Efficient index caching for data dissemination in mobile computing environments

Jen-Jou Hung, Yungho Leu *

Department of Information Management, National Taiwan University of Science and Technology, Taipei 10672, Taiwan

Received 15 April 2004; received in revised form 11 March 2005; accepted 11 March 2005
Available online 2 June 2005

Abstract

Due to the limited power supply of mobile devices, much research has been done on reducing the power consumption of mobile devices in mobile computing environments. Since supporting indices on the broadcast data items can effectively reduce the power consumption of the mobile clients, most of the existing research on data broadcasting has focused on designing efficient indexing schemes. In this paper, we propose to cache indices on the mobile clients and to use cached indices to facilitate the access of the broadcast data items. To manage the cache usage of the mobile clients, we propose two cache management policies. The lower-level-index-first policy caches the index nodes that are at the lower level of the index tree, while the cut-plane-first policy caches the index nodes that belong to a cut-plane of the index tree. Through experiments, we compare the performance of the two proposed policies with some existing policies in terms of tuning time and access time. The experiments show that index caching significantly reduces the tuning time of a mobile client without increasing its access time. In terms of tuning time, the experiments show that, when the access pattern of a mobile client is not skew, the cut-plane-first policy outperforms the lower-level-index first policy, LRU and LRFU. On the contrary, when the mobile client has a small cache and its access pattern is skew, the lower-level-index-first policy outperforms the cut-plane-first policy, LRU and LRFU. In terms of access time, the lower-level-index first policy outperforms the cut-plane-first policy, LRU and LRFU.

© 2005 Elsevier Inc. All rights reserved.

Keywords: Index caching; Energy efficiency; Data broadcast; Power consumption

1. Introduction

Due to the advent of wireless networks, mobile computing has gained its popularity in the market. As a result, more and more users are able to access data from wireless networks. In the near future, people will not be confined to work in a fixed area. Instead, they can roam with a mobile client, which can be a small portable computing unit, to retrieve data anywhere and at any time.

Most of today’s mobile computing environments are designed in a way that one mobile server provides service for a large number of mobile clients. In such an environment, numerous mobile clients may overrun the mobile server, a phenomenon called the scalability problem. To deal with this problem, data broadcast technique was proposed. With the data broadcast technique, the mobile server continually broadcasts data items in a broadcast channel and the mobile clients access their required data items by listening to the broadcast channel. The tuning time and the access time are two performance measures in data broadcasting environments. The tuning time is defined to be the time required for a mobile client to tune in to the broadcast channel to download a data item (Imielinski et al., 1997). On the other hand, the access time is the time elapsed from the moment a mobile client requests a data item to the

* Corresponding author. Tel.: +886227376770; fax: +88622736777.
E-mail address: yhl@cs.ntust.edu.tw (Y. Leu).

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2005.03.006
moment when the required data item is downloaded by the mobile client. According to Imielski et al. (1997), the power consumption of a mobile client in downloading a data item is in proportion to tuning time. To reduce the access time of a mobile client, Acharya et al. (1995), Vaidya and Hameed (1996, 1997) and Jiang and Vaidya (1999) proposed several algorithms for designing the optimal broadcast schedule. Su and Tassiulas (2000) proposed a scheduling algorithm which took into account the access patterns and the caching policies of the mobile clients.

Caching of data items in a mobile client is also considered for reducing tuning time and access time (Acharya et al., 1995; Acharya et al., 1996; Su and Tassiulas, 2000; Yajima et al., 2001). However, in real life applications, the contents of the data items change frequently. This nullifies the effect of data caching. Compared with the contents of the data items, the broadcast structure is not changed so frequently. For example, the stock price of a company in a stock market may change in every second, while the list of companies in the stock market changes at most once a day. In this paper, we study the ways of reducing tuning time by caching indices. Having cached indices on a mobile client, the mobile client can use the cached indices instead of the indices from the broadcast channel to access its required data items. Preventing access indices from the broadcast channel can effectively reduce tuning time. Caching indices on a mobile client necessitates the management of cache space of the mobile client. Though many existing cache management policies such as LFRU and LRU can be readily used, they do not take the index structure of the broadcast data item into account. As a result, the existing policies may not be suitable for index caching. In this paper, we propose two new cache management policies. The LLIF policy caches the leaf index nodes of the index tree, and the CPF policy caches a cut-plane of the index tree. We perform extensive experiments to study the performance of cache management policies in terms of tuning time and access time. The contribution of this paper include the following: (a) we proposed the way of using cached indices to access broadcast data items to reduce tuning time, and (b) we proposed two new cache management policies and studied their performance through simulation. In Hung and Leu (2003), we briefly described the way of using a cached index to reduce the tuning time of a mobile client. In this paper, we give the details of the proposed policies and perform an extensive study on the performance of the proposed policies.

The rest of this paper is organized as follows. In Section 2, we review the related concept of data broadcasting. In Section 3, we propose the way of using cached indices to reduce tuning time. In Section 4, we present the new policies for the cache management of the mobile clients. In Section 5, we compare the performance of the cache management policies. Finally, we conclude this paper and outline the directions for future research in Section 6.

2. Preliminaries

2.1. The broadcast structure

In a data broadcasting environment, each broadcast data item is identified by a key which is a unique integer number (Imielski et al., 1997). Data items are broadcasted according to a broadcast structure. The smallest unit of broadcasting in the broadcast structure is a bucket. Since the buckets are of the same size, it takes the same time to transmit any two different buckets. Therefore, a bucket is usually used as a unit of time in the literature (Imielski et al., 1997).

Two types of buckets exist in a broadcast structure—data buckets and index buckets. A broadcast cycle, also called a bcast, consists of a sequence of index buckets and data buckets. The time for broadcasting a complete bcast is called a bcast_length. Each bucket in a broadcast structure contains a header with the following information: a bucket_id, a bcast_pointer, an index_pointer and a bucket_type. The bucket_id is the bucket identifier used to identify this bucket, which is defined to be the offset from the beginning of the bcast to this bucket. The bcast_pointer, used as an index to the next bcast, is defined to be the offset from this bucket to the beginning of the next bcast. The index_pointer, used as an index to the next index segment, is defined to be the offset from this bucket to the beginning of the next index segment. Finally, the bucket_type indicates whether this bucket is a data bucket or an index bucket.

2.2. Index organization

An index structure is shown in Fig. 1. The bottommost level of the index structure contains data buckets only. Above the data buckets is an index tree. To access a data item, a mobile client first retrieves the root, R, and then finds a path to the leaf index node that contains the index information for the desired data item.

Example 1. To access the 16th data item, denoted by item\textsubscript{16}, in Fig. 1, a mobile client has to access the root R, a\textsubscript{1}, b\textsubscript{2}, c\textsubscript{6} and item\textsubscript{16} in the listed order.

To ease the following discussion, we shall call the set of data items that can be indexed by an index node the range of the index node.

Example 2. The range of a\textsubscript{1} contains item\textsubscript{1} through item\textsubscript{27}, that is, a mobile client has to access a\textsubscript{1} before it can retrieve any data item from item\textsubscript{1} to item\textsubscript{27}. 
In a broadcast environment, data are disseminated sequentially according to a broadcast schedule. Fig. 2(a) is a broadcast schedule for the indices and data items in Fig. 1. An index bucket contains a sequence of index entries arranged in ascending order of the Attribute.Key (Imielinski et al., 1997). An index entry is a pair of (Attribute.Key, Offset). To access a data item with attribute_key X, the mobile client searches in the index node for entry\[i\], the \[i\]th entry in the index bucket, where entry\[i\].attribute_key \leq X < entry\[i + 1\].attribute_key. Then, the mobile client follows the offset of entry\[i\], denoted by offset\[i\], to retrieve the next node (could be another index node or the desired data item). We give the following example to illustrate the contents of an index node.

Example 3. Referring to Fig. 2(b), node R has three children, \(a_1\), \(a_2\) and \(a_3\). The bucket_ids of \(a_1\), \(a_2\) and \(a_3\) are 2, 43, and 84, respectively. The bucket id of R is 1. The range of \(a_1\) is \([1, \ldots, 27]\). The range of \(a_2\) is \([28, \ldots, 54]\). Finally, the range of \(a_3\) is \([55, \ldots, 81]\). The index entries contained in R are \{(1,1), (28,42), (55,83)\}. If a mobile client wants to access any data item from item\(_1\) to item\(_{27}\), it has to access the first index entry of R. Then, it will switch itself to a doze mode until bucket \(a_1\) arrives.

The procedure for a mobile client to access a data item X is listed in the following:

1. The mobile client tunes in to the broadcast channel to fetch a complete bucket.
2. After receiving its first complete bucket, the mobile client switches itself to a doze mode and stays in the doze mode until the next R arrives.
3. After receiving R, the mobile client looks up in the index bucket for an index entry to the next node. Then, the mobile client switches back to doze mode again and waits for next node to arrive.
4. If the received node in step 3 is an index bucket, the mobile client goes to step 3; otherwise, the mobile client retrieves its desired data item and terminates the procedure.
3. Index caching

In this section, we first discuss how to use a cached index to reduce the tuning time. Then, we examine the dependency of the tuning time reduction and the location of an index node in an index tree.

3.1. Using a cached index

In order to access a data item, a mobile client needs to traverse the index tree from root \( R \) to the data item. This process incurs a lot of energy consumption in accessing related index nodes. If these index nodes are cached, then we can use the cached index nodes rather than the index nodes from the broadcast channel to access a required data item. This may significantly reduce the tuning time.

However, unlike an index node from the broadcast channel, an index node in the cache (i.e., a cached index node) cannot be used without modification. Note that the offset field of an index entry means that a mobile client can access the next related node in offset buckets\(^1\) away from the current bucket. However, when a mobile client tunes in to the broadcast channel, it is very unlikely that the time when the mobile client tunes in coincides with the time when the index node is broadcast. Therefore, the offset value cannot be used without modification. Fortunately, we can adjust this value to make it useful. The idea behind the adjustment is to map an offset value to a bucket_id in the \( \text{bcast} \) structure. To illustrate, we suppose that a mobile client has a cached index bucket, denoted by \( b_{\text{cached}} \), and it uses offset\([i]\) of \( b_{\text{cached}} \) to access the next related bucket. In the following, we use \( b_{\text{next}} \) to represent the next related bucket. The bucket_id of \( b_{\text{next}} \) can be calculated through the following equation:

\[
\text{bucket_id} = b_{\text{cached}} \cdot \text{offset}[i] + b_{\text{cached}} \cdot \text{bucket_id}.
\]

Example 4. Suppose that a mobile client wants to access \( \text{item}_{2} \) in Fig. 1. The mobile client needs to access \( R, a_{1}, b_{1}, c_{1} \). Assume that the mobile client has index \( a_{1} \) in its cache. According to Fig. 2, the bucket_id of \( a_{1} \) is 2, and the next related bucket is \( b_{1} \). In \( a_{1} \), the offset to \( b_{1} \) is 1. The bucket_id of \( b_{1} \) can be calculated as follows.

\[
\text{Bucket_id of } b_{1} = 2 + 1 = 3.
\]

Before using a cached index bucket, the mobile client has to access a complete bucket from the broadcast channel. To ease our discussion, we use \( b_{\text{access}} \) to represent the first complete bucket received by the mobile cli-ent. If the bucket_id of \( b_{\text{access}} \) is smaller than that of \( b_{\text{next}} \), this implies that bucket \( b_{\text{next}} \) has not been broadcast yet. The mobile client can, therefore, access \( b_{\text{next}} \) in this \( \text{bcast} \). That is, \( b_{\text{next}} \) will arrive at the mobile client in \( b_{\text{next}} \cdot \text{bucket_id} - b_{\text{access}} \cdot \text{bucket_id} \) time units. If the bucket_id of \( b_{\text{access}} \) is larger than that of \( b_{\text{next}} \), the mobile client misses the \( b_{\text{next}} \) in this \( \text{bcast} \), and it has to access \( b_{\text{next}} \) in the next \( \text{bcast} \). The offset in this case becomes \( b_{\text{next}} \cdot \text{bucket_id} + b_{\text{access}} \cdot \text{bcast_ptr} \). Bear in mind that the \( \text{bcast_ptr} \) is the offset from the current bucket to the beginning of the next \( \text{bcast} \). After adding \( b_{\text{access}} \cdot \text{bcast_ptr} \) to \( b_{\text{next}} \cdot \text{bucket_id} \), we derive the offset of the next related bucket. The following example illustrates the adjustment of offset.

Example 5. Fig. 3 shows two cases (case a and case b) in which a mobile client wants to access \( \text{item}_{2} \) (the second data bucket in Fig. 1), and the mobile client has index \( a_{1} \) in its cache. These two cases differ in the time when the mobile client first tunes in. In case a, the bucket_id of \( b_{\text{access}} \) is equal to 1, and the mobile client tunes in before the next related bucket, \( b_{1} \), is broadcasted; thus, the offset from \( b_{\text{access}} \) to \( b_{\text{next}} \) equals \( 3 \cdot 1 = 2 \). In case b, the bucket_id of \( b_{\text{access}} \) equals 15, and the mobile client tunes in after the next related bucket, \( b_{1} \), is broadcasted. The \( \text{bcast_ptr} \) of \( b_{\text{access}} \) in this case, equals 108. The mobile client has to access the next related bucket, \( b_{1} \), in the next broadcast cycle; thus, the offset from \( b_{\text{access}} \) to \( b_{\text{next}} \) equals \( 108 + 3 = 111 \).

3.2. Tuning time reduction and the location of an index node

The effectiveness of a cached index node depends on its location in the index tree. We will illustrate this using the following example.

Example 6. Assume that we are given an index structure as shown in Fig. 1, and a mobile client wants to access \( \text{item}_{16} \). We will compare the tuning time for accessing \( \text{item}_{16} \) in the following two different cases:

(a) The mobile client contains index \( a_{1} \) in its cache.
(b) The mobile client contains index \( c_{6} \) in its cache.

In case a, the mobile client has to proceed with the following procedure: \( \text{tunes in the broadcast channel}; \) accesses \( \text{item}_{2} \); accesses \( \text{item}_{16} \). The tuning time of retrieving the index nodes for accessing \( \text{Item}_{16} \) equals \( 1.5 + 2 = 3.5 \).

In case b, the mobile client has to proceed with the following procedure: \( \text{tunes in the broadcast channel}; \) accesses \( \text{item}_{16} \). The tuning time in this case equals \( 1.5 + 0 = 1.5 \).

\(^1\) Note that we use \( \text{bucket} \) as the unit of time in this paper.
Referring to Example 6, if the mobile client has enough memory to cache all its required index nodes, the tuning time will be reduced to 1.5, i.e., the optimal value.

4. Cache management

In the previous section, we assume that the cache size of a mobile client is enough to store all required index nodes. However, some mobile hosts, PDAs for example, may not have enough main memory to cache all the required index nodes. Thus, cache management policies have to be considered for mobile devices with limited main memory.

4.1. Cache management policies

From Example 6, we have the following observations that guide us to design suitable cache management policies:

Observation 1. To access a data item in the range of an index node, the index node is more effective than its ancestors in terms of tuning time reduction.

Observation 2. An index node will be less useful if any of its child nodes is also in the cache.

Observation 1 is evident from Example 6. Observation 2 is due to the fact that a child node can be used to access some of the data items in the range of its parent node. The following example illustrates this.

Example 7. Refer to Fig. 1. Assume that the mobile client has only the root $R$ in its cache. In this case, the mobile client can use $R$ to access any data item from $item_1$ to $item_{81}$. However, if the mobile client has also $a_1$ in its cache, it can use $a_1$ instead of $R$ to access any data item from it item$_1$ to item$_{27}$. This is because $a_1$ is at the lower level of the index tree and, therefore, is more effective than $R$. In this case, $R$ will only be used for accessing any data item from $item_{28}$ to $item_{81}$.

Based on these two observations, we present two cache management policies in the following.

4.2. The lower-level-index-first management policy

Observation 1 states that a lower level index node is more effective in tuning time reduction than an upper level index node. The Lower-Level-Index-First (abbreviated as LLIF) policy retains as many lower level index nodes as possible in the cache. In LLIF, the cache space are organized into two lists—the upper level list and the end list. Both lists are managed by the LRU replacement policy. An index node having no child in the cache is called an end index node. The end index nodes are stored in the end list. Nodes that are not end nodes are stored in the upper level list. On receiving a new index node from the channel, the mobile client first inserts the index node at the tail of the end list. Then, the mobile client moves the parent of the index node, if one exists, from the end list to the tail of the upper level list. This is because the parent node is no longer an end index node if any of its children is also in the cache. Note that if the parent index node of the received index node is in the end list, it must be at the tail of the end list. This is due to the fact that before the mobile client can access the index node, it must access the parent of the index node. Therefore, when a mobile client receives a new index node, it directly searches the tail of the end list for the parent of the new index node. LLIF dictates that if the cache is full when a new index node is received, the index node at the head of the upper level list is replaced. The following example illustrates the LLIF policy.

Example 8. Refer to the index structure in Fig. 1. Suppose that the cache size is 6, and a mobile client wants to access $item_{22}$ and $item_{34}$ in sequence. After receiving $item_{22}$, the contents of the end list and the upper level list are shown in Fig. 4(a). The scenario for the mobile client to access $item_{34}$ is described in the following procedures:

1. The mobile client receives $a_2$ and inserts it after the tail of the end list, as Fig. 4(b).
2. As in Fig. 4(c), the mobile client receives $b_4$; inserts $b_4$ after the tail of the end list. Then, the mobile client moves index $a_2$, the immediate parent of $b_4$, to the upper level list.
3. When $c_{12}$ is received, the cache is full; the mobile client deletes the index node from the head of the upper level list and inserts $c_{12}$ after the tail of the end list. This scenario is shown in Fig. 4(d).

\[ \text{In this paper, we assume that the cache space is dynamically allocated. However, the cache size limits the amount of memory space that can be allocated to a mobile client.} \]
Due to the fact that when a mobile client accesses a new index node \( x \) from the broadcast channel, it needs only to investigate the node at the tail of the end list to see whether \( x \)’s parent also exists in the end list. Therefore, the computational complexity of LLIF is \( O(1) \).

### 4.3. The cut-plane-first management policy

Observation 2 states that in order to use the cache space efficiently, a mobile client should prevent the ranges of any two cached-index nodes from overlapping. In the following, we present a cache management policy, called the cut-plane first (abbreviated as CPF) policy, that efficiently uses the cache space by avoiding the ranges of the cached index nodes from overlapping.

The CPF policy is based on the concept of cut-planes. Basically, a cut-plane is a set of index nodes of the index tree which represents a horizontal cut of the index tree. As discussed above, to maximize the utilization of the cache space, one should prevent the ranges of any two index nodes from overlapping. To this end, we define a cut-plane as a set of index nodes which satisfies two conditions: (1) the union of the ranges of the index nodes in a cut-plane should contain all data items in the broadcast; and (2) the ranges of any two index nodes in a cut-plane do not overlap.

#### Example 9

Refer to Fig. 5. \( S_1 \) contains \( R, a_1, a_2, b_1, b_2, b_3, b_4, c_7 \) and \( c_8 \). \( S_1 \) is not a cut-plane since the ranges of some index nodes are not disjoint. \( S_2 \), which contains \( b_1, b_2, b_3, c_7 \) and \( c_8 \), is a cut-plane. \( S_2 \) has the same index power as that of \( S_1 \), but with fewer index nodes.

The CPF policy can be stated in the following.

**The CPF policy:** A cached index node which does not belong to the current cut-plane will be replaced when the cache is full.

In the following, we present an algorithm to implement the CPF policy. The algorithm consists of two phases—cut-plane initialization and cut-plane adjustment.

#### 4.3.1. Phase 1: Cut-plane initialization

In the CPF policy, the cache space of a mobile client is organized into two lists—the cut-plane list and the non-cut-plane list. The cut-plane list (abbreviated as the CP list) stores the cached index nodes that belong to the current cut-plane, while the non-cut-plane list (abbreviated as the NCP list) stores the index nodes that do not belong to the current cut-plane. Initially, both the CP list and the NCP list are empty. When \( R \), the root of the index tree, is received, the mobile client inserts \( R \) into the CP list which constitutes the first cut-plane. When a new index node \( x \) is received, the mobile client first inserts it into the NCP list. If \( x \)’s parent is in the CP list, the mobile client searches for a possible substitution of \( x \)’s parent by \( x \) and \( x \)’s sibling in the NCP list. A node \( i \) in the CP list is replaced by its child index nodes in the NCP list if all of its child nodes exist in the NCP list. To perform the substitution, the mobile client removes node \( i \) from the CP list, and moves the child nodes of node \( i \) from the NCP list to the CP list. The following algorithm (Fig. 6) summarizes the procedure of cut-plane initialization.

We use the following example to illustrate the cut-plane initialization phase.

**Example 10.** Refer to the index tree in Fig. 5. Assume that a mobile client has \( R \) and \( a_1 \) in its cache. \( R \) is in the CP list and \( a_1 \) is in the NCP list. When \( a_2 \) is received, the mobile client inserts \( a_2 \) into the NCP list; Fig. 5 shows that \( a_1 \) and \( a_2 \) constitute the set of all the children of \( R \). \( R \) is therefore replaced by \( a_1 \) and \( a_2 \).

Note that the number of index nodes in a cut-plane increases as the initialization phase continues. It will come to a stage that for any node in the CP list the free space plus the space occupied by the NCP list is not enough to allocate all of its child nodes. In this case, the cut-plane initialization phase terminates, and the CPF policy proceeds with a fine-tuning phase called the cut-plane adjustment phase.

#### 4.3.2. Phase 2: Cut-plane adjustment

The cut-plane adjustment is based on the concept of access probability of an index node. The access probability of an index node is defined to be the probability...
of using this index node to access a data item from the broadcast channel. Accordingly, the access probability of a non-leaf index node is equal to the summation of the access probability of all of its child index nodes. Recall that, from observation 1, a lower level index node is more effective than an upper level index node in terms of tuning time reduction. To reduce the tuning time, we would choose to cache the lower level index nodes for data items with high access probability. On the other hand, to reserve the cache space for index nodes with high access probability, we would reduce the cache space used for index nodes with low access probability. One way to achieve this is to cache the upper level index nodes for data items with low access probability. The following example illustrates this.

Example 11. Fig. 7(a) shows the cache content of a mobile client. Suppose that the access probability of $a_2$ is 0.3, while the summation of access probabilities of $b_5$ and $b_6$ is 0.1. To reduce tuning time, the mobile client should cache $a_3, b_3$ and $b_4$ instead of $a_2, b_3$ and $b_6$, as illustrated in Fig. 7(b). By replacing $b_5$ and $b_6$ with $a_3$, the mobile client needs to spend one extra time unit (in terms of bucket) in accessing data items indexed by either $b_5$ or $b_6$. On the other hand, by replacing $a_2$ with $b_3$ and $b_4$, the mobile client saves one unit of time in accessing the data items indexed by either $b_3$ or $b_4$. The expected saving in the replacement is $1 * 0.3 - 1 * 0.1 = 0.2$ time unit.

The adjustment procedure consists of two steps. The first step is to identify the set of index nodes to be replaced and the set of index nodes to be replaced with. The second step carries out the replacement. The procedure of the first step is shown in the following:

1. Search the CP list for an index node that is not a leaf node in the index tree. If more than one such index node is found, choose the one with the highest access probability. Let $i$ denote the node that is found.
2. Search the CP list for a set of index nodes which constitute the set of all immediate children of any index node. If more than one of such sets are found, choose the one with the lowest total access probability. Let $s$ denote this set.
3. If the access probability of $i$ is higher than the summation of the access probabilities of the index nodes in $s$, replace $i$ with its children, and replace the nodes in $s$ with the parent of $s$.

In the second step, the mobile client waits for the parent of the nodes in $s$ to be broadcast in the broadcast channel. After receiving the parent node, the mobile client removes all the nodes in $s$ from the CP list and inserts the parent node of $s$ into the CP list. Subsequently, the mobile client retrieves the child nodes of $i$ from the broadcast channel and inserts them into the CP list. When all the child nodes of $i$ are in the CP list, the mobile client removes $i$ from the CP list and terminates this round of adjustment. The algorithm of cut-plane adjustment is shown in Fig. 8.

Once the current round of adjustment completes, the mobile client starts another round of adjustment.

Note that, in any phase of the algorithm, when the cache is full and a new index node is received, the head of the NCP list is replaced.
4.3.3. The computational complexity of the CPF algorithm

The computational complexity of an index caching replacement algorithm is to compute the time complexity of the algorithm in selecting a victim in the cache to make room for an incoming index node. For the proposed CPF algorithm, the replacement can happen in either of the two phases. When the replacement happens in the cut-plane initialization phase, the mobile client needs to search in the CP list for the parent of the incoming index node, and to search in the NCP list for all the sibling node of the incoming index node. The worst case is that the mobile client needs to compare all the cached index nodes to find the parent and the siblings of the incoming index node. The time complexity of this case is $O(n)$, where $n$ is the size of the cache.

When the replacement happens in the cut-plane adjustment phase, the mobile client needs to search in the CP list for an index node which is not a leaf and has the highest access probability. Since the length of the CP list is at most $n$, the time complexity of this operation is $O(n)$. In addition, the mobile client needs to search in the CP list for a set of siblings with the lowest total access probability. This operation needs to group the cached index nodes by different parents, and to calculate the access probability of each group. The time complexity of this operation is $O(n \log n)$. In summary, the complexity of the replacement operation in the cut-plane adjustment phase is $O(n \log n)$.

5. Experimental results

In this section, we compare the performance of the proposed policies through simulations. Each simulation consists of a software module for the broadcast server, a module for a mobile client, and a module for the broadcast channel. In the simulation, we assume that the data items are indexed by the EPR indexing scheme (Imielinski et al., 1997). The EPR indexing scheme is one of the distributed indexing schemes for data dissemination in a mobile computing environment. In the EPR indexing scheme, the index tree is divided into two parts: the replicated part and the non-replicated part. The replicated part consists of the top $r$ levels of the index tree, and rest of the index tree constitutes the non-replicated part. An index node in the replicated part is broadcast more than
once in a broadcast cycle, while an index node in the non-replicated part is broadcasted only once in a broadcast cycle. Table 1 shows the default values of the parameters in each simulation. The database to be broadcast contains $N_D$ buckets. The level of the index tree is $L$ and the degree of the index tree is $D$. We assume that the replicated part of the index tree consists of the top $R$ levels.

In each simulation, a mobile client listens to the broadcast channel for its desired data items. We assume that the access of the broadcast data items by a mobile client follows a Zipf distribution (Ijiri and Simon, 1977). The broadcast data items are numbered from 1 to $N_{data}$, where $N_{data}$ is the number of the broadcast data items. In a Zipf distribution, the access probability of a data item $x$ is determined by the following probability function:

$$\text{Zipf}(x) = \begin{cases} \frac{1}{x^\theta} & x = 1 \sim N_{data}, \\ \frac{1}{\sum_{j=1}^{N_{data}} (j^\theta)} & \text{otherwise}. \end{cases}$$

In the Zipf distribution, the degree of the data access skew is in proportion to $\theta$. A large $\theta$ gives a high skew Zipf distribution. When $\theta = 0$, it reduces to a uniform distribution.

In the study of the performance of the index caching replacement policies, we compare the performance of LLIF and CPF with the performance of least-recently/frequently-used (LRFU) (Lee et al., 2001) and LRU.

### Table 1

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter of the Zipf distribution ($\theta$)</td>
<td>0–2</td>
</tr>
<tr>
<td>Number of data buckets ($N_D$)</td>
<td>4096 buckets</td>
</tr>
<tr>
<td>Cache size of the client ($C$)</td>
<td>34–1000</td>
</tr>
<tr>
<td>Height of the index tree ($L$)</td>
<td>3–8</td>
</tr>
<tr>
<td>Degree of the index tree ($D$)</td>
<td>3–8</td>
</tr>
<tr>
<td>Replicated part of the index tree ($R$)</td>
<td>2</td>
</tr>
</tbody>
</table>

The LRFU replacement policy is known as the best page-based cache replacement policy.

In the following subsections, we report the results of the simulations. We compare the tuning times and access times of the replacement policies in experiments 1 and 2, respectively, for different cache sizes. Experiment 3 and 4 are devoted to the study of the effect of access skew on the tuning times and the access times, respectively, of the replacement policies.

#### 5.1. Experiment 1: Effect of cache size on tuning time

In this experiment, the cache size ranges from 34 to 1000. We use an index tree with degree of 4 and height of 6. The number of data buckets is 4096. The number of index nodes in the index tree is set to 1360. Fig. 9(a) shows the average tuning times for accessing a data item given $\theta = 0$. Fig. 9(b) shows the average tuning times for accessing a data item given $\theta = 1.5$.

Fig. 9(a) shows that CPF outperforms other policies when there is no access skew. On average, the tuning time of CPF is about 6% lower than that of LRFU, and 14% lower than that of LRU. Fig. 9(b) shows that when the data access is very skew ($\theta = 1.5$), CPF has the longest tuning time. It also shows that LLIF outperforms CPF for all different cache sizes, and LLIF performs equally well as LRFU does. On average, the tuning time of LLIF is about 14% lower than that of CPF and 12% lower than that of LRU.

Fig. 9(a) shows that LLIF has the longest tuning time among all policies. However, as the cache size increases, the tuning time of LLIF decreases faster than any other policies. It shows that, as the cache size reaches 1000, LLIF has the shortest tuning time. This is because LLIF tends to cache the lower level index nodes. When the cache size is small, LLIF has low hit rate for index caching. However, as the cache size increases, the hit rate increases. Furthermore, because a leaf index node is more effective than a non-leaf index node in tuning time reduction, LLIF, which caches many lower level index nodes, excels other policies when the cache size is large.

![Fig. 9. The tuning time for different cache sizes: (a) $\theta = 0$, (b) $\theta = 1.5$.](image)
Fig. 9(a) shows that CPF outperforms any other policies when the data access is not skew. However, when the data access is skew (Fig. 9(b)), CPF has the longest tuning time. This is because when the data access is skew, the mobile client only requests a small number of the broadcast data items. The locality in data accessing results in the high performance of LRFU, LRU and LLIF. However, the locality slows down the process of cut-plane initialization and cut-plane adjustment in CPF, which, in turn, deteriorates the performance of CPF.

5.2. Experiment 2: Effect of cache size on access time

Figs. 10(a) and 10(b) show the average access times given that $\theta = 0$ and $\theta = 1.5$, respectively. Figs. 10(a) and 10(b) both show that LLIF has the shortest access time among the five policies. When $\theta = 0$, the access time of LLIF is about 2% lower than those of LRU and LRFU and about 1% lower than that of CPF. This is because a non-leaf index node is less effective in access time reduction than a leaf index node. That is, using a non-leaf index node to access a data item requires a mobile client to access more index nodes than using a leaf index node. Since LLIF caches leaf index nodes first, it therefore has the shortest access time. Note that the difference on the access times of different policies is not significant. This is because that the access time is determined by the time when the desired data item is retrieved. The cached index nodes are less effective in the reduction of access time.

Fig. 10(a) shows that the access time is determined by the cache size when the data access is not skew. More specifically, when the cache size is less than 400 buckets, the access time of LRU is shorter than that of CPF. As the cache size increases, the difference between the access times of LRU and CPF is diminishing. When the cache size reaches 600 buckets, the access time of CPF becomes shorter than that of LRU. This is because when the cache size is small, CPF caches only the upper level index nodes of the index tree. In this case, the access time of CPF is large. As the cache size increases, CPF caches many low level index nodes. The access time is therefore reduced. On the other hand, the decrease in the access time of LRU as the cache size increases is not significant due to the range overlapping problem.

5.3. Experiment 3: Effect of data access skew on tuning time

In experiment 3, we study the effect of data access skew on tuning time. Figs. 11(a) and 11(b) show the tuning times for different policies given that the cache size is fixed at 68 and 200, respectively. Both figures show that
CPF has the shortest tuning time when the data access pattern is not skew \((\theta < 1)\). Fig. 11(a) shows that LLIF has the shortest tuning time when the data access pattern is skew \((\theta \geq 1.8)\) and the cache size is small. Fig. 11(a) also shows that when the cache size is small and \(\theta\) is between 1 and 1.5, LRFU has the shortest tuning time. The experiment also shows that as the degree of the data access skew increases the tuning times of all policies decrease.

5.4. Experiment 4: Effect of data access skew on access time

In this experiment, we study the effect of data access skew on access time. In the experiment \(\theta\) is varied from 0 to 2 with an increment of 0.2, and cache size is set to 200. Fig. 12 shows that when \(\theta < 1.5\), LLIF has the shortest access time. However, the difference on the access times of different policies is not significant. Note that the access times of all policies start to increase when \(\theta > 0.6\). This is because when \(\theta\) exceeds a certain threshold, the mobile client has a high tendency to access the same data item in any two consecutive requests. Since the time between two consecutive broadcasts is \(\text{bcast\_length}\), the access times of all policies increase and converge at \(\text{bcast\_length}\).

5.5. Summary of the Effects of cache size and data access skew

We summarize the results of the previous experiments in Tables 2 and 3. Table 2 lists the policy that outperforms other policies in terms of tuning time for each combination of cache size and degree of data skew, while Table 3 lists the policy that outperforms other policies in terms of access time. Notices that, if the difference of the performance of two policies is less than 1%, they are regarded as performing equally well.

Table 2

<table>
<thead>
<tr>
<th>(C/\theta)</th>
<th>0</th>
<th>0.2</th>
<th>0.4</th>
<th>0.6</th>
<th>0.8</th>
<th>1.0</th>
<th>1.2</th>
<th>1.4</th>
<th>1.6</th>
<th>1.8</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF/LRFU</td>
<td>LRFU</td>
<td>LRFU</td>
<td>LRFU</td>
<td>LLIF</td>
</tr>
<tr>
<td>68</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF/LRFU</td>
<td>LRFU</td>
<td>LRFU</td>
<td>LRFU</td>
<td>LLIF</td>
<td>LLIF</td>
</tr>
<tr>
<td>200</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF/LRFU</td>
<td>CPF/LRFU</td>
<td>CPF/LRFU</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>400</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>600</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>800</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>1000</td>
<td>CPF/LLIF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF</td>
<td>CPF/LLIF</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

A ‘\(\ldots\)’ means that there is no winner.

Table 3

<table>
<thead>
<tr>
<th>(C/\theta)</th>
<th>0</th>
<th>0.2</th>
<th>0.4</th>
<th>0.6</th>
<th>0.8</th>
<th>1.0</th>
<th>1.2</th>
<th>1.4</th>
<th>1.6</th>
<th>1.8</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>LLIF</td>
<td>LLIF</td>
<td>LLIF</td>
</tr>
<tr>
<td>68</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>LLIF</td>
<td>LLIF</td>
<td>LLIF</td>
<td>LLIF</td>
</tr>
<tr>
<td>200</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>400</td>
<td>LLIIF</td>
<td>CPF/LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>600</td>
<td>LLIIF</td>
<td>CPF/LLIIF</td>
<td>CPF/LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>LLIIF</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>800</td>
<td>LLIIF</td>
<td>CPF/LLIIF</td>
<td>CPF/LLIIF</td>
<td>CPF/LLIF</td>
<td>CPF/LLIF</td>
<td>CPF/LLIF</td>
<td>CPF/LLIF</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>1000</td>
<td>CPF/LLIIF</td>
<td>CPF/LLIIF</td>
<td>CPF/LLIF</td>
<td>CPF/LLIF</td>
<td>CPF/LLIF</td>
<td>CPF/LLIF</td>
<td>CPF/LLIF</td>
<td>CPF/LLIF</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

A ‘LL/LR’ means that LLIIF and LRFU are winners.
5.6. Experiment 5: Effect of height of index tree

To study the effects of the height of the index tree, we use six index trees that have the same degree but have different levels. Notice that, as the height of an index tree increases, the number of data buckets and index buckets in the corresponding broadcast increases. Since the number of the index nodes in an index tree is in proportion to the height of the index tree, the cache size is, therefore, in proportion to the height of the index tree. We set the cache size to be 10% of the total index nodes. Fig. 13(a) shows the average tuning time with different heights of index tree. From Fig. 13(a), we observe that, when a mobile client accesses a data item without using cached indices, the tuning time increases as the height of the index tree increases. This is because the mobile client has to access all the index nodes from the root to the related leaf of the index tree. Thus, as the height of the index tree increases, the mobile client has to access more index nodes before accessing a required data item. Consequently, the tuning time increases.

Second, we observe that for LLIF, the tuning time increases as the height of the index tree increases, while LRU, LRFU and CPF do not exhibit the same phenomenon. This is because the hit ratio of LLIF may be less than 1. When a required index is not in the cache (i.e., a cache miss occurs), the mobile client has to access all the index nodes from the root to a related leaf. Therefore, the tuning time increases as the height of the index tree increases. In contrast, the tuning time of LRU, LRFU and CPF is not sensitive to the height of the index tree. Note that the hit ratios of LRU, LRFU and CPF always equal 1. This is because LRU and LRFU always keep the root, \( R \), in its cache. Since the range of \( R \) covers all the data buckets, the hit ratio equals 1. The same reason applies to CPF since CPF always keeps a cut-plane and the range of the cut-plane covers all the data buckets.

Since the hit ratios of CPF, LRFU and LRU equal 1, an index entry always exists in the cache for CPF, LRFU and LRU for a broadcast data item. Furthermore, since the cache size is set to be in proportion to the number of the levels of the index tree, the cache size increases as the number of levels of the index tree increases. As the cache size increases, CPF will obtain a lower cut-plane, which, in turn, reduces the tuning time. Similarly, as the cache size increases, LRU and LRFU will retain many lower level indices in their caches. Consequently, the tuning time is reduced. However, these reductions are offset by the increase on the height of the index tree. This is because when the height of the index tree increases, a mobile client needs to access more index nodes to access a required data item. This explains why the tuning times of LRU, LRFU and CPF remain unchanged as the height of the index tree changes.

Fig. 13(b) shows the access time for different heights of the index tree. It shows that the access time increases as the height of the index tree increases. However, the difference in access time for five policies is not significant.

5.7. Experiment 6: Effect of degree of index tree

In this experiment, we study the effect of the degree of the index tree on the tuning time and access time. We vary the degree of the index tree from 3 to 8, and set the height of the index tree to 4. The cache size is set to 10% of the size of the index nodes. \( \theta \) is set to 0. Fig. 14(a) shows the average tuning time for accessing a data item using different index trees. Fig. 14(a) shows that the tuning time of LLIF is not sensitive to the degree of the index tree. This is because the hit ratio of LLIF policy remains unchanged with respect to changes on the degree of the index tree. Note that the cache size is set to 10% of the size of the index tree. When the degree of the index tree increases, so does the cache size. The hit ratio is, therefore,
unchanged. For LRU, LRFU and CPF, the tuning time decreases as the degree of the index tree increases.

Fig. 14(b) shows that the access times of these three policies increase as the degree of the index tree increases. This is because when the degree of the index tree increases, both the size of the index tree and the number of the data items increase. This results in the increase in the length of the broadcast cycle. When the length of the broadcast cycle increases, so does the access time. Note that, among the three policies, LLIF has the shortest access time.

6. Conclusions and future work

In this paper, we propose to use cached indices to facilitate data access in a data dissemination environment. By reusing cached indices, the tuning time of a mobile client can be significantly reduced. To use the cache space of the mobile clients efficiently, cache management is required. The traditional cache management policies, such as LRFU and LRU, do not consider the index structure of the broadcast data items. In this paper, we propose two cache management policies that take the index structure of the broadcast data items into account. The LLIF policy caches the leaf index nodes of the index tree because a leaf index node is more effective in reducing tuning time, while the CPF policy caches the index nodes in a cut-plane of the index tree to prevent the range overlapping problem of different index nodes.

We conducted several experiments to compare the performance of the LLIF, CPF, LRFU and LRU policies. The performance the cache management policies depend on the size of the cache space and the degree of the data access skew of the mobile clients. According to the experiments, CPF excels all other policies in tuning time given that the degree of the data access skew is not significant. In contrast, LLIF outperforms all other cache policies when the cache size is small and when the data access skew is significant. When the degree of the data access skew is average, that is, $1.0 < \theta < 1.6$, and the cache size is small (<68), LRFU has shortest tuning time. In terms of access time, LLIF outperforms all other policies. However, the difference in access times of all policies is insignificant. Besides, the LLIF policy is very sensitive to the level of the index tree, while The CPF, LRFU, LRU policies are not sensitive to the level of the index tree. As expected, the access time for both policies increases as the level of the index tree increases.

There are issues remained to be addressed. To further reduce the access time of the mobile clients, data items should be broadcasted with different frequencies according to their access probabilities. When the access probability is considered, the index scheme is different from what we have addressed here. We plan to design new cache management policies for such an indexing scheme.

References

