An Error-Resilient and Tunable Distributed Indexing Scheme for Wireless Data Broadcast

Jianliang Xu, *Member*, *IEEE*, Wang-Chien Lee, *Member*, *IEEE*, Xueyan Tang, *Member*, *IEEE*, Qing Gao, and Shanping Li

Abstract—Access efficiency and energy conservation are two critical performance concerns in a wireless data broadcast system. We propose in this paper a novel parameterized index called the *exponential index* that has a linear yet distributed structure for wireless data broadcast. Based on two tuning knobs, index base and chunk size, the exponential index can be tuned to optimize the access latency with the tuning time bounded by a given limit, and vice versa. The client access algorithm for the exponential index under unreliable broadcast is described. A performance analysis of the exponential index is provided. Extensive ns-2-based simulation experiments are conducted to evaluate the performance under various link error probabilities. Simulation results show that the exponential index substantially outperforms the state-of-the-art indexes. In particular, it is more resilient to link errors and achieves more performance advantages from index caching. The results also demonstrate its great flexibility in trading access latency with tuning time.

Index Terms—Index structure, data broadcast, energy conservation, mobile computing.

1 Introduction

Wireless data broadcast has received a lot of attention from industries and academia in recent years. It has been available as commercial products for many years (e.g., StarBand [20] and Hughes Network [21]). In particular, the recent announcement of the *smart personal objects technology* (SPOT) by Microsoft [16] further highlights the industrial interest in and feasibility of utilizing broadcast for wireless data services. With a continuous broadcast network (called DirectBand Network) using FM radio subcarrier frequencies, SPOT-based devices (e.g., PDAs and watches) can continuously receive timely information such as stock quotes, airline schedules, local news, weather, and traffic information.

Access efficiency and *energy conservation* are two main performance issues for the clients in a wireless data broadcast system. Access efficiency concerns how fast a request is satisfied, and energy conservation concerns how to reduce a mobile client's energy consumption when it accesses the data of interest. While access efficiency is a constantly tackled issue in most system and database research, energy conservation is very critical due to the limited battery capacity on mobile clients [25]. To facilitate energy conservation, a mobile device typically supports two operation modes: *active mode* and *doze mode*. The device normally operates in the active mode; it can switch to the doze mode to save energy when the system becomes idle. For example, a typical wireless PC card, ORiNOCO, consumes 60 mW during the doze mode and 805-1,400 mW during the active mode [25]. In the literature, two performance metrics, namely, *access latency* and *tuning time*, have been used to measure access efficiency and energy conservation, respectively [6], [9], [10]:

- Access latency. The time elapsed between the moment when a query is issued and the moment when it is responded.
- Tuning time: The amount of time a mobile client stays *active* to receive the requested data.

To retrieve a data item from wireless data broadcast, the mobile client has to continuously monitor the broadcast until the data arrives. This will consume a lot of energy since the client has to remain *active* during its waiting time. A solution to this problem is *air indexing*. The basic idea is to include some index information about the arrival times of data items on the broadcast channel. By accessing the index, the mobile client is able to predict the arrival of its desired data. Thus, it can stay in the doze mode during waiting time and tune into the broadcast channel only when the data item of interest arrives. Several traditional disk-based indexing techniques such as B^+-tree have been extended for air indexing [4], [10], [19]. However, existing designs are mostly based on centralized tree structures, which are not performance efficient for the sequential-access broadcast media. Specifically, to start an index search, the client needs to wait until it reaches the root of the next broadcast search tree; also, in case of link errors during index search, the client

- J. Xu and Q. Gao are with the Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, KLN, Hong Kong. E-mail: {xujl, qgao}@comp.hkbu.edu.hk.
- W.-C. Lee is with the Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802. E-mail: wlee@cse.psu.edu.
- X. Tang is with the School of Computer Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 639798. E-mail: asxytang@ntu.edu.sg.
- S. Li is with the College of Computer Science, Zhejiang University, Hangzhou 310027, China. E-mail: shan@cs.zju.edu.hk.

Manuscript received 6 Jan. 2005; revised 6 June 2005; accepted 8 Sept. 2005; published online 18 Jan. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0007-0105.

has to restart the search from the next broadcast root.[1] To reduce such access delay, multiple replicated indexes are usually interleaved with data broadcast. Nevertheless, the air indexing solution has the drawback of lengthening the broadcast cycle due to the indexing information. In other words, there is a trade-off between access latency and tuning time.

Different application scenarios may require different performance trade-offs. For example, in an office environment, users may favor a short latency since they can easily recharge the batteries; while in the airport, users may prefer to conserve energy at the cost of a longer latency. As such, we need a *tunable* air indexing scheme to accommodate different requirements. A good air indexing scheme should be able to facilitate *latency bounded tuning* and *tuning-time bounded tuning*. In general, a shorter tuning time is expected when a longer latency can be tolerated, and vice versa. However, most of the existing indexing techniques are not flexible in the sense that the trade-off between tuning time and access latency is not adjustable based on application specific requirements.

In this paper, we propose a novel parameterized index called the *exponential index*. The proposed exponential index is very efficient because it naturally facilitates the index replication by sharing links in different search trees and, thus, minimizes storage overhead. Moreover, it has a linear yet distributed structure which suits the sequential-access broadcast environment very well. It not only allows searching to start at any index segment but also makes recovering an index search from a link error quickly. Based on its two tuning knobs, the exponential index can also be easily adjusted to optimize the access latency (or tuning time) with the tuning time (or access latency) bounded by a given limit.[2]

Wireless transmission is error-prone. Data might be corrupted or lost due to many factors like signal interference, etc. The client access algorithm for the exponential index under unreliable broadcast is described. We also provide a performance analysis of the exponential index in terms of the access latency and tuning time under unreliable wireless broadcast environments. Extensive experiments are conducted to compare the exponential index with two state-of-the-art air indexing schemes, i.e., the distributed tree [10] and the flexible index [9], under various link error probabilities. Simulation results show that the proposed exponential index substantially outperforms the existing indexing schemes. In particular, it is more resilient to link errors and achieves more performance advantages from index caching. The results demonstrate its great flexibility in trading access latency with tuning time.

The rest of this paper is organized as follows: Section 2 gives the background for indexing data on broadcast channels and reviews the related work. In Section 3, we introduce the proposed exponential index for clustered broadcast and explain how to tune different trade-offs between tuning time and access latency. We compare the proposed index with the existing indexes in Section 4. Section 5 extends the exponential index to nonclustered broadcast and index caching. Finally, the paper is concluded in Section 6.

1. This problem can be alleviated a little by enhancing the index with some auxiliary data structure [23].

2. Note that it is not necessary for the whole system to employ a single tuning parameter. A wireless network typically consists of a large number of cells. The users in the same cell are likely to have similar performance favors. Thus, each cell can independently employ its own tuning parameter and apply the proposed work.

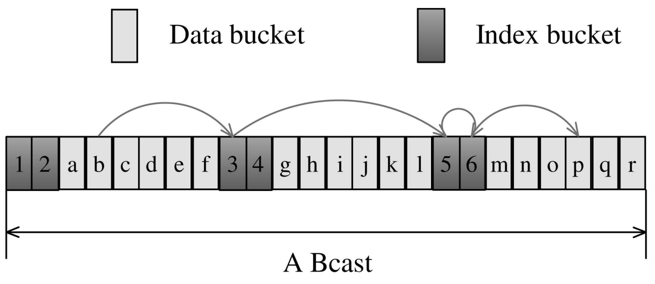


Fig. 1. Data organization on a wireless broadcast channel.

2 Background

2.1 Preliminaries

Consider a data dissemination system that periodically broadcasts a collection of data items (e.g., stock quotes) to mobile clients through a wireless broadcast channel. Each data item is a tuple of attribute values and can be identified by a key value. Similar to [10], the smallest access unit of a broadcast is referred to as a *bucket*, which physically consists of a fixed number of packets—the basic unit of message transfer in the network. Each bucket is identified by a sequentially increased id number. We distinguish between *index buckets* that hold the index and possibly some data if space permits, and *data buckets* that hold the data (one or more items) only. A sequence of multiplexed index buckets and data buckets constitute a *bcast*, in which each data item appears at least once (see Fig. 1). Bcasts can be classified as *flat broadcast*, where each item appears exactly once, and *skewed broadcast*, where some items may appear more than once. A bcast is repeatedly broadcast on the wireless channel.

To facilitate a data search via an air index, each data bucket includes an offset to the beginning of the next index bucket. Taking Fig. 1 as an example, the general access protocol for retrieving data involves the following phases:

- Initial probe. The client tunes into the broadcast channel at bucket b and determines when the next index bucket 3 is broadcast. The client tunes in again at bucket 3.
- Index search. The client selectively accesses a number of index buckets (i.e., index buckets 3, 5, and 6) to find out when to get the desired data held in bucket p.
- Data retrieval. When bucket p arrives, the client downloads it and retrieves the desired data. Thus, the access latency is 19 buckets; as the client needs to stay active for buckets b, 3, 5, 6, and p only, the tuning time is five buckets.

There are two basic data organizations with respect to an attribute within a bcast: *clustered broadcast* and *nonclustered broadcast*. A sequence of data items are *clustered* if all the data items with the same value of the attribute appear consecutively; otherwise, they are *nonclustered*. Clustered broadcast corresponds to flat broadcast with respect to the primary attribute, whereas nonclustered broadcast corresponds to flat broadcast with respect to a secondary attribute or skewed broadcast [10]. Without loss of generality, data items in clustered broadcast can be arranged in ascending order of the attribute values. For nonclustered broadcast, a bcast can be partitioned into a number of segments called *metasegments*, each of which holds a sequence of items with nondescending (or nonascending) values of the attribute [10]. Thus, when we look at each individual metasegment, the data items are clustered on that attribute and the indexing techniques developed for clustered broadcast are still applicable to a metasegment. Therefore, we mainly focus on clustered broadcast when we describe the proposed index in Section 3 and extend the technique to nonclustered broadcast in Section 5.

2.2 Related Work

Several disk-based indexing techniques have been extended for air indexing. Imielinski et al. redesigned the B^+-tree such that the leaf nodes store the arrival times of the data items [10]. A *distributed indexing* method was proposed to efficiently replicate and distribute the index tree in a bcast. Chen et al. and Shivakumar and Venkatasubramanian considered unbalanced tree structures to optimize energy consumption for nonuniform data access [4], [19]. These structures minimize the average index search cost by reducing the number of index searches for hot data at the expense of spending more on cold data. Tan and Yu discussed data and index organization under skewed broadcast [24]. Hashing and signature methods have also been suggested for wireless broadcast that supports equality queries [7], [9]. Hu et al. showed that the signature method is particularly attractive for multiattribute indexing [6]. However, none of these techniques is flexible in tuning access latency and tuning time. Moreover, as they are extended from disk-based environments, which support random access, they are not natural for broadcast environments, where only sequential access is allowed and, hence, tedious adaptation is needed.

A flexible indexing method was proposed in [9]. The *flexible index* first sorts the data items in ascending (or descending) order of the search key values and then divides them into p segments. The first bucket in each data segment contains a *control index*, which is a binary index mapping a given key to the segment containing the key, and a *local index*, which is an m-entry index mapping a given key to the bucket containing the key within the current segment. By tuning the parameters of p and m, mobile clients can achieve either a good tuning time or a good access latency. However, [9] does not make it clear how *flexibility* can be measured. As we shall see in Section 4, the flexibility of this indexing method is quite limited.

While all the aforementioned work assumed an error-free broadcast environment, Tan and Ooi [23] investigated air indexing techniques for unreliable data broadcast. They enhanced the distributed tree index and the flexible index to efficiently deal with link errors. In this paper, we generalize our previously proposed exponential index [27] to an unreliable broadcast environment where link errors may occur. The proposed exponential index differs from the flexible index in at least three aspects: 1) Instead of binary spaced indexing, the exponential index allows indexing spaces to be exponentially partitioned at any base value. 2) The exponential index intelligently exploits the available bucket space for indexing, whereas the flexible index blindly incurs overhead. 3) The exponential index allows the current bcast to index into the next bcast to support an efficient search, but the flexible index indexes the data within the current bcast only.

Other related work investigated different aspects of broadcast, including data scheduling [5], [15], semantic broadcast [12], broadcast of location-dependent data [28], consistency management [13], [17], and cache management [14], [18], [26]. There also exist studies on designing error correction codes to improve data transmission reliability [8]. Complementary to these studies, we propose error-resilient data indexing methods to facilitate data accesses even if the broadcast is not reliable.

3 The Exponential Index

This section presents a new air indexing method called the *exponential index*. We focus on clustered broadcast in this section and shall extend the proposed index to nonclustered broadcast in Section 5. We first illustrate the basic idea of the exponential index by an example and then generalize it with two tunable parameters: *index base* and *chunk size* (to be defined later). Next, we analyze the performance of the generalized exponential index. Finally, we show how to adjust the trade-off between tuning time and access latency for the exponential index.

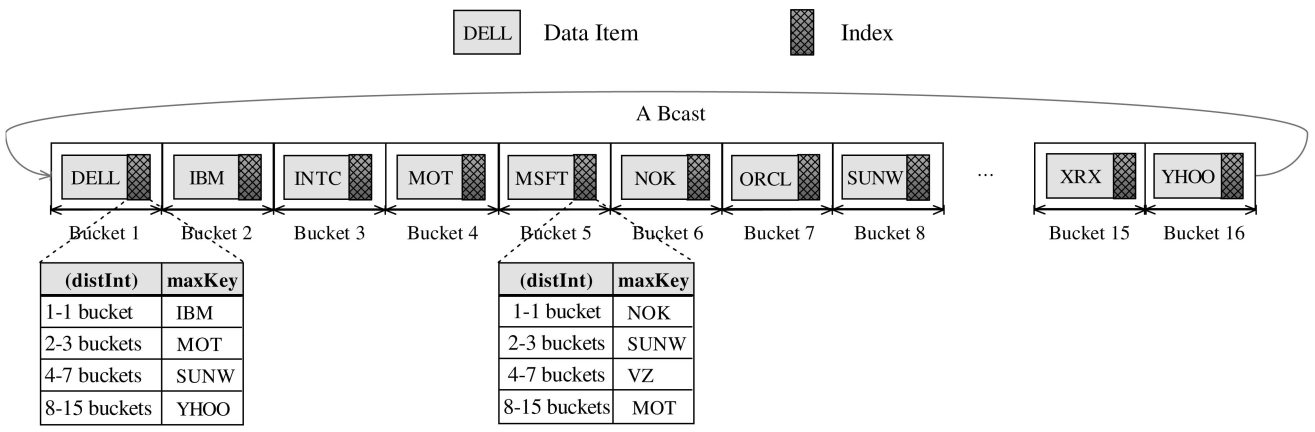
3.1 A Motivating Example

Consider a server that periodically broadcasts stock information (e.g., stock ticks, prices, trading volumes, etc). Suppose the server maintains 16 stock items that are arranged in a bcast in ascending order of their identifiers.

For simplicity, each bucket is assumed to accommodate only one stock item and some index information.[3] As shown in Fig. 2, a bcast consists of 16 buckets. Each bucket contains a data part and an *index table*. The index table consists of four *entries* (rows). Each entry indexes a *segment* of buckets in the form of a tuple $\{distInt, maxKey\}$, where $distInt$ specifies the distance range of the buckets from the current bucket (measured in the unit of *buckets*), and $maxKey$ is the maximum key value of these buckets. The sizes of the segments grow exponentially. The first entry describes a single bucket segment (i.e., the next bucket) and, for each $i > 1$, the ith entry describes the segment of buckets that are 2^{i-1} to $2^i - 1$ away (i.e., 2^{i-1} buckets). Note that the $distInt$ values need not be maintained in the index table since they can be inferred from their entry ids. The key range of the buckets indexed by the ith entry is given by the $maxKey$ values of the $(i-1)$th and ith entries.

Suppose that a client issues a query for item "NOK" right before item "DELL" (i.e., bucket 1) is broadcast. The client

3. As mentioned, a bucket may accommodate one or more data items (i.e., stock items here), depending on the bucket capacity.



(distInt)	maxKey
1-1 bucket	IBM
2-3 buckets	MOT
4-7 buckets	SUNW
8-15 buckets	YHOO

(distInt)	maxKey
1-1 bucket	NOK
2-3 buckets	SUNW
4-7 buckets	VZ
8-15 buckets	MOT

Fig. 2. A simple exponential index.

tunes into the broadcast channel and first retrieves the index table in bucket 1 (i.e., the left index table in Fig. 2). Since "NOK" falls between the second $maxKey$ "MOT" and the third $maxKey$ "SUNW," the target item must lie in the buckets that are four to seven away. The client then stays in the doze mode until bucket 5 is broadcast and examines the item in bucket 5. As the target item cannot be found in bucket 5, the client further checks the index table in bucket 5 (i.e., the right index table in Fig. 2). Since "NOK" matches the first $maxKey$, the target item must be in the next bucket. Therefore, the client completes the query by accessing bucket 6. The total tuning time for the query is three buckets (i.e., buckets 1, 5, and 6). Similarly, if a client wants to access item "SUNW" right before bucket 1 is broadcast, it can get the desired data by searching buckets 1, 5, 7, and 8. As we shall show in Section 3.3, the worst tuning time for this exponential index is $\lceil log_2(N-1)+1\rceil$ buckets when the data broadcast is error-free, where N is the total number of buckets in a bcast. In our example, where $N=16$, the worst tuning time is five buckets.

As mentioned, wireless data broadcast is unreliable. A bucket might be corrupted during broadcasting. We can handle such link errors easily with the exponential index. Again, using Fig. 2 as an example, a client searches for "NOK" from the first bucket. If the broadcast is error-free, the client accesses buckets 1, 5, and 6, as discussed above. However, if bucket 1 is corrupted, the client immediately restarts the search from the next bucket (i.e., bucket 2). Thus, the client accesses bucket 1 (corrupted) and buckets 2 and 6 to get the desired data. If both buckets 1 and 2 are corrupted, the client restarts the search from bucket 3 and accesses buckets 5 and 6. Hence, there is only a small performance penalty. This is indeed an advantage over the centralized tree index, where, if the root is corrupted, the client has to wait until the root is next broadcast before restarting the search, thus causing a significant access delay.

We can observe several nice properties of the exponential index from this simple example:

- The index has a linear yet distributed structure. Hence, it immediately enables an index search from the next index bucket (i.e., the next bucket with an index table), thereby saving access latency. The index bucket where a search starts represents the root of a search tree for the indexed data on the air.
- The index is naturally replicated in such a way that an index link is shared by different search trees, i.e., two index searches traversing through different search trees (i.e., starting with different root buckets) may use the same index links in the searching processes. Thus, the storage overhead of the index is minimized.
- The worst tuning time is the logarithm of the bcast length under error-free broadcast.
- The index can recover an index search from a link error quickly thanks to its distributed structure.

In addition, the next section shows that the indexing overhead can be controlled by adjusting the index structure (via an exponential base) and the number of index buckets.

3.2 The Generalized Exponential Index

In the above example, the sizes of the indexed segments exponentially increase by a base of two (hereafter referred to as the *index base*). To generalize the exponential index, the index base can be set to any value $r \geq 1$. Specifically, as shown in Fig. 3, the ith entry in the index table describes the maximum key value of a segment of r^{i-1} buckets (i.e., the buckets that are

$$\left\lfloor \sum_{j=1}^{i-2} r^j + 1 \right\rfloor = \left\lfloor \frac{r^{i-1}-1}{r-1} + 1 \right\rfloor$$

to $\lfloor \sum_{j=1}^{i-1} r^j \rfloor = \lfloor \frac{r^i-1}{r-1} \rfloor$ away).

Since the exponential index maintains an index table in each bucket, the bucket capacity to accommodate data items is reduced, thereby increasing the bcast length. To reduce such indexing overhead, we group I buckets into a data chunk and build the exponential index on a per-chunk basis (i.e., including one index table in each chunk). This decreases the number of index tables in a bcast and the number of

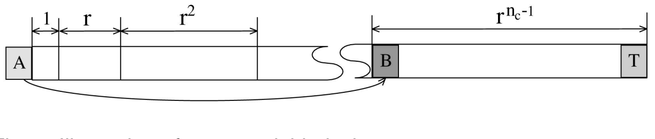


Fig. 3. Illustration of exponential indexing space.

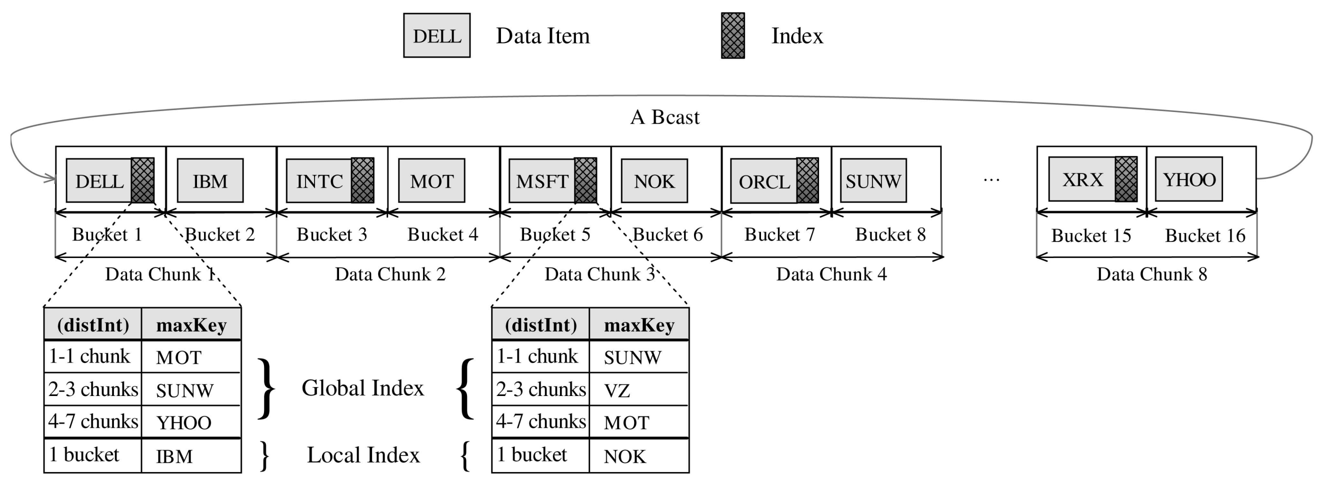

DELL Data Item
Index
A Bcast
DELL IBM INTC MOT MSFT NOK ORCL SUNW ... XRX YHOO
Bucket 1 Bucket 2 Bucket 3 Bucket 4 Bucket 5 Bucket 6 Bucket 7 Bucket 8 Bucket 15 Bucket 16
Data Chunk 1 Data Chunk 2 Data Chunk 3 Data Chunk 4 Data Chunk 8
(distInt) maxKey
1-1 chunk MOT
2-3 chunks SUNW
4-7 chunks YHOO
1 bucket IBM
Global Index
Local Index
(distInt) maxKey
1-1 chunk SUNW
2-3 chunks VZ
4-7 chunks MOT
1 bucket NOK

Fig. 4. The generalized exponential index ($r = 2$, $I = 2$).

entries in each index table. However, the price to pay for per-chunk indexing is that an average of $\frac{I-1}{2}$ buckets need to be searched to locate a data item within a data chunk.

To remedy this, we propose, in a data chunk, to construct a plain index for all buckets, where an index entry is used to describe the maximum key value for each bucket. With the plain index, if we do not consider link errors, the intrachunk tuning time is either one (for the first bucket in the chunk) or two buckets (for the other buckets). In this way, the index table for each data chunk is split into two parts: a *global index* for the other data chunks and a *local index* for the $I - 1$ buckets within the local chunk. Fig. 4 shows an example of the generalized exponential index, where the index base r is set at 2 and the chunk size I is set at 2.

We now describe the client access protocol under unreliable data broadcast. Assume that each data bucket includes an offset to the next index bucket (i.e., the first bucket of the next chunk). The client access protocol follows the same three phases described in Section 2.1. We discuss the access protocol for the proposed exponential index using an example (see Algorithm 1 for a formal description). Again, suppose that the client makes a query for item "NOK" right before the bucket containing item "DELL" is broadcast. Since the requested item is not in the current bucket, the client checks the local index. Because "NOK" is larger than the maximum key value "IBM" in the local index, the client proceeds to check the global index. In the global index, "NOK" lies in the key range specified by the second entry, hence, the client goes into the doze mode and waits for the second next index bucket (i.e., bucket 5). In the index table of bucket 5, "NOK" falls in the key range specified by the first local index entry. Therefore, the client accesses the next bucket (i.e., bucket 6) to complete the query. The total tuning time is three buckets. As another example for the same index search, now suppose bucket 1 is corrupted. The client continues to access the next bucket 2 to find out the index bucket 3. From there, it proceeds to search bucket 5 and accesses bucket 6 to retrieve the desired data. If bucket 6 is also corrupted, the client waits for one bcast to download bucket 6 in the next broadcast cycle.

Algorithm 1 Client Access Protocol for the Exponential Index under Unreliable Broadcast

```
1: sequentially access the broadcast until an error-free
       bucket is retrieved
2: if the bucket is a data bucket then
3:     go into the doze mode and wake up at the next index
           bucket
4:     if the index bucket is corrupted then
5:         go into the doze mode again, wait for $(I - 1)$
               buckets to retrieve the next index bucket, and
               goto Line 4
6:     end if
7: end if // so far an error-free index bucket is retrieved
8: for each data item in the index bucket do
9:     if it is the requested data item then
10:        stop the search and the query is finished
11:    end if
12: end for
13: // check the local index in the index table:
14: if the requested data item is within the key range
        specified by the $i$th local entry then
15:    go into the doze mode, wait for $i - 1$ buckets to
       retrieve the $i$th data bucket, and goto Line 26
16: end if
17: // check the global index in the index table:
18: if the requested data item is within the key range
        specified by the $i$th global entry then
19:    go into the doze mode and wait for
           $(\lfloor \frac{r^{i-1}-1}{r-1} + 1 \rfloor \cdot I - 1)$ buckets to retrieve the index
           bucket of the $(\lfloor \frac{r^{i-1}-1}{r-1} + 1 \rfloor)$th chunk
20:    if the retrieved index bucket is corrupted then
21:        go into the doze mode, wait for $(I - 1)$ buckets to
           access the next index bucket, and goto Line 20
22:    else
23:        goto Line 8 to repeat this search procedure
24:    end if
25: end if // so far the data bucket containing the desired
        item is located
26: if the data bucket is corrupted then
27:    go into the doze mode, wait for $(IC - 1)$ buckets to
```

TABLE 1
Summary of Notations

Notation	Description
N	number of data items
B	capacity of a data bucket without an index
B'	capacity of a data bucket with an index
p	probability of a bucket being corrupted
s_o	size of a data item
s_e	size of an index entry
I	chunk size, i.e., # buckets in a data chunk
r	index base
C	number of chunks in a bcast
s_i	size of index table for each chunk
n_i	number of entries in an index table
n_b	number of local index entries for local buckets
n_c	number of global index entries for data chunks
$E(d)$	average access latency
$E(t)$	average tuning time
$O(t)$	worst tuning time

```
            access the same bucket in the next broadcast
            cycle, and goto Line 26
28: else
29:     retrieve the data and the query is finished
30: end if
```

There are two tuning knobs for the generalized exponential index: index base r and chunk size I. These two parameters offer the exponential index great flexibility in tuning access latency against tuning time. In general, the number of index entries and, hence, the indexing overhead increases with decreasing index base r, and the tuning time decreases with r. Moreover, the larger the chunk size I, the less the tuning time but the longer the initial index probing time. A detailed performance analysis is provided in the next section.

3.3 Performance Analysis

This section analyzes the access latency and tuning time of the exponential index. We assume that the access probabilities of data items are uniformly distributed and the initial points to tune in the broadcast channel are randomly distributed over the bcast. Table 1 summarizes the notations used in the analysis.

Let B denote the number of data items that a data bucket can hold. Since an index table needs to occupy the space used to store data items, fewer items can be accommodated by a bucket with an index table. Let B' denote the number of items such a bucket can hold. The value of B' is a function of the parameters of I and r. Note that B' is an integer and r is a real number. An arbitrary r may not result in an index table of a size equal to a multiple of the data item size. Since the tuning time generally decreases with the index base r, it is desirable to adjust r according to B' to fully exploit the available space for an index table.

Given B and B', the number of entries in an index table follows:

$$n_i \le \frac{(B-B')\cdot s_o}{s_e}, \tag{1}$$

where s_o and s_e are the sizes of a data item and an index entry, respectively.

Since a data chunk consists of I buckets, the number of local index entries is simply given by: $n_b = I-1$. Thus, we obtain the number of global index entries:

$$n_c = n_i - n_b \le \frac{(B-B')\cdot s_o}{s_e} - I + 1. \tag{2}$$

As a data chunk consists of $I-1$ buckets without index tables and one bucket with an index table, it can hold a total of $B(I-1)+B'$ data items. Hence, the number of data chunks in a bcast is given by:

$$C = \left\lceil \frac{N}{B(I-1)+B'} \right\rceil. \tag{3}$$

The index table in each chunk indexes all the other chunks in a bcast; thus, we must have

$$\sum_{i=1}^{n_c} r^{i-1} = \frac{r^{n_c}-1}{r-1} \ge C-1. \tag{4}$$

Therefore, given B' and I, the smallest value of r can be obtained by numerically solving the following inequality:

$$r^{n_c} + (1-C)r + C - 2 \ge 0. \tag{5}$$

Now, we derive the average access latency and tuning time given B' and I. The average access latency is obtained as follows (see the Appendix, which can be found on the Computer Society Digital Library at http://www.computer.org/tkde/archives.htm, for detailed derivation):

$$\begin{aligned} E(d) =& \frac{IC}{2} + 1 + \frac{IC\cdot p}{1-p} \\ &+ \left(\frac{I}{2} + \frac{IC\cdot p}{1-p}\right)\cdot\frac{B(I-1)}{B(I-1)+B'}. \end{aligned} \tag{6}$$

The average tuning time is given by (see the Appendix, which can be found on the Computer Society Digital Library at http://www.computer.org/tkde/archives.htm, for detailed derivation):

$$\begin{aligned} E(t) =& \frac{2I-1}{I(1-p)} \\ &+ \frac{1}{C}\left(t(0) + \sum_{l=1}^{C-1}\left(t(l) + \frac{t(C-1)\cdot p}{1-p}\right)\right) \\ &+ \frac{B(I-1)}{(B(I-1)+B')\cdot(1-p)}, \end{aligned} \tag{7}$$

where

$$t(l) = \begin{cases} 0, & \text{if } l = 0; \\ t(l-x)\cdot(1-p) + t(l-1)\cdot p + 1, & \text{if } l > 0, \end{cases} \tag{8}$$

where x is the maximum value less than or equal to l in the set of $\{1, 2, \lfloor r+2\rfloor, \cdots, \lfloor\frac{r^{n_c-1}-1}{r-1}\rfloor + 1\}$.

From (5) and (7), it is not difficult to see that, with the same value of B', the smaller is the value of r, the less is the average tuning time in general. Therefore, in performance optimization and tuning, we examine only the smallest values of r that result in an index table whose size is a multiple of the data item size, rather than testing all possible values of r.

To have more intuition on tuning time, we also derive the worst tuning time, assuming the broadcast is error-free.[4] Suppose the client initially tunes into data chunk A and is interested in some data item in chunk T (see Fig. 3). The initial search space is C (approximately $\frac{r^{n_c}-1}{r-1}$) buckets. According to the index table in A, the search will be guided to a certain range of sequential data chunks whose size is at most r^{n_c-1} chunks (when T falls in the last index entry). Thus, the search space is reduced by a factor of at least

$$\frac{\frac{r^{n_c}-1}{r-1}}{r^{n_c-1}} = \frac{r - r^{1-n_c}}{r-1} \approx \frac{r}{r-1}. \tag{9}$$

Then, the client will access the first chunk in the refined search space (e.g., chunk B in Fig. 3) and trim the search space again by a factor of at least $\frac{r}{r-1}$ through examination of B's index table. The procedure is repeated until the refined search space contains one data chunk only. Therefore, at most $\lceil log_{\frac{r}{r-1}}(C-1)\rceil + 1$ buckets are accessed to reach the target chunk. If a chunk contains more than one bucket, we might need one more bucket access to probe the first index bucket and another one to locate the desired data item after reaching the target chunk. Therefore, the tuning time is bounded by:

$$O(t) = \begin{cases} \lceil log_{\frac{r}{r-1}}(C-1)\rceil + 1, & \text{if } I = 1; \\ \lceil log_{\frac{r}{r-1}}(C-1)\rceil + 3, & \text{if } I > 1. \end{cases} \tag{10}$$

3.4 Performance Tuning

As mentioned before, tuning time and access latency are two conflicting performance measures; they cannot be minimized at the same time. To cater for different application scenarios, we need tunable indexing structures that optimize either the tuning time or the access latency with a certain performance requirement on the other metric. The proposed exponential index can be employed to serve this purpose. Specifically, we are interested in tuning the performance along two dimensions:

- **Latency-bounded tuning**. Given a limit L on the average access latency, how can the parameters (i.e., r and I) of the exponential index be tuned to obtain the minimum tuning time?
- **Tuning-time bounded tuning**. Given a limit T on the average tuning time, how can the parameters of the exponential index be tuned to achieve the shortest access latency?

Based on the analysis presented in the last section, the optimal solutions to the above two problems can be obtained by searching the optimal values of B' (recall that r is a function of B' as shown by (2), (3), and (5)) and I. Thus, the latency-bounded tuning problem is defined as follows:

TABLE 2
Parameter Settings

Parameter	Setting	Parameter	Setting
N	30,000	B	10, 80
s_o	16, 128 bytes	s_e	4 bytes
p	0 – 10%		

$$\min_{I=\{1,2,\cdots,\lceil\frac{N}{B}\rceil\},B'=\{0,1,\cdots,\lfloor B-\frac{I\cdot s_e}{s_o}\rfloor\}} E(t), \tag{11}$$

$$\text{s.t.} \quad E(d) \leq L. \tag{12}$$

The tuning-time-bounded tuning problem is defined as follows:

$$\min_{I=\{1,2,\cdots,\lceil\frac{N}{B}\rceil\},B'=\{0,1,\cdots,\lfloor B-\frac{I\cdot s_e}{s_o}\rfloor\}} E(d), \tag{13}$$

$$\text{s.t.} \quad E(t) \leq T. \tag{14}$$

It is easy to see that these two search problems have a worst-case time complexity of $\mathcal{O}(\frac{N}{B}\cdot B) = \mathcal{O}(N)$.

4 Performance Evaluation

This section evaluates the performance of the proposed exponential index. We developed a simulator based on ns-2 to simulate the GPRS wireless network [2], [11]. We would like to compare the exponential index against the state-of-the-art indexes (i.e., the distributed tree [10] and the flexible index [9] enhanced for unreliable broadcast [23]) and to investigate its ability to adjust the trade-off between access latency and tuning time.

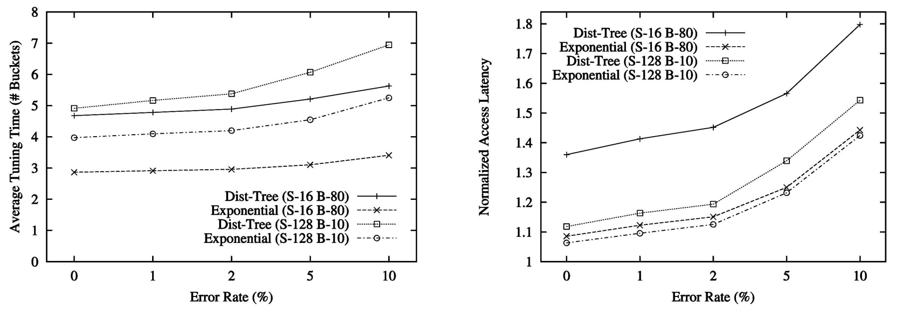
In the simulation, a broadcast server and a client are simulated, and the user requests are sequentially issued and processed by the client.[5] We set the system parameters similar to those in [9], [10], [23]. The database size is set at 30,000 items. The link error probability ranges from 0 to 10 percent. Flat broadcast is employed to broadcast the data items. Without loss of generality, we assume that the access distribution over the data items is uniform.[6] For the exponential index and the flexible index, an index entry contains a key value only; hence, its size is set to 4 bytes. For the distributed tree, the index entry size is set to 8 bytes since it contains a key value as well as the offset to the bucket containing the key value. We have evaluated different combinations of item size s_o and bucket capacity B. Due to space limitations, we report the results for two informative settings only, i.e., 1) $s_o = 16$ bytes, $B = 80$ (denoted as "S-16 B-80") and 2) $s_o = 128$ bytes, $B = 10$ (denoted as "S-128 B-10"). Recall that each data bucket includes an offset to the beginning of the next index bucket. For simplicity, we omit this overhead since it is very small and exists in all the indexing schemes under investigation. The system parameter settings are summarized in Table 2.

We compare the indexing schemes in terms of the tuning time and access latency, both of which are measured in the

4. The worst tuning time is ∞ when considering link errors.

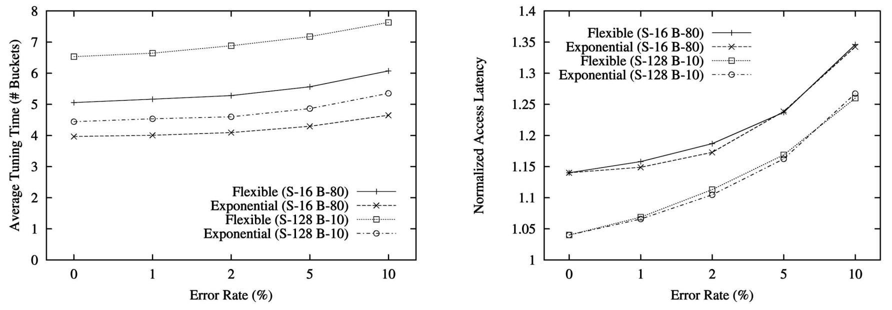
5. Note that the data access rate will not have an impact on the overall performance.

6. With flat broadcast, the average performance of the indexes would remain the same no matter what the data access pattern is.



(a) (b)

Fig. 5. Exponential versus distributed tree index. (a) Average tuning time. (b) Normalized access latency.



(a) (b)

Fig. 6. Exponential versus flexible index (chunk size = one bucket). (a) Average tuning time. (b) Normalized access latency.

unit of buckets. To make a clear comparison, the access latency of an indexing scheme shown in the results is normalized by the latency of a nonindex scheme, i.e., $\lceil \frac{N}{2B} \rceil$. The results reported for all indexes under investigation were obtained from simulation. We have also calculated the results for the exponential index based on the analysis presented in Section 3; the simulation results match the analytical results, which confirms the correctness of the analysis.

4.1 Comparison with the Distributed Tree Index

This set of experiments compares the proposed exponential index to the distributed tree, which is a nonflexible scheme. To compare the tuning time, we first measure the performance of the distributed tree given the default system setting; we then obtain the best tuning time for the exponential index by tuning the index base and chunk size such that its access latency is no higher than that of the distributed tree. Fig. 5a shows the average tuning time as a function of link error probability. As expected, the tuning time increases with increasing error probability for both index schemes. The exponential index outperforms the distributed tree by 25-42 percent.

Similarly, to compare the access latency, we tune the parameters of index base and chunk size to obtain the best result for the exponential index while making sure its tuning time is no worse than that of the distributed tree. As shown in Fig. 5b, the exponential index achieves a better performance than the distributed tree for all cases.

4.2 Comparison with the Flexible Index

This section compares the proposed exponential index to the flexible index in terms of their effectiveness in reducing the tuning time. Note that these two schemes have a similar performance for a local data search within a chunk. Therefore, to facilitate the comparison, we set the chunk size to one bucket to observe their performance differences for global index search across data chunks. For the exponential index, we adjust the index base r such that it achieves a similar access latency to that of the flexible index. Fig. 6a and Fig. 6b show the average tuning time and the normalized access latency, respectively.

As shown in Fig. 6, with the same (or even less) access latency (Fig. 6b), the exponential index consistently outperforms the flexible index in terms of the tuning time (Fig. 6a). The improvement is more significant for the setting of item size 128 bytes, bucket capacity 10 (i.e., S-128, B-10) than the setting of item size 16 bytes, bucket capacity 80 (i.e., S-16, B-80). This can be explained as follows: The flexible index employs a binary control index, which blindly incurs overhead without considering the available space. Thus, the larger the item size, the higher the probability of leaving large internal fragments. On the other hand, the exponential index adjusts the parameter of r according to the available space for indexing. Hence, with a large item size, it can fully utilize the large item space to achieve a better performance.

4.3 Flexibility of the Indexes

This section investigates the indexes' ability to adjust the trade-off between access latency and tuning time. First, we look at the tuning-time-bounded tuning problem. It is

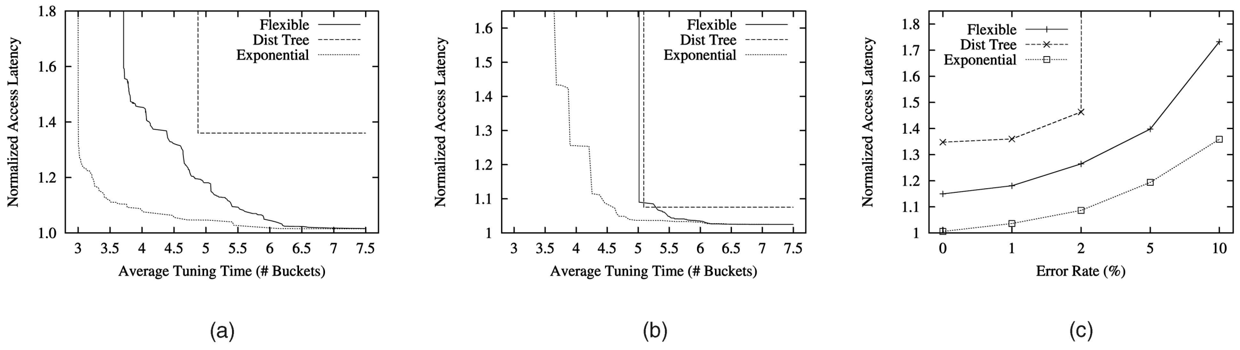


Fig. 7. Access latency with bounded tuning time. (a) Link error = 1 percent, S-16, B-80. (b) Link error = 1 percent, S-128, B-10. (c) Link error = 0-10 percent, S-16, B-80.

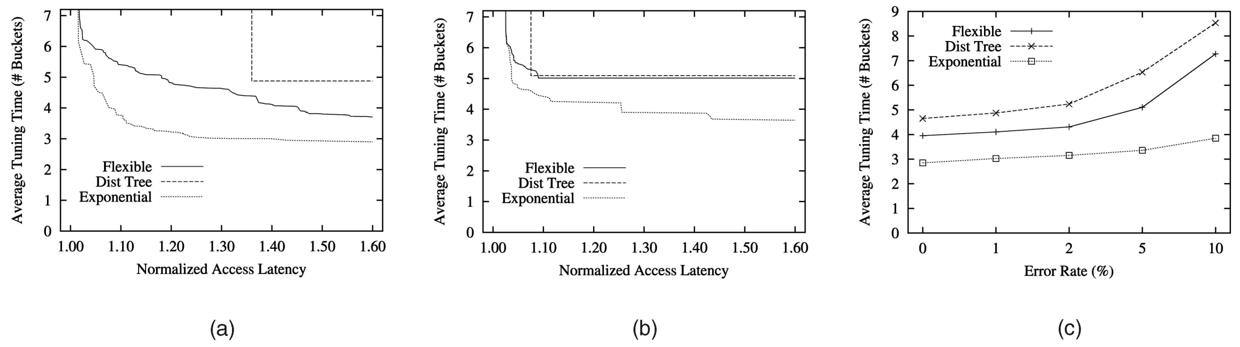


Fig. 8. Tuning time with bounded access latency. (a) Link Error = 1 percent, S-16, B-80. (b) Link error = 1 percent, S-128, B-10. (c) Link error = 0-10 percent, S-16, B-80.

desirable that the longer is the tuning time allowed, the shorter is the access latency achieved. Fig. 7a and Fig. 7b, respectively, show the results for the settings of S-16, B-80 and S-128, B-10 under the default link error probability 1 percent. As expected, the distributed tree is not flexible: It is impossible for it to achieve a tuning time shorter than 4.8 buckets, and the latency remains the same after this point. While the flexible index is able to trade access latency for tuning time; obviously, the exponential index performs even better. With the same tuning time requirement, the exponential index achieves a shorter (or the same) access latency. For a similar reason to that explained in Section 4.2, the improvement of the exponential index over the flexible index is more remarkable for the setting with a larger item size.

To examine the indexes' resilience to link errors, we show in Fig. 7c the achieved latency with a bounded tuning time of 5.0 under a variety of link error probabilities. As can be seen, the performance improvement of the exponential index over the other two indexes becomes more significant (e.g., from 12 to 20 percent against the flexible index) with increasing error probability, implying the exponential index is more resilient to link errors. This confirms our claim that the exponential index recovers an index search from a link error more quickly (as discussed in Section 3).

Next, we examine the latency-bounded tuning problem. We expect to achieve a shorter tuning time by tolerating a longer latency. As shown in Fig. 8a and Fig. 8b, the distributed tree obtains a better performance than the flexible index only at normalized access latencies of 1.07-1.09 for the setting of S-128, B-10. The exponential index performs the best throughout the range of bounded latencies tested.

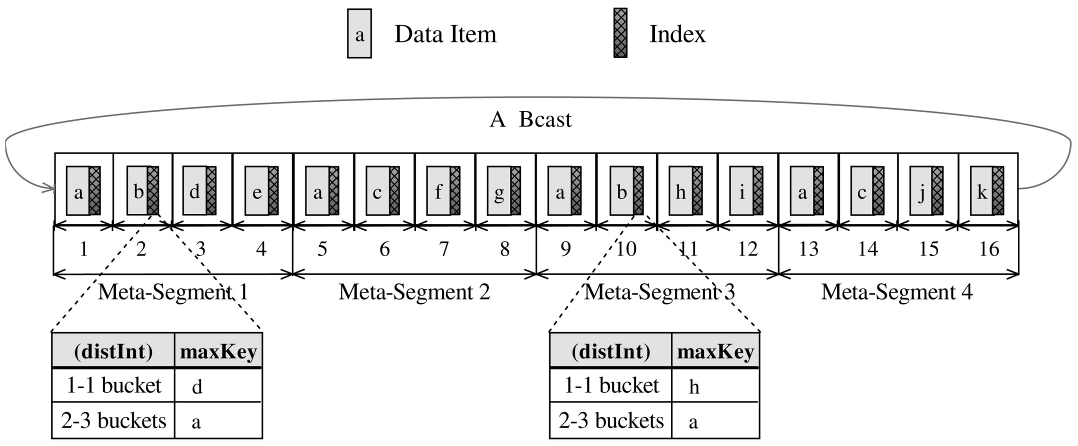
Fig. 8c shows the tuning time under a variety of link error probabilities when the access latency is bounded at 1.4. Once again, the exponential index demonstrates a stronger resilience to link errors. As the link error probability is increased from 0 to 10 percent, the performance improvement of the exponential index over the distributed index and the flexible index increases from 36 percent to 53 percent and 26 percent to 44 percent, respectively.

5 Extensions

The previous sections have focused on clustered broadcast, which is capable of indexing the primary attribute in flat broadcast. This section extends the proposed exponential index to nonclustered broadcast, which is useful for indexing secondary attributes and skewed broadcast. In addition, we leverage the idea of index caching to further improve the access performance.

5.1 Nonclustered Broadcast

As discussed in Section 2, for a nonclustered broadcast, a bcast can be partitioned into a number of clustered segments (i.e., metasegments). The number of metasegments in a bcast for an attribute is called the *scattering factor* (denoted by M) [10]. Without loss of generality, we assume the items are sorted in ascending order of the attribute values in each metasegment. Similar to clustered broadcast, the exponential index can be applied to each metasegment.



(distInt)	maxKey
1-1 bucket	d
2-3 buckets	a

(distInt)	maxKey
1-1 bucket	h
2-3 buckets	a

Fig. 9. Indexing nonclustered broadcast ($r = 2,\ I = 1$).

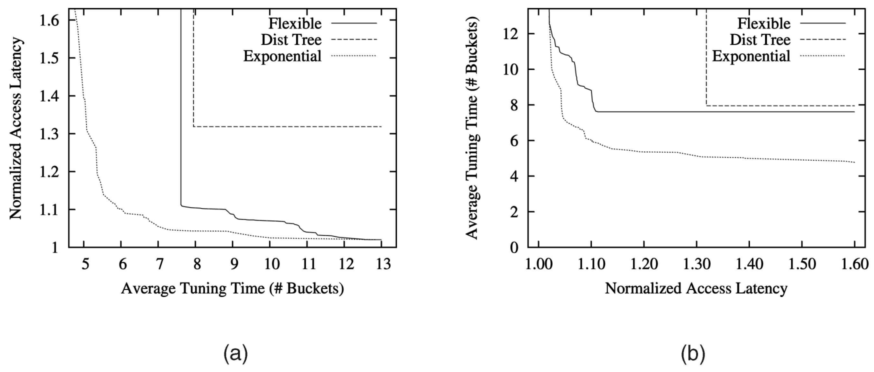


Fig. 10. Flexibility of the indexes for nonclustered broadcast (S-16, B-80, link error = 1 percent). (a) Tuning-time bounded tuning. (b) Access-latency bounded tuning.

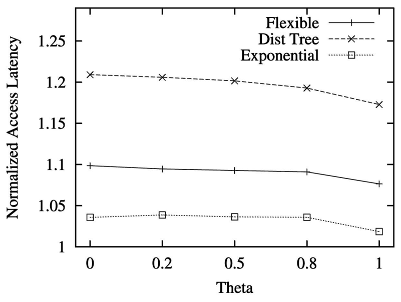
Instead of indexing a whole bcast, each index table for nonclustered broadcast covers the buckets up to the farthest one in the next metasegment whose attribute value is less than that of the current bucket. In the example shown in Fig. 9, the index table in bucket 2 indexes buckets 3-5, and the index table in bucket 10 indexes buckets 11-13.

The client access protocol remains the same except that a query continues to search the next segment if the target item is not found in the current metasegment. The maximum number of metasegments to be searched is M. Thus, based on (10), the tuning time of a query is bounded by $\mathcal{O}(Mlog_{\frac{r}{r-1}}S)$, where S is the number of chunks in a metasegment.

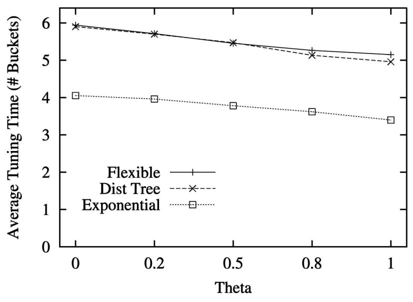
We now evaluate the flexibility of the exponential index for nonclustered broadcast. We simulate a broadcast-disk system [1] which consists of three disks with 800, 1,600, and 23,600 data items, respectively. The three disks are interleaved in a bcast but rotate at different speeds: The first disk rotates at a speed twice as fast as the second one and four times as fast as the third disk. In other words, the items in the first, second, and third disks are broadcast 4, 2, and 1 time(s), respectively, in each bcast. The resulting bcast has four metasegments, each of which contains 7,500 clustered items. The access probability of each item is set proportional to its broadcast frequency. The distributed tree and the flexible index are included for comparison. For both of the index schemes, an index is built for each metasegment; similar to the exponential index, if the item of interest is not found in the current metasegment, the search continues in the next metasegment until the item is retrieved.

As shown in Fig. 10, the exponential index has the best performance among all indexes under investigation. Compared with the clustered broadcast case (Fig. 7a and Fig. 8a), the improvement here is more significant. For example, in tuning-time bounded tuning, the exponential index achieves a tuning time of 4.9 buckets at a normalized latency of 1.6, whereas the flexible index cannot get a tuning time shorter than 7.6 buckets (cf. 3.0 versus 3.7 buckets in Fig. 7a). This is mainly because the exponential index allows the current metasegment to index into the next metasegment to support continuous search, thus improving search efficiency; in contrast, the distributed tree and the flexible index constrain the indexing space within a metasegment only and, hence, when the item is not found in the current metasegment, the search has to restart from the index root in the next metasegment.

To evaluate the impact of skewed data access, a Zipf distribution (with a skewness parameter θ) is used to simulate client access behavior [26]. The broadcast-disk system described earlier is employed to generate broadcast program. Fig. 11 shows the results under different settings of θ. We set the bounded tuning time at 11 for Fig. 11a and the bounded access latency at 1.8 for Fig. 11b. It can be observed the exponential index consistently outperforms the distributed tree and the flexible index. This demonstrates that the exponential index is robust to skewed data access.



(a)



(b)

Fig. 11. Impact of skewed access for nonclustered broadcast (S-16, B-80, link error = 1 percent). (a) Tuning-time bounded tuning. (b) Access-latency bounded tuning.

5.2 Index Caching

We now discuss how to cache the index to further improve data access performance. In particular, we shall show that the exponential index can benefit from more performance advantages from index caching than the other indexes.

The client access protocol with index caching is similar to that without it. Upon a data request, the client first tunes into the broadcast and retrieves the first index bucket by following Algorithm 1. If the index (table) is not cached yet, it is cached on the client. Then, it follows the exponential index to determine which index bucket to access next. If the next index (table) is in the cache, it can be accessed immediately from the cache. Otherwise, as usual, the client tunes into the broadcast to retrieve it. An issue here is how to tell if the next index (table) is cached or not, or, equivalently, how to figure out the id of the next index (table). Recall that each bucket contains a sequentially increased id number. As such, the id of the next index (table) can be computed from the id of the current and the indexing distance. The saving for index data retrieval due to index caching can also be achieved by the distributed tree and the flexible index.

Yet, the exponential index can do more than that thanks to its nice data structure. Since multiple search trees are embedded in the exponential index, we can make use of this feature to further refine the search space with caching. We take one example to illustrate this additional advantage. Assume again the broadcast program is as shown in Fig. 2. To retrieve item "SUNW" starting from the "DELL" without index caching, the client accesses buckets 1, 5, 7, and 8, as discussed in Section 3.1. Now, suppose that the index table of bucket 3 is cached. By first accessing bucket 1, we know that the desired item must be stored in buckets 5 through 8. Moreover, we figure out from the cached index table that the item must be in buckets 7 through 10. Hence, the desired item should lie in buckets 7 and 8. Therefore, we next access buckets 7 and 8 to retrieve the item "SUNW." Compared to the no-caching case, one bucket access can be saved. With more index tables cached, more performance improvement can be expected. The index cache access algorithm is formally described in Algorithm 2.

Algorithm 2 Index Cache Access Algorithm

```
1:  tune into the broadcast and retrieve the first index
        bucket $B$ by following steps 1 to 7 Algorithm 1
2:  Denote by $\mathcal{R}$ the range of a bcast
3:  for each cached index table $\tau$ do
4:      calculate the segment of buckets, $\mathcal{R}_\tau$, containing the
            desired item
5:      $\mathcal{R} = \mathcal{R} \cap \mathcal{R}_\tau$
6:  end for
7:  calculate from $B$ the segment of buckets, $\mathcal{R}_i$, containing
        the desired item
8:  $\mathcal{R} = \mathcal{R} \cap \mathcal{R}_i$
9:  access the cache or tune into the broadcast to access the
        first bucket $B$ in $\mathcal{R}$
10: repeat Lines 7-9 till the desired item is retrieved
```

To focus on the performance of index caching, we assume there are no index updates in the experiments.[7] We employ LRU as the cache replacement policy. Fig. 12 shows the tuning time as a function of cache size, where the flat broadcast is employed and the bounded access latency is set at 1.4. The cache size is represented as a percentage of the size of the distributed tree index. As observed, as the cache size is increased from 0 to 15 percent, all schemes improve their performance. Among the three schemes, the exponential index reduces the tuning time most significantly as expected (i.e., 24 versus 11 percent for distributed tree and 15 percent for flexible index). As a result, its performance gain over the distributed tree and the flexible index is increased from 62 to 96 percent and from 37 to 56 percent, respectively.

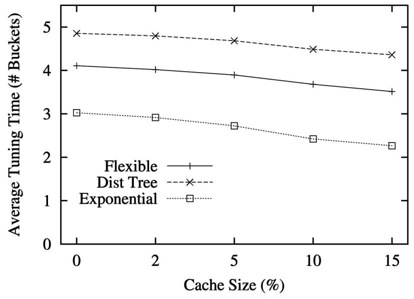


Fig. 12. Performance as a function of cache size (S-16, B-80, link error = 1 percent).

7. The cache consistency issue for data broadcast has been well studied in the literature. This is out of the scope of this paper; however, interested readers are referred to [3], [22].

6 Conclusions

This paper has investigated the use of air indexing techniques to improve the efficiency of energy consumption on mobile devices in an unreliable broadcast system. We have proposed a novel parameterized index scheme called the exponential index. It has a linear yet distributed structure which suits the broadcast environment very well. The distributed property of the exponential index enables a search to start immediately from an arbitrary index table in the broadcast as well as to restart quickly in case of a link error. The energy consumption of mobile clients is also very efficient (i.e., the tuning time is logarithmically proportional to the bcast length). Moreover, the access latency and tuning time of the exponential index can be adjusted by two tuning knobs: index base and chunk size.

We have provided an analytical model to derive the access latency and tuning time of the exponential index and analyzed how to minimize the access latency (or tuning time) with a bounded tuning time (or access latency). We have demonstrated via simulations that

1. the exponential index substantially outperforms two state-of-the-art air indexing schemes,
2. it is more resilient to link errors,
3. it benefits from more performance advantages from index caching, and
4. it achieves a greater flexibility in adjusting the trade-off between access latency and tuning time.

There are a number of issues that deserve further study. The proposed index exponentially partitions the indexing space, yet the optimal partitioning remains an open problem. The exponential index does not differentiate the performance requirements of individual clients or items. We plan to investigate client-based and item-based flexible indexes. In addition, we are interested in exploring the research issues of balancing access latency and tuning time in a multichannel data broadcast environment.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments. Professor Michael Franklin at UC Berkeley offered many valuable suggestions on an earlier version of this paper. Jianliang Xu's work was partially supported by grants from the Research Grants Council of the Hong Kong SAR, China (Project Nos. HKBU 2115/05E and HKBU FRG/04-05/II-26). Wang-Chien Lee's work was supported in part by US National Science Foundation grant IIS-0328881. Xueyan Tang's work was supported in part by a grant from Nanyang Technological University (Grant No. CE-SUG 1/04). This work was done when Shanping Li was a visiting scholar at Hong Kong Baptist University.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, "Broadcast Disks: Data Management For Asymmetric Communications Environments," *Proc. ACM SIGMOD Conf. Management of Data,* pp. 199-210, May 1995.

[2] C. Bettstetter, H.-J. Vogel, and J. Eberspacher, "GSM Phase 2+ General Packet Radio Service GPRS: Architecture, Protocols, and Air Interface," *IEEE Comm. Surveys,* vol. 2, no. 3, 1999.

[3] G. Cao, "A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 5, pp. 1251-1265, Sept./Oct. 2003.

[4] M.-S. Chen, K.-L. Wu, and P.S. Yu, "Optimizing Index Allocation for Sequential Data Broadcasting in Wireless Mobile Computing," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 1, pp. 161-173, Jan./Feb. 2003.

[5] C.-L. Hu and M.-S. Chen, "Dynamic Data Broadcasting with Traffic Awareness," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS '02),* pp. 112-119, July 2002.

[6] Q.L. Hu, W.-C. Lee, and D.L. Lee, "Power Conservative Multi-Attribute Queries on Data Broadcast," *Proc. 16th Int'l Conf. Data Eng. (ICDE '00),* pp. 157-166, Feb. 2000.

[7] Q.L. Hu, W.-C. Lee, and D.L. Lee, "A Hybrid Index Technique for Power Efficient Data Broadcast," *Distributed and Parallel Databases (DPDB),* vol. 9, no. 2, pp. 151-177, Mar. 2001.

[8] W.C. Huffman and V. Pless, *Fundamentals of Error-Correcting Codes.* Cambridge Univ. Press, 2003.

[9] T. Imielinski, S. Viswanathan, and B.R. Badrinath, "Power Efficient Filtering of Data on Air," *Proc. Fourth Int'l Conf. Extending Database Technology (EDBT '94),* pp. 245-258, Mar. 1994.

[10] T. Imielinski, S. Viswanathan, and B.R. Badrinath, "Data on Air—Organization and Access," *IEEE Trans. Knowledge and Data Eng.,* vol. 9, no. 3, pp. 353-372, May/June 1997.

[11] R. Jain, "GPRS Simulations Using NS—Network Simulator," master's thesis, Dept. of Electrical Eng., India Inst. of Technology—Bombay, http://www.isi.edu/nsnam/ns/ns-contributed.html, June 2001.

[12] K.C.K. Lee, H.V. Leong, and A. Si, "Semantic Data Broadcast for a Mobile Environment Based on Dynamic and Adaptive Chunking," *IEEE Trans. Computers,* vol. 51, no. 10, pp. 1253-1268, Oct. 2002.

[13] V.C.S. Lee, J.K. Ng, J.Y.P. Chong, and K.-W. Lam, "Maintaining Temporal Consistency in Broadcast Environments," *Proc. Fifth IEEE Int'l Conf. Mobile Data Management (MDM '04),* Jan. 2004.

[14] V. Liberatore, "Caching and Scheduling for Broadcast Disk Systems," Technical Report 98-71, Inst. for Advanced Computer Studies, Univ. of Maryland at College Park (UMIACS), Dec. 1998.

[15] V. Liberatore, "Multicast Scheduling for List Requests," *Proc. IEEE INFOCOM '02 Conf.,* pp. 1129-1137, June 2002.

[16] DirectBand Network, Microsoft Smart Personal Objects Technology (SPOT), http://www.microsoft.com/resources/spot/, 2005.

[17] E. Pitoura and P. Chrysanthis, "Exploiting Versions for Handling Updates in Broadcast Disks," *Proc. Conf. Very Large Data Bases (VLDB '99),* pp. 114-125, 1999.

[18] Q. Ren, M.H. Dunham, and V. Kumar, "Semantic Caching and Query Processing," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 1, pp. 192-210, Jan./Feb. 2003.

[19] N. Shivakumar and S. Venkatasubramanian, "Energy-Efficient Indexing for Information Dissemination in Wireless Systems," *ACM/Baltzer J. Mobile Networks and Applications (MONET),* vol. 1, no. 4, pp. 433-446, Dec. 1996.

[20] StarBand, http://www.starband.com/, 2005.

[21] Hughes Network Systems, DIRECWAY Homepage, http://www.direcway.com/, 2005.

[22] K.L. Tan, J. Cai, and B.C. Ooi, "An Evaluation of Cache Invalidation Strategies in Wireless Environments," *IEEE Trans. Parallel and Distributed Systems,* vol. 12, no. 8, pp. 789-807, Aug. 2001.

[23] K.L. Tan and B.C. Ooi, "On Selective Tuning in Unreliable Wireless Channels," *J. Data and Knowledge Eng.,* vol. 28, no. 2, pp. 209-231, Nov. 1998.

[24] K.L. Tan and J.X. Yu, "Energy Efficient Filtering of Nonuniform Broadcast," *Proc. 16th Int'l Conf. Distributed Computing Systems (ICDCS '96),* pp. 520-527, May 1996.

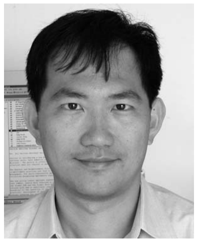
[25] M.A. Viredaz, L.S. Brakmo, and W.R. Hamburgen, "Energy Management on Handheld Devices," *ACM Queue,* vol. 1, no. 7, pp. 44-52, Oct. 2003.

[26] J. Xu, Q.L. Hu, W.-C. Lee, and D.L. Lee, "Performance Evaluation of an Optimal Cache Replacement Policy for Wireless Data Dissemination," *IEEE Trans. Knowledge and Data Eng.,* vol. 16, no. 1, pp. 125-139, Jan. 2004.

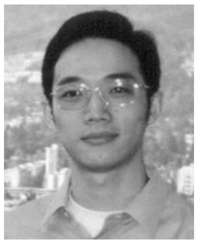
[27] J. Xu, W.-C. Lee, and X. Tang, "Exponential Index: A Parameterized Distributed Indexing Scheme for Data on Air," *Proc. Second ACM/USENIX Int'l Conf. Mobile Systems, Applications, and Services (MobiSys '04),* pp. 153-164, June 2004.

[28] J. Xu, B. Zheng, W.-C. Lee, and D.L. Lee, "Energy Efficient Index for Querying Location-Dependent Data in Mobile Broadcast Environments," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03),* pp. 239-250, Mar. 2003.

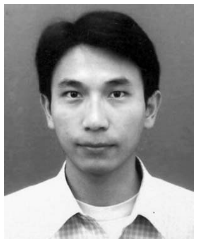
Jianliang Xu received the BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998 and the PhD degree in computer science from the Hong Kong University of Science and Technology in 2002. He is currently an assistant professor in the Department of Computer Science at Hong Kong Baptist University. His research interests include mobile and pervasive computing, wireless sensor networks, and distributed systems, with an emphasis on data management. He has published more than 40 technical papers in these areas, many in prestigious journals and conferences, including ACM SIGMOD, MobiSys, IEEE ICDE, INFOCOM, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Parallel and Distributed Systems*, and *VLDB Journal*. He is a coeditor of a book entitled *Web Content Delivery* published by Springer. He has also served as a session chair and program committee member for many international conferences, including IEEE INFOCOM. He is a member of the IEEE.



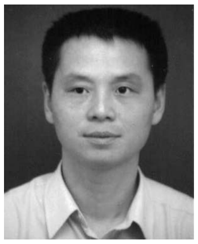
Wang-Chien Lee received the BS degree from the Information Science Department, National Chiao Tung University, Taiwan, the MS degree from the Computer Science Department, Indiana University, and the PhD degree from the Computer and Information Science Department, the Ohio State University. He is an associate professor of computer science and engineering at Pennsylvania State University. Prior to joining Penn State, he was a principal member of the technical staff at Verizon/GTE Laboratories, Inc. Dr. Lee performs cross-area research in database systems, pervasive/mobile computing, and networking. He is particularly interested in developing data management techniques for supporting complex queries in a wide spectrum of networking and mobile environments, such as peer-to-peer networks, mobile ad hoc networks, wireless sensor networks, and wireless broadcast systems. He has served as a guest editor for several journal special issues on mobile database-related topics, including the *IEEE Transactions on Computers*, *IEEE Personal Communications Magazine*, *ACM MONET*, and *ACM WINET*. He was the founding program committee cochair for the International Conference on Mobile Data Management. He is a member of the IEEE, the IEEE Computer Society, and the ACM.



Xueyan Tang received the BEng degree in computer science and engineering from Shanghai Jiao Tong University, Shanghai, China, in 1998 and the PhD degree in computer science from the Hong Kong University of Science and Technology in 2003. He is currently an assistant professor in the School of Computer Engineering at Nanyang Technological University, Singapore. He has served as a program committee member of IEEE INFOCOM '04 and WWW '05. He is an editor of a book entitled *Web Content Delivery* published by Springer. His research interests include mobile and pervasive computing, wireless sensor networks, Web and Internet, and distributed systems, particularly the data management aspects in these areas. He is a member of the IEEE.



Qing Gao received the BS degree in computer science from Zhejiang University, Hangzhou, China, in 2000. He is pursuing a PhD degree in computer science at Zhejiang University and currently is an exchange research student at Hong Kong Baptist University. His research interests include mobile communications, sensor networks, and grid computing.



Shanping Li received the BEng and MS degrees in computer science and engineering from Zhejiang University, Hangzhou, China, and the PhD degree in computer science from Zhejiang University in 1993. He is currently a professor in the College of Computer Science at Zhejiang University. He was a visiting scholar at Hong Kong Baptist University in Feb. 2004. His research interests include semantic Web, grid computing, and pervasive computing.