A Novel Hash-Based Streaming Scheme for Energy Efficient Full-Text Search in Wireless Data Broadcast*

Kai Yang[1], Yan Shi[1] Weili Wu[1], Xiaofeng Gao[2], and Jiaofei Zhong[1]

[1] The University of Texas at Dallas, Richardson TX 75080, USA
{kxy080020,yanshi,weiliwu,fayzhong}@utdallas.edu
[2] Georgia Gwinnett College, Lawrenceville, GA 30043
xgao@ggc.edu

Abstract. *Full-Text Search* is one of the most important and popular query types in document retrieval systems. With the development of The Fourth Generation Wireless Network (4G), *wireless data broadcast* has gained a lot of interest because of its scalability, flexibility, and energy efficiencies for wireless mobile computing. How to apply full-text search to documents transmitted through wireless communications is thus a research topic of interest. In this paper, we propose a novel data streaming scheme (named *Basic-Hash*) with hash-based indexing and inverted list techniques to facilitate energy and latency efficient full-text search in wireless data broadcast. We are the first work utilizing hash technology for this problem, which takes much less access latency and tuning time comparing to the previous literature. We further extend the proposed scheme by merging the hashed word indices in order to reduce the total access latency (named *Merged-Hash*). An information retrieval protocol is developed to cope with these two schemes. The performances of *Basic-Hash* and *Merged-Hash* are examined both theoretically and empirically. Simulation results prove their efficiencies with respect to both energy consumption and access latency.

1 Introduction

Full-text search is a popular query type that is widely used in document retrieval systems. Many commercial database systems have included full-text search as their features. For example, SQL Server 2008 provides full-text queries against character-based data in SQL Server tables [1]. Oracle Text [3] also gives powerful support for full-text search applications. Many full-text search techniques in different application areas have been proposed in literatures [2][5][6][11].

With the rapid development of mobile devices and quick rise of The Fourth Generation Wireless Network (4G), mobile communication has gained popularity due to its flexibility and convenience. Limited bandwidth in wireless communication and limited energy supply for mobile devices are the two major concerns of mobile computing. That is why *wireless data broadcast* becomes an attractive

* This work is supported by NSF grant CCF-0829993 and CCF-0514796.

data dissemination technique for mobile communication. In a wireless data broadcast system, Base Stations (BS) broadcast public information to all mobile devices within their transmission range through broadcast channels. Mobile clients listen to the channels and retrieve information of their interest directly when they arrive. This scheme is bandwidth efficient because it utilizes most of the bandwidth as downlink and requires little uplink traffics. It is also energy efficient because receiving data costs much less energy than sending data.

Mobile devices usually have two modes: *active mode* and *doze mode.* In active mode, a device can listen, compare, and download the required data; while in doze mode, it turns off antennas and many processes to save energy. The energy consumed in active mode can be 100 times of that in doze mode [15]. In general, there are two major performance criteria for a wireless data broadcast system: *access latency* and *tuning time.* Access latency refers to the time interval between a client first tunes in the broadcast channel and it finally retrieves the data of interest, which reflects the system's time efficiency; tuning time is the total time a client remains in active mode, which indicates the system's energy efficiency.

How to apply full-text search in wireless data broadcast is an interesting but challenging topic. Since data broadcast is especially suitable for public information such as news report and traffic information, full-text search can be a very useful feature desired by mobile clients. For example, a mobile user may want to browse all news related to "FIFA", or all local traffic information that includes "accidents". Full-text search for traditional disk-storage data has been well studies [9][12][4]. However, in wireless data broadcast, the data are stored "on the air" rather than on the disk, which posts new challenges to full-text search. In disk-based storage, documents are stored in physical space, so clients can "jump" among different storage slots with little cost; while in on-air storage, documents are stored sequentially along the time line, which posts much more cost for clients to search back and forth. Traditional full-text search techniques cannot not be adopted directly because of this difference. On the other hand, since existing index techniques for wireless data broadcast [10][17][8][19][20] are mainly based on predefined structured data with key attributes, they also cannot be directly applied for full-text search which uses arbitrary words as search keys. Therefore, new design of indexing schemes are needed to facilitate full-text search in order to ensure both time efficiency and energy efficiency.

To the best of our knowledge, [7] is the only published research on full-text query processing in wireless data broadcast. They firstly utilized *inverted list* in processing full-text queries on a wireless broadcast stream, and then proposed two methods: *Inverted-List* and *Inverted-List + Index-Tree* which was extended to $(1, \alpha)$ and $(1, \alpha(1, \beta))$. They made use of an inverted list to guide full-text search and a tree-based index to locate the key word in the inverted list. However, this method is not energy efficient enough because it might take a long tuning time to locate the key word in the inverted list. It is also not latency efficient enough due to the duplication of tree-based index.

Inverted list is a mature indexing method for full-text search [18][22][14]. It is a set of word indices which guides clients to find specific documents containing a specific word. In this paper, we apply inverted list as a guide for full-text search, but implement hash function instead of searching tree as indexing method, to avoid lengthening broadcast cycle and redundant tuning time for locating target word index. Note that hash function is used to index "word indices" in an inverted list, which is the "index of indices". So the index designed in this paper is a hierarchical index scheme with two levels: (1) inverted list, the index for documents, and (2) hash function, the index for word indices in an inverted list.

Compared with tree-based indexing technique, hash-based indexing for word indices is more flexible and space efficient for full-text search in wireless data broadcast. A hash function only takes several bytes while a searching tree may take thousands of bytes depending on its design. Hash-based index is more suitable for full-text search because the nature of full-text search uses arbitrary words as search keys. Based on this idea, we propose a novel data streaming scheme named *Basic-Hash* to allocate inverted list and documents on the broadcast channel. *Basic-Hash* is further improved to another streaming scheme named *Merged-Hash*, by merging the hashed word indices to reduce access latency. A client retrieval protocol is also developed corresponding to the two schemes. We are the first work utilizing hash technology for full-text search in wireless data broadcast. We also provide detailed theoretical analysis to evaluate the performance of *Basic-Hash* and *Merged-Hash*, and then implement many numerical experiments. Simulation results prove the efficiency of these two schemes with respect to both energy and access latency.

To summarize, our main contributions include:

1. We are the first work implement inverted list and hash function for full-text search in wireless data broadcast. We propose two novel wireless broadcast streaming schemes, namely, *Basic-Hash* and *Merged-Hash*, to facilitate full-text query on broadcast documents. For each scheme, we develop algorithms for inverted index allocation, document allocation and query protocol.
2. We discuss how to turn collision issues of hash functions into advantage and utilize appropriate collisions to reduce the access latency of full-text query.
3. We analyze the performances of *Basic-Hash* and *Merged-Hash* theoretically by computing the expected access latency and tuning time for full-text queres on broadcast streams created based on these two schemes.
4. We implement simulations for the proposed systems and analyze their performances by simulation results.

The rest of the paper is organized as follows: Sec. 2 presents related works on wireless data broadcast, full-text search involving inverted list techniques, and recent research on full-text search for wireless data broadcast systems; Sec. 3 introduces the system model and some preliminaries; Sec. 4 first discusses the *Basic-Hash* broadcast streaming scheme to facilitate full-text search and then extends *Basic-Hash* to *Merged-Hash* to improve the performance; Sec. 5 theoretically analyzes the performances of *Basic-Hash* and *Merged-Hash*; Sec. 6 empirically analyzes *Basic-Hash* and *Merged-Hash* based on simulation results; and Sec. 7 concludes the paper and proposes future research directions.

2 Related Work

Wireless data broadcast has gained many attentions during the past few years. Imielinski et al. first gave an overview on wireless data broadcast systems in [10]. They also proposed a popular *B^+-tree based distributed index* to achieve energy efficiency. Many different index methods were proposed thereafter. Yao et al. [17] proposed an *exponential index* which has a linear but distributed structure to enhance error-resilience. In [21], the trade-off between confidentiality and performance of *signature-based index* was discussed. Hash-based index for wireless data broadcast was also proposed in [16]. All these index techniques, however, focus only on structured data with predefined key attributes. They cannot be applied directly to guide full-text queries.

Inverted list is a popular structure in document retrieval systems and a well-known technique for full-text search. Tomasic et al. [14] studied the incremental updates of inverted lists by dual-structure index. Scholer et al. [13] discussed the compression of inverted lists of document postings which contains the position and frequency of indexed terms and developed two approaches to improve the document retrieval efficiency. Zobel et al. gave a survey on inverted files for text search engines in [22]. Zhang et al. [18] studied how to process queries efficiently in distributed web search engines with optimized inverted list assignment. Most research works on inverted list are based on disk-storage documents. For on-air documents, modifications are needed to adjust to on-air storage features.

Chung et al. [7] firstly applied inverted list for full-text search in wireless data broadcast. They also combined tree-based indexing technique with inverted list for full-text query on broadcast documents. However, the construction of a searching tree and the duplication of inverted list will extend the total length of a broadcast cycle heavily, resulting additional access latency. Moreover, the average search time for a searching tree heavily relies on the depth of the tree, which is much more than the searching time of a hash function. Therefore, we replace the search tree with hash function design and construct a more efficient data streaming scheme for full-text search in wireless data broadcast.

3 Preliminary and System Model

3.1 System Model

For simplification, we only discuss the situation for one Base Station (BS) with one communication channel. The broadcast program will not update during a period of time. The BS will broadcast several documents periodically in cycle. Each document only repeats once in a broadcast cycle. Let D denote a set of t documents to be broadcast. $D = \{doc_0, doc_1, \cdots, doc_{t-1}\}$. Each doc_i will be broadcast as several *buckets* on a channel, each with different size. Here bucket is the smallest logical unit on a broadcast channel. Assume y_i is the size of doc_i, measured by buckets, and $Y = \{y_0, y_1, \cdots, y_{t-1}\}$.

There are altogether v non-duplicated words in D, denoted as $K = \{k_0, k_1, \cdots, k_{v-1}\}$. Let w be the length of each word measured in bytes (here we assume on average, each word has the same length).

Besides documents, we also need to insert indices to form a full broadcast cycle. As mentioned in Sec. 1, we will apply inverted list and hash function together as a searching method. A hash function will be appended to each of the bucket, while the inverted list will be split into word indices and interleave with document buckets. After all the process, we will form a whole broadcast cycle, consisting of a sequence of broadcast buckets. Each bucket will have a continuous sequence number starting from 0. Let *bcycle* denote this bucket sequence, and $|bcycle|$ denote the whole number of buckets in one broadcast cycle.

3.2 Inverted List

To facilitate full-text search in wireless data broadcast systems, we apply inverted list technique. For full-text search, each word can be related to several documents and each document contains usually more than one word. To resolve such many-to-many relationship between documents and words, inverted list has been popularly used as an index in data retrieval systems [18][22][14].

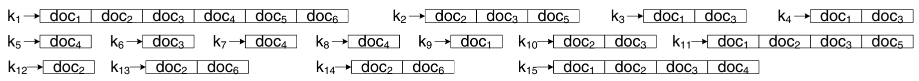

Fig. 1. Example of an Inverted List for Broadcast System

Let I be an inverted list composed of v entries representing v non-duplicated words in D. In each entry, a word k_i is linked with a set of document address pointers which can guide clients to find documents containing this word. We name each entry a *word index*, denoted as e_i. The number of pointers in each word index depends on how many documents contain this word and is not necessarily the same. Fig. 1 is an example of an inverted list generated from 6 documents. Each pointer indicates the time offset from the index to the target document. Clients can tune off during this offset, and tune on again to save some energy.

Instead of treating an inverted list as a whole, we split it into a set of word indices, each of which is a pair of a word and a list of offsets that points to documents containing the word, that is, $I = \{e_i, i = 0, \cdots, v-1\}$ where $e_i =< k_i : doc\_addr\_offset\_list_i >$.

For each word index $e_i \in I$, s_i denotes its number of document address offsets. Assume a is the length of one document address pointer measured in bytes, then the length of e_i can be easily computed as $w + as_i$.

3.3 Hash Function and Collisions

There are many hash functions that can hash strings into integers. In this paper, only hashing a word into an integer is not enough. We need to hash a word into a bucket on a broadcast cycle. If $|bcycle|$ denotes the length of a broadcast cycle, we should map the integer result to a sequence number between 0 and $|bcycle|$-1.

Collisions can be quantified by *collision rate* γ. In this paper, a collision happens when a word index is hashed to the same bucket as some previously hashed word indices were. In this case, the collision rate $\gamma =$ total number of collisions$/v$. In most hashing applications, collisions are what designers try to avoid because it increases the average cost of lookup operations. However, it is inevitable whenever mapping a large set of data into a relatively small range. While most hash function applications struggle for collision issues, our method is much more collision-tolerant. In fact, appropriate collisions are even beneficial. This is because in our method, hash scheme is only used to allocate the word indices in the inverted list and usually the sizes of most word indices are not exactly the size of a bucket. Therefore, more than one word indices hashed into the same bucket may help increasing the bucket utilization factor, which will reduce the length of a broadcast cycle and the average access latency of query processing. We will do a more detailed discussion on the collision issue in Subsec. 4.1.

3.4 Data Structure of a Bucket

A bucket is composed of two parts: *header* and *payload*. Header records basic information of a bucket such as bucket id and sequence number, while payload is the part of a bucket to store data. In our model, there are two different types of buckets: *index bucket* which stores word indices, and *document bucket* which stores documents. Index and document buckets have the same length and header structure. Hence, we can use the number of buckets to measure both Y and $|bcycle|$. If the total number of index buckets within one broadcast cycle is $|IB|$ and the total number of document buckets is $|DB|$, then we should have:

$$|bcycle| = |DB| + |IB| \, . \tag{1}$$

Next, we will illustrate the detailed design of index and document buckets. Assume a bucket is capable of carrying l bytes information. For both index and document buckets, the header contains the following information in Tab. 1.

Table 1. Information in a Bucket Header

Item	Description
TYPE:	whether the bucket is an index or document bucket.
LEFT:	length of unused space, measured in bytes.
END:	whether the index or document ends.
MERGEP:	time offset of merged index bucket.
SQ:	sequence number of a bucket. In this method, SQ is also the hash value.
OFFSET:	distance to the next bucket containing the same document.
HASH:	hash function to compute sequence number of target index bucket.

For a document bucket, the payload is part of a document or a complete document, depending on the document size. For an index bucket, the payload may contain a part of a long word index or several short word indices. Fig. 2 illustrates the whole view of a broadcast cycle and details of index and document buckets. Here grey block **D** denotes document buckets, while white block **I** denotes an index bucket. Each document doc_i has y_i document buckets, but they may be separated by some index bucket, not necessarily consecutively broadcasted.

Table 2. Symbol Description

Sym	Description	Sym	Description
D	document set. $D = \{doc_0, \cdots, doc_{t-1}\}$	a	document pointer size.
Y	document length. $Y = \{y_0, \cdots, y_{t-1}\}$	l	bucket size.
K	word set. $K = \{k_0, \cdots, k_{v-1}\}$	t	number of documents.
I	an inverted list. $I = \{e_0, \cdots, e_{v-1}\}$	v	number of keywords.
S	$S = \{s_0, \cdots, s_{v-1}\}$, where s_i is the number of document pointers in e_i.	w	word size.
		γ	collision rate.
$Avg(S)$	average number of document pointers in each word index.	δ	merge rate.
		$bcycle$	one broadcast cycle.
$\|IB\|$	number of index buckets.	$\|bcycle\|$	length of $bcycle$.
$\|DB\|$	number of document buckets.	$\|ibcycle\|$	initial broadcast cycle length.
$\|MIB\|$	number of index buckets after merging.	$Avg(AL)$	average access latency.
$\|AKE\|$	average length of the keyword entry.	$Avg(TT)$	average tuning time.

For convenience, Tab. 2 lists all symbols used in this paper. Some will be defined in the following sections.

4 Hash-Based Full-Text Search Methods

In this section, we will introduce the construction of two data streaming schemes for full-text search, which are *Basic-Hash* and *Merged-Hash*. Data streaming scheme is a preprocessing before documents are broadcasted on channels. It will interleave documents and inverted list as a whole data stream, allocate index buckets and data buckets according to the predefined hash function, and then setup corresponding address pointer and other information for clients to search words of interest and retrieve target documents.

In the following subsections, we will discuss the construction of Basic-Hash and Merged-Hash, with detailed algorithm description, examples, and scenario discussion. Finally, we propose an information retrieval protocol for mobile/ wireless clients to retrieve their interest documents.

4.1 Basic-Hash Data Streaming Scheme

Full-text query processing can be achieved by adding the inverted list onto the broadcast channel. If we put the inverted list directly in front of the documents, the average tuning time can be dramatically long because the client needs to go through the inverted list one by one to find the word index of interest. This tuning time overhead can be reduced by a two-level index scheme which adds another level of index for the inverted list.

In this subsection, we propose a novel two-level index scheme for full-text search called *Basic-Hash* method. The idea is to hash all word indices in the inverted list onto the broadcast channel. The documents and word indices are interleaved with each other. We choose hashing rather than tree-based indexing as the index for the inverted list because it is faster and doesn't occupy much space. Once a client tunes in the broadcast channel, it reads the hash function in the header of a bucket and computes the offset to the target word index immediately. The tuning time to reach the word index of interest is only 2 in the ideal case. Hash function also takes little space from the header of each bucket, without occupying extra index buckets.

It takes three steps to construct a *bcycle* using *Basic-Hash* broadcast scheme:

Step 1: *Index Allocation.* Hash all word indices onto broadcast channel (Alg. 1);
Step 2: *Document Allocation.* Fill empty buckets with documents (Alg. 2).
Step 3: *Pointer setup.* Set offset information for pointers in word indices and document/index buckets.

Hash Function. Before index allocation and document allocation, we need to construct our hash function. Recall that there are t documents to broadcast, each with y_i buckets. I is split into v entries, each with a document address offset list of size s_i. Each word has w bytes, each document address pointer is a bytes, and each bucket contains l bytes (we ignore the length of the header). Initially, we do not know $|bcycle|$, so we will use an estimated $|ibcycle|$ to represent the length of a broadcast cycle. It is easy to know, $|ibcycle| = \sum_{i=0}^{t-1} y_i + \frac{1}{l}\{a \sum_{j=0}^{v-1} s_j + v \times w\}$. Then the hash function should be:

$$Hash(string) = hashCode(string) \mod |ibcycle|.$$

In the above equation, the input string will be a word (string), while the output is an integer between 0 and $|ibcycle| - 1$. This function maps a word index to a broadcast bucket with sequence number equal to hashed value.

Index Allocation. Algorithm 1 describes **Step 1** of Basic-Hash: hashing all word indices onto broadcast channel. Let **A** denote the bucket sequence array. Initially, **A** contains $|ibcycle|$ empty bucket with consecutive sequence number starting from 0. We use $A[0], \cdots, A[|ibcycle| - 1]$ to represent each bucket. The main idea of Alg. 1 is: firstly, sort I by s_i, $i = 0, \cdots, v-1$ in increasing order as $I' = \{e'_0, \cdots, e'_{v-1}\}$. Next, hash each e'_i onto the channel in order.

The sorting process guarantees that during allocation, if more than one word index are hashed to the same bucket, the shorter word index will be assigned to the bucket first and longer word indices will be appended thereafter, which helps reducing the average tuning time for a client to find the word index of interest.

Since each bucket has l bytes capability, it may include more than one word index. Once encountering a collision, we will append e'_i right after the existing index. If this bucket is full, then find the next available bucket right forward and insert e'_i. It is also possible that there is no enough space for e'_i. In this situation, we will push other word indices in buckets with higher sequence number, and "insert" e'_i. An example is shown in Fig. 3.

Fig. 3 illuminates several scenarios for index allocation. In (a), word index e_5 should be insert into bucket $A[9]$. Since $A[9]$ is empty, we directly insert e_5 into it. If the size of e_5 is larger than $A[9]$, the rest part will be appended to $A[10]$ or more buckets. In (b), e_2, e_3, and e_5 have been allocated already, and we are going to insert e_7 to $A[7]$. Since e_2 is already allocated at $A[7]$, and there is still enough space left in $A[7]$, we append e_7 after e_2. In (c), e_1 should be inserted

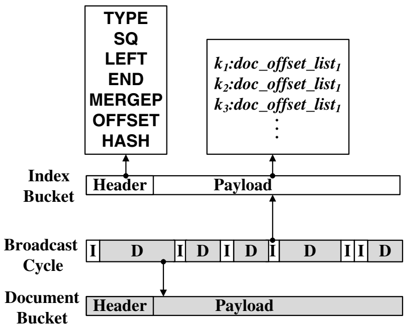

Fig. 2. Whole View of *bcycle*

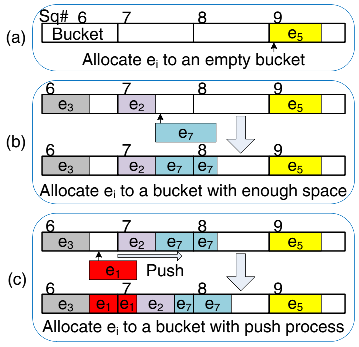
(a)
Sq# 6 7 8 9
Bucket
e5
Allocate ei to an empty bucket
(b)
e3 e2 e5
e7
e3 e2 e7 e7 e5
Allocate ei to a bucket with enough space
(c)
e3 e2 e7 e7 e5
e1 Push
e3 e1 e1 e2 e7 e7 e5
Allocate ei to a bucket with push process

Fig. 3. Index Allocation Scenario Analysis

Algorithm 1. Index Allocation For *Basic-Hash*

```
input: $I$, $\mathbf{A}$;
output: $\mathbf{A}$ filled with word indices;
1: sort $I$ by $s_i$ increasingly to $I' = \{e'_0, \cdots, e'_v\}$;
2: for $i = 0$ to $v - 1$ do
3:     $sq = Hash(k'_i)$;
4:     check whether $A[sq]$ is full, if yes, set $sq = sq + 1$ until $A[sq]$ is not full;
5:     insert $e'_i$ into $A[sq]$, if there is not enough space, then push data from $A[sq+1]$
   forward until $e'_i$ can be successfully inserted.
6: end for
```

into $A[6]$. e_2, e_3, e_5, and e_7 has already been allocated onto the channel before inserting e_1. Note that $A[6]$ does not have enough space for e_5. Thus from $A[7]$, all the data entries should be moved forward until e_1 can fit into the channel. Since there is some unfilled space between e_7 and e_5, e_5 will not be influenced. Detailed description of index allocation is illustrated in Alg. 1.

Data Allocation. Alg. 2 discusses how to allocate documents after index allocation process. Since each bucket can be either an index bucket or a document bucket, we cannot append documents to these buckets which already contain indices. Therefore, starting from $A[0]$, we will scan each bucket in order, and insert documents from doc_0 to doc_{t-1} sequentially to the empty buckets. Each doc_i will take y_i buckets. We use doc_i^j to denote the j^{th} bucket for doc_i in short.

Pointer Setup. Besides index allocation and data allocation, we need to setup the offset (address) information for pointers inside each word index, as well as **OFFSET** in each bucket header (since each document doc_i will be split into y_i buckets, and may not be consecutively allocated, we need another pointer to figure out this information). Pointers for word indices in buckets and **OFFSET** in headers can only be setup after the index and data allocation, because we did not know the locations of documents before that. To fill **OFFSET** in headers, we can scan reversely from the last bucket of the *bcycle*, record the sequence number of each doc_i^j, and then fill the offset information.

Algorithm 2. Document Allocation for *Basic-Hash*

```
    input: $D$, $\mathbf{A}$;
    output: a complete broadcast stream $\mathbf{A}$;
1:  $sq = 0$, $j = 0$;
2:  for $i = 0$ to $t - 1$ do                                   ▷ Insert $doc_i$ onto channel
3:      while $j < y_i$ do
4:          if $A[sq]$ is empty then append $doc_i^j$ to $A[sq]$; $j = j + 1$; $sq = sq + 1$;
5:          else $sq = sq + 1$; end if
6:      end while
7:      $j = 0$;                                                ▷ Reset intermediate variable
8:  end for
```

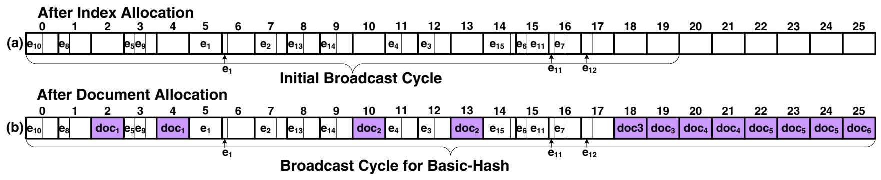

Fig. 4. Index and Document Allocation of Hash-Based Method

An Example. We apply *Basic-Hash* to the document set described in Fig. 1 as an example to demonstrate the complete streaming procedure. Assume $w = a =$ 4bytes. We can then compute the length of each word index. Assume $l =$ 24bytes, each of doc_1, doc_2, doc_3 and doc_4 takes 2 buckets, doc_5 takes 3 bucket, and doc_6 takes 1 bucket. Based on the above information, we can compute $|ibcycle| = 20$. Following Alg. 1, we first allocate all word indices onto the channel. Then, we use Alg. 2 to allocate 6 documents. The broadcast cycle after index and document allocations is presented in Fig. 4 respectively. Finally, after *bcycle* is constructed, we need to fill the necessary header and pointer information to each bucket.

4.2 Merged-Hash Data Streaming Scheme

Basic-Hash method can dramatically shorten the average tuning time of the search process than the *Inverted List* method in [7]. The average access latency, however, is much longer. The reason is that in *Inverted List* method, all the word indices are combined together, and inserted into consecutive buckets. On the other hand, *Basic-Hash* method separates them and maps each index respectively into disconnect buckets, which makes $|bcycle|$ much longer. For example, if the document number is 1000, total number of words is 500, the inverted list may only occupy 100 bucket; while a non-conflict hash function maps words in different buckets from each other, which occupies 500 buckets. The $|bcycle|$ expands from 1100 to 1500, and the average access latency is thus influenced.

Merged-Hash Algorithm. *Merged-Hash* aims at reducing average access latency by reducing the number of index buckets. Compared with *Basic-Hash*, *Merged-Hash* has one more step: *Merge Word Index*. It will be performed between

Alg. 1 and 2. The purpose is to combine adjacent index fragments into one bucket to make full use of bucket space. Merge process can reduce $|bcycle|$ without increasing average tuning time.

The idea of *Merge Word Index* algorithm is: starting from the last index bucket $A[i]$, if its closest previous index $A[j]$ can be merged into $A[i]$, then append $A[j]$ to $A[i]$, and delete $A[j]$. Repeat this process until either $A[i]$ is full or its closest previous index $A[j]$ is full or cannot be merged into $A[i]$. Next, find another $A[i]$ and repeat the above process, until all index buckets have been scanned. The detailed description is showed in Alg. 3.

Algorithm 3. Merge Word Index

```
   input: A;
   output: A with merged word indices;
1: find the last non-full index $A[i]$, set $M = A[i]$;
2: while $M$ is not full do
3:     find its closest previous index $A[j]$;
4:     if $A[j]$ is not full and $M$ has enough space for $A[j]$ then
5:             append $A[j]$ to $M$, delete $A[j]$;
6:     else $M = A[j]$, Break; end if
7: end while
8: repeat Line 2 to Line 7 until all index buckets have been scanned.
```

An Example. We also use an example shown in Fig. 5(a) to illustrate Alg. 3. Merge process begins at bucket 17 with M moving backwards. We can see that word indices in bucket 16 is merged in bucket 17 because there is enough space for them to append. And we also observe that bucket 15 does not follow the step of bucket 16, because after bucket 16 is merged to bucket 17, their is not enough space any more to append the whole indices in bucket 15. For each bucket that is merged to another bucket, MERGEP in header indicates the offset between these two buckets for clients to keep track of the index. For instance, MERGEP in bucket 16's header is 1 and MERGEP in bucket 1's header is 2. Note that the merging operation is based on bucket instead of index. Hence, it is possible that an index will be split after merging. In such cases, we also need MERGEP to direct clients. For example, the second part of e_1 in bucket 6 is merged to bucket 7, so it is separated from its first part in bucket 5. With the help of MERGEP, the tuning time to read such an index only increases by 1, while merging operation dramatically reduces the $|bcycle|$.

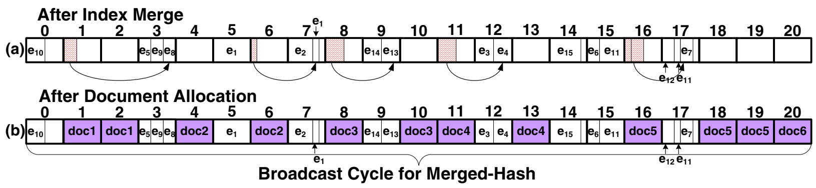
After Index Merge
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
(a)
e10 e5 e9 e8 e1 e2 e14 e13 e3 e4 e15 e6 e11 e7
e1
e12 e11
After Document Allocation
(b)
e10 doc1 doc1 e5 e9 e8 doc2 e1 doc2 e2 doc3 e14 e13 doc3 doc4 e3 e4 doc4 e15 e6 e11 doc5 e7 doc5 doc5 doc6
e1
e12 e11
Broadcast Cycle for Merged-Hash

Fig. 5. Index and Document Allocation of Merged-Based Method

After *Basic-Hash*, the broadcast channel looks as Fig. 4(a), with 13 index buckets. If we merge index buckets following Alg. 3, the number of index buckets will decrease to 9, as shown in Fig. 5(a). Comparing Fig. 4(b) and 5(b), we see that $|bcycle|$ after merging is reduced to 20, which is the same as $|ibcycle|$; while $|bcycle|$ without merging is 25, which is 20% longer than $|ibcycle|$.

4.3 Information Retrieval Protocol

After document allocation, the whole *bcycle* is built. Next, we discuss information retrieval protocol. A mobile client will firstly access onto the channel, read the current bucket and get hash function. Next, it computes a sequence number hashed from target word w, and waits until this bucket appears. Then it will follow the direction of bucket pointers to find the word index containing the information of w, and read every offset inside the *doc_offset_list*. Finally, it waits according to these time offsets and download the requested documents one by one. The detailed description of this algorithm is illustrated in Alg. 4.

Algorithm 4. Information Retrieval Protocol

```
   input: keyword $w$;
   output: a set of documents containing $w$;
1: read current bucket $cb$, get hash function $hash(\cdot)$; compute $sq = hash(w)$;
2: if $A[sq]$ is not the current bucket then wait for the $A[sq]$; end if
3: read $A[sq]$, follow index pointer to find $e_i$ with $w$.
4: read all the addresses of document containing $w$;
5: for each document offset do wait and download the document buckets; end for
```

5 Performance Analysis

In this section, we will give theoretical analysis for both *Basic-Hash* and *Merged-Hash* with respect to the average access latency and tuning time.

5.1 Analysis for Basic-Hash

For an inverted list, the average number of documents linked to a word is $Avg(S) = \sum_{i=0}^{v-1} s_i/v$. The average length of a word index $|AKE| = w + aAvg(S)$.

Theorem 1. *The average access latency of* Basic-Hash *is*

$$\left(\frac{1}{2} + \frac{Avg(S)}{Avg(S)+1}\right)\left(\sum_{i=0}^{t-1} y_i + \left\lceil \frac{|AKE|}{(1-\gamma)l} \right\rceil (1-\gamma)v\right). \tag{2}$$

Proof. In *Basic-Hash*, average access time (Avg(AL)) is the sum of *probe wait* and *bcast wait*, where *probe wait* denotes the latency of finding target word index bucket and *bcast wait* is the time needed to download all documents containing the requested word. If documents are uniformly distributed on the channel,

$$Avg(AL) = \frac{|bcycle|}{2} + \frac{Avg(S)|bcycle|}{Avg(S)+1}. \tag{3}$$

From Eqn. (1), we have

$$|bcycle| = \sum_{i=0}^{t-1} y_i + \left\lceil \frac{|AKE|}{(1-\gamma)l} \right\rceil v(1-\gamma) \tag{4}$$

Combining Eqn. (3) and (4), we can derive Eqn. (2).

Theorem 2. *The average tuning time of* Basic-Hash *is* $1 + \left\lceil \frac{|AKE|}{(1-\gamma)l} \right\rceil + \frac{Avg(S)}{t} \sum_{i=0}^{t-1} y_i$.

Proof. The average tuning time (Avg(TT)) includes time of 1) initial probing, 2) reading target index bucket and 3) downloading target documents. Initial probing takes time 1. After initial probe, the client computes the hashed value, dozes and tunes in the hashed bucket directly. The time needed to read the target index bucket is $\lceil |AKE|/((1-\gamma)l) \rceil$. On average, there are $Avg(S)$ documents containing a word, so the time needed to download these documents is $Avg(S) \sum_{i=0}^{t-1} y_i/t$. Summing the above three parts, we can get the conclusion.

5.2 Analysis for Merged-Hash

For *Merged-Hash* scheme, index buckets are merged according to Alg. 3 after word indices are hashed to the channel. We define $|MIB|$ to represent the total number of index buckets after merging, and merge rate $\delta = |MIB|/|IB|$ to indicate the effect of merging. δ is bounded between $[\lceil 1/|AKE| \rceil, 1]$.

Theorem 3. *The access latency of* Merged-Hash *is*

$$\left(\frac{1}{2} + \frac{Avg(S)}{Avg(S)+1} \right) \left(\sum_{i=0}^{t-1} y_i + \left\lceil \frac{|AKE|}{(1-\gamma)l} \right\rceil (1-\gamma)v\delta \right). \tag{5}$$

Proof. The access latency of *Merged-Hash* scheme is also computed as Eqn. (3). The difference is how to get $|bcycle|$. In *Merged-Hash*, the length of a *bcycle* is:

$$|bcycle| = |DB| + |MIB| = \sum_{i=0}^{t-1} y_i + \left\lceil \frac{|AKE|}{(1-\gamma)l} \right\rceil (1-\gamma)v\delta.$$

Theorem 4. *The tuning time of* Merged-Hash *is* $2 + \delta \left\lceil \frac{|AKE|}{(1-\gamma)l} \right\rceil + \frac{Avg(S)}{t} \sum_{i=0}^{t-1} y_i$.

Proof. Similar as *Basic-Hash*, the tuning time for *Merged-Hash* is also composed of three parts. If the word index of interest did not merge with any other index, the tuning time is exactly the same as in *Basic-Hash*. If the word index merged with other indices, it means the size of this index is smaller than an index bucket. So it takes 1 unit time to read. Therefore, the average tuning time to read word index is $(1-\delta)+\delta\lceil |AKE|/((1-\gamma)l) \rceil$. Combining with the tuning time for initial probing and document downloading, we can prove Thm. 4.

6 Simulation and Performance Evaluation

In this section, we will evaluate the *Basic-Hash* and *Merged-Hash* methods by simulation results. We also compare *Merged-Hash* method with *Inverted List* and *Inverted List + Tree Index* methods in [7]. The performance metrics used are average access latency (AAL) and average tuning time (ATT).

The simulation is implemented using Java 1.6.0 on an Intel(R) Xeon(R) E5520 computer with 6.00GB memory, with Windows 7 version 6.1 operating system. We simulate a base station with single broadcast channel, broadcasting a database of 10,000 documents with a dictionary of 5,000 distinct words. For each group of experiments, we generate 20,000 clients randomly tuning in the channel and compute the average of their access latency and tuning time.

6.1 Comparison between *Basic-Hash* and *Merged-Hash*

We use two experiments to compare the performance of *Basic-Hash* and *Merged-Hash*. In the first experiment, we vary the size of the word dictionary from 1,000 to 5,000, while the number of documents is fixed to 10,000. This simulates how similar a set of documents are. Documents with more similar topics may have more words in common, which results in a smaller dictionary. The content of each document is randomly generated from the dictionary. The repetitions of a word in a document is uniformly distributed between 1 and 5. The number of non-replicated words contained in a document is set between 1 and 50.

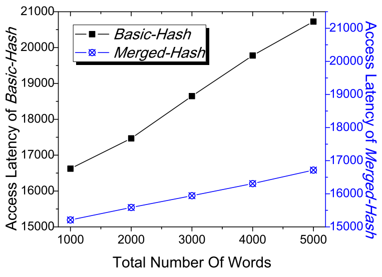

Fig. 6. AAL w.r.t. Word No.

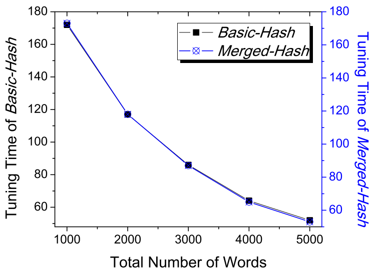

Fig. 7. ATT w.r.t. Word No.

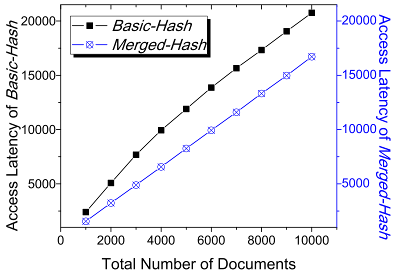

Fig. 8. AAL w.r.t. Document No.

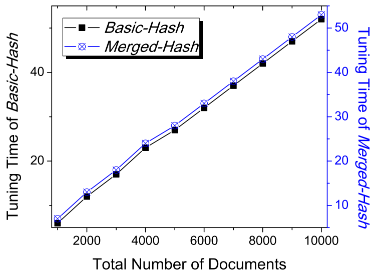

Fig. 9. ATT w.r.t. Document No.

Fig. 6 shows average latency of *Basic-Hash* and *Merged-Hash* in this setting. Obviously, *Merged-Hash* has a much shorter access latency than *Basic-Hash*. This verifies our prediction that by merging hashed indices, we can reduce $|bcycle|$ and thus reduce average access latency. In fact, when the dictionary size is 5000, $|bcycle|$ of *Basic-Hash* is 14047 while tit is only 11284 for *Merged-Hash*. When the total number of words increases, the advantage of *Merged-Hash* compared with *Basic-Hash* becomes more obvious.

Fig. 7 presents the average tuning time. We can observe that no matter how the dictionary size changes, the average tuning times of *Basic-Hash* and *Merged-Hash* are very similar with each other. This is because merging word indices do not have much impact on the time for reading a word index of interest.

The second experiment is to evaluate the influence of document set size to the performances of proposed two streaming scheme. We generate D in the same way. Then, randomly choose subsets of D to form eight smaller-sized document set ranging from 1,000 to 9,000. Fig. 8 indicates that *Merged-Hash* performs better than *Basic-Hash* with respect to average access latency no matter how large the document set size is. The difference between *Merged-Hash* and *Basic-Hash* first increases as the number of documents increases, then almost remains unchanged after the document set size reaches 6000. Similar as the first experiment, whatever document set size is, the difference between average tuning times of these two streaming schemes is negligible.

6.2 Comparison with Other Methods

In [7], the authors proposed two full-text search method: *Inverted List* method (*IL*) and *Inverted List + Tree Index* method (*IL+TI*). For a fair comparison, we set the simulation environment exactly the same as in [7]. We generate 10,000 documents, each of size 1024 bytes. The contents of documents are randomly generated from 4703 distinct words. The bucket size is 1024 bytes. The repetitions of a word in a document is 1 to 5, in a uniform way. The $Avg(S)$ is 51, which is also the same as in [7]. All results are averaged based on 20,000 clients.

Table 3. Comparison of three full-text search methods

	IL	*IL+TI*	*Merged-Hash*
average access latency	14901	16323	16312
average tuning time	916	91	54

Tab. 3 compares the average access latency and tuning time of *IL*, *IL+TI* and *Merged-Hash*. Compared with *IL*, both *IL+TI* and *Merged-Hash* can dramatically reduce average tuning time by indexing the inverted list. *Merged-Hash* costs even 40.7% less average tuning time than *IL+TI*. Therefore, *Merged-Hash* is the most energy efficient scheme among three. This verifies our analysis that hashing can speed up the searching within the inverted list and consequently reduce tuning time. The average access latency of *Merged-Hash* are slightly longer (9.5%) than *IL*. The reason is that although hashing itself does not require dedicated index bucket, hashing word indices into different buckets may not make full use of the bucket capacity. Therefore, $|MIB|$ may be larger than the number of buckets needed to fill in a complete inverted list. However, *Merged-Hash* still has very similar average access latency with *IL+TI*.

7 Conclusion

In this paper, we proposed two novel wireless data broadcast streaming schemes: *Basic-Hash* and *Merged-Hash*, which provide a two-level indexing to facilitate the full-text query processing in the wireless data broadcast environment. The proposed methods utilizing hash technique to index the inverted list of document broadcasted, which itself is an index for full-text search. For each scheme, we designed detailed index allocation and document allocation algorithms, together with a corresponding querying processing protocol. The performances of these two schemes were analyzed both theoretically and empirically. Simulation results indicate that *Merged-Hash* is the most energy-efficient streaming scheme among all broadcast schemes for full-text search in existing literatures. In the future, we plan to extend *Merged-Hash* to increase the utilization ratio of index buckets in order to further reduce the access latency and tuning time of full-text query processing. We also plan to explore how to adopt other traditional full-text search methods to the wireless broadcast environment.

References

1. `http://msdn.microsoft.com/en-us/library/ms142571.aspx`
2. Amer-Yahia, S., Shanmugasundaram, J.: Xml full-text search: challenges and opportunities. In: VLDB 2005 (2005)
3. Asplund, M.: Building full-text search applications with oracle text, `http://www.oracle.com/technology/pub/articles/asplund-textsearch.html`
4. Atlam, E.S., Ghada, E.M., Fuketa, M., Morita, K., Aoe, J.: A compact memory space of dynamic full-text search using bi-gram index. In: ISCC 2004 (2004)
5. Blair, D.C., Maron, M.E.: An evaluation of retrieval effectiveness for a full-text document-retrieval system. Commun. ACM 28(3), 289–299 (1985)
6. Brown, E.W., Callan, J.P., Croft, W.B.: Fast incremental indexing for full-text information retrieval. In: VLDB 1994, pp. 192–202 (1994)
7. Chung, Y.D., Yoo, S., Kim, M.H.: Energy- and latency-efficient processing of full-text searches on a wireless broadcast stream. IEEE Trans. on Knowl. and Data Eng. 22(2), 207–218 (2010)
8. Chung, Y.C., Lin, L., Lee, C.: Scheduling non-uniform data with expected-time constraint in wireless multi-channel environments. J. Parallel Distrib. Comput. 69(3), 247–260 (2009)
9. Faloutsos, C., Christodoulakis, S.: Signature files: an access method for documents and its analytical performance evaluation. ACM Trans. Inf. Syst. 2(4), 267–288 (1984)
10. Imielinski, T., Viswanathan, S., Badrinath, B.r.: Data on air: Organization and access. IEEE Trans. on Knowl. and Data Eng. 9(3), 353–372 (1997)
11. Kim, M.S., Whang, K.Y., Lee, J.G., Lee, M.J.: Structural optimization of a full-text n-gram index using relational normalization. The VLDB Journal 17(6), 1485–1507 (2008)

12. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Trans. Inf. Syst. 14(4), 349–379 (1996)
13. Scholer, F., Williams, H.E., Yiannis, J., Zobel, J.: Compression of inverted indexes for fast query evaluation. In: SIGIR 2002, pp. 222–229 (2002)
14. Tomasic, A., García-Molina, H., Shoens, K.: Incremental updates of inverted lists for text document retrieval. SIGMOD Rec. 23(2), 289–300 (1994)
15. Viredaz, M.A., Brakmo, L.S., Hamburgen, W.R.: Energy management on handheld devices. Queue 1(7), 44–52 (2003)
16. Xu, J., Lee, W.C., Tang, X., Gao, Q., Li, S.: An error-resilient and tunable distributed indexing scheme for wireless data broadcast. IEEE Trans. on Knowl. and Data Eng. 18(3), 392–404 (2006)
17. Yao, Y., Tang, X., Lim, E.P., Sun, A.: An energy-efficient and access latency optimized indexing scheme for wireless data broadcast. IEEE Trans. on Knowl. and Data Eng. 18(8), 1111–1124 (2006)
18. Zhang, J., Suel, T.: Optimized inverted list assignment in distributed search engine architectures. In: Parallel and Distributed Processing Symposium, International, p. 41 (2007)
19. Zhang, X., Lee, W.C., Mitra, P., Zheng, B.: Processing transitive nearest-neighbor queries in multi-channel access environments. In: EDBT 2008: Proceedings of the 11th International Conference on Extending Database Technology, pp. 452–463 (2008)
20. Zheng, B., Lee, W.C., Lee, K.C., Lee, D.L., Shao, M.: A distributed spatial index for error-prone wireless data broadcast. The VLDB Journal 18(4), 959–986 (2009)
21. Zheng, B., Lee, W.C., Liu, P., Lee, D.L., Ding, X.: Tuning on-air signatures for balancing performance and confidentiality. IEEE Trans. on Knowl. and Data Eng. 21(12), 1783–1797 (2009)
22. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. 38(2), 6 (2006)