RTSI: An Index Structure for Multi-Modal Real-Time Search on Live Audio Streaming Services

Zeyi Wen[†1], Xingyang Liu[‡2], Hongjian Cao[‡3], Bingsheng He [†4]
[†] *National University of Singapore, Singapore*
[1] wenzy@comp.nus.edu.sg, [4] hebs@comp.nus.edu.sg
[‡] *Shanghai Jiao Tong University, China*
[2] billyliu@sjtu.edu.cn [3] chj5chj5@sjtu.edu.cn

***Abstract*—Audio streaming services (e.g., Mixlr, Ximalaya, Lizhi and Facebook Live Audio) have become increasingly popular due to the wide use of smart phones. More and more people are enjoying live audio broadcasting while they are doing various kinds of activities. On the other hand, the data volume of live audio streams is also ever increasing. Searching and indexing these audio streams is still an important and open problem, with the following challenges: (i) queries on the large number of audio streams need to be answered in real-time; (ii) a live audio stream is inserted into the index continuously to enable live audio streams to appear in query results, and the number of insertions is large which often becomes a performance issue. Existing studies on audio search either oversimplify the problem or simply ignore searching live audio streams. Moreover, existing studies do not explore the multi-modal property of audio streams. In this application paper, we propose a multi-modal and unified log structured merge-tree (a.k.a. LSM-tree which consists of multiple inverted indices) based index to support intensive insertions and real-time search on live audio stream applications. Our index natively supports two major types of indexing techniques in audio search: text based indexing and sound based indexing. The key technical challenges we need to address are that (i) a live audio stream may appear in multiple inverted indices due to the streaming nature, (ii) relevance, popularity and freshness of each audio stream need to be maintained in a way that allows fast accesses, and a query often matches to a large number of audio streams since audio streams usually contain many unique terms, and (iii) massive insertions are happening alongside with queries. To address the above challenges, we propose an index (called *RTSI*) which avoids traversing multiple inverted indices to compute the score of an audio stream. In RTSI, we propose various techniques to address the technical challenges. First, we use one inverted list which contains the sorted score of popularity, freshness and relevance, such that we can compute the top-k query results efficiently. Second, we devise an upper bound for the unchecked audio streams, such that the query answering process can be terminated early. Third, we create mirrors for the indices that need to be merged, such that queries can be answered in real-time when the indices are merging. We conduct extensive experiments on audio streams obtained from Ximalaya. The experimental results show that RTSI can answer a large number of queries in a real-time manner while concurrently handling massive insertions.**

I. Introduction

Thanks to many audio services available (Mixlr [1], Ximalaya [2], Lizhi [3], and Facebook Live Audio), live audio broadcasting has become an important part of people's daily life. People listen to audio streams (particularly live audio streams) almost everywhere (e.g., gymnasium, transport and home). According to the recent reports [4], there are 0.6 billion Chinese users of those audio streaming services, and the yearly user increment rate is about 29.5% in Ximalaya. It is very important that users can search for the audio streams, especially live audio streams, of their interests in real-time. Users tend to be interested in more recent content. For example, they are interested in the audio streams on the more recent events. Figure 1 shows an example of a user using our proposed audio search service and the query answering process at back-end. At the front-end, the user can use keyword or voice search to look for audio streams of interest; at the back-end, the index is built on text and audio (which is represented by a set of phonetic lattices [5]) for live audio streams. Audio indexing and search is still an important and open problem, and audio search using human voice is still not a mature technique [6]. Thanks to the recent success of deep learning technologies in speech recognition [7], transcribing audio streams to text is more usable. Some popular speech recognition systems include Google Speech API, Bing Speech API, and Baidu Yuyin [8]. Audio indexing and search using transcribed text is regaining more and more attention [9], [10].

However, searching and indexing the live audio streams are challenging because (i) queries on the large number of the audio streams need to be answered in real-time; (ii) live audio streams are inserted into the index continuously to enable live audio streams to appear in the query results, and the number of insertions is large. Indexing and search on audio content is difficult to meet the real-time requirement due to the massive insertions from the live audio streams. Related work on audio search either oversimplifies the problem or simply ignores searching live audio streams. First, some existing studies mainly compare keywords with titles/categories/tags of the audio streams for better efficiency [11], [12]. Titles/categories/tags are not informative for searching relevant audio streams, and hence many related audio streams are not retrieved in this approach. Other related studies cannot search live broadcasting audio streams in real-time [13], [14], and

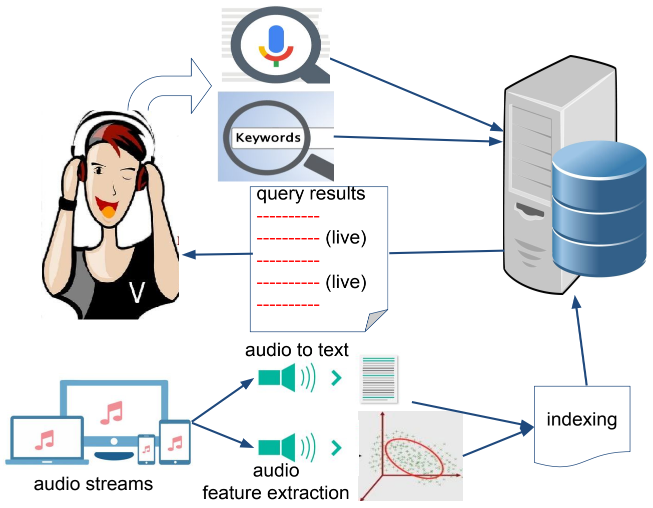


Fig. 1. Overview of users and the audio search service

hence the live audio streams cannot be retrieved until the audio streams finish broadcasting. Moreover, the existing studies do not support multi-modal search.

In this paper, we leverage recent successful technologies from machine learning (e.g., speech recognition) and databases (e.g., log structured merge-tree indexing) to address the problem of the emerging application of live audio streaming. We propose a multi-modal and unified log structured merge-tree (a.k.a. LSM-tree which consists of multiple inverted indices) based [15] index that supports intensive insertions and real-time search for live audio streams while considering relevance, popularity and freshness. Our multi-modal search includes keyword and voice search, and is powered by our index on the transcribed text and exacted sound features (i.e., phonetic lattices are represented using Mel-Frequency Cepstrum Coefficients (MFCC) [16]) as illustrated in Figure 1. The key technical challenges we need to address are as follows. First, a live audio stream may appear in multiple inverted indices, because the live audio stream is inserted into the index continuously to allow the live audio streams to appear in query results. Second, we need to maintain audio information (e.g., term frequency) in the inverted indices for fast search, and a query often matches to a large number of audio streams, because an audio stream is usually long and contains various words. Third, massive insertions are happening alongside with queries.

To address those challenges, we propose an index called *RTSI*. To address the first challenge, RTSI stores the audio information in the inverted list, such that we avoid traversing multiple inverted indices when computing the score of an audio stream. To address the second challenge, RTSI exploits an inverted list which contains the sorted information for relevance, popularity and freshness, such that the information required to compute scores can be easily obtained. We devise an upper bound for the audio streams that we have not computed their scores, such that the query answering process can be terminated early if the upper bound score is smaller than the lowest score in the top-k candidate result set. To address the third challenge, RTSI creates mirrors with the minimum memory consumption for the inverted indices that are currently needed to be merged and organized them into a mirror set, such that queries can be answered in real-time when the indices are under merging. Moreover, we propose to use Huffman coding [17] to reduce the memory consumption of the index.

In this application paper, we demonstrate that the success of machine learning (in speech recognition) and indexing techniques enables real-time and multi-modal search on the live audio stream applications. In summary, our contributions are as follows.

- We propose a multi-modal and unified index called RTSI for live audio streams and supporting real-time queries. Our index natively supports two major types of audio indexing techniques: text based indexing and sound based indexing.
- To address the challenge of an audio stream appearing in multiple indices, RTSI stores the audio information in the inverted index for efficient audio stream score computation. To address the challenge of a query matching to a large number of audio streams, we propose an upper bound for the unchecked audio streams, such that the query answering process can be terminated earlier.
- To further optimize our index, we leverage various techniques including using one inverted list to contain the sorted information for relevance, popularity and freshness, Huffman encoding, a lazy deletion, and a mirror set to support concurrent insertion and querying.
- We conduct extensive experiments to study the efficiency of RTSI in comparison with the latest index called "LSII" on real-time search. Our experimental results show that RTSI can respond queries in real-time while handling a large number of insertions. Compared with LSII, RTSI is more memory efficient, and is faster in index update and answering queries. Moreover, RTSI is inclined to be less sensitive to different settings than LSII.

The remainder of this paper is structured as follows. We present the background and related work on audio search and real-time search in Section II. Then we discuss the overview of our RTSI index in Section III and elaborate the implementation of RTSI in detail in Section IV. Our comprehensive experimental results are provided in Section V. Finally, we conclude the paper and discuss the future work in Section VI.

II. Background and Related Work

In this section, we first present some background on live audio streaming services, and audio indexing and search. Then we discuss the related work on indexing techniques for real-time search.

A. Live audio streaming services and audio indexing

Live audio streaming services have become very popular due to the wide use of smart phones and better network connections [18], [19]. Some examples of live audio streaming

TABLE I
CHINESE MARKET SHARE AND YEARLY USER INCREMENT OF AUDIO STREAMING SERVICES

audio streaming service	**market share**	**yearly user increment**
Ximalaya FM	25.8%	29.5%
Qingting FM	20.7%	32.5%
Tingban FM	13.8%	17.1%
Lizhi FM	6.9%	68.3%
Douban FM	5.2%	15.1%
Fenghuang FM	4.3%	34.6%

services are Mixlr [1], Ximalaya [2], Lizhi [3] and Facebook Live Audio. According to the recent reports [4], there are 0.6 billion Chinese users of those audio streaming services, and the yearly user increment rate is about 29.5% in Ximalaya. Table I shows more details about the market share and yearly user increment of the major audio streaming services in China [4]. Many listeners use the search functionality in the websites to find the audio streams of their interest. Searching for relevant live audio streams in real-time is very important, because some broadcasters immediately remove the audio streams once the broadcasting in completed. Existing audio search studies are based on one of the two mainstream methods: sound based search and text based search [12].

Nagano et al. [20] proposed a purely sound based method to retrieve audio streams. Their key idea is to compare the query voice with the audio streams using the similarity of sound signals. Other studies [21], [22] propose to convert audio streams into phonetic lattices (i.e., sound units) and build indices on the lattices. Liu et al. [23] proposed a signature based technique to index audio for efficient search. The signature can be produced by locality sensitive hashing on audio features, or by audio summarization [24]. The signature based technique can be viewed as a simplified version of lattices based method. These techniques are not designed for real-time live audio stream search, because they consider the set of audio streams static and no more audio streams are coming in after the index is built.

Another mainstream method for audio indexing and search is based on text. Audio streams can be represented using simple text such as audio stream titles, comments and categories [25], and then the index is built on the text. The advantage of representing audio streams using simple text is that indexing and search can be performed efficiently. The disadvantage is that simple text is not informative to represent the audio streams, and hence many relevant audio streams are not retrieved when answering user queries. Some studies [13], [26] use full text based approaches to audio search, where audio streams are transcribed into text via speech recognition tools. A key issue of full text based approach is that the query results are highly sensitive to the accuracy of the speech recognition tools. With the recent advancement of speech recognition technologies, the transcribed text from audio streams is becoming more and more accurate [27]. Many recent studies have used transcribed text to help address their problems [9], [10].

B. Indexing techniques for real-time search

Indexing and search are well studied problems in information retrieval [28]. In this subsection, we discuss related studies about indexing techniques for real-time search. Inverted index has been a key technique in indexing and search problems [29], and it has played a key role in real-time search engines [30]. The log structured merge-tree (LSM-tree) which uses multiple inverted indices have been used in applications with a large number of insertions and queries [15]. A more recent work extends the LSM-tree to a more general purpose data structure called bLSM [31]. Wu et al. [32] proposed a LSM-tree based approach (called *LSII*) for indexing microblgs (i.e., tweets) and supporting real-time search. Magdy et al. designed a data management system for microblogs [33], [34].

The existing real-time search approaches discussed above are for indexing microblogs or short text where a microblog appears in only one inverted index. In live audio stream indexing problems, data arrives in a long stream manner and an audio stream may appear in multiple inverted indices to enable live audio streams to appear in query results. The issue of an audio stream appearing in multiple inverted indices makes those existing studies not applicable to live audio indexing. Although those approaches are not applicable to the live audio indexing problem, they give us inspiration on the design of our RTSI index. Next, we present more details of two key techniques (i.e., LSM-tree and LSII) that we will use in the later sections.

1) The log structured merge-tree (LSM-tree) indexing: As the live audio broadcasting services have high volume of new audio streaming data coming in and hence a large number of insertions to the indices, we use the LSM-tree to enable fast insertion and real-time search. An example of the LSM-tree index is shown in Figure 2. As we can see from the figure, the LSM-tree has multiple inverted indices. The second inverted index I_1 is ρ times larger than I_0 and is $\frac{1}{\rho}$ of I_2. This log structured amortized the cost of merging indices and improves the overall insertion and search efficiency. As we will show in the complexity analysis of our RTSI index, the insertion cost is approximately the cost of inserting a term to I_0. This property makes LSM-tree index popular in real-time search applications [32], [35].

2) The LSII index: The LSII index was proposed by Wu et al. [32] for supporting real-time search on tweets. Because users of Twitter may take popularity, freshness and relevance into consideration when issuing queries, LSII maintains three sorted inverted list for each term of the inverted index. Figure 3 gives an example of the inverted indices in LSII. It is important to point out that the first inverted index I_0 has only one invert list for each term, and the inverted list is sorted by freshness (i.e., time stamps of tweets). Similar to the traditional LSM-tree index, LSII can handle insertions very fast. When I_0 is full and needs to be merged with I_1, two extra sorted inverted lists are created for I_0 such that the inverted lists of I_0 and I_1 can be merged.

When answering queries, LSII looks for the top-k results first in I_0, then I_1, etc. The query answering process is

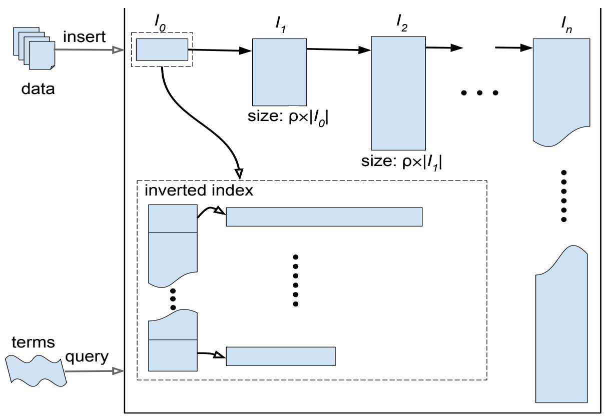


Fig. 2. The log structured merge tree

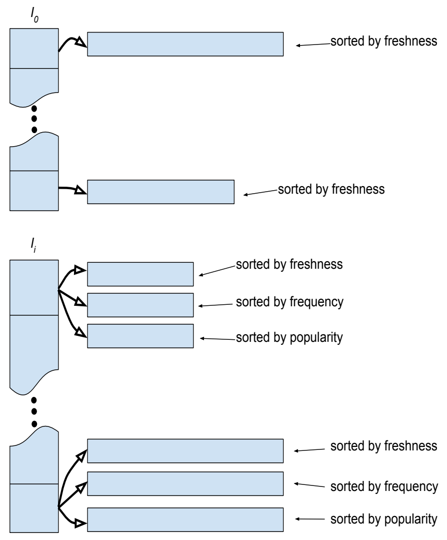


Fig. 3. Each term associated with three sorted inverted lists

relatively simple because a tweet can only appear in one inverted index. The update of tweet (e.g., updating popularity) is handled via a hash table where all the information of tweets is stored.

LSII cannot address the technical challenges of live audio stream services. First, LSII is for indexing microblogs (e.g., tweets) where a microblog only appears in one inverted index. In contrast, audio streams are much longer than tweets (e.g., thousands of characters vs 140 characters) and may appear in more than one inverted indices, which makes query answering and update operation much more challenging. As a result, most of the techniques (e.g., the bound for top-k results pruning, storing all the information of tweets in a hash table) used in LSII need to be redesign to solve the audio indexing and search problem. Second, it is unclear how the mirrors are constructed when the inverted indices need to be merged. In practice, the cost of constructing the mirror is high, especially creating mirrors for the large inverted indices (e.g., the last inverted index of the LSM-tree). Third, LSII does not compress the three inverted lists for each term, which is fine for tweet indexing. This compression is necessary for audio stream indexing, because the inverted lists tend to be long as the audio streams are likely to contain many terms.

III. Overview of our RTSI index

In this section, we present an overview of our indexing technique to search audio streams, particularly live audio streams.

A. Goal of our index

Users of audio streaming services often need to search for live audio streams, and require the query response to be real-time. Moreover, users may use different types of queries (i.e., query by keywords and query by voice). Hence, our index aims at providing real-time search for live audio streams, and supports both keyword and voice queries (i.e., multi-modal search). Next, we show the overview of our index.

B. Overview of RTSI indexing and search

We leverage recent successful technologies from machine learning (e.g., speech recognition) and databases (e.g., LSM-tree indexing) to address the problem of the emerging application of live audio streaming. Figure 4 shows the overview of our audio indexing and query answering using RTSI. As we can see from the figure, the top part is the indexing process and the bottom part is the query answering process. When building the indices, the live audio streams are converted into text by Baidu Yuyin speech recognition and converted into phonetic lattices [21]. The phonetic lattices are then represented using Mel-Frequency Cepstrum Coefficients (MFCC) [16]. Then, the RTSI index is built for text and phonetic lattices such that both voice and keyword search are supported.

When handling queries, users can use either keywords or voice as input. Our query processor converts keywords into voice or voice to keywords. Then, the voice is decoded into phonetic lattices. We retrieve from the indices all the audio streams which contain the keywords or phonetic lattices. Finally, we compute the score for each audio stream and obtain the top-k audio streams and respond the user queries. It is worthy noting that the multi-modal functionality is implemented by using two LSM-trees (cf. Figure 2): one for search by keywords and the other for search by voice.

C. Challenges

Due to the large number of insertions and queries in audio broadcasting service platforms, the LSM-tree index (cf. Figure 2) is a good candidate for providing real-time indexing and querying. However, using the LSM-tree for indexing audio streams has the following technical challenges. First, a live audio stream may appear in multiple inverted indices. This is because the audio streams are long, and some audio streams may last for over two hours. The live audio streams are inserted into the LSM-tree index continuously to enable live audio streams to appear in the query results. Second, we need to maintain information for relevance, popularity and

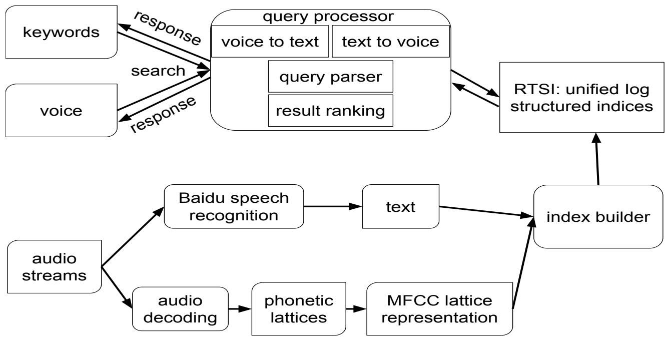


Fig. 4. Indexing and querying overview

freshness in the inverted indices for real-time search. Users tend to be interested in more recent or more popular content. For example, they are interested in the audio on the more recent events. So, freshness, popularity as well as relevance need to be considered and be accessed efficiently. Third, a query often matches to a large number of audio streams, because an audio stream is usually long and contains various words. As a result, many audio streams are likely to contain the query terms. Fourth, massive insertions are happening alongside with queries. A live audio stream is inserted into the index continuously, and results in many insertions. The audio streaming service commonly hosts many such live audio streams, and hence the number of insertions is massive. On the other hand, the number of queries is also very large due to the large number of listeners.

IV. Design and implementation

In this section, we first present our solution overview, and then we elaborate the technical details of RTSI.

A. Solution overview

To address the challenges discussed in Section III, we propose an index called *RTSI*. RTSI maintains the information of an audio stream in one inverted index, such that computing the score of an audio stream is fast and we avoid accessing multiple inverted indices to compute the score. RTSI maintains one inverted list to represent three sorted inverted lists (corresponding to relevance, popularity and freshness of an audio stream) for each term in the inverted index. Once users issue a query, the inverted lists are retrieved from the inverted index (i.e., I_0) which consists of the latest inserted audio streams, and then the scores of the top-k audio streams are computed based on the audio information. These top-k results are verified with a bound to check if the query process can be terminated. The scoring function we use is as follows.

$$f(q, p) = \omega_p \cdot pop(p) + \omega_r \cdot rel(q, p) + \omega_f \cdot frsh(p) \quad (1)$$

where q is a query and p is an audio stream; ω_p, ω_r and ω_f are the weight of popularity, relevance and freshness, respectively. Popularity is measured by the play counters, the number of “like”s, etc.; freshness is measured by the audio time stamp; relevance is measured by Term Frequency Inverse Document Frequency (TF-IDF) [36]. RTSI supports concurrent insertion, merge and queries through the creation of mirrors and partially locking the inverted index.

B. The RTSI index

To allow fast access to the audio information for computing score, we store the information needed for score computation into the inverted lists. More specifically, we store the term frequency into the inverted list which is sorted in descending order (cf. Figure 3). Then, we maintain a hash table for popularity and freshness of each audio stream. This hash table of RTSI is small because the number of keys in the hash table equals to the number of audio streams. More formally, suppose the number of audio streams is $|P|$ and the average number of terms in an audio stream is $|\bar{T}|$. Then, the number of keys of the hash table of RTSI is $|P|$, which is irrelevant to the number of terms in the audio stream because the term frequency information is stored in the inverted lists.

With the hash table for popularity and freshness of each audio streams in RTSI, we can easily obtain popularity and freshness. For computing the score of an audio stream (cf. Equation 1), we also need to obtain the total frequency of a term that matches the query. However, obtaining the total term frequency of a term may requires accessing multiple inverted indices, because an audio streams may appear in multiple inverted indices. To avoid accessing multiple inverted indices for improving query efficiency, we maintain another small hash table which keeps track of the existing term frequency of a term. This hash table is small because it only stores the live audio streams which consists of only a small proportion of all the audio streams in the audio streaming platform. Please note that although the audio streams in the audio streaming platform are originally from live audio streams, the number of live audio streams in a given time stamp is often not large.

Next, we first present the insertion and query answering algorithms of RTSI. Then, we analyze the time complexity of insertion and query answering algorithms. Finally, we discuss other implementation issues of RTSI. Table II summarizes some notations used in this subsection.

1) Insertion to RTSI: The pseudo-code of the insertion algorithm is given in Algorithm 1. Given a term to insert, we first find the inverted list where the term will be allocated. Then we append the term together with the term frequency to the inverted list (Line 2). After that, we update the latest location of term t of the audio stream on the hash table for the live audio streams (Line 3). We merge two indices if the size of index I_0 exceeds the memory limit (Line 5 to 7). The merge operation is discussed next.

In Algorithm 1, we need to invoke the merge procedure. We provide the pseudo-code of the merge procedure in Algorithm 2. Specifically, given two inverted indices to merge, we first construct the mirror of the smaller index (i.e., I'_i in Line 2). The mirror I'_i and I_{i+1} is used for handling queries. We next insert all the terms in I_{i+1} to I_i, such that I_{i+1} is read-only and can support concurrent queries. More

TABLE II
NOTATIONS USED IN RTSI IMPLEMENTATION

name	description
ω_p	the weight of popularity
ω_r	the weight of relevance
ω_f	the weight of freshness
ρ	ratio of the the LSM-tree
δ	the threshold to trigger the merge of I_0
id	audio stream identifier
pop	audio stream popularity
$frsh$	audio stream freshness
tf	term frequency
t	a query term or a term in audio streams
$htbl$	hash table
$IdxSet$	inverted indices of the LSM-tree
I_i	the i-th inverted index of the LSM-tree
sc	score of an audio stream
p	an audio stream
$info$	pop, $frsh$ and tf of an audio stream
L_m	a set of inverted index mirrors

Algorithm 1: Insertion in RTSI

```
Input: the id, significance, freshness, a term, the term frequency of the audio $id$, $pop$, $frsh$, $t$ and $tf$, respectively; a set of indices: $IdxSet$; ratio of log structure: $\rho$; a hash table for live audio streams: $htbl$
Output: Updated index set: $IdxSet$
1 IList $\leftarrow$ getInvertedList($t$, $I_0$);
2 append(IList, $t$, $id$, $pop$, $frsh$, $tf$);
3 updateHashTable($htbl$, $id$, $t$, IList);
4 maxSize $\leftarrow \delta$, $i \leftarrow 0$;
5 while $size(I_i) > maxSize$ do
6     Merge(I$_i$, I$_{i+1}$);
7     maxSize $\leftarrow$ maxSize $\times\rho$, $i \leftarrow i+1$;
```

Algorithm 2: Merging in RTSI

```
Input: Two indices: $I_i$ and $I_{i+1}$; a mirror set: $L_m$
Output: Updated LSM-tree: $IdxSet$
1 $I_i' \leftarrow I_i$
2 StoreToMirrorList($L_m$, $I_i'$);
3 foreach $t \in I_{i+1}$ do
4     $3IList1 \leftarrow$ getInvertedList($t$, $I_i$);
5     $3IList2 \leftarrow$ getInvertedList($t$, $I_{i+1}$);
6     if IsSorted($3IList1$) $\neq$ true then
7         $3IList1 \leftarrow$ Sort($3IList1$);
8     $3IList1 \leftarrow$ CombineLists($3IList1$, $3IList2$);
9     StoreToIndex($I_i$, $3IList1$);
10 $I_{i+1} \leftarrow I_i$;        /* replace $I_{i+1}$ by new $I_i$ */
11 RemoveFromMirrorList($L_m$, $I_i'$);
```

specifically, for each term t in I_{i+1}, we get the three inverted lists corresponding to t in I_i and three inverted lists in I_{i+1} (Line 4 and 5). If the inverted list in I_i is sorted, then we combine those inverted lists (Line 8). Otherwise, we create three new inverted lists sorted by popularity, freshness and term frequency, respectively; then we combine those inverted lists (Line 8). After the combination, the inverted lists are stored to I_i (Line 9). When the combination is completed, I_{i+1} is replaced by I_i, and the mirror I_i' is removed (Line 10 and 11).

Discussion of the merge operation: When creating a mirror for the inverted index, we can create the mirror for I_i or I_{i+1}. In our implementation, we create the mirror for I_i, because I_i is smaller and hence more efficient for the creation. Furthermore, if we merge I_0 and I_1, we need to create two extra sorted inverted lists for I_0 since I_0 only has one sorted inverted list for freshness (cf. Figure 3).

2) Answering queries: Here, we present the details of answering queries in RTSI. Without loss of generality, we suppose the query contains two terms. Given two query terms t_1 and t_2, we first compute the potentially largest score of audio streams in inverted indices except I_0 (Line 2 to 6). Note that this value is easy to compute because the inverted lists are sorted descendingly, and the value is used for pruning the unchecked audio streams. We then obtain the inverted lists from the inverted index (Line 8 and 9). For each audio stream appears in the inverted lists, we get the audio stream information, compute the score of the audio stream, and save the audio stream into the top-k candidate result set (Line 11 to 15). After obtaining the top-k candidate result set, we compute the lowest score in the result set and compare it with the possible largest score of the unchecked audio streams (Line 16 to 18). If the lowest score of the audio stream in top-k candidate result set is greater than the largest possible audio stream in the unchecked audio streams, the query processing is terminated and the top-k candidate result set serves as the final query results.

3) Time complexity analysis: Here, we analyze the time complexity of insertion and query in RTSI. We first introduce some notations to help the cost analysis. Support I_0 has m_0 number of unique terms, each term has an inverted list of length $\bar{n}$ in average, the whole dataset has M number of unique terms, the ratio of the LSM-tree is ρ (i.e., $\rho = \frac{|I_1|}{|I_0|}$), and the total number of inverted indices of the LSM-tree is $(l+1)$.

Insertion cost analysis of RTSI: In the average case, an insertion does not lead to a merge of index, and the insertion only affects I_0. Therefore, the insertion cost is $\mathcal{O}(\log m_0)$ which is the look-up cost for the term in I_0. Note that the term is appended to the inverted list with constant cost, and is sorted based on the arrival time stamp. If the insertion leads to a merge of index, the average merge cost is $\mathcal{O}(m_0\bar{n}\log_\rho \frac{M}{m_0})$, where $\log_\rho \frac{M}{m_0}$ is the number of merge operations of inserting the whole dataset into the RTSI index and $m_0\bar{n}$ is the average cost of a merge operation. A more detailed cost analysis on the merge process is available in Appendix -A. Note that the number of merge operations is much smaller than the number of insertions (i.e., $\log_\rho \frac{M}{m_0} \ll M\bar{n}$), and therefore the insertion cost is approximately $\mathcal{O}(\log m_0)$.

Query cost analysis of RTSI: In the best case, the top-k

Algorithm 3: Answering query in RTSI

```
Input: terms: $t_1$, $t_2$; LSM-tree: $IdxSet$; the number of
       top results: $k$
Output: audio result set: $res$
1  $sc^{\top} \leftarrow 0$
2  foreach $I \in IdxSet \setminus I_0$ do
3      $pop, frsh, tf_1, tf_2 \leftarrow$ GetMaxScoreInfo($I$, $t_1$, $t_2$);
4      $sc \leftarrow$ CompScore($pop, frsh, tf_1, tf_2$);
5      if $sc > sc^{\top}$ then
6          $sc^{\top} \leftarrow sc$;
7  foreach $I \in IdxSet$ do
8      $3IList1 \leftarrow$ getInvertedList($t_1$, $I$);
9      $3IList2 \leftarrow$ getInvertedList($t_2$, $I$);
10     while $3IList1 \neq \phi$ && $3IList2 \neq \phi$ do
11         $p_1$, $p_2$, $p_3 \leftarrow$ GetTop3($3IList1$);
12         $p_4$, $p_5$, $p_6 \leftarrow$ GetTop3($3IList2$);
13         $info_1$ to $info_6 \leftarrow$ Find($htbl$, $p_1$, ..., $p_6$, $t_1$, $t_2$);
14         $sc_1$ to $sc_6 \leftarrow$ ComputeScore($info_1$ to $info_6$);
15         SaveTopK($p_1$ to $p_6$, $sc_1$ to $sc_6$, $res$);
16         $sc^{\perp} \leftarrow$ GetLowestScore($res$);
17         $sc \leftarrow$ GetNextLargestScore($p_1$ to $p_6$);
18         if $sc^{\perp} \geq sc^{\top}$ && $sc^{\perp} \geq sc$ then
19             return;
```

query results are obtained from I_0. So the cost is $\mathcal{O}(\bar{n}\log m_0)$, where $\log m_0$ is the cost for finding the term in I_0 and $\bar{n}$ is the average length of the inverted lists of I_0. In the worst case, all the inverted indices are searched. The total cost is $\mathcal{O}(l\bar{n}\log m_0+\bar{n}l^2)$. The derivation of the total cost and a more detailed cost analysis on query are available in Appendix -B. In practice, l is small thanks to the log structured property. Therefore, the average case time complexity is $\mathcal{O}(\bar{n}\log m_0)$.

4) Other design issues: Here we discuss other design issues of RTSI including concurrency, compressing three inverted lists into one list, lazy deletion, Huffman coding to compress index size, and pruning audio streams when ansering queries.

Concurrent query and insertion: As the number of insertions and queries is large, we should not block queries/insertions when handling insertions/queries or merging indices. We propose to support queries while the indices are updating. For an insertion to I_0, we only lock the inverted list which the insertion occurs. The rest of the inverted lists can be used to respond queries without data inconsistency. When merging I_0 and I_1, the queries are handled using I'_0 and I_1 where I'_0 is the mirror of I_0. To support concurrent insertions and queries, we create an empty I_0 to receive insertions.

Compressing the three inverted lists: As we can see from Figure 3, there are three inverted lists for each term in inverted indices I_1, I_2, ..., I_l. The inverted lists store the identifiers (i.e., id) of audio streams, and each audio stream identifier is stored three times in the three inverted lists of each term. We propose to use a compact form to represent the three inverted lists by only one inverted list. Each node in the inverted list stores an identifier and three pointers to enable the reconstruction of three sorted inverted lists.

Lazy deletion: Some audio streams may be deleted from the audio streaming platform. A naive approach is to find all the terms of the audio stream from the index, and delete the terms. The cost of this approach for deletion is very high, because first we need to search for all the terms of the audio streams and second the inverted lists containing the deleted audio stream need to be sorted after deletion. To reduce the cost of deletion, we propose the lazy deletion technique which postpones the delete operations until the inverted indices need to be merged. When we are merging two inverted lists, we ignore the audio streams which are deleted.

Pruning and Huffman coding: When answering queries, users are usually only interested in top-k results. We propose to store the highest popularity score, freshness score and term frequency in each inverted list, such that we can compute an upper bound for all the unchecked audio streams in that inverted index. Moreover, we use Huffman coding to reduce the memory consumption of the inverted indices. The coding is more effective when the number of inverted indices is large, since different inverted indices tend to contain the same set of terms.

V. Experimental studies

In this section, we empirically study our proposed index: RTSI. We first describe the experimental settings with an extension to the LSII index [32] for indexing and search on live audio streams, such that we can have thorough understanding of the advantages of RTSI. Then, we report the results of the overall performance of RTSI, sensitivity study and effectiveness of individual optimization. Finally, we investigate the quality of the query results and summarize the key findings.

A. Experiment settings

We use a dataset obtained from Ximalaya [2]—an audio streaming service. The dataset contains 80 thousand audio streams. The total length of the audio streams is about 2,133 hours, and the average length of an audio stream is about 16 minutes. We used Baidu Yuyin speech recognition services[1] to transcribe the audio streams into text, and keywords into voice to enable multi-modal search. The transcribed text consists of 32 million words (excluding stop words), the average number of unique words in each audio stream is about 400. The number of phonetic lattices is also about 30 million from the audio streams. Each insertion to the index, we insert all the text and phonetic lattices from 60 seconds of the audio stream. Apart from the audio streams, we also obtained other information of each audio stream, such as title, time stamp, tags, comments and other data to measure the popularity of an audio stream.

The extended LSII: As we have discussed earlier, LSII was proposed to index tweets for real-time search, and cannot

[1]http://yuyin.baidu.com

TABLE III
VARIABLES AND THEIR DEFAULT VALUES

name	description	value
ω_p	the weight of popularity	0.2
ω_r	the weight of relevance	0.6
ω_f	the weight of freshness	0.2
ρ	ratio of LSM-tree	2
n_l	the # of live audio streams	10,000
n_i	the # of audio streams to insert	10,000
n_q	the # of queries	10,000
k	top-k query results	40
S_{I_0}	the # of terms in I_0	2M
n_a	the # of audio streams	40,000

be used for audio search because audio streams are much longer than microblogs. To handle the issue of an audio stream appearing in multiple inverted indices, we extend LSII to support live audio stream indexing and search. The extended LSII (also called it "LSII" hereafter[2]) maintains a hash table to keep track of the audio information including popularity, freshness and frequency of each term. The information of an audio stream is later used to compute the score of the audio stream when answering queries. For each term t that needs to be inserted, we search the first inverted index (i.e., I_0) in the LSM-tree. If the term appears in I_0, we append the term to the inverted list. Otherwise, a new inverted list is created for I_0 and then the term t is appended to the new inverted list. Then, we update the hash table which keeps track of information for computing the score of an audio stream. After the insertion, if the size of I_0 exceeds the maximum size limit, I_0 is merged with I_1. The merge process is similar to that of RTSI. For query answering, we suppose the query has two terms t_1 and t_2 for the ease of presentation. The search starts from I_0 in the index set. Given the terms t_1 and t_2, LSII obtains the inverted lists from the inverted index I. Then, the top-6 potentially most similar audio streams (since two terms correspond to six inverted lists, and one audio stream from each inverted list is selected) to the query from the inverted lists are selected. Then, the hash table is used to obtain the information of the 6 audio streams, and compute their scores. After the score computation, LSII saves the audio streams into the top-k candidate result set. Then, the lowest score in the top-k candidate result set is compared with the upper bound score of the unchecked audio streams. If the lower bound score of the candidate result set is greater than the upper bound score of the unchecked audio streams, the query answering process is terminated; otherwise LSII continues the process until all the inverted indices are searched.

We have conducted experiments to study the overall performance of RTSI and LSII, and study the sensitivity of varying variables of the indices. Since there are many variables involved in our experiments, whenever the variables are not specified in the experiments, we use the default values. The relevant variables and their default values are shown in Table III.

[2]For some risk of confusion, we use "LSII" for the "extended LSII".

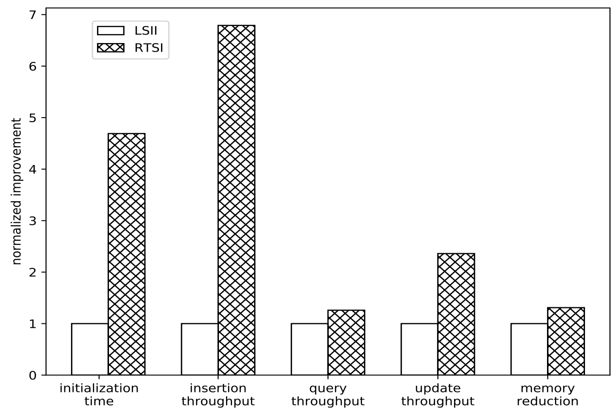


Fig. 5. Overall improvement of RTSI

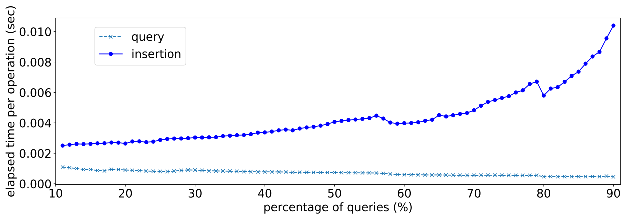
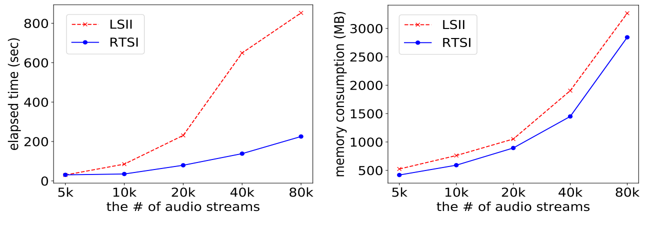


Fig. 6. Varying the percentage of queries

B. Overall comparison

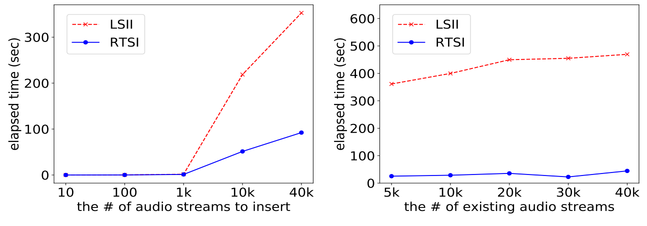
We first study the overall performance of our RTSI index in comparison with LSII. We demonstrate the improvement of RTSI over LSII in terms of initialization, insertion, query, update and memory consumption. Figure 5 shows the normalized improvement comparison, where normalized improvement is computed by $\frac{\text{elapsed time/memory reduction/throughput}}{\text{LSII elapsed time/memory reduction/throughput}}$. As we can see from the results, the improvement is the most significant when handling insertions and updates. Please note that initialization mainly consists of insertions, so we also observe significant improvement in initialization. It is worthy to point out that RTSI is more efficient when handling queries and is more memory efficient, thanks to our techniques of storing scoring information in the inverted lists instead of a big hash table.

Moreover, we study the correlation of queries and insertions. To study the correlation, we initialized the index with 40 thousand audio streams. Then, we set the total number of operations (including queries and insertions) to 40 thousand. We varied the percentage of queries in the 40 thousand operations. For example, when the percentage of queries is 10, the percentage of insertions is 90 of the total number of operations. As we can see from Figure 6, the elapsed time per query is stable when the percentage of insertions varies significantly (i.e., from 90% to 10%). In comparison, the elapsed time per insertion decreases as the total number of insertions increases (from 90% at the left side of the figure to 10% at the right side of the figure). We can also see that there are merge processes triggered when the number of insertions is about 20%, 35%, and 45% of the total number of operations.



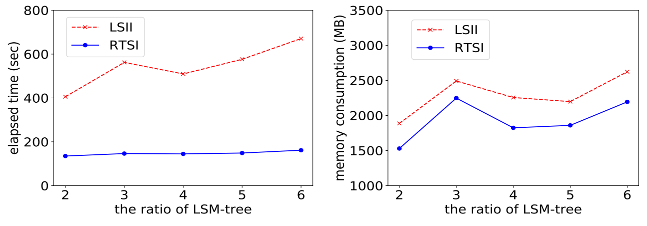
(a) initialization time (b) init. memory consumption

Fig. 7. Initialization: varying the # of audio streams



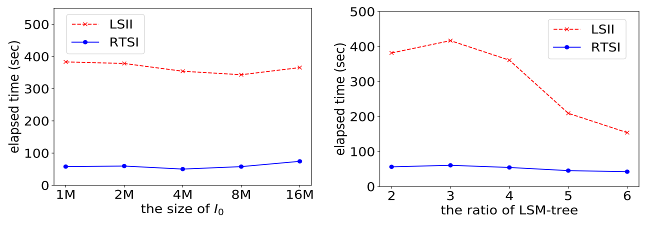
(a) varying # of inserted audios (b) varying # of existing audios

Fig. 9. Insertion: varying # of inserted and # of existing audio streams



(a) initialization time (b) init. memory consumption

Fig. 8. Initialization: varying ratio of LSM-tree



(a) varying size of I_0 (b) varying LSM-tree ratio

Fig. 10. Insertion: varying size of I_0 and LSM-tree ratio

C. Sensitivity studies

In this subsection, we study the sensitivity of RTSI to demonstrate that RTSI outperforms LSII in various settings. We conducted experiments on index initialization, insertion, querying, update and individual optimization. In each group, we compare the performance of LSII and RTSI while varying involved variables related to the experiments.

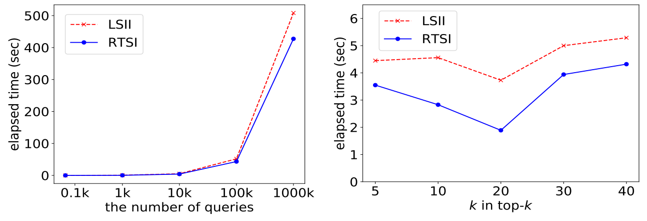
1) Index initialization: In this set of experiments, we study the performance of building the initial index with existing audio streams. We first investigate the effect of the number of audio streams on the performance of RTSI and LSII. Figure 7 shows the elapsed time and the maximal resident memory required to initialize indexes, with the number of audio streams varying from 5,000 to 80,000. As we can see from the result, the elapsed time and memory consumption of RTSI and LSII increases as the number of audio streams increases, but RTSI increases much more slowly than LSII in elapsed time. We also evaluate the effect of the size of I_0 on the performance of LSII and RTSI. Figure 8 shows that RTSI constantly outperforms LSII in different ratios of LSM-tree in terms of elapsed time and memory. We observe fluctuation of memory consumption while varying the ratio (cf. Figure 8b). This is because the ratio directly affects the number of inverted indices in LSM-tree and hence the total memory consumption.

2) Insertion: In this set of experiments, we study the performance of RTSI and LSII on insertion. First, we measure the effect of the number of audio streams to insert on the performance of RTSI and LSII, with the number of audio streams varying from 10 to 40,000. As illustrated by Figure 9a, the insertion cost of LSII increases much more dramatically than that of RTSI. The insertion cost of RTSI is stable regardless of index size as shown in Figure 9b, where the index size is measured by the number of audio streams in the index. Figure 10 shows the effect of size of I_0 and ratio of LSM-tree on the elapsed time. According to the figures, RTSI is again stable while varying I_0 and the ratio of LSM-tree. In comparison, LSII is more sensitive to varying the size of I_0 and ratio of LSM-tree.

3) Query: In this set of experiments, we investigate the sensitivity of varying various variables related to query answering. As we can see from Figure 11, RTSI consistently outperforms LSII in different number of queries and using different k (i.e., different number of top results). This result is impressive because RTSI is not only faster in insertion as discussed in Section V-C2 but also more efficient in answering queries than LSII.

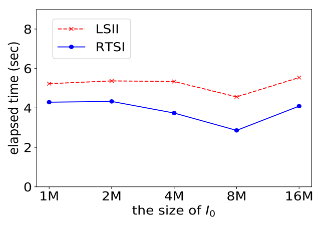
To have a better understanding of the sensitivity of RTSI in answering queries, we conducted experiments by varying the size of I_0, ratio of LSM-tree, the weight of freshness and the number of existing audio streams in the index (i.e., varying index size). Figure 12 shows the results. In summary, our RTSI is much more efficient when handling queries in various situations than LSII, thanks to getting rid of storing all the audio information in a hash table (cf. Section IV-B).

4) Update: In this set of experiments, we study the efficiency of handling audio stream updates (e.g., the increase of

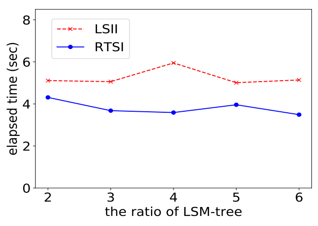


(a) varying # of queries (b) varying k

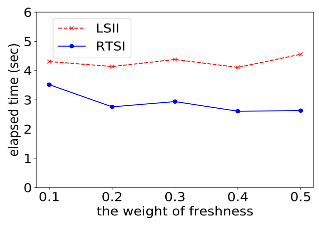
Fig. 11. Query: varying number of queries and k



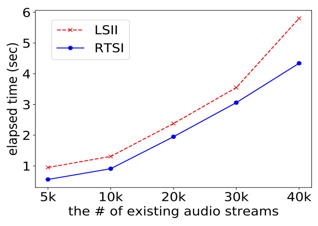
(a) varying the size of I_0



(b) varying ratio of LSM-tree

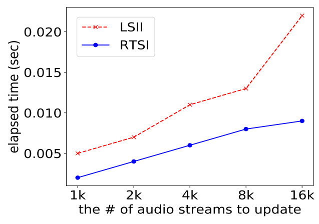


(c) varying the weight of freshness

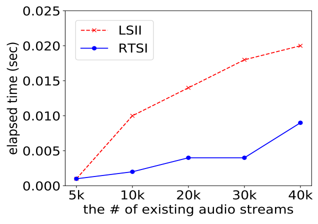


(d) varying # of existing audios

Fig. 12. Query: sensitivity study on query answering



(a) varying # of audios to update

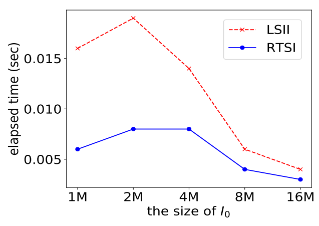


(b) varying # of existing audios

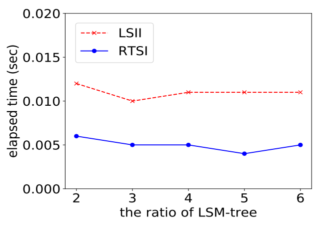
Fig. 13. Update: varying # of updated audios and existing audios

the play counter of an audio stream). As expected, our RTSI index again much faster than LSII when handling updates as shown in Figure 13. An observation to the figure is that the increase of the number of updates or index size causes the update cost to increase more dramatically in LSII due to the large hash table. In comparison, RTSI increases much more slowly than LSII.

We further study the effect of varying I_0 and the ratio of LSM-tree. Figure 14 shows the results. According to the results, RTSI is less sensitive than LSII when varying I_0 and the ratio of LSM-tree. This property is intriguing, because we often do not know which value is the best for a certain variable, and RTSI tends to perform well in different settings.

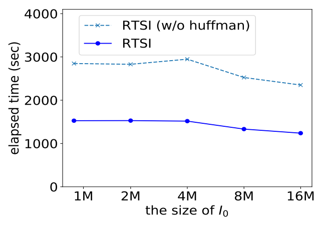


(a) varying the size of I_0

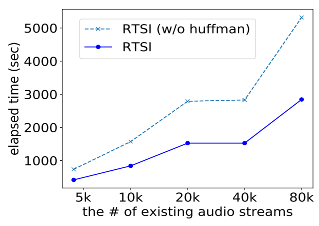


(b) varying the ratio of LSM-tree

Fig. 14. Update: varying the size of I_0 and ratio of LSM-tree

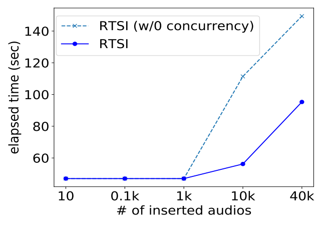


(a) varying the size of I_0

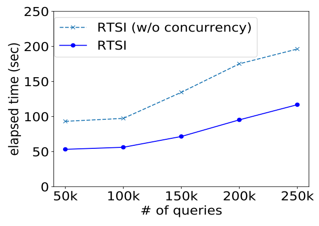


(b) varying the # of existing audios

Fig. 15. Huffman coding: varying the size of I_0 and # of existing audios



(a) varying the number of insertions



(b) varying the number of queries

Fig. 16. Concurrency: varying the number of insertions and queries

D. *Effect of other optimization*

In this last set of experiments, we study the individual optimizations: Huffman coding, concurrent querying and insertions, and the bound for improving query efficiency.

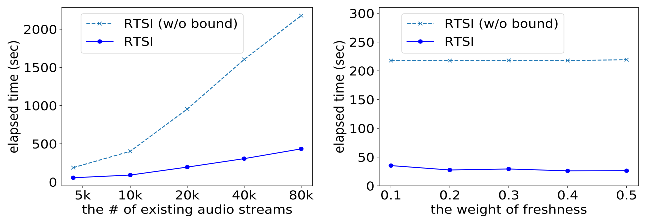
1) Memory efficiency of Huffman coding: Figure 15 shows the effectiveness of using Huffman coding for reducing memory consumption of the index. As we can see from the figures, the memory consumption of the indices is significantly reduced. An observation from Figure 15b is that the improvement over memory consumption is more significant as the number of audio streams increases.

2) Concurrent querying and insertion: Here, we study the effectiveness of concurrent processing on various settings. Figure 16 shows the results. When varying the number of insertions, we set the number of queries to 100 thousand; when varying the number of queries, we set the number of insertions to 10 thousand. As we can see from the figures, concurrent processing brings improvement on various scenarios. The improvement is very significant when the number of queries and insertions is large as shown in Figure 16b.

3) Effectiveness of bound for top-k query: Finally, we study the effectiveness of the upper bound of score of the unchecked audio streams. Figure 17 shows the results. As we can see from the results, our proposed upper bound is very effective, which makes the query time steady when the number of audio streams in index increases.

E. *Evaluation on the query results*

Here we attempt to evaluate the quality of the query results, which is difficult to conduct due to the lack of a public benchmark. We use a compromised evaluation as follows. We manually collected 100 keyword queries, and collected their top-40 query results retrieved from RTSI. Then, we asked 20 students to give the satisfaction score to 10 query



(a) varying the # of existing audios (b) varying the weight of freshness

Fig. 17. Bound: varying the # of existing audios and weight of freshness

results, which means that every query result was scored by two students. The students gave score to the query based on (i) whether the results are relevant to the query, and (ii) whether live audio streams are retrieved. Finally, we compute the average satisfaction score for the 100 queries. The average score is about 85 out of 100, which we think is reasonably good. This initial evaluation demonstrates the reasonable quality of RTSI.

F. Summary of experiments

The key findings are summarized as follows. First, RTSI outperforms LSII (cf. Section V-A) in initialization, insertion, update and query, and RTSI is also more memory effecient than LSII, thanks to RTSI's avoidance of storing all the audio information in a hash table. Second, compared with LSII, RTSI tends to be less sensitive to the variables such as the ratio of the LSM-tree and the number of audio streams. RTSI is inclined to perform well in various settings. This property is intriguing, because given a live audio stream indexing problem, the best settings for the problem are unknown beforehand. The other techniques such as Huffman coding and pruning are quite effective for improving the overall performance of RTSI. Finally, RTSI can return quality query results.

VI. Conclusion and future work

Live audio streams have been becoming increasingly popular in recent years. Providing real-time search on the live audio streams has been a challenging problem. Related work on audio indexing and search either oversimplifies the problem or simply ignores searching live audio streams. In this paper, we have leveraged recent successful technologies from machine learning (e.g., speech recognition) and databases (e.g., LSM-tree indexing) to address the problem of the emerging application of live audio streaming. We have proposed a multi-modal and unified index called RTSI to enable live audio stream search. RTSI addresses the challenges of audio indexing and search, such as an audio stream appearing in multiple inverted indices and responding queries in real-time while handling massive insertions. We have conducted extensive experiments to study the efficiency and effectiveness of RTSI. The experimental results show that our RTSI index can answer a large number of queries in a real-time manner while handling a large number of insertions from live audio streams. The top-k results obtained by RTSI are high quality. This study demonstrates that the success of machine learning (in speech recognition) and indexing techniques enables real-time and multi-modal search on the live audio stream applications. The proposed techniques are able to improve the timeliness and user experience of live audio stream applications.

We plan to extend this work in the following three directions. First, we will develop a demonstration with a user friendly interface, such that general public audiences can understand the various functionalities of our RTSI index. Second, we will evaluate our techniques on more real-world datasets such as audio streams from Facebook Live Audio or Lizhi. Third, we plan to develop a benchmark of the audio streams for other researchers to compare different live audio search techniques.

Acknowledgement

This work is supported by a MoE AcRF Tier 1 grant (T1 251RES1610) in Singapore.

References

[1] http://mixlr.com/, "Mixlr: broadcasting live audio made simple."
[2] http://www.ximalaya.com/, "Ximalaya: enabling users to share audio and personal radio stations."
[3] http://www.lizhi.fm/, "Lizhi FM: a Chinese podcast platform."
[4] http://www.jianshu.com/p/4451fe576637 and http://www.jianshu.com/p/56799862074d, "Analysis to the live audio streaming services (in Chinese)."
[5] P. Yu and F. T. B. Seide, "A hybrid word/phoneme-based approach for improved vocabulary-independent search in spontaneous speech," in *Eighth International Conference on Spoken Language Processing*, 2004.
[6] D. S. Blancas and J. Janer, "Sound retrieval from voice imitation queries in collaborative databases," in *Audio Engineering Society Conference: 53rd International Conference: Semantic Audio*. Audio Engineering Society, 2014.
[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[8] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.
[9] R. Shadiev and Y.-M. Huang, "Facilitating cross-cultural understanding with learning activities supported by speech-to-text recognition and computer-aided translation," *Computers & Education*, vol. 98, pp. 130–141, 2016.
[10] R. Shadiev, W.-Y. Hwang, N.-S. Chen, and H. Yueh-Min, "Review of speech-to-text recognition technology for enhancing learning," *Journal of Educational Technology & Society*, vol. 17, no. 4, p. 65, 2014.
[11] M. Levy and M. Sandler, "Music information retrieval using social tags and audio," *IEEE Transactions on Multimedia*, vol. 11, no. 3, pp. 383–395, 2009.
[12] G. Richard, S. Sundaram, and S. Narayanan, "An overview on perceptually motivated audio indexing and classification," *Proceedings of the IEEE*, vol. 101, no. 9, pp. 1939–1954, 2013.
[13] B. Logan, P. Moreno, J.-M. v. Thong, and E. Whittaker, "An experimental study of an audio indexing system for the web," in *Sixth International Conference on Spoken Language Processing*, 2000.
[14] A. Wang *et al.*, "An industrial strength audio search algorithm." in *Ismir*, vol. 2003. Washington, DC, 2003, pp. 7–13.
[15] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[16] F. Zheng, G. Zhang, and Z. Song, "Comparison of different implementations of mfcc," *Journal of Computer Science and Technology*, vol. 16, no. 6, pp. 582–589, 2001.
[17] D. E. Knuth, "Dynamic huffman coding," *Journal of algorithms*, vol. 6, no. 2, pp. 163–180, 1985.
[18] J. C. Tang, G. Venolia, and K. M. Inkpen, "Meerkat and periscope: I stream, you stream, apps stream for live streams," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 2016, pp. 4770–4780.

[19] M. Z. Rafique, T. Van Goethem, W. Joosen, C. Huygens, and N. Nikiforakis, "It's free for a reason: Exploring the ecosystem of free live streaming services," in *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS 2016)*. Internet Society, 2016, pp. 1–15.

[20] H. Nagano, R. Mukai, T. Kurozumi, and K. Kashino, "A fast audio search method based on skipping irrelevant signals by similarity upper-bound calculation," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE, 2015, pp. 2324–2328.

[21] O. Siohan and M. Bacchiani, "Fast vocabulary-independent audio search using path-based graph indexing," in *Ninth European Conference on Speech Communication and Technology*, 2005.

[22] V. Gupta, J. Ajmera, A. Kumar, and A. Verma, "A language independent approach to audio search," in *Twelfth Annual Conference of the International Speech Communication Association*, 2011.

[23] W. Liu, T. Mei, and Y. Zhang, "Instant mobile video search with layered audio-video indexing and progressive transmission," *IEEE Transactions on Multimedia*, vol. 16, no. 8, pp. 2242–2255, 2014.

[24] J. Haitsma and T. Kalker, "A highly robust audio fingerprinting system." in *Ismir*, vol. 2002, 2002, pp. 107–115.

[25] R. Typke, F. Wiering, and R. C. Veltkamp, "A survey of music information retrieval systems," in *Proc. 6th International Conference on Music Information Retrieval*. Queen Mary, University of London, 2005, pp. 153–160.

[26] F. Kurth, A. Ribbrock, and M. Clausen, "Efficient fault tolerant search techniques for full-text audio retrieval," in *Audio Engineering Society Convention 112*. Audio Engineering Society, 2002.

[27] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, 2013, pp. 6645–6649.

[28] H. Schütze, "Introduction to information retrieval," in *Proceedings of the international communication of association for computing machinery conference*, 2008.

[29] V. N. Anh and A. Moffat, "Inverted index compression using word-aligned binary codes," *Information Retrieval*, vol. 8, no. 1, pp. 151–166, 2005.

[30] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-time search at twitter," in *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 2012, pp. 1360–1369.

[31] R. Sears and R. Ramakrishnan, "blsm: a general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 217–228.

[32] L. Wu, W. Lin, X. Xiao, and Y. Xu, "Lsii: An indexing structure for exact real-time search on microblogs," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 2013, pp. 482–493.

[33] A. Magdy, L. Alarabi, S. Al-Harthi, M. Musleh, T. M. Ghanem, S. Ghani, and M. F. Mokbel, "Taghreed: a system for querying, analyzing, and visualizing geotagged microblogs," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2014, pp. 163–172.

[34] A. Magdy, R. Alghamdi, and M. F. Mokbel, "On main-memory flushing in microblogs data management systems," in *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 2016, pp. 445–456.

[35] J. Wang, Y. Zhang, Y. Gao, and C. Xing, "plsm: A highly efficient lsm-tree index supporting real-time big data analysis," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE, 2013, pp. 240–245.

[36] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, 2003, pp. 133–142.

A. RTSI merge time complexity

For the ease of presentation, suppose the ratio of the LSM-tree is 2. Assume we build $(l+1)$ inverted indices in the LSM-tree, which means we have I_l, I_{l-1}, $\ldots$, I_0. Let us deduce the total number of merge processes triggered during the creation of the $(l+1)$ inverted indices. The construction of I_l needs two I_{l-1} to merge, so the number of merge process occurs is 2^0. The construction of two I_{l-1} needs to merge four I_{l-2}, and hence the number of merge processes occurs is 2^1. Therefore, we can calculate the total number of merge processes as follows.

$$\begin{aligned}\text{\# of merge processes} &= 1 + 2^1 + 2^2 + \cdots + 2^{l-1} \\ &= 2^l - 1\end{aligned}$$

Suppose I_0 has m_0 number of unique terms, and each term has an average length of $\bar{n}$. Then the size of I_0 is $(m_0\bar{n})$. The size of I_1 is $(2^1 \times m_0\bar{n})$, the size of I_2 is $(2^2 \times m_0\bar{n})$, ..., the size of I_{l-1} is $(2^{l-1} \times m_0\bar{n})$.

Now, let us calculate the average index size of a merge process. When we merge two I_{l-1} inverted indices to get I_l, which only happens once because we only create one I_l during building the LSM-tree. The index size of the merge process is $2 \times (2^{l-1} \times m_0\bar{n})$. Since we have created two I_{l-1} inverted indices (which are later merged to obtain I_l), we need to merge four I_{l-2}. So, the merging size is $2^2 \times (2^{l-2} \times m_0\bar{n})$. From this pattern, we know that we need to create 2^{l-1} number of I_1 inverted indices, the size of merging I_0 to obtain I_1 is $(2^1 \times m_0\bar{n})$, and hence the total size of indices to merge for I_l creation is $2^{l-1} \times (2^1 \times m_0\bar{n})$. So we can deduce the average merging size by the following equation.

$$\begin{aligned}&= \frac{2 \times (2^{l-1} \times m_0\bar{n}) + \cdots + 2^{l-1} \times (2^1 \times m_0\bar{n})}{1 + 2^1 + 2^2 + \cdots + 2^{l-1}} \\ &= \frac{l \times 2^l m_0\bar{n}}{2^l - 1} \\ &\approx l \times m_0\bar{n} = m_0\bar{n}\log_2\frac{M}{m_0}\end{aligned}$$

To generalize the result above to the ratio ρ instead of 2, we get $m_0\bar{n}\log_\rho\frac{M}{m_0}$. Note that the number of merge operations is much smaller than the number of the number of insertions (i.e., $\log_\rho\frac{M}{m_0} \ll M\bar{n}$), and therefore the insertion cost is approximately $\mathcal{O}(\log m_0)$.

B. RTSI query time complexity analysis

For the ease of presentation, suppose the ratio of the LSM-tree is 2. As the best case analysis is simple, we omit the details of the best case here and focus on the worst case time complexity. We know the size of I_0 is m_0 (i.e., m_0 number of terms). Then the size of I_1, I_2, ..., I_l are $2^1 \times m_0$, $2^2 \times m_0$, ..., $2^l \times m_0$, respectively. So, the cost of search cost on the inverted indices is $\mathcal{O}(\log m_0 + \log(2^1 \times m_0) + \cdots + \log(2^l \times m_0)) = \mathcal{O}(l\log m_0 + \frac{l^2}{2})$. Each term has an inverted list of $\bar{n}$, and therefore the worst case time complexity of query is $\mathcal{O}(l\bar{n}\log m_0 + \bar{n}l^2)$. In practice, l is small thanks to the log structured property. The average case time complexity is $\mathcal{O}(\bar{n}\log m_0)$.