Exponential Index: A Parameterized Distributed Indexing Scheme for Data on Air

Jianliang Xu[*]
Dept. of Computer Science
Hong Kong Baptist University
Kowloon Tong, Hong Kong
xujl@comp.hkbu.edu.hk

Wang-Chien Lee
Dept. of Computer Sci. & Eng.
Penn State University
University Park, PA 16801
wlee@cse.psu.edu

Xueyan Tang
School of Computer Eng.
Nanyang Technological Univ.
Singapore 639798
asxytang@ntu.edu.sg

ABSTRACT

Wireless data broadcast has received a lot of attention from industries and academia in recent years. Access efficiency and energy conservation are two critical performance concerns in a wireless data broadcast environment. To improve the efficiency of energy consumption on mobile devices, traditional disk-based indexing techniques such as B^+-tree have been extended to index broadcast data on a wireless channel. However, existing designs are mostly based on centralized tree structures. Most of these indexing techniques are not flexible in the sense that the trade-off between access efficiency and energy conservation is not adjustable based on application specific requirements. We propose in this paper a novel parameterized index, called the *exponential index*, which can be tuned to optimize the access latency with the tuning time bounded by a given limit, and vice versa. The proposed index is very efficient because it facilitates replication naturally by sharing links in multiple search trees and thus minimizes storage overhead. Experimental results show that the exponential index not only achieves better performance than the state-of-the-art indexes but also enables great flexibility in trade-offs between access latency and tuning time.

Categories and Subject Descriptors

H.3.1 [**Information Systems**]: Information Storage and Retrieval—*content analysis and indexing*; C.2.0 [**Computer Systems Organization**]: Computer-Communication Networks—*general*

General Terms

Algorithms, Performance

[*]Jianliang Xu's work was supported in part by grants from Hong Kong Research Grants Council (Grant HKUST6179/03E) and Hong Kong Baptist University (Grant FRG/02-03/II-34).

Keywords

Index structure, data broadcast, energy conservation, mobile computing

1. INTRODUCTION

Owing to the widespread deployment of wireless networks and fast-improving capabilities of mobile devices, there has been an increasing interest in wireless data services among both industrial and academic communities in recent years. There are two fundamental information delivery approaches for wireless data services: *point-to-point access* and *periodic broadcast* [6]. Point-to-point access employs a basic client-server model, where the server is responsible for processing a query and returning the result to the user via a dedicated point-to-point channel. Periodic broadcast, on the other hand, has the server actively pushing data to the users. The server determines the data and its schedule for broadcast. A user listens to a broadcast channel to retrieve data based on his queries and, thus, is responsible for query processing.

Point-to-point access is particularly suitable for light-loaded systems when contention for wireless channels and server processing is not severe. However, as the number of users increases, the system performance deteriorates rapidly. Compared with point-to-point access, broadcast is a very attractive alternative [1, 10]. It allows simultaneous access by an arbitrary number of mobile clients and thus allows efficient usage of the scarce wireless bandwidth. Moreover, in some wireless environments (e.g., satellite-based systems), communication capacities are asymmetric. The *downlink* communication capacity is much greater than the *uplink* communication capacity. With periodic broadcast, users need not submit queries to the server, which alleviates the burden on the limited uplink.

Wireless data broadcast services have been available as commercial products for many years (e.g., StarBand [18] and Hughes Network [19]). In particular, the recent announcement of the *smart personal objects technology* (SPOT) by Microsoft [15] 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. In this paper, we focus on wireless data broadcast.

Access efficiency and *energy conservation* are two critical issues for the users in a wireless data broadcast system. Access efficiency concerns how fast a request is satisfied, while energy conservation concerns how to reduce a mobile client's energy consumption when it accesses the data of interest. In the literature, two performance metrics, namely *access latency* and *tuning time*, are used to measure access efficiency and energy conservation, respectively [7, 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.

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 [16, 21]. Moreover, only a modest improvement in battery capacity of 20-30% is expected over the next few years. 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 [21].

To retrieve a data item in wireless data broadcast, a 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, mobile clients are able to predict the arrivals of their desired data. Thus, they can stay in the doze mode during waiting time and tune into the broadcast channel only when data items of interest arrive. Several traditional disk-based indexing techniques such as B$^+$-tree have been extended for air indexing [3, 10, 17]. However, existing designs are mostly based on centralized tree structures. To start a search, a client needs to wait until it reaches the root of the next broadcast search tree. To reduce this waiting latency, multiple replicated indexes are usually interleaved with data broadcast. The drawback of this solution is that broadcast cycles are lengthened due to the additional indexing information. As such, there is a trade-off between access latency and tuning time.

Different application scenarios may require different performance trade-offs: some may favor a short latency at the cost of energy consumption; others may prefer to conserve energy by tolerating a long 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*, which can be easily adjusted to optimize the access latency (or tuning time) with the tuning time (or access latency) bounded by a given limit. 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 and, hence, allows searching to start at any index segment. This suits the sequential-access broadcast environment very well.

A performance analysis of the exponential index in terms of the access latency and tuning time is provided. A wide range of experiments are also 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]. Experimental results show that the proposed exponential index not only achieves better performance than the existing indexing schemes but also enables great flexibility in trade-offs between access latency and 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 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 discusses a number of practical issues of applying the proposed index to a real broadcast system. Finally, the paper is concluded in Section 6.

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 networks. We distinguish between *index buckets* that hold the index and possibly some data if space permits, and *data buckets* which 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 Figure 1). The bcast is repeatedly broadcast on the wireless channel.

Bcasts can be classified as *flat broadcast*, where each item appears exactly once, and *skewed broadcast*, where some items may appear more than once. While skewed broadcast is useful for reducing the average access latency for non-uniform data access, its effectiveness depends heavily on the accuracy of the estimates of user access patterns [1]. In contrast, flat broadcast is simple. Moreover, even though the original user access pattern may show non-uniformity, flat broadcast is justified in the following two scenarios: 1) the non-uniform pattern is flattened out if the client employs a local cache and the data are not updated [13]; 2) flat broadcast offers very good performance for queries requesting multiple items [14].

To facilitate a search for data items via an air index, each data bucket includes an offset[1] to the beginning of the next index bucket. Taking Figure 1 as an example, the general ac-

[1]Offset denotes the relative distance of the index bucket from this bucket.

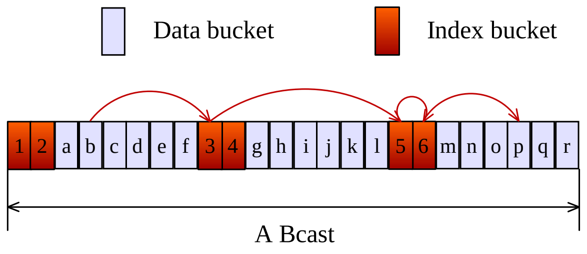


Figure 1: Data Organization on Wireless Broadcast Channels

cess protocol for retrieving data involves the following steps:

- Initial probe: The client tunes into the broadcast channel at bucket b and determines when the next index bucket is broadcast.
- Index search: The client tunes into the broadcast channel again at index bucket 3 and 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.

There are two basic data organizations with respect to an attribute within a bcast: *clustered broadcast* and *non-clustered 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 *non-clustered*. Clustered broadcast corresponds to flat broadcast with respect to the primary attribute, whereas non-clustered broadcast corresponds to flat broadcast with respect to secondary attributes or skewed broadcast [10]. Without loss of generality, data items in clustered broadcast can be arranged in ascending order of the attribute values. For non-clustered broadcast, a bcast can be partitioned into a number of segments called *meta-segments*, each of which holds a sequence of items with non-descending (or non-ascending) values of the attribute [10]. Thus, when we look at each individual meta-segment, the data items are clustered on that attribute and the indexing techniques developed for clustered broadcast can still be applied to a meta-segment. Therefore, we mainly focus on clustered broadcast when we describe the proposed index in Section 3 and extend the technique to non-clustered broadcast in Section 5.1.

2.2 Related Work

Several disk-based indexing techniques have been extended for air indexing. Imielinski *et al.* applied the B$^+$ index tree, where the leaf nodes store the arrival times of the data items [10]. The *distributed indexing* method was proposed to efficiently replicate and distribute the index tree in a bcast. Specifically, the index tree is divided into a replicated part (the upper levels of the tree) and a non-replicated part (the lower levels). The index tree is broadcast every $\frac{1}{d}$ of a bcast. Each broadcast consists of the replicated part and the non-replicated part that indexes the data items immediately following it. As such, each node in the non-replicated part appears only once in a bcast and, hence, reduces the replication cost and access latency while achieving a good tuning time.

Chen *et al.* and Shivakumar *et al.* considered unbalanced tree structures to optimize energy consumption for non-uniform data access [3, 17]. 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 [20]. Hashing and signature methods have also been suggested for wireless broadcast that supports equality queries [9, 12]. Hu *et al.* showed that the signature method is particularly attractive for multi-attribute indexing [7, 8]. However, none of these techniques is flexible in adjusting access latency and tuning time. Moreover, as they are extended from disk-based environments, which enable 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 value to the segment containing that key, and a *local index*, which is an m-entry index mapping a given key value to the buckets 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.

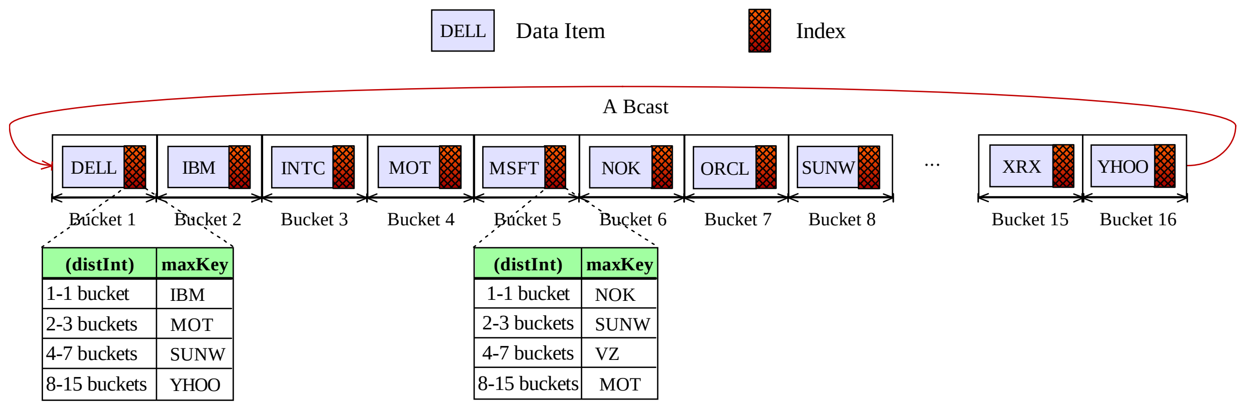
The proposed exponential index enhances the flexible index in at least three aspects: 1) instead of binary spaced indexing, the exponential index allows indexing spaces to be exponentially partitioned at any base value; 2) the exponential index intelligently exploits the available bucket space for indexing, whereas the flexible index blindly incurs overhead; and 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.

Khanna *et al.* [5] considered broadcast scheduling and indexing in an integrated fashion. In their approach, the broadcast sequence of data items is first scheduled based on their access rates. An index tree of the broadcast sequence is then built in a bottom-up manner. Since the broadcast sequence may not be key-ordered, the authors proposed to store the range of the keys appearing in the subtrees (rather than the maximum key only) at the internal nodes. To make the range representation more accurate, they assumed that the keys of data items are designated by the server and made known to the clients. However, this does not suit the practice in which the keys are predetermined and represent some semantics of data items and, thus, the applicability of their approach is limited.

Other related work includes data scheduling [4, 14], semantic broadcast [11], hybrid broadcast [6], cache management [22, 24], and broadcast of location-dependent data [23]. These studies complement our studies in different aspects.

3. THE EXPONENTIAL INDEX

This section presents a new air indexing method, called the *exponential index*. We focus on clustered broadcast in



(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

Figure 2: A Simple Exponential Index

this section and shall extend the proposed index to non-clustered broadcast in Section 5.1. 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.[2] As shown in Figure 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 the 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 tunes into the broadcast channel and first retrieves the index table in bucket 1 (i.e., the left index table in Figure 2). Since "NOK" falls between the second $maxKey$ "MOT" and the third $maxKey$ "SUNW," the target item must lie in the buckets that are 4 to 7 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 Figure 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 3 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 such a distributed index is $\lceil log_2(N-1)+1\rceil$ buckets, where N is the total number of buckets in a bcast. In our example, where $N = 16$, the worst tuning time is 5 buckets.

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 tuning time is logarithmically proportional to the bcast length.

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 2 (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 Figure 3, the ith entry in the index table describes the maximum key value of a segment of around r^{i-1} buckets (i.e., the buckets that are $\lfloor\sum_{j=1}^{i-2} r^j + 1\rfloor = \lfloor\frac{r^{i-1}-1}{r-1} + 1\rfloor$ 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

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

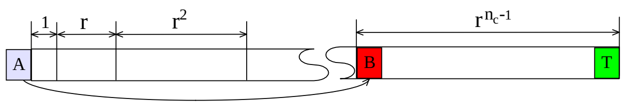


Figure 3: Illustration of Exponential Indexing Space

is reduced, thereby increasing the bcast length. To reduce the 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 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, the intra-chunk tuning time is either 1 (for the first bucket in the chunk) or 2 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. Figure 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. Assume that each data bucket includes an offset to the first bucket of the next chunk. The client access protocol follows the same three steps described in Section 2.1. We discuss the index search step 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 first bucket of chunk 3 (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 again 3 buckets.

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 Section 3.3.

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.

Algorithm 1 Index Search for the Exponential Index

```
1: wait until the first bucket of the next chunk is broadcast
2: for each data item in the bucket do
3:   if it is the requested data item then
4:     stop the search and the query is finished
5:   end if
6: end for
7: check the local index in the index table:
8: if the requested data item is within the key range specified by the $i$th entry then
9:   go into the doze mode, wait for the $i$th bucket, retrieve the data, and the query is finished
10: end if
11: check the global index in the index table:
12: if the requested data item is within the key range specified by the $i$th entry then
13:   go into the doze mode, wait for $(\lfloor \frac{r^{i-1}-1}{r-1} + 1 \rfloor \cdot I - 1)$ buckets to retrieve the first bucket of the $(\lfloor \frac{r^{i-1}-1}{r-1} + 1 \rfloor)$th chunk, and repeat this search procedure
14: end if
```

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
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

Table 1: Summary of Notations

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:

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

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

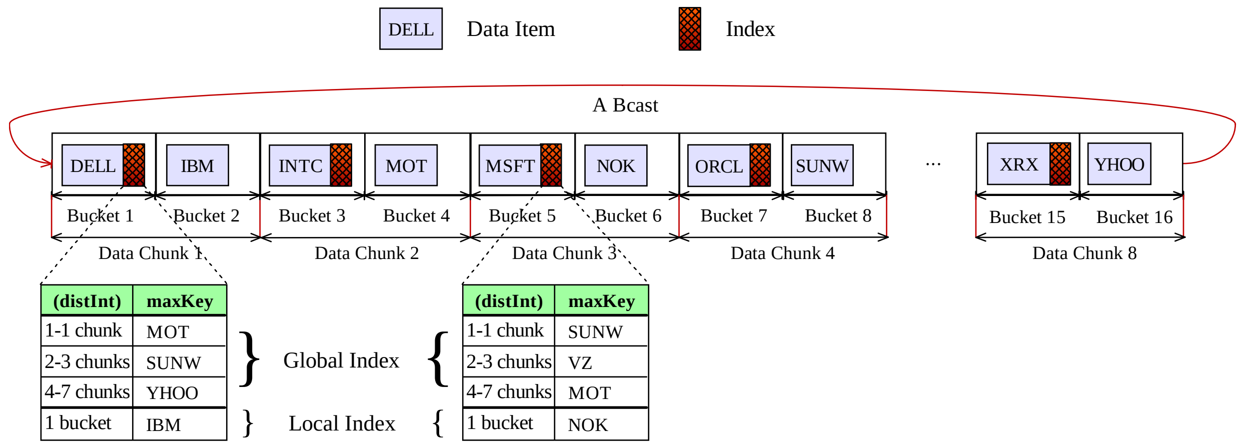


Figure 4: The Generalized Exponential Index (r=2, I=2)

obtain the number of global index entries:

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

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:

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

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

$$\sum_{i=0}^{n_c - 1} r^i = \frac{r^{n_c} - 1}{r - 1} \geq C - 1. \tag{4}$$

Therefore, the smallest value of r can be obtained by numerically solving the following inequality:

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

The average access latency is the initial index probe time plus half the bcast length, i.e.:

$$E(d) = \frac{I}{2} + \frac{IC}{2}. \tag{6}$$

Next, we derive the average tuning time. To do so, we derive the tuning time $t(l)$ for a data chunk that is l chunks away from the current chunk. This can be computed recursively:

$$t(l) = \begin{cases} 1, & \text{if } l = 0; \\ t(l - x) + 1, & \text{if } l > 0, \end{cases} \tag{7}$$

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

The average tuning time is thus given by:

$$E(t) = \frac{(I - 1)}{I} + \frac{B(I-1)}{B(I-1) + B'} + \frac{1}{C} \sum_{l=0}^{C-1} t(l), \tag{8}$$

where the first and second terms represent the average tuning time for locating the first bucket with an index table and the local bucket within a data chunk, respectively, and the third represents the average tuning time for locating the desired chunk. From (7) and (8), 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. Suppose the client initially tunes into data chunk A and is interested in some data item in chunk T (see Figure 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 Figure 3) and trim the search space by another factor of at least $\frac{r}{r-1}$ through the 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 certain performance requirements 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:

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

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

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

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

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

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 would like to investigate its performance compared with the state-of-the-art indexes (i.e., the distributed tree [10] and the flexible index [9]) as well as its ability to adjust the trade-off between access latency and tuning time.

We set the system parameters similar to those in [9, 10]. The database size ranges from 300 to 300,000 items. Flat broadcast is employed to broadcast the data items. We assume that the access distribution over the data items is uniform. For the exponential index and the flexible index, an index entry contains a key value only;[3] 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 for 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 unit of buckets. To make a fair comparison, the access latency of an indexing scheme shown in the results is normalized by the latency of a non-index scheme, i.e., $\lceil\frac{N}{2B}\rceil$. The results reported for the exponential index were obtained based on the analysis presented in Section 3.

[3]In the original proposal of the flexible index [9], an index entry also contains an offset, which is not necessarily included and can be derived from the number of buckets in a bcast.

Parameter	Setting	Parameter	Setting
N	300 – 300,000	B	10, 80
s_o	16, 128 bytes	s_e	4 bytes

Table 2: Parameter Settings

4.1 Comparison with the Distributed Tree Index

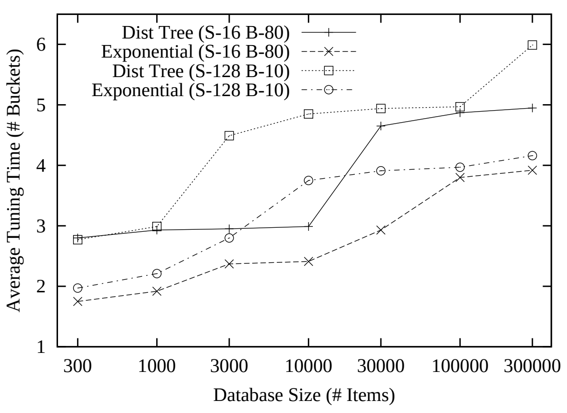
This set of experiments compares the proposed exponential index to the distributed tree, which is a non-flexible scheme. To compare the tuning time, we first compute the performance of the distributed tree given the system settings; we then obtain the best tuning time for the exponential index by tuning the index base and chunk size such that its access latency is no higher than that of the distributed tree. Figures 5a and 5b respectively show the average tuning time and the worst tuning time under different database sizes. As expected, the tuning time worsens with increasing database size for both index schemes. The exponential index performs no worse than the distributed tree in the worst cases and achieves a much better average tuning time for all database sizes tested.

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 Figure 5c, the exponential index outperforms the distributed tree in all cases. In particular, the performance improvement is up to 60% when the database size is small. This is mainly because at small database sizes, the index tree is degenerated into a single bucket and, hence, we cannot partially replicate the index tree to reduce the access latency.

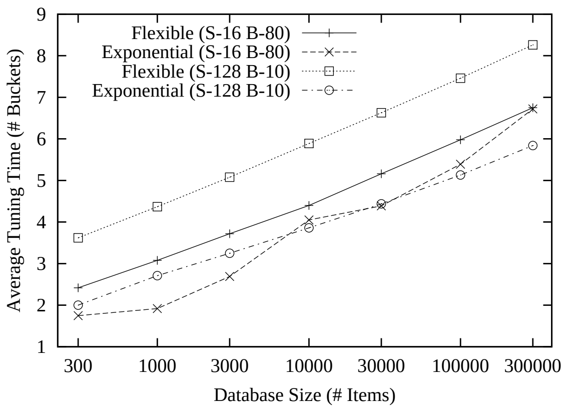
4.2 Comparison with the Flexible Index

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

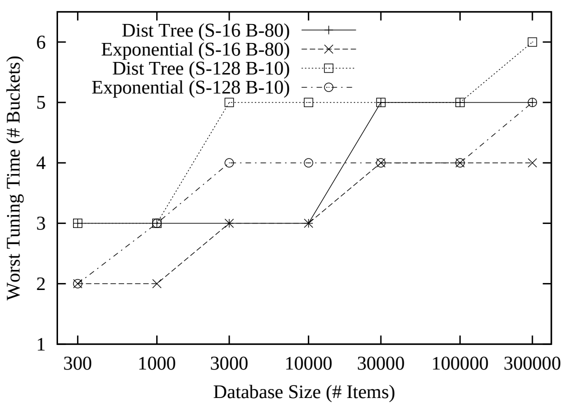
As shown in Figure 6, under the same access latency, the exponential index consistently outperforms the flexible index in terms of the tuning time. The improvement is more significant for the setting of item size 128 bytes, bucket capacity 10 (i.e., S-128, B-10) than the setting of item size 16 bytes, 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



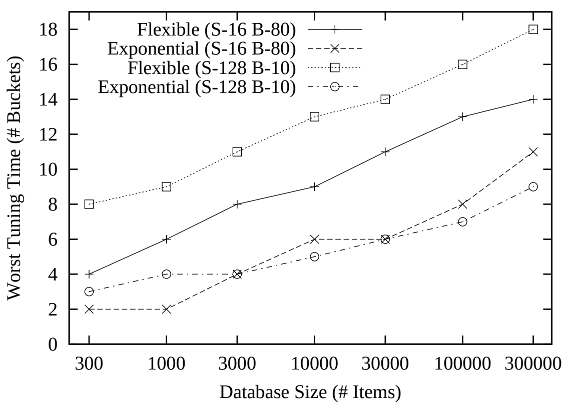
(a) Average Tuning Time



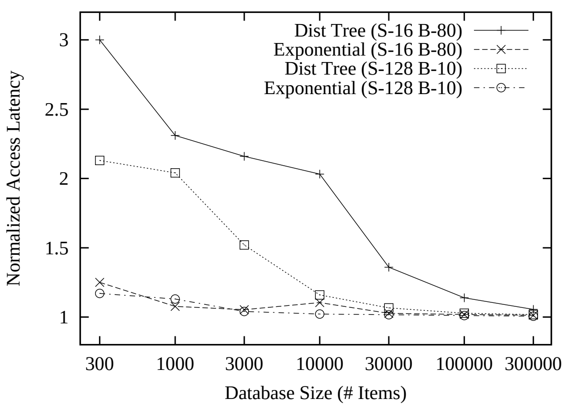
(a) Average Tuning Time



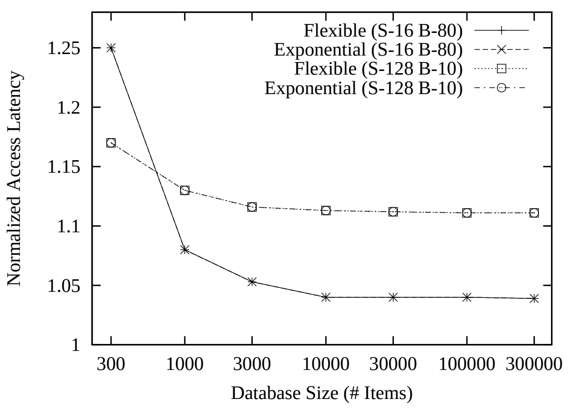
(b) Worst Tuning Time



(b) Worst Tuning Time



(c) Normalized Access Latency

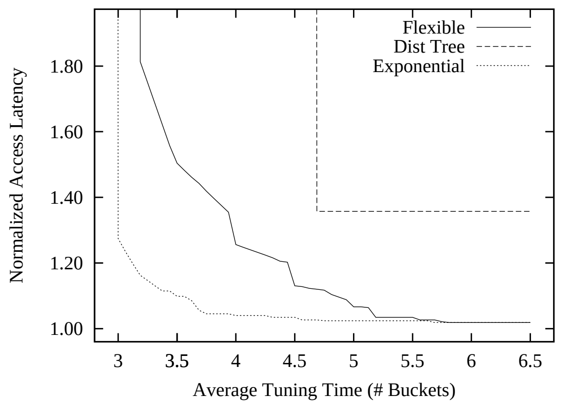


(c) Normalized Access Latency

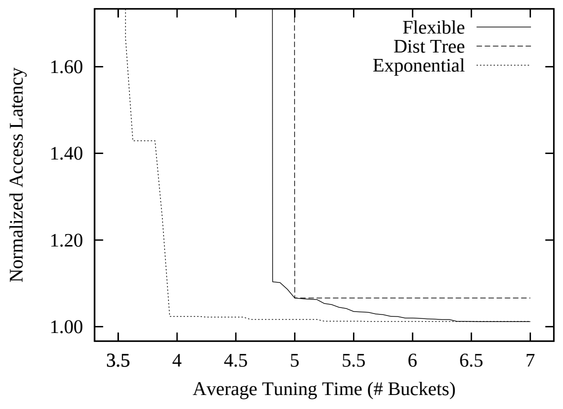
Figure 5: Comparison with the Distributed Tree Index

Figure 6: Comparison with the Flexible Index (Chunk Size=1 Bucket)

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.



(a) Item Size=16 bytes, Bucket Capacity=80



(b) Item Size=128 bytes, Bucket Capacity=10

Figure 7: Access Latency with Bounded Tuning Time (Database Size=30,000)

Hence, at large item sizes, 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. We set the database size at 30,000 items. First, we look at the tuning-time-bounded tuning problem. It is desirable that the longer is the tuning time allowed, the shorter is the access latency achieved. Figures 7a and 7b show the results for the settings of S-16, B-80 and S-128, B-10, respectively. As expected, the distributed tree is not flexible: it is impossible for it to achieve a tuning time shorter than 4.6 buckets and 5.0 buckets for these two settings, respectively, and the latency remains the same after these two points. While the flexible index is able to trade the latency for the tuning time requirement, obviously the exponential index performs even better: with the same tuning time requirement, the exponential index achieves a shorter (or the same) access latency. In particular, for the setting of S-128, B-10, the flexible in-

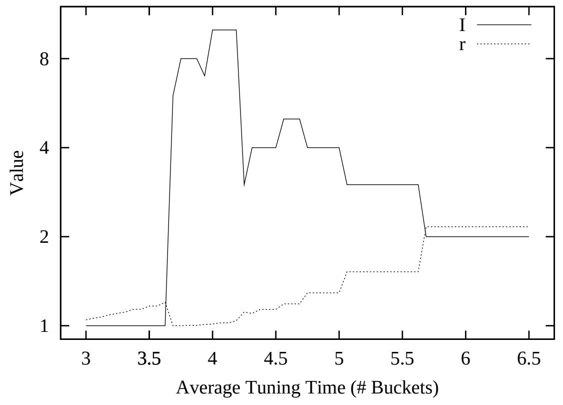
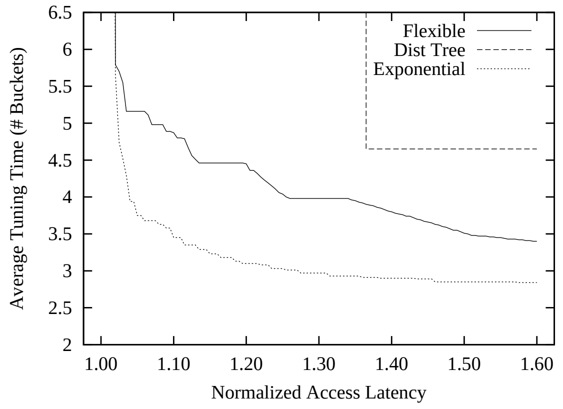
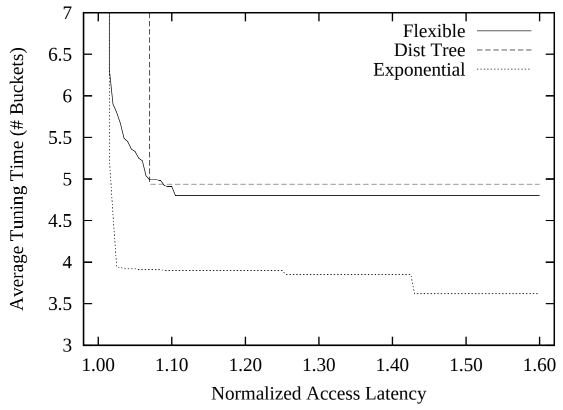


Figure 8: The I and r Values Set in the Exponential Index with Bounded Tuning Time (Item Size=16 bytes, Bucket Capacity=80)



(a) Item Size=16 bytes, Bucket Capacity=80



(b) Item Size=128 bytes, Bucket Capacity=10

Figure 9: Tuning Time with Bounded Access Latency (Database Size=30,000)

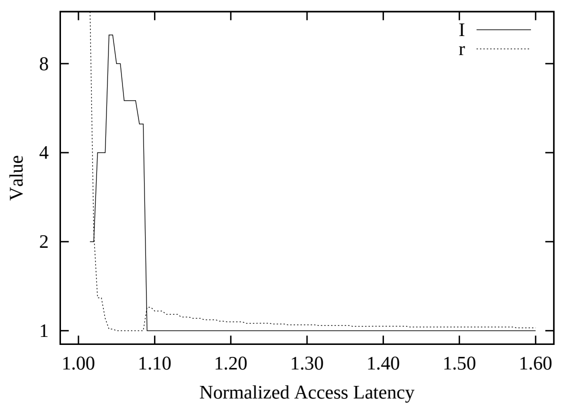


Figure 10: The I and r Values Set in the Exponential Index with Bounded Access Latency (Item Size=16 bytes, Bucket Capacity=80)

dex cannot manage a tuning time shorter than 4.8 buckets, whereas the exponential index can achieve a tuning time of less than 4 buckets at 3% access latency overhead on top of the non-index scheme.

To gain more insight into the exponential index, we show in Figure 8 the values of chunk size I and index base r selected for each bounded tuning time. To achieve a tuning time smaller than 3.6, I is set at 1 and a small value of r (less than 1.2) is employed. In these cases, the access latency overhead is a little bit high (see Figure 7a). By relaxing the tuning time, the indexing overhead, in particular the global index size, and, hence, the access latency overhead, can be reduced by increasing I and r alternatively. It is interesting to note that to obtain the least indexing overhead with a bounded tuning time, an increase of r generally leads to a decrease of I, and vice versa.

Next, we examine the latency-bounded tuning problem. We expect to achieve a shorter tuning time by tolerating a longer latency. As shown in Figures 9a and 9b, the distributed tree only obtains a better performance than the flexible index at normalized access latencies of 1.07–1.09 for the setting of S-128, B-10. Again, the proposed exponential index performs the best throughout the range of bounded latencies tested. 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 (i.e., S-128, B-10).

Figure 10 shows the values of I and r selected by the exponential index for each bounded access latency. At a small access latency overhead (less than 10% of the non-index scheme), a large value of I or r is required. When a higher access latency overhead is allowed, the values of I and r are decreased to achieve the best tuning time.

5. PRACTICAL ISSUES

5.1 Non-Clustered Broadcast

The previous sections have focused on clustered broadcast, which is capable of indexing the primary attribute in flat broadcast. We now extend the proposed exponential index to non-clustered broadcast, which is useful for indexing secondary attributes and skewed broadcast.

As discussed in Section 2, for non-clustered broadcast, a bcast can be partitioned into a number of clustered segments (i.e., meta-segments). The number of meta-segments 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 meta-segment. Similar to clustered broadcast, the exponential index can be applied to each meta-segment. Instead of indexing a whole bcast, each index table for non-clustered broadcast describes the buckets up to the farthest one in the next meta-segment whose attribute value is less than that of the current bucket. In the example shown in Figure 11, 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 meta-segment. The maximum number of meta-segments 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 buckets in a meta-segment.

5.2 Mode Transition Time

Recall that each index entry in the exponential index implicitly contains a distance offset. The client relies on the distance offsets to selectively tune in the broadcast. For the index shown in Figure 2, suppose a client wants to search "MSFT" right before "DELL" (i.e., bucket 1) is broadcast. By accessing the first bucket, the client finds that the next segment to search is 4 buckets away and, thus, switches to the doze mode to save energy. Ideally, the client can sleep in the doze mode for $d \times T_b$ time and wake up afterwards, where d is the distance to the next bucket to search and T_b is the time taken to broadcast a bucket. However, in determining the sleeping time, we need to take into consideration the mode transition time, in particular the delay caused by switching from the doze mode to the active mode. Therefore, the sleeping time should be set at $d \times T_b - T_w$, where T_w is the mode transition delay. In case the computed sleeping time is a negative value, the client should stay active to wait for the arrival of the next bucket to search. Fortunately, the mode transition delay T_w is generally negligible compared to the bucket broadcast time T_b. For example, in GPRS/GSM, T_w and T_b are on the order of 1 ms and 100 ms, respectively [2, 10].

5.3 Index Computation Time

With the exponential index, after downloading an index bucket, the client computes the distance offset to the next bucket to search and then determines the sleeping time. We can simply apply the binary search over the local index, and the global index if necessary, to compute the distance offset. There is some computational delay. As such, the client cannot go to the doze mode immediately after accessing an index bucket, because otherwise it may miss the next bucket to search if it happens to be the next bucket to broadcast. Instead, the client should stay on until the sleeping time is obtained. Due to the simple structure of the exponential index, this delay is insignificant. As observed in our experiments, for a database size of 300,000 items, the average delay is about 3 μs on a Pentium 4 1.8GHz CPU, which is much shorter than the bucket broadcast time of about 120

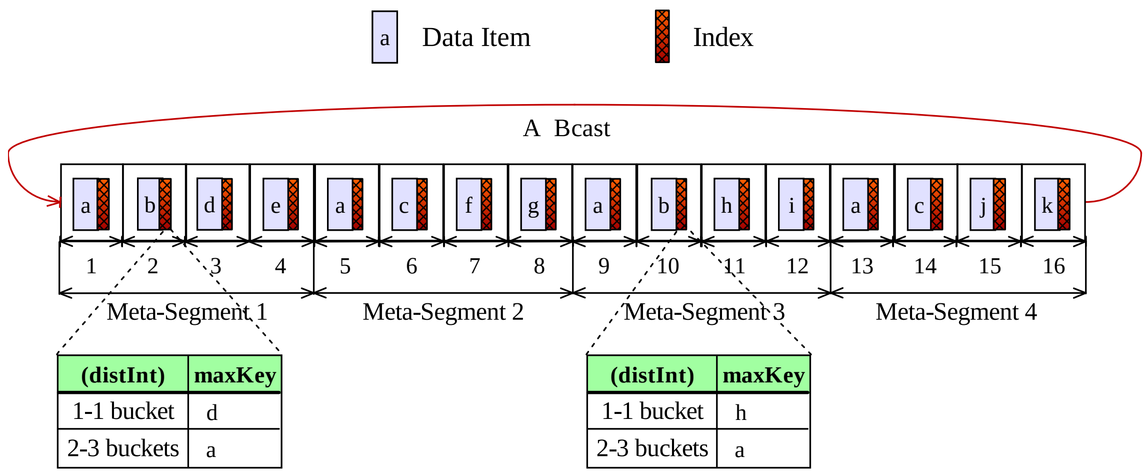


Figure 11: Indexing Non-Clustered Broadcast (r=2, I=1)

ms with a data rate 115 Kbps in GPRS [2].

5.4 Link Errors

Wireless environments are error prone. A bucket might be corrupted during broadcasting. We are able to handle link errors easily with the exponential index due to its distributed structure. Using Figure 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 in Section 3.1. However, if bucket 1 is corrupted, the client can immediately restart 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 can restart the search from bucket 3 and access buckets 5 and 6. Hence, there is only a small performance penalty. This is indeed an advantage over the tree index, where, if the root is corrupted, the client has to wait until the next root is broadcast before restarting the search, thus incurring additional access latency.

5.5 Data Updates

Periodic broadcast is targeted at applications such as stock quotes, airline schedules, and traffic information. The data values (e.g., stock prices) may change frequently. If the search key values (e.g., stock ids) are not updated, we need not update the index structure. If the search key values are changed, we shall reorganize the bcast structure as well as the index structure. Therefore, we expect the exponential index, like other existing indexes, works well for applications where the non-search key values may change but the search key values do not change often.

6. CONCLUSIONS

Data broadcast has been considered an attractive information dissemination method in pervasive computing environments due to its scalability and low cost. This paper has investigated the use of air indexing techniques to improve the efficiency of energy consumption on mobile devices in a broadcast environment. Several air indexing techniques had been developed for wireless data broadcast. However, most of the existing indexing techniques, based on centralized tree structures, are not flexible in adjusting the trade-off between access efficiency and energy conservation based on application specific requirements. Thus, in this study, we have focused on the problems of latency-bounded tuning and tuning-time-bounded tuning. To the best of our knowledge, this is the first comprehensive study that addresses these two tuning problems.

We have proposed a novel parameterized index scheme, called the exponential index, to meet different application requirements on access latency and tuning time. The exponential index is very efficient because it naturally facilitates the index replication by sharing links in different search trees and therefore minimizes storage overhead. Moreover, 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 quickly from an arbitrary index table in the broadcast. The energy consumption of mobile clients is also very efficient (i.e., the tuning time is logarithmically proportional to the bcast length). Finally, 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 measure 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) by tuning the parameters of the index base and chunk size. Through experiments, we have demonstrated that the exponential index outperforms two state-of-the-art air indexing schemes, the distributed tree and the flexible index, in terms of the access latency and tuning time. In addition, the results show that the exponential index can achieve great flexibility in adjusting the trade-off between access latency and tuning time.

We are building a prototype in a wireless LAN environment and plan to evaluate the performance of the proposed exponential index in a real environment. The proposed index can adjust the trade-off between access latency and tuning time on a per-application basis. We plan to investigate per-client flexible indexes. We also plan to employ the exponential index for location-based services in pervasive computing environments. We are particularly interested in exploiting the processing of window queries and nearest neighbor search in wireless data broadcast environments. In addition, we are interested in exploring the research issues of balancing access latency and tuning time in a multi-channel data broadcast environment.

7. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments, which improved the presentation of this paper. Special thanks go to Prof. Michael Franklin at UC Berkeley, who provided many constructive suggestions in the revision of this paper.

8. REFERENCES

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 199–210, San Jose, CA, May 1995.

[2] J. Cai and D. J. Goodman. General Packet Radio Service in GSM. *IEEE Communications Magazine*, 35(10):122–131, October 1997.

[3] M.-S. Chen, K.-L. Wu, and P. S. Yu. Optimizing index allocation for sequential data broadcasting in wireless mobile computing. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(1):161–173, January/February 2003.

[4] C.-L. Hu and M.-S. Chen. Dynamic data broadcasting with traffic awareness. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'02)*, pages 112–119, Vienna, Austria, July 2002.

[5] S. Khanna and S. Zhou. On indexed data broadcast. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC'98)*, pages 463–472, Dallas, TX, 1998.

[6] Q. L. Hu, D. L. Lee, and W.-C. Lee. Performance evaluation of a wireless hierarchical data dissemination system. In *Proceedings of the 5th Annual ACM International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 163–173, Seattle, WA, August 1999.

[7] Q. L. Hu, W.-C. Lee, and D. L. Lee. Power conservative multi-attribute queries on data broadcast. In *Proceedings of the 16th International Conference on Data Engineering (ICDE'00)*, pages 157–166, San Diego, CA, February 2000.

[8] Q. L. Hu, W.-C. Lee, and D. L. Lee. A hybrid index technique for power efficient data broadcast. *Distributed and Parallel Databases (DPDB)*, 9(2):151–177, March 2001.

[9] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Power efficient filtering of data on air. In *Proceedings of the 4th International Conference on Extending Database Technology (EDBT'94)*, pages 245–258, Cambridge, UK, March 1994.

[10] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Data on air – Organization and access. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 9(3):353–372, May/June 1997.

[11] K. C. K. Lee, H. V. Leong, and A. Si. A semantic broadcast scheme for a mobile environment based on dynamic chunking. In *Proceedings of 20th IEEE International Conference on Distributed Computing Systems (ICDCS'00)*, pages 522–529, Taipei, Taiwan, April 2000.

[12] W.-C. Lee and D. L. Lee. Using signature techniques for information filtering in wireless and mobile environments. *Journal of Distributed and Parallel Databases (DPDB), Special Issue on Database and Mobile Computing*, 4(3):205–227, July 1996.

[13] V. Liberatore. Caching and scheduling for broadcast disk systems. Technical Report 98-71, Institute for Advanced Computer Studies, University of Maryland at College Park (UMIACS), December 1998.

[14] V. Liberatore. Multicast scheduling for list requests. In *Proceedings of IEEE INFOCOM'02*, pages 1129–1137, New York, NY, June 2002.

[15] DirectBand Network. Microsoft Smart Personal Objects Technology (SPOT). [Online]. Available: http://www.microsoft.com/resources/spot/.

[16] E. Shih, P. Bahl, and M. J. Sinclair. Wake on wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th Annual ACM International Conference on Mobile Computing and Networking (MobiCom'02)*, pages 160–171, Atlanta, GA, September 2002.

[17] N. Shivakumar and S. Venkatasubramanian. Energy-efficient indexing for information dissemination in wireless systems. *ACM/Baltzer Journal of Mobile Networks and Applications (MONET)*, 1(4):433–446, December 1996.

[18] StarBand. [Online]. Available: http://www.starband.com/.

[19] Hughes Network Systems. DIRECWAY homepage. [Online]. Available: http://www.direcway.com/.

[20] K. L. Tan and J. X. Yu. Energy efficient filtering of nonuniform broadcast. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96)*, pages 520–527, Hong Kong, May 1996.

[21] M. A. Viredaz, L. S. Brakmo, and W. R. Hamburgen. Energy management on handheld devices. *ACM Queue*, 1(7):44–52, October 2003.

[22] 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 Transactions on Knowledge and Data Engineering (TKDE)*, 16(1):125–139, January 2004.

[23] J. Xu, B. Zheng, W.-C. Lee, and D. L. Lee. Energy efficient index for querying location-dependent data in mobile broadcast environments. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE'03)*, pages 239–250, Bangalore, India, March 2003.

[24] L. Yin, G. Cao, C. Das, and A. Ashraf. Power-aware prefetch in mobile environments. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'02)*, pages 571–578, Vienna, Austria, July 2002.