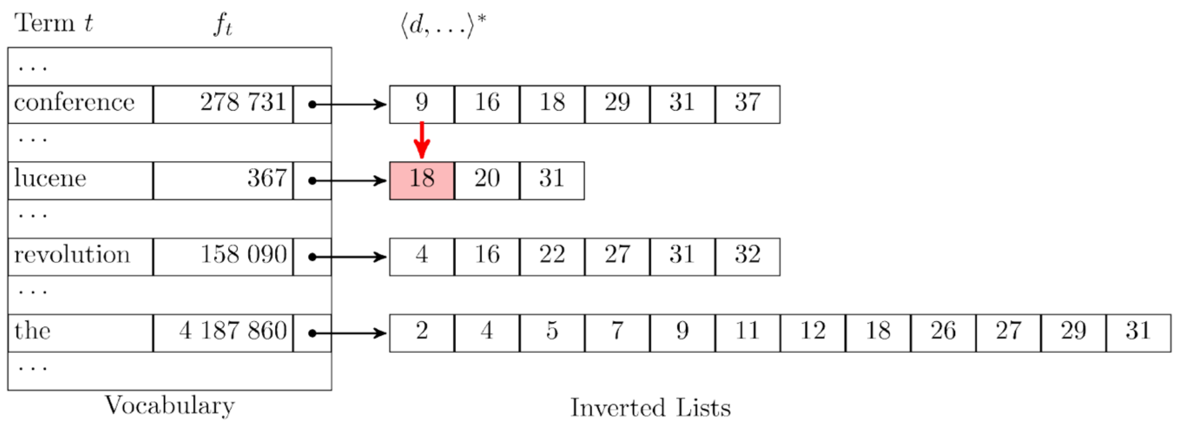

Figure 3: Inverted index of documents[3]

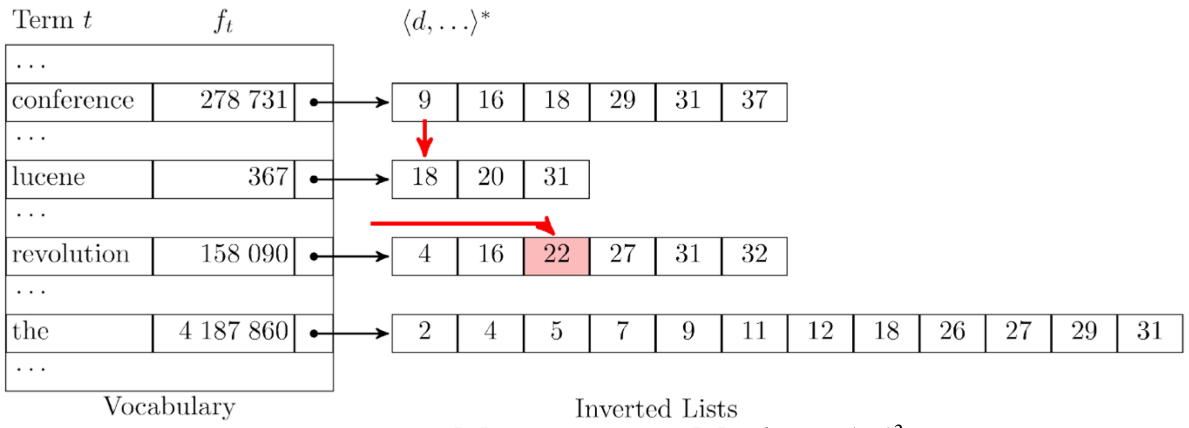

Figure 4: posting[1].DID = posting[1].advance(18)[3]

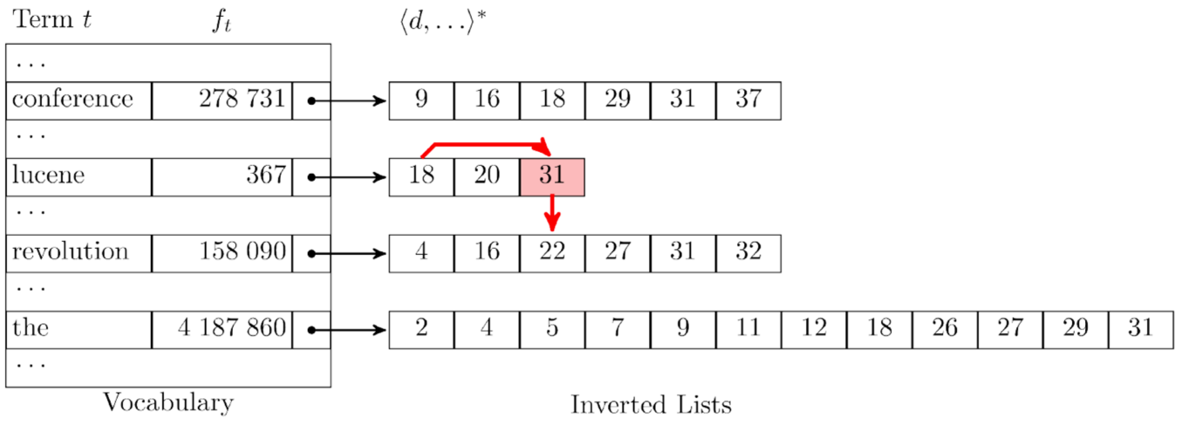

Figure 5: posting[0].DID = posting[0].advance(22)[3]

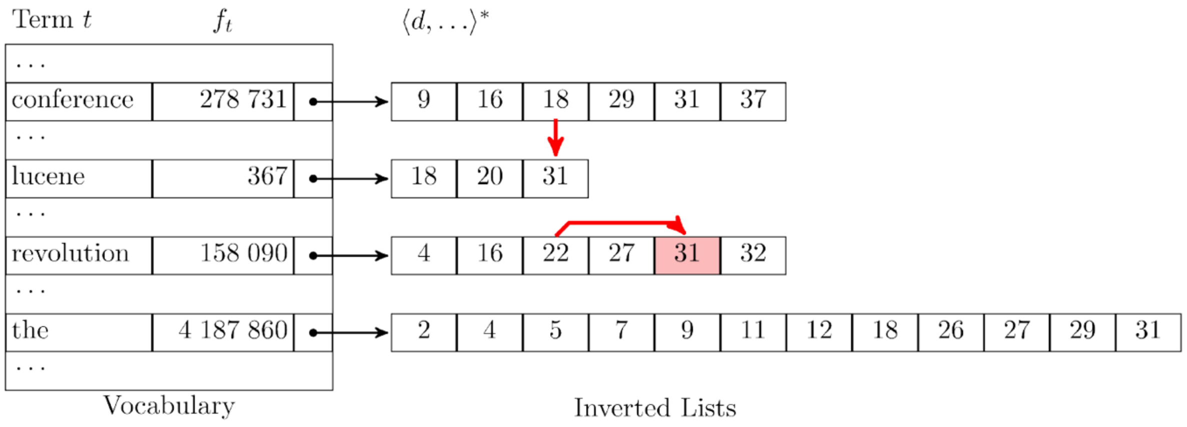

Figure 6: posting[1].DID = posting[1].advance(31)[3]

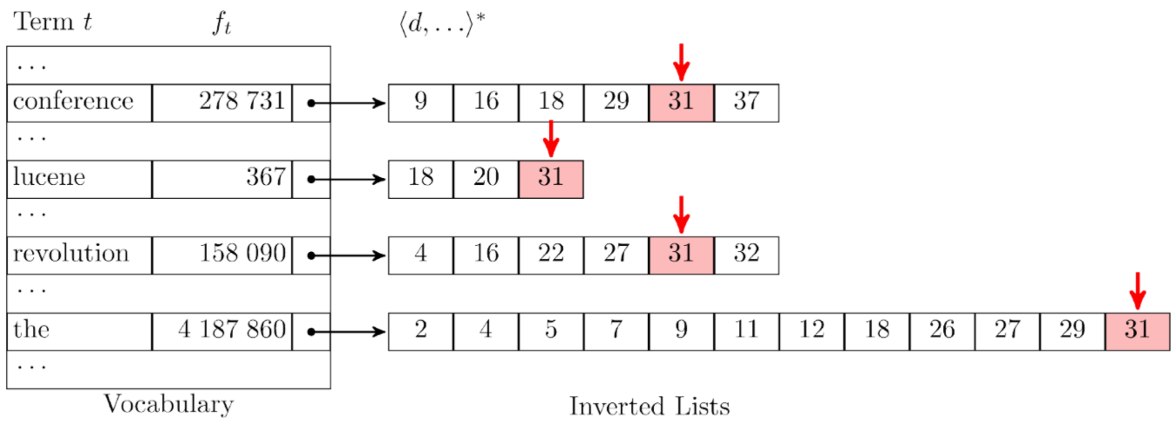

Figure 7: All posting lists have the same DID; a match found[3]

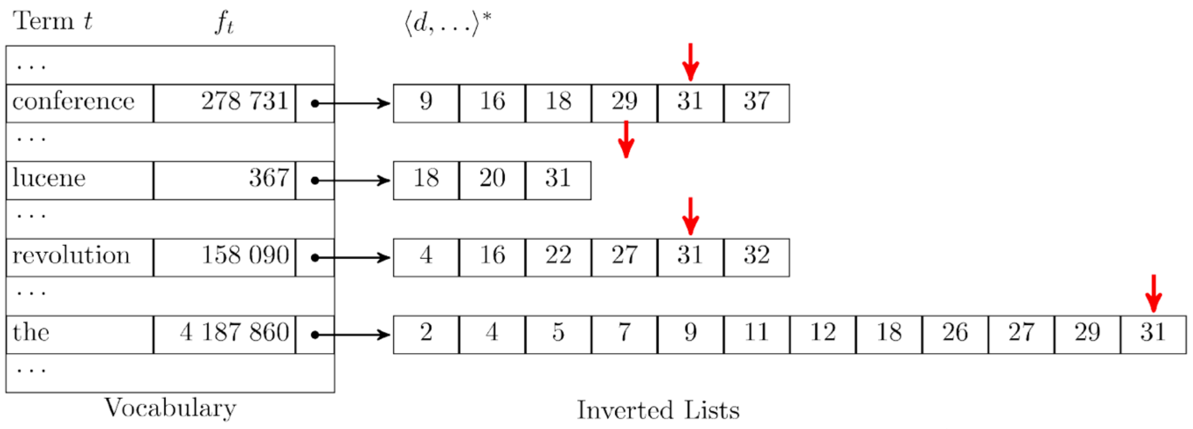

Figure 8: The lead reaches its end, the search is completed[3]

With the support of Lucene, a powerful text search engine library, Luwak can accurately parse very general queries and handle more complicated searches than terms exact match. For example, Luwak can conduct range search and wildcard search. To stay applicable to various use cases, its code is unavoidably kept general for all kinds of queries and search requirements. In other words, its performance in a specific use case might be worse than programs developed for solving this case. In this project, we build our system upon the framework of Luwak, intending to make some problem-specific improvements, while still follow the workflow of Luwak for easier future integration.

4. Motivation

Having a general idea of pub-sub systems and the workflow Luwak, we next discuss what we want to achieve and the motivation of our plan for this project more specifically in this section.

The overall goal of this project is improving search performance by selecting additional terms, rather than just the representative term for each query. Besides the first layer of non-candidates filtering by PresearcherQuery, we add a second layer of filtering taking advantage of the additional indexed terms, so that fewer candidate queries need to be fully evaluated in the final step, which is time expensive. As described in the above section, Luwak originally builds an inverted index for subscribed queries only on their representative terms. After pre-search, there is still a large portion of candidate queries that in fact have no matches in the documents. The time spent on searching matches for such queries is unnecessary. The plan is to improve the efficiency of pre-search, reduce the number of candidate queries as many as possible, and avoid the unnecessary time consumption, at a cost of some acceptable extra space.

Before getting into our methodology, some more details about Luwak pre-search need to be clarified. First, the sorting scheme for query terms has a significant effect on the efficiency of pre-search. A different order may imply a different set of representative terms, so the number of candidate queries output by the first layer of filtering varies. Since we do not want to break the workflow of Luwak, those extra terms we decide to index should be stored with the corresponding query id as a customized java data structure in posting lists as shown in Figure 9 (using the same example as Figure 2), under the assumption that for each query we index one more term in addition to the first one). With the customized data structure, we can still utilize the first layer of filtering as in the unmodified version of Luwak, because the first layer of filtering involves only the key set of the inverted index, and the key set is unchanged. We call the customized data structure QueryNote. We add one more layer of filtering that takes the initial candidates output by operating PresearcherQuery on subscribed queries as input. In the next section, we will discuss the implementation of the second layer of filtering. We also mention that Luwak evaluates candidate queries one-by-one in step 3, which is also unnecessary if there are duplicate queries. This can be optimized by sorting queries alphabetically in advance, so that duplicate queries are next to each other in the input of our system, and then evaluating the group of duplicate queries as a batch only once.

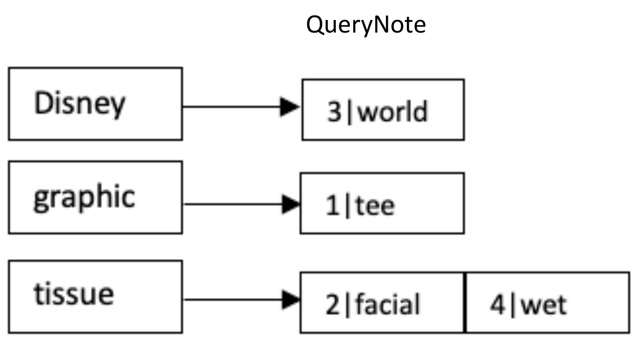

Figure 9: Inverted index of queries built with QueryNote

5. Our Solution

In this section, we will discuss our solution, starting with an introduction of the sorting scheme we apply to sort terms in a query, and a data structure employed by the solution algorithm. Then, we will explain how the optimization of evaluating duplicate queries as a batch works. Finally, we will get to the filtering algorithm and the query term collection algorithm.

5.1 Weight of Query Terms

Luwak provides a few sorting schemes based on the length of terms or term frequency. The basic idea for any sorting scheme is that a term expected to be less frequent in incoming documents should have a higher weight and a higher rank. Though it is unrealistic to assume that we have accurate statistics about incoming documents, we can still make a reasonable induction from the statistics of a training set of documents.

Note that we are not interested in how many times a term occurs in a document but in how many documents contain the term at least once. Document frequency is the proper measurement for our case. We decide the weight of a query term using the formula:

$$w(t) = \log [n / df(t)] + 1$$

Equation 1: Weight of term t

In the formula, `n` is the total number of documents in the training set, and `df(t)` is the document frequency of `t` with respect to the training set of documents. A rarer term `t` that has a smaller df(t) is assigned a higher weight by the formula, and has a higher rank in the query. Thus, a term that is more helpful in filtering out a query has a higher rank. Such a sorting scheme ensures that Luwak is more likely to filter out a query using its representative term in the first layer of filtering, and we can use the least number of extra terms to have a query filtered out in the second layer of filtering, if the order of term rareness remains the same with respect to the test set.

5.2 Inverted Bit Map of Documents

To obtain response rates within an acceptable range (say, less than one second), each batch of documents is usually in a size, say a few hundred documents. At this batch size, the cost of building an inverted bit map for each batch when the documents come in is acceptable in terms of both time and space. In an inverted bit map, each key is a distinct term in the incoming batch, and a value is a bit array that has the same length as the batch size. Each bit of a bit array represents a document in the batch. If a bit is set to 1, it means its corresponding document contains the key term that has a mapping to the bit array, otherwise the document does not contain the key term. The number of keys in a bit map is the number of distinct terms in the batch, and each value is an array with the same length, so the size of a bit map is equal to the number of distinct terms times the input batch size, which is at most a hundred Megabytes in memory.

From the bit map, we can deduce promising documents that might be matches for a query by doing AND operations among bit arrays whose keys are terms in the query. For example, suppose we decide to collect the first two terms of the sorted query *{+lucene +revolution +conference +the}*, and the incoming batch has 10 documents, promising documents are {2, 6, 7} for the partial query *{+lucene +revolution}*. The interesting part of the inverted bit map of the 10 documents is shown in Figure 10. An AND operation between the two bit arrays results in {0, 0, 1, 0, 0, 0, 1, 1, 0, 0}. Promising documents of the complete query should be a subset of {2, 6, 7}. How many AND operations should be conducted depends on what extra terms we collect for each query. The more terms we collect for a query, the fewer promising documents remain after the AND operations. When no promising documents are left, we know the query can be filtered out. In most cases, we only need to check some of the query terms, and the number of promising documents for a query would go to zero. The time of doing one AND operation between two bit arrays is much cheaper than running a query against documents for matches. Therefore, the idea of indexing more terms for each query to get as least candidate queries as possible can still be beneficial, even though there is some additional time cost in doing AND operations between bit arrays.

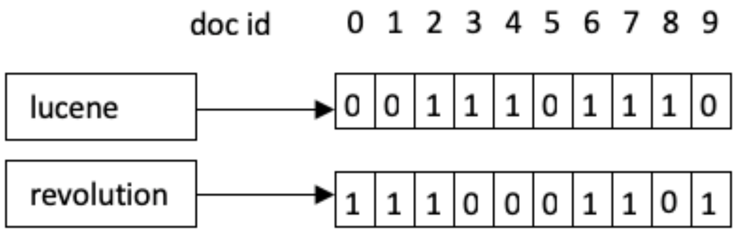

Figure 10: Inverted bit map of 10 documents.

5.3 Evaluating Duplicate Queries

The subscribed queries are sorted alphabetically as they are processed by our system. The initial candidate queries output by the first layer of filtering are in the order of increasing id, which means if a group of duplicate queries are candidates, they are next to each other in the output of the first filtering layer. Thus, we can cache duplicate queries after the first and before the second layer of filtering to save repeated calculations, and then send the batch of duplicate queries $\{q_a, \ldots, q_b\}$ to the second layer of filtering, which treats them as one query q_a. Eventually if the query q_a is filtered out, other queries in the batch are discarded as well. If matches are found for q_a, we can simply output matches for $\{q_{a+1}, \ldots, q_b\}$.

5.4 The Second Layer of Filtering in Pre-search

Originally, Luwak searches PresearcherQuery against subscribed queries for candidates. This is regarded as the first layer of filtering in pre-search. We add a second layer of filtering utilizing the extra indexed terms and the inverted bit map of documents to further reduce the number of candidates. The input to the second layer is the cached output of the first layer. Therefore, the representative terms of the input queries of the second layer must exist in the current document batch. The `bitMap.get()` method returns the mapped bit array of the extra term if it exists in the key set of the bit map, otherwise the method returns an all-zeros bit array.

When the bit array `partialQueryBitArray` that records the status of promising documents has no true bit, the `query` should be filtered out because there are no promising documents left for the query. If a query has all its terms except for the first one stored in the QueryNote, but cannot be filtered out during the iterations, the query still does not need a full evaluation. The outputMatches() helper takes partialQueryBitArray and query as input, picks out all true bits in partialQueryBitArray and the corresponding documents {d_1, d_2, …, d_k}, and outputs matches {(query, d_1), (query, d_2), …, (query, d_k)}. Queries that have to go through full checking to get matches are handled by matchQuery() method, which is the beginning of step 3.

```
Function filterCandidates(query, queryNote)
partialQueryBitArray ← bitMap.get(query.fstTerm).copy()
if (queryNote.numExtraTerms > 0) {
    j ← 0
    for (; j < queryNote.numExtra; j++) {
        extraTerm ← queryNote.extraTerms[j]
        partialQueryBitArray.and(bitMap.get(extraTerm))
        if (partialQueryBitArray.isEmpty()) {
            return
        }
    }
}
if (queryNote.numExtra == (query.size()-1)) {
    outputMatches(query, partialQueryBitArray)
    return
}
matchQuery(query)
```

5.5 Query Term Collection Algorithm

As we index an extra term for a query, the probability that it occurs in a document together with the already indexed terms decreases. In other words, the probability that the query could be filtered increases. The fact that the cardinality of a bit array after doing another AND operation should be not greater than the cardinality of the bit array before the operation is a proof. If at some point the cardinality reaches 0, it means the query cannot be a candidate and it does not need a full evaluation.

The order of terms is determined by Equation 1. What the query term collection algorithm needs to do is deciding when to stop collecting more terms. In fact, the idea of the algorithm is quite straightforward. For each sorted query, it iterates over all its terms from the highest weight to the lowest, aiming to return an integer, at which the combination of the current and the already indexed terms probabilistically maximizes time benefits.

The query term collection algorithm we devise employs a cost function to make the decision. The cost function incorporates the time cost of step 2 and step 3 of the workflow. With the cost function and a training set of documents, we can calculate an expected number that probabilistically maximizes time benefits, which equals the original time cost minus the current expected time cost. The way we do the calculation can be summarized as: for every term in a query, the system estimates the probability that the combination of the current and the previous terms may occur in one document in a batch, based on the statistics of the training set of documents. The expected performance gain of indexing this many extra terms can be computed by applying the probability (partial query probability) to the cost function. For every query, we choose the number of indexed terms that brings the highest expected performance gain.

If terms in the same query are assumed to be independent, we can use the document frequency with respect to the training set to approximate partial query probabilities as follows:

$$Prob(t) = df(t)/n$$

Equation 2: Probability of term t

$$Prob(t_1, t_2, ..., t_k) = \Pi_{i=1}^{k} Prob(t_i)$$

Equation 3: Partial query probability under independence assumption

The accuracy of the approximated probabilities can significantly affect the performance of the algorithm, and the independence assumption in fact is a bit coarse for our case (its performance is depicted by the "No matrix" curve in Figure 15). In order to capture the relations among terms, and to more accurately predict the probabilities, we build a word co-occurrence matrix on the training set of documents. Such a matrix could be too large to load to the memory, even for a small set of documents. However, we only need a good approximation to get better estimates than with the independence assumption. In Section 6, we will discuss how we utilize a simplified version of the word co-occurrence matrix that still provides a better approximation to true probabilities.

The method `decideNumTermsCollect()` is called when the system is trying to digest the subscribed queries (step 1). After calling the method, the algorithm repeatedly evaluates the partial query probability of the fractional query, each time with an extra term added. The evaluation starts from the second term because the first term of each query must occur in the batch, which is assured by the first layer of filtering. If the current term in combination with the previous one is represented in the word matrix, its probability is replaced with the stored probability. Otherwise, its probability is calculated using Equation 2. Meanwhile, every new partial query probability is recorded. With the list of recorded probabilities, we can calculate which one of them can probabilistically maximize time benefits through the helper method findMaximumGain().

```
Function decideNumTermsCollect()
partialProb ← 1
probsList ← ArrayList<Double>
probsList.add(1)
j ← 1
for (; j<query_size; j++) {
    preTerm ← query[j-1]
    curTerm ← query[j]
    phraseProb ← wordMatrix.findProb((preTerm, curTerm))
    if (phraseProb > 0) {
        partialProb ← partialProb * phraseProb
    }
    else {
        partialProb ← partialProb * Prob(curTerm)
    }
    probsList.add(partialProb)
}
return findMaximumGain(probsList)
```

Helper method findMaximumGain() calculates performance gain using a cost function. Suppose those previous indexed terms did not succeed in filtering the query, as one more extra term is indexed, one more AND operation needs to be conducted (variable i in the cost function refers to the number of AND operations, $time_{AND}$ is the time cost of doing one AND operation), and the probability that this query cannot be filtered out

($Prob_{not\_filtered}$) after adding the term should be updated. $time_{check}$ is the original time cost of checking this query as a candidate against documents. Overall, expected time benefits after indexing one new extra term equals to the original cost ($time_{check}$) minus the current expected cost ($i * time_{AND} + Prob_{not\_filtered} * time_{check}$). There is one special case when the expected cost should be ($i * time_{AND} + Prob_{not\_filtered} * time_{avoid}$) for the last term of every query, because matches can be directly deduced from the bit map if all terms of a query have been indexed. The time spent on outputMatches() in the filtering algorithm to pick out satisfying documents is denoted as $time_{avoid}$.

Whether the expected gain would increase after indexing one more term depends on the decreasing rate of $Prob_{not\_filtered}$, except for the last term. For clearer explanation purposes, we use P_k to denote the partial query probability of indexing the first k terms, and P_{k+1} to denote the partial query probability of indexing the first k+1 terms. If $(P_k - P_{k+1}) * time_{check} > time_{AND}$, the expected gain increases. Otherwise, the expected gain decreases. The decreasing rate of $Prob_{not\_filtered}$ fluctuate from term to term, which makes the values of expected gain fluctuate accordingly. There are a few reasons that may make the algorithm become greedy and decide to collect all terms of a query. One important reason is that $time_{avoid}$ is much smaller than $time_{check}$ in our system, so the algorithm has to avoid full evaluation sometimes.

Cost Function

$$\text{Benefit} = time_{check} - i * time_{AND} - Prob_{not\_filtered} * time_{check}$$

```
Function findMaximumGain(probsList)
maxGain ← 0
index ← 0
j ← 1
for (; j<query_size; j++) {
        if (j != (query_size-1))
                gain ← $time_{check}$ * (1 – probsList[j]) – j * $time_{AND}$
        else
                gain ← $time_{check}$ – j * $time_{AND}$ – probsList[j] * $time_{avoid}$
        if (gain >= maxGain) {
                maxGain ← gain
                index ← j
        }
}
return index
```

A simplified version of word matrix helps improve the accuracy of predicted probabilities of queries that do not mostly consist of very common terms, but in real cases there are a large portion of queries mostly consisting of very common terms, such as *{+what +is +new +today +in +the +world}*. Terms in such queries (hereafter referred to as common queries) usually have broader correlations, not restricted to just two terms. Word co-occurrence matrix can only help capture relations of two-term phrases. Therefore, we need special rules for common queries, or in other words, we need special rules to handle broader correlations among common terms.

5.5.1 Rules for Common Queries

There are (query_size - 1) iterations of the for loop in the decideNumTermsCollect() method. We classify a query as a common query if in every iteration the current two-term phrase or the current single term is considered frequent. For a two-term phrase, if it can be found in the word matrix, it is considered frequent, because the matrix only holds very frequent phrases. For a single term, if its probability is in the range of top 0.67% of the sorted probabilities of all single distinct terms in the training set, it is considered frequent. The reason that we chose 0.67% as the threshold will be explained in Section 6.3.

A common query as a whole may have a comparable probability with any one of its subparts. In other words, indexing more terms for a common query is less likely to help in filtering it out. Thus, it is usually better to either not collect any extra terms and push it to do a full evaluation using the original routine (no additional space cost), or to index all its terms and calculate final document matches by doing AND operations among all corresponding bit arrays. The latter can avoid full evaluations. Which one to choose depends on the time cost of the two methods. Our experiments show that conducting one conjunctive query search on a batch of hundreds of documents (500 - 1000) is more expensive than doing 60 AND operations. That is, the latter option should be better in terms of time cost for a query shorter than 60 words, which is dominant in our dataset.

5.5.2 A Small Greedy Improvement

Theoretically, a partial query probability `partialProb` does not depend on batch sizes, because it is decided by the statistics of the training set of documents, which we assume to be consistent with the test set. However, the expected number of documents in a batch that contain all the terms in the partial query is related to the batch size. The expected number of documents that contain all the terms is equal to `[1 - (1 -`

partialProb)^{BatchSize}] * BatchSize, which means a larger batch is prone to have more documents meeting the requirement. In the second layer of filtering, the number of terms that the term collection algorithm decides to index may maximize time benefits, but is not probabilistically large enough to filter out the query, because the expected number of promising documents for the partial query ([1 - (1 - partialProb)^{BatchSize}] * BatchSize) is still greater than one. We modify the collection algorithm a bit to take advantage of the batch size information, and put a greedy threshold to the range that findMaximumGain() works to find the maximum. The adjusted version of query term collection algorithm that incorporates the additional rules for common queries and the greedy threshold is shown as follows. The gain maximization function findMaximumGain() no longer considers all terms in a query, instead it starts from doc_th, at which [1 - (1 - partialProb)^{BatchSize}] * BatchSize just becomes smaller than 1. It means the gain is maximized within the range, in which the expected number of satisfying documents is smaller than 1. The greedy threshold might not be helpful in other scenarios, but it helps reduce runtimes in our experiments.

The process of query term collection is a subpart of step 1 in the Luwak workflow. It ends before the real search begins.

```
Function decideNumTermsCollect()
partialProb ← 1
probsList ← ArrayList<Double>
probsList.add(1)
numFrequent ← 0
doc_th ← query_size
j ← 1
for (; j<query_size; j++) {
        preTerm ← query[j-1]
        curTerm ← query[j]
        phraseProb ← wordMatrix.findProb((preTerm, curTerm))
        if (phraseProb > 0) {
                partialProb ← partialProb * phraseProb
                numFrequent += 1
        }
        else {
                partialProb ← partialProb * Prob(curTerm)
                if (Prob(curTerm) > freq_threshold) {
                        numFrequent += 1
                }
        }
        probsList.add(partialProb)
        if ([1 - (1 - partialProb)$^{BatchSize}$] * BatchSize < 1 && j < doc_th) {
                doc_th ← j
        }
}
if (numFrequent == (query_size-1)) {
        return (query_size-1)
}
else {
        return findMaximumGain(probsList, doc_th)
}
```

```
Function findMaximumGain(probsList, doc_th)
maxGain ← 0
index ← 0
j←doc_th
for (; j<query_size; j++) {
        if (j != (query_size-1))
                gain ← $time_{check}$ * (1 – probsList[j]) – j * $time_{AND}$
        else
                gain ← $time_{check}$ – j * $time_{AND}$ – probsList[j] * $time_{avoid}$
        if (gain >= maxGain) {
                maxGain ← gain
                index ← j
        }
}
return index
```

6. Experimental Results

In this section we will present results of the experiments that we designed and implemented to evaluate our solution, and compare its performance to the original routine of Luwak (the latest version 1.6.0). We also address the questions of how our solution performs in different scenarios and how much effect each functional part of the solution has on the overall performance. We ran experiments on a 3.4GHz Intel(R) Core(TM) i7 processor with 31GB of RAM. All algorithms were implemented in Java, because Luwak is written in Java.

6.1 Datasets

We use AOL queries[4] to simulate the subscription database. In the AOL dataset, there are more than 7 million distinct queries. We randomly extract a part of them as required by different experiment settings (duplicate queries are allowed). All the queries should have more than one term. Otherwise, our solution of indexing extra terms is meaningless. To simulate events, we use 100,000 documents randomly selected from the ClueWeb09 dataset[5]. Half of them are used for training, and the other half is used for testing. We only consider documents that have less than 5000 terms. This restriction should be applicable to most daily use cases. Regardless of training or testing, documents are divided into batches of a certain size and fed to the system one batch at a time. A summary

[4] http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection/
[5] https://lemurproject.org/clueweb09/

of statistics of the two sets is shown in the following two tables. Documents with more than 5000 terms are removed from the sets (256 in the training set and 1 in the test set).

number of words	<= 200	200 – 600	600 – 1000	1000 - 5000
count	9386	18087	8971	13300

Table 1: Numbers of documents of different lengths in the training set (49744 in total)

number of words	<= 200	200 – 600	600 – 1000	1000 - 5000
count	9400	18268	8987	13344

Table 2: Numbers of documents of different lengths in the test set (49999 in total)

6.2 Pruned Word Co-Occurrence Matrix

The word co-occurrence matrix is built on the training set of documents. A full version of the matrix occupies about 3GB space, which is very inefficient to use in high-throughput and low-latency systems. Therefore, we prune the matrix and only keep pairs of words that co-occur in at least 200 documents in the training set. Those pairs along with their predicted probabilities of occurring in a test document are recorded in a plain text file. The predicted probability of a pair of terms $\underline{p}$ is calculated by: df($\underline{p}$) / n, where df($\underline{p}$) is the document frequency of *p* with respect to the training set, and *n* is the total number of documents in the training set. The plain text file is about 180MB, which can be kept in memory. The threshold used to prune the matrix can be adjusted according to the system requirements on runtimes and space. A larger matrix can provide more information of term dependency, which is helpful for improving the time performance. Also, the matrix could of course be quantized and compressed for additional size reduction.

To test the accurateness of our way of predicting probabilities, we calculate the mean squared error of the predicted probabilities. The ground truth probability of a partial query *partialQuery* is defined as df(*partialQuery*) / *n*, where df(*partialQuery*) is the number of documents in the test set that contain all terms of the *partialQuery*, and *n* is the total number of documents in the test set. The mean squared error is computed over the queries whose representative terms exist in the test documents, because the queries that can be removed by the first layer of filtering have nothing to do with our solution. The mean squared error is 0.0064 if we assume term independency. After applying the pruned version of co-occurrence matrix, the error is reduced to 0.0054. The improvement in probabilities may seem small, but it results in improved time performance.

6.3 Threshold of Deciding Frequent Terms

We chose top 0.67% as the threshold because single terms whose probabilities (df(*t*) / *n*, *n* is the total number of documents in the training set) are in this range cover over 65% of the occurrences of words in the query datasets that we use for experiments (each dataset contains at least a half million queries). In language models, a small portion of the vocabulary in fact covers most of word occurrences. A tighter threshold would result in a larger number of full evaluations. In contrast, a looser threshold would result in more queries classified as common queries, and many more extra terms being indexed by our algorithm. Similarly, the tradeoff can be adjusted according to how much space one is willing to sacrifice for better runtimes. The threshold 0.67% is a good enough choice for our experiments, taking both time and space into consideration.

6.4 Statistical Data in the Cost Function

In the cost function we proposed, there are a few parameters where we have not explained how to get their values: $\mathtt{time}_{\mathtt{check}}$, $\mathtt{time}_{\mathtt{AND}}$ and $\mathtt{time}_{\mathtt{avoid}}$. From the way Luwak does checking (described in Section 3.3), we know $\mathtt{time}_{\mathtt{check}}$ is positively correlated to the number of times that the inverted index of documents has to be probed (proportional to the average length of queries times the average length of posting lists). In contrast, $\mathtt{time}_{\mathtt{avoid}}$ is negatively correlated to query length. The fact is $\mathtt{time}_{\mathtt{avoid}}$ is positively correlated to the average number of matches of subscribed queries. The average number of matches goes down for longer queries. On the other hand, the average number of matches is correlated to batch sizes. Queries tend to have less matches, given a smaller batch of documents. As for $\mathtt{time}_{\mathtt{AND}}$, it depends on the speed of CPU, the size of RAM, and the length of bit arrays, which is equal to the input batch size, but not on any query properties.

The three parameters should be relatively stable in controlled experimental environments. Thus, we can get statistical numbers of them by taking the average of measurement results of pre-run experiments on the training set. The pre-run experiments use the same setups as the corresponding evaluation experiments. In the pre-run experiments, the variances of the three parameters are lower than 2%, which validates this approach. The time cost of an AND operation for batch sizes of 200, 400, 600, 800, 1000, and 1200, is 82ns, 87ns, 95ns, 103ns, 110ns, and 117ns respectively. In Table 3 and Table 4, we show some results of the pre-run experiments, which we later apply in the evaluation experiments. Workload variables are introduced in the next subsection. The combined set of queries specified in Table 5 refers to a set we randomly extracted from AOL and involves queries of different lengths.

$size_{Eb}$ / l_S, u_S	200	400	600	800	1000	1200
2-term query	6870ns	7381ns	7745ns	8778ns	9040ns	10201ns
3-term query	8560ns	9574ns	10328ns	10600ns	11136ns	12196ns
4-term query	10695ns	11656ns	12645ns	13027ns	13690ns	14496ns
5-term query	12535ns	13093ns	14520ns	14513ns	15521ns	16509ns
>5-term query	16446ns	16882ns	17457ns	18211ns	18719ns	19201ns

Table 3: $time_{check}$ (row is query, column is batch size)

$size_{Eb}$ / l_S, u_S	200	400	600	800	1000	1200
2-term query	2268ns	3282ns	4067ns	5618ns	6786ns	7921ns
3-term query	1215ns	1789ns	2066ns	2721ns	3168ns	3744ns
4-term query	968ns	1343ns	1414ns	1902ns	2030ns	2254ns
5-term query	803ns	1048ns	1141ns	1258ns	1392ns	1512ns
>5-term query	855ns	1046ns	1098ns	1300ns	1490ns	1597ns

Table 4: $time_{avoid}$ (row is query, column is batch size)

	2-term	3-term	4-term	5-term	6-term	>6-term
count	410943	301249	181609	98999	51290	55835

Table 5: Length distribution of our queries

6.5 Performance Comparison

To evaluate our algorithms, we use two fundamental cost metrics: document throughput, and required memory. We assume that all data structures required for the algorithms fit in the main memory of our machine.

We implemented a workload generator to emit documents to the pub-sub system according to a workload specification. Documents are emitted to the system in fixed-size batches right after the system responds to the previous batch. Since there is no break between two processing intervals, our system operates under a key assumption of a high input rate of documents. The batch size of documents to submit to the system is denoted by $size_{Eb}$. Queries are loaded into the system as a whole set at the beginning. The total number of subscribed queries is n_S and the domain of the length of queries is described by a lower bound l_S and an upper bound u_S. Timings are taken in milliseconds, starting just before a batch of documents is submitted to the system, and ending right after the system responds to the submitted batch. The timings include the processing of the batch of documents, but the process of collecting extra terms and building inverted index on queries is excluded. As we explained earlier, queries are parsed and stored in advance of real search. In order not to obscure the effect of extra indexing, all loaded queries are long-lived.

Figure 11(a) compares overall system throughput between our full solution Pre-Search and the original method in Luwak. They use the same sorting scheme discussed in Section 5.1. The query set to load is the one specified in Section 6.4. The document set used to randomly generate batches is the test set of documents. The workload specification is: (n_S = 1099925, l_S = 2, u_S is unlimited, $size_{Eb}$ varies). The same workload is used in all figures until Figure 15 except for Figure 13. The dark green curve annotated as Luwak_Qbatch shows the performance of evaluating duplicate queries as a batch without using the second layer of filtering. The yellow curve shows the performance of FullyIndexed method that greedily indexs all terms for every query, and simply uses AND operations to get matches instead of applying full candidate checking. As we can see, the method that Luwak originally uses needs longer than 4 seconds to respond to a small batch consisting of just 200 documents. Its response time is much larger than the other three methods. Luwak_Qbatch improves Luwak by more than twice on the throughput. Figure

11(b) zooms in to show the difference between Pre-Search and FullyIndexed. Their throughput performance is comparable. The FullyIndexed approach shows the best timings, while Luwak has the poorest performance. Our pre-search solution lies in between these two, but is very close to the best timings. Looking at the experiments whose input batch size is 600 for analysis, the document throughput of our system when loaded with 1,099,925 queries is 84 documents/s (Luwak), 797 documents/s (Pre-Search) and 840 documents/s (FullyIndexed). The throughput becomes larger for a larger batch size. It is 125 documents/s (Luwak), 1049 documents/s (Pre-Search) and 1125 documents/s (FullyIndexed) for the input batch size of 1200. The increase of throughput for larger batch sizes is understandable, as each batch requires a certain time overhead in processing documents (building inverted index). Adding new postings to existing lists in a larger batch costs less than allocating space to non-existing keys and values, which happens in smaller batches. If a high document input rate is guaranteed, in every second, fewer number of batches submitted implies a smaller overhead and more time spent on looking for matches, and thus a higher throughput, but also higher latency for notifying users of query matches.

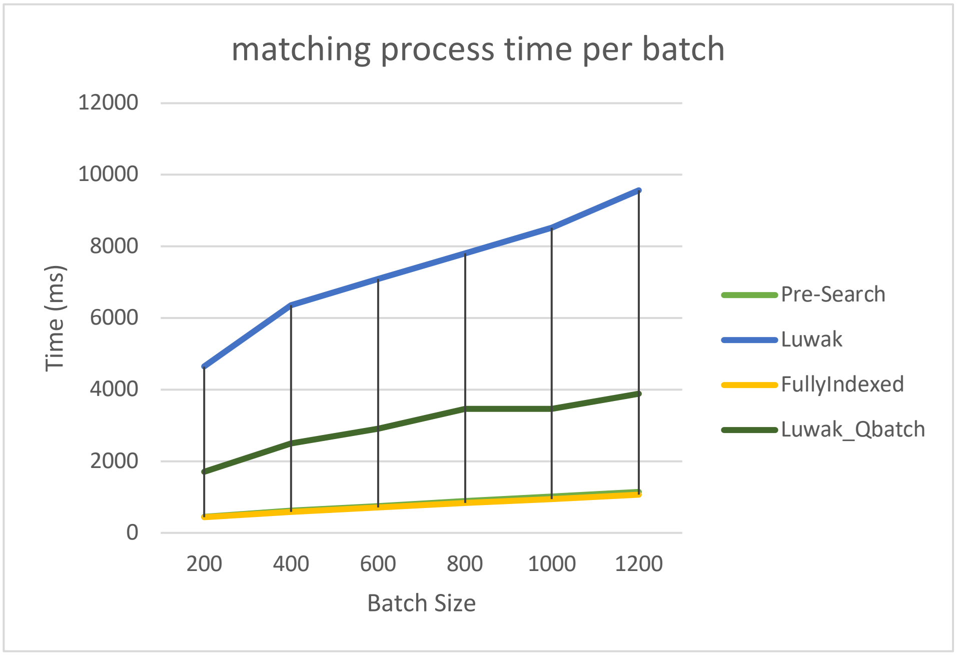

Figure 11(a): Timings, n_S = 1099925, $size_{Eb}$ varies

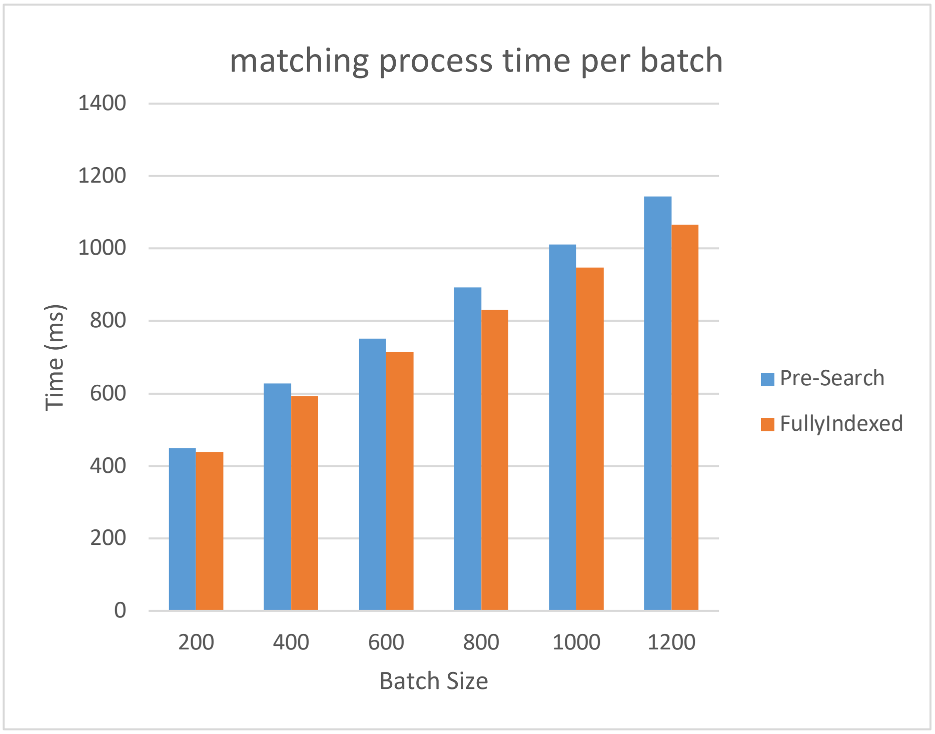

Figure 11(b): Zoom-in on timings, n_S = 1099925, $size_{Eb}$ varies

Figure 12(a) and 12(b) shows the memory utilization during match search. Before we collect the numbers, we free the memory of data structures that are used in query processing but no longer needed for match search. As expected, the original Luwak algorithm requires the least amount of memory, because it does not need to store the extra terms within its inverted index of queries. We give this part of memory consumption a name, called memory base, as it is a part of all algorithms. Memory base comprises the inverted index of queries (subs) and document inverted index (docs). They add up to about 800 to 1000 MB in our experiments. The number slightly varies with different batch sizes. The FullyIndexed algorithm requires the largest memory. The Pre-Search algorithm lies in between the middle again. The difference between the memory utilization of FullyIndexed and Pre-Search is small: about 1 Megabytes for various batch sizes. It is mainly caused by the difference between the total number of extra terms the two algorithms index. Other than the memory base, there is a small part of extra memory cost for both the FullyIndexed and Pre-Search algorithms during search. It is the inverted document bit map, which takes about 72 MB memory size. The average runtime memory use of Pre-Search for 600 documents per batch is illustrated as Figure 13. The FullyIndexed and Pre-Search algorithms having other batch sizes as input share a similar distribution. The word co-occurrence matrix, used only in query processing, is transformed to a hash map in our system, which occupies about 226 MB memory. Our primary goal is to optimize the time performance of our system and this amount of memory usage (about 1 GB during match search) is very acceptable for state of the art machines and experiments. Figure 14 demonstrates the total number of extra terms indexed by the FullyIndexed algorithm in comparison to the Pre-Search algorithm. Pre-Search uses approximately 2/3 of the extra terms, and can achieve a comparable document throughput as indexing all terms of queries, though the reduction in memory usage is limited. The limited reduction is due to a technique we use to optimize the data structure QueryNote. As specified earlier, the extra indexed terms of a query are stored with the corresponding query id in a QueryNote. Instead of storing the terms as an array of strings, we convert them to an array of bytes in UTF-8 encoding, which saves a significant amount of memory. We calculate the number of bytes required to represent all the extra terms indexed by Pre-Search, which is about 22 million. The number of bytes required to represent all the extra terms indexed by FullyIndexed is about 23 million. The FullyIndexed algorithm needs 1 more million bytes than the Pre-Search, which is consistent with the results of our experiments.

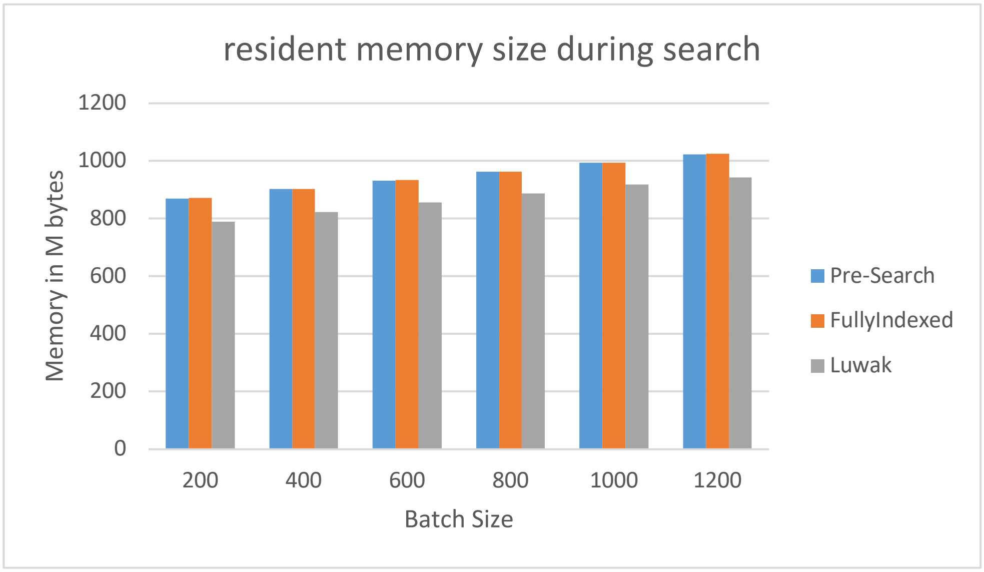

Figure 12(a): Memory size, n_S = 1099925, $size_{Eb}$ varies

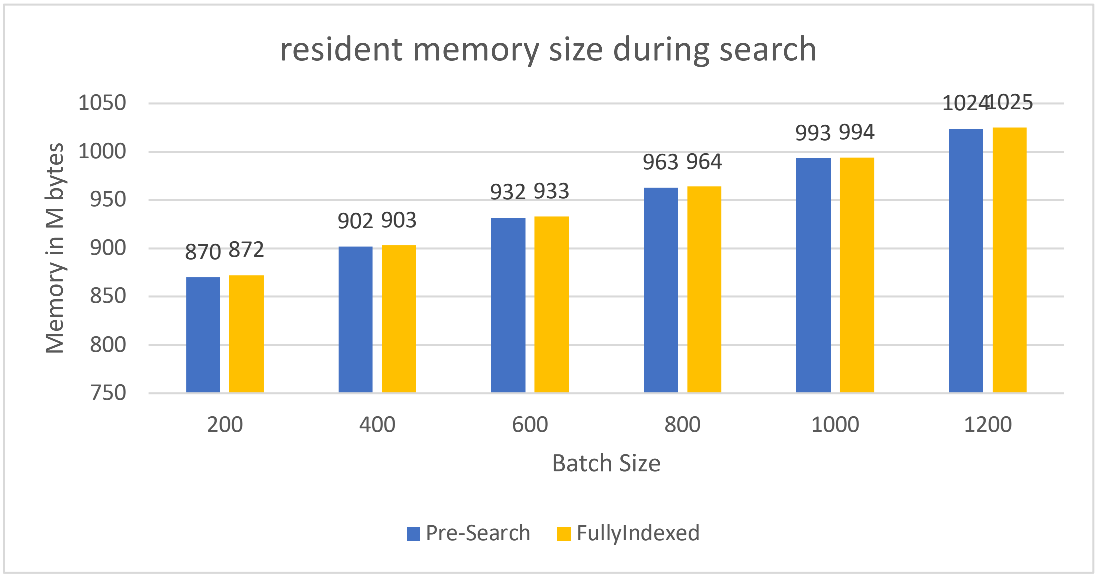

Figure 12(b): Zoom-in on memory size, n_S = 1099925, $size_{Eb}$ varies

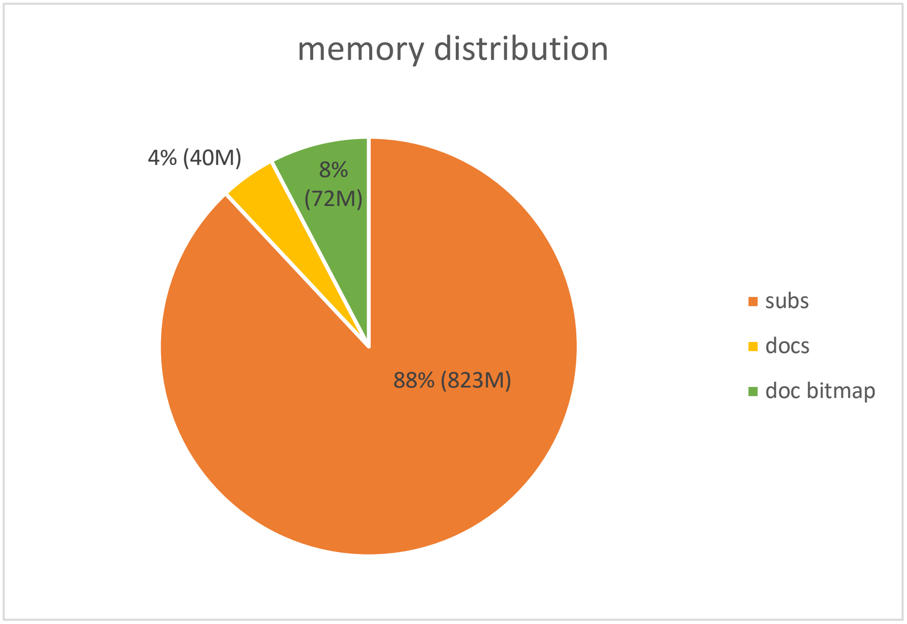

Figure 13: Memory distribution, n_S = 1099925, $size_{Eb}$ = 600

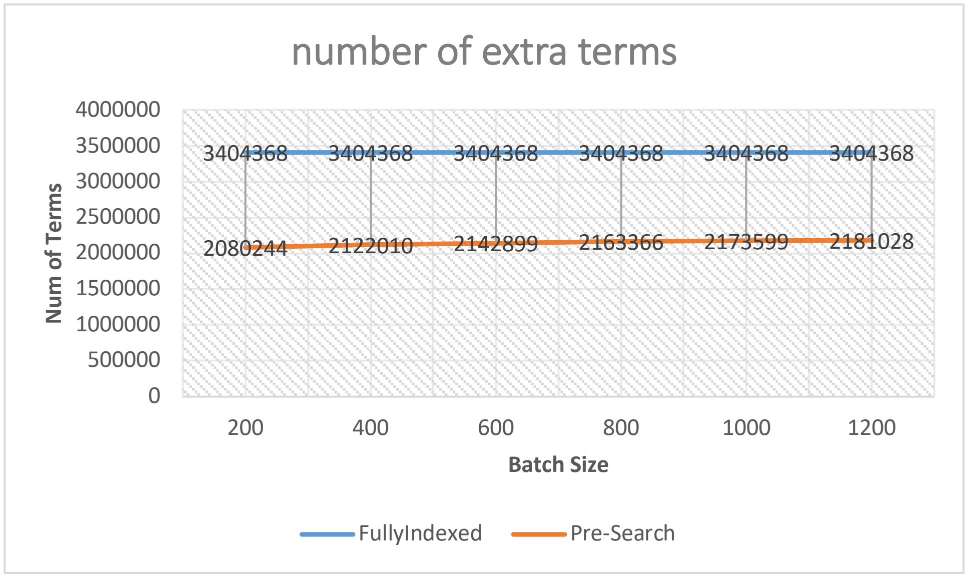

Figure 14: Number of extra terms, n_S = 1099925, $size_{Eb}$ varies

Next, we will analyze how much effect each functional part of the solution has on the overall time performance. The functional parts we applied to our solution are: word co-occurrence matrix, common query rules, and the greedy threshold to narrow the range of doing the maximization. The way they work was discussed in Section 5.5. Every time we leave one functional part out and rerun the experiments under the same setups. From Figure 15 we can see that every functional part contributes to the overall performance. The Word co-occurrence matrix and the greedy threshold have limited effect for small batches (in size 200 or smaller). Their effect on larger batches is more obvious (tens of milliseconds slower). The application of common query rules is the most influential on timings. The matching process is one to two hundred milliseconds slower if we leave the rules out. To sum up, every functional part is indispensable, in order to make our solution more competitive on solving realistic pub-sub problems.

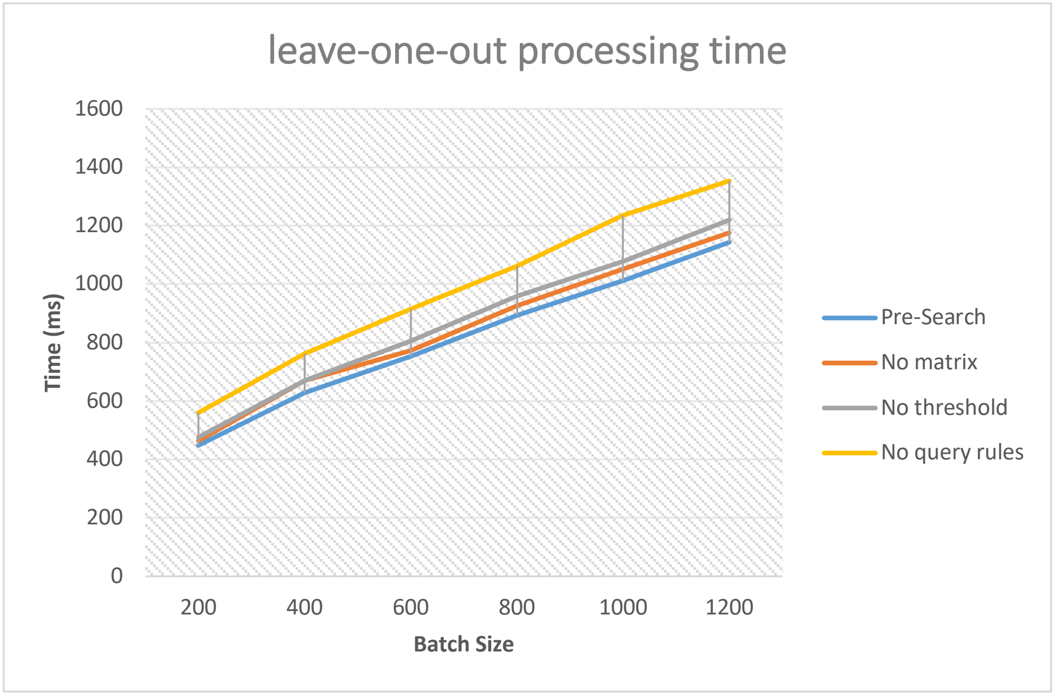

Figure 15: Leave-one-out time performance, n_S = 1099925, $size_{Eb}$ varies

Finally, we demonstrate the scalability of the Pre-Search, FullyIndexed, and Luwak algorithms for a range of 0.5 to 4 million queries, where the input batch size is fixed at 600. All query sets follow a similar distribution in terms of the length as the set specified in Table 5. Figure 16 shows that Luwak performs the worst in terms of the scalability in query workloads. Pre-Search and FullyIndexed seem to again have similar performance. In Figure 17, we notice that the FullyIndexed algorithm is faster than Pre-Search when loaded with 0.5 and 1 million queries. However, their performance reverses when loaded with 2 million or more queries. Thus, the scalability of Pre-Search is in fact slightly better than the FullyIndexed algorithm.

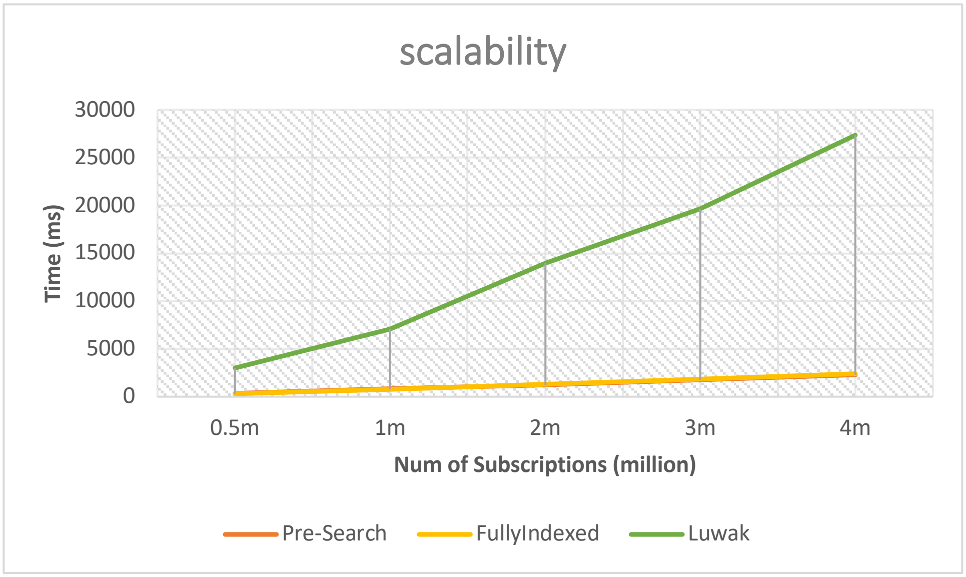

Figure 16: Scalability, $size_{Eb}$ = 600, n_S varies

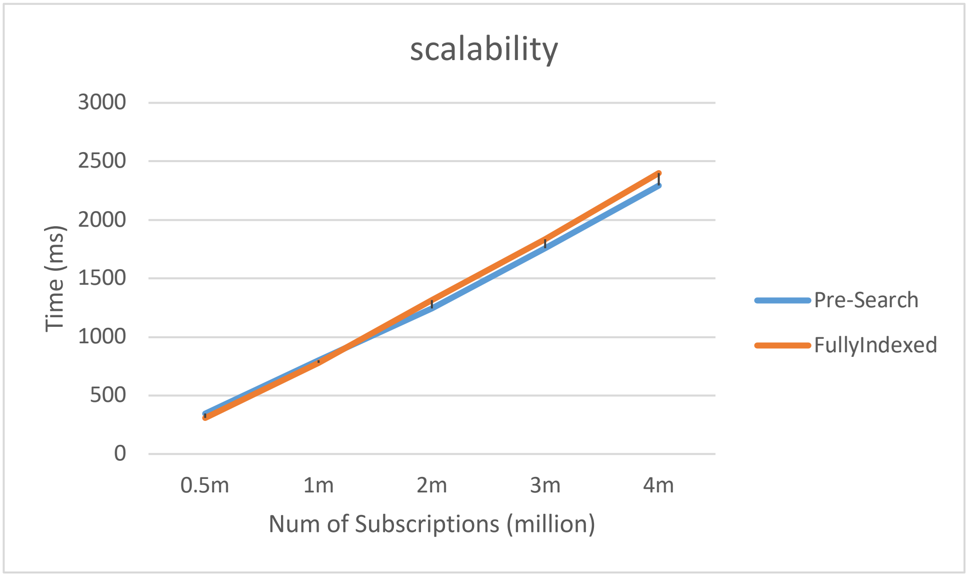

Figure 17: Zoom-in on scalability, $size_{Eb}$ = 600, n_S varies

7. Conclusion

In this thesis, we studied fast subscription processing based on the Luwak open-source system. We proposed a more efficient pre-search method for the two-level matching process of Luwak. The idea is a little similar to the algorithm in [4], which identifies candidate documents at the first level to reduce the number of full document evaluations. In contrast, we identify candidate queries in the pre-search phase to minimize the number of full query evaluations. Compared to the original Luwak approach of doing one shallow layer of candidate filtering, our method achieves on average a 98% reduction in the number of full evaluations. The sharp reduction brings significant time benefits. We have experimentally shown that our solution improves the original Luwak approach by 8 to 10 times on match search speed when loaded with one million queries, at an acceptable cost of extra memory (about 100 MB during match search). Meanwhile, our algorithm shows a better scalability for larger query workloads (larger than 1 million) than the full indexing of all query terms.

There are faster and more up-to-date ways of doing Boolean query evaluation, such as WAND and block-max WAND [8], which are now available in Lucene as of version 8.0. However, these algorithms have not been integrated into Luwak, and Luwak is still using the slow counting-based evaluation algorithm in step 3. This is a bottleneck to achieve the higher throughput that is required by many modern practical applications.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC ’99).

[2] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: an index structure for high-dimensional data. In Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96).

[3] S. Bittner and A. Hinze. The arbitrary Boolean publish/subscribe model: making the case. In Proceedings of the 2007 International Conference on Distributed Event-Based Systems (DEBS '07).

[4] Andrei Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In Proceedings of the 12th ACM Conference on Information and Knowledge Management, 2003.

[5] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In Proceedings of the 23rd International Conference on Software Engineering (ICSE '01).

[6] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In Proceedings of the 2003 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '03).

[7] L. Chen, G. Cong, X. Cao, K.-L. Tan. Temporal spatial-keyword top-k publish/subscribe. In IEEE 31st International Conference on Data Engineering, 2015.

[8] S. Ding, T. Suel. Faster top-k document retrieval using block-max indexes. In Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '11).

[9] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. ACM Comput. Surv.35, 2 (June 2003), 114-131.

[10] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01).

[11] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, S. Venkatesan, and J. Y. Zien. Efficiently evaluating complex boolean expressions. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10).

[12] Google Alerts. https://www.google.com/alerts

[13] A. Guttman. R-trees: dynamic index structure for spatial searching. In Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84).

[14] N. Khan, M. Alsaqer, S. Salehian. The 10 Vs, Issues and Challenges of Big Data. In Proceedings of the 2018 International Conference on Big Data and Education (ICBDE '18).

[15] H. Leung and H. Jacobsen. 2003. Efficient matching for state-persistent publish/subscribe systems. In Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '03).

[16] Guoli Li, S. Hou, and H. Jacobsen. 2005. A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems Based on Modified Binary Decision Diagrams. In Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS '05).

[17] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient Matching for Web-Based Publish/Subscribe Systems. In Proceedings of the 7th International Conference on Cooperative Information Systems (CooplS '02).

[18] W. Rjaibi, K. R. Dittrich, and D. Jaepel. 2002. Event matching in symmetric subscription systems. In Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '02).

[19] M. Sadoghi and H. Jacobsen. 2011. BE-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11).

[20] S. Whang, H. Garcia-Molina, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and R. Yerneni. 2009. Indexing Boolean expressions. Proc. VLDB Endow. 2, 1 (August 2009), 37-48.

[21] Tak W. Yan and H. García-Molina. 1994. Index structures for selective dissemination of information under the Boolean model. ACM Trans. Database Syst. 19, 2 (June 1994), 332-364.

[22] Tak W. Yan and H. Garcia-Molina. 1999. The SIFT information dissemination system. ACM Trans. Database Syst. 24, 4 (December 1999), 529-565.