Efficient Data Dissemination to Mobile Clients in E-Commerce Applications*

Ruchi Agrawal and *Panos K. Chrysanthis*
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
{ruchi,panos}@cs.pitt.edu

Abstract

Mobile commerce is the next growing area in electronic commerce and mobile computing. These are sophisticated, data intensive mobile applications whose success strongly depends on the efficiency by which data are disseminated to a large number of mobile users. Different techniques have been put forward of which the most promising are the push-based techniques that explore the asymmetry in wireless communication and the reduced energy consumption of the receiving mode on mobile devices. This paper proposes a new broadcast indexing scheme, termed "Constant-Size I-node Distributed Indexing" (CI), that offers more energy savings in practical applications. Our detailed simulation results indicate that CI which is a variant of the currently best performing Distributed Indexing, outperforms the latter for broadcast sizes of 12,000 or fewer data items, reducing the access time up to 25% and tuning time up to 15%.

1 Introduction

In the last decade, the emergence of the Web led to an unprecedented number of new applications, most notably Electronic Commerce (E-Commerce) applications such as advertising, eAuctions, on-line stock trading, to name few. As the popularity of these applications grows, so does the need for efficient and scalable dissemination of data to a large number of users. This need takes a new form when we consider the new class of mobile users.

In mobile environments, data dissemination is more challenging due to the limited, wireless bandwidth available for communication, frequent disconnections and low power withholding capacity of mobile devices. Different techniques have been put forward to address this problem, of which the most promising are the *push-based* techniques that explore the asymmetry in wireless communication and the reduced energy consumption of the receiving mode on mobile devices [4, 8]. Servers have both much larger bandwidth available than mobile devices and more power to transmit large amounts of data. In push-based techniques, the server repetitively broadcasts data to the mobile users without a specific request. Mobile users monitor the broadcast channel and retrieve data as they arrive on the broadcast channel. If data is properly organized to cater to the needs of all users, this scheme makes an effective use of the low wireless bandwidth [6, 1, 7] This can be achieved, for example, by treating user requests as feedback and adjusting the broadcast content to satisfy requests with similar data requirements by different groups of users [12].

A common approach in push-based techniques is to consider "air" as virtual disk. But in broadcasting data on the air, access is sequential, as it is in tape drives. So, in order to access the required data, a user has to be in *active mode*, waiting for the data to appear on broadcast. Hand-held and mobile devices are typically capable to switch from *active mode* to *doze mode* which requires much less energy. The power consumption of mobile devices is a key issue as the lifetime of a battery is expected to increase by only 20% over the next 10 years [11]. Hence, the energy efficient way to access data is to *tune in selectively* in order to find out the correct position of data on broadcast and then go into doze mode until the data appear on the broadcast. This requires some form of directory information to be broadcasted along with data, making the broadcast self-descriptive. This directory identifies data items by some key value and gives the location of the actual data on broadcast. Several broadcast organizations to encode this directory structure have been proposed. These include incorporating hashing in broadcasts [7], using signature techniques [10] and broadcasting *index* information along with data [6, 3, 8].

In this paper, we propose a new indexing scheme, called *Constant-size I-node Distributed Indexing* (CI), that performs much better with respect to the tuning time and access

*This work is supported in part by National Science Foundation under grant IRI-9502091 and IIS-9812532.

time for broadcast sizes in practical applications. This new scheme minimizes the amount of coding required for constructing an index in order to correctly locate the required data on broadcast, thus decreasing the size of the index and consequently the access time as well. Our detailed simulation results indicate that CI which is a variant of the currently best performing *Distributed Indexing* [6, 8], outperforms it for broadcast sizes of 12,000 or fewer data items, reducing the access time up to 25% and tuning time by 15%.

In the next section after introducing the broadcast model, we review Distributed Indexing and then in Section 3, we present our new broadcasting indexing scheme. In Section 4, we discuss in detail our simulation experiments. We conclude with a summary in Section 5.

2 The Broadcast Model

In a broadcast dissemination environment, a data server periodically broadcasts data items to a large client population. Each period of the broadcast is called a *broadcast cycle* or *bcycle*, while the content of the broadcast is called a *bcast*. Each mobile client (MC) listens to the broadcast and fetches data as they arrive. We assume that all updates are performed at the server and disseminated from there.

The smallest logical unit of a broadcast is called a *bucket*. Buckets are analogous to blocks in disks. Each bucket has a header that includes useful information. The exact content of the bucket header depends on the specific broadcast organization. Information in the header usually includes that whether the bucket contains data or index, the offset to the beginning of the next bcast, the offset of the bucket from the beginning of the bcast, and the offset to the beginning of the next index segment.

A data bucket may contain several data items which correspond to database records (tuples). We assume that all data items can be identified by the values of some key attributes and that keys are known to the clients. The filtering process is simply matching the key values. Index buckets contain the necessary control information for filtering that leads to the actual location of the specified key value. We will discuss below the contents of index nodes of each indexing scheme in our study.

The efficiency of accessing data on air can be characterized by two parameters.

- *Tuning Time:* The amount of time spent by a user in active mode (listening to channel) and
- *Access Time:* The total time elapsed from the moment a client requests data identified by ordering key, to the time the client reads that data on channel.

Ideally, we would like to reduce both tuning time, and access time. However, this is not possible because any improvement in tuning time requires additional information to be broadcast which increases the bcycle and hence the access time. On the other hand, the best access time is achieved when only data are broadcast and without any indexing. Clearly, this is the worst case for tuning time. In this paper, our goal is to develop an indexing scheme which provides the best balance between the tuning and access time.

2.1 Distributed Indexing

Distributed Indexing (DI) [6, 8] is currently considered the best indexing for broadcast data. It reduces tuning time at a small cost in access time by distributing and selectively replicating part of an index over a broadcast.

In DI, the index is organized as a tree with all the data buckets appearing as leaf nodes. One data bucket can have one or more data items. Index information is added at non-leaf nodes such that each node has the complete information about its children (their position on the broadcast cycle and the keys for the data they are addressing). The tree is divided into two parts:

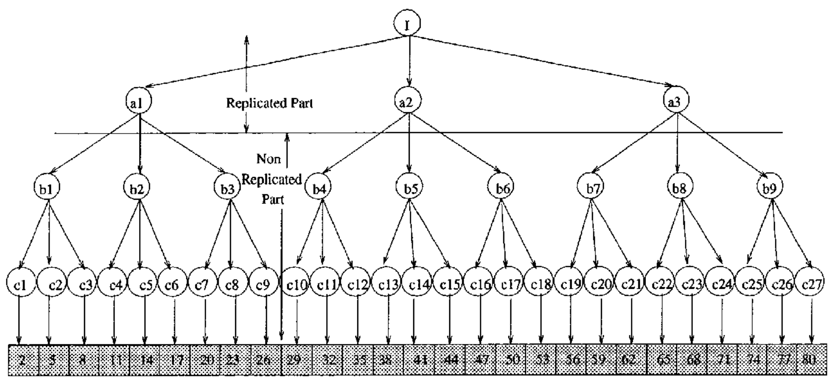
- *The replicated part* – the top r levels of the index tree
- *The non-replicated part* – bottom (k-r) levels

The tree nodes of the $(r+1)^{th}$ level are called *non-replicated roots* (NRR). Figure 1 illustrates an example of a DI tree as given in [6] with 81 data items. The index tree has four levels and each node has 3 children. Level 1 is the root depicted by *I*. Level 2 is depicted by $\{a_1, a_2, a_3\}$. Level 3 $\{b_1, ..., b_9\}$ are non-replicated roots. $\{c_1$ to $c_{27}\}$ depict fourth level nodes and finally 0 - 80 data items, grouped in 3, are shown in shaded nodes.

The idea is to construct the linear broadcast such that each node rooted in a non-replicated root will appear only once in the whole broadcast just in front of the set of data buckets it indexes (points to). Each node of the index tree which appears above a non-replicated root is replicated n times where n is the number of the children that node has, forming the paths from the root to non-replicated roots. In this way, each b_i in NRR forms an index segment of broadcast having the information about its children after it (Ind(b_i)) and the information about its ancestors before it (Rep(b_i)). Thus, the broadcast can be expressed as sequence of triples, each of which corresponds to the index segment of each non-replicated root b_i in the tree and the pertaining data segment (Data(b_i)) [6]:

$< Rep(b_i), Ind(b_i), Data(b_i) >$
$\forall b_i \in NRR = \{b_1, b_2, ..., b_t\}$ in left to right order, where

- Rep(b_1) = Path(I,b_1), b_1 is the first bucket in NRR and I is the root of the index tree.
- Rep(b_t) = Path(LCA(b_{t-1}, b_i), b_t) for i = 2, $\cdots$, t where b_i is the i^{th} index in NRR of tree.



NRR = {b1,b2,b3,b4,b5,b6,b7,b8,b9}

Rep(b1) = { I, a1}	Ind(b1) = { b1,c1,c2,c3 }	Data(b1) = { 0,..................8}
Rep(b2) = { a1 }	Ind(b2) = { b2,c4,c5,c6 }	Data(b2) = {9,.....................17 }
Rep(b3) = { a1 }	Ind(b3) = { b3,c7,c8,c9 }	Data(b3) = { 18 , ..., 26 }
Rep(b4) = { I,a2 }	Ind(b4) = {b4,c10,c11,c12 }	Data(b4) = { 27 , ... ,35 }
Rep(b5) = { a2 }	Ind(b5) = {b5,c13,c14,c15 }	Data(b5) = { 26 , ... , 44 }
Rep(b6) = { a2 }	Ind(b6) = {b6,c16,c17,c18 }	Data(b6) = {45 ,..................., 53 }
Rep(b7) = { I,a3 }	Ind(b7) = {b7,c19,c20,c21 }	Data(b7) = { 54 , ..., 62 }
Rep(b8) = { a3 }	Ind(b8) = {b8,c22,c23,c24 }	Data(b8) = { 63 , ..., 71 }
Rep(b9) = { a3 }	Ind(b9) = {b9,c25,c26,c27 }	Data(b9) = { 72 , ..., 80 }

Figure 1. Distributed Indexing Tree

- Path(c, b): the sequence of index nodes along the path from index node c to b (excluding b).
- LCA(b_i, b_k): the least common ancestor of b_i, and b_k in the index tree.
- Data(b_i): the set of data buckets indexed by b_i.
- Ind(b_i): the part of the tree below b_i (including b_i).

Figure 2 depicts the linear broadcast corresponding to the DI tree in Figure 1. The index segments are shown in white. In the example, Ind(b_3) consists of 3^{rd} NRR, i.e., b_3 and its children nodes {c7,c8,c9} and Data(b_3) consists of all data items {18 ... 26} that the node b_3 covers in the tree. Rep(b_3) = Path(LCA(b_2, b_2),b_3) = a_1. Note that for simplicity of presentation, we are assuming that each tree node is stored in a separate index bucket. In our experimental evaluation, we take into consideration that the size of all tree nodes is not the same and more than one might be fitted in a bucket. Similarly, we assume that each data item is stored in a separate data bucket.

In the replicated part of the tree, not an entire parent node of a child node is replicated on the bcast. For example in Figure 2, for a replicated node a1, not the entire I_1 is replicated. In order to determine if a data item has already appeared on the broadcast, *control information* is attached to replicated nodes, e.g., I_1 and a1. The control information records what part of the broadcast is left behind, what lies ahead which is not covered by the particular node and when the start of the next broadcast is. Thus, if needed data is missed, the next start of broadcast can be determined, or if data is ahead in the current broadcast, then the index bucket having the right information can be located.

Let us consider the case in which a mobile client MC searches for key 7 and tunes into the data bucket with key 8. The index pointer in that bucket leads the MC to the next index segment which starts at the second_a1 (Figure 2). The control information in the second_a1 specifies that for key values $\leq$ 8, tune into *begin* (which is the start of the next broadcast), for key values > 26, tune into I_2, and for any other value (i.e., 8< key $\leq$ 26) search the value range pairs in second_a1 for the appropriate next pointer. Given that MC searches for key 7, it goes into doze mode until the start of the next bcast. Next, MC tunes into I_1 (begin). The first range pair covers the values 0-26, for which key 7 is in range, so MC follows pointer_1 which leads it to first_a1. In first_a1, the first range pair covers values 0-8 for which key 7 is again in range so MC follows the pointer_1 and tunes into b1. In this node, the third pair with range 6-8 covers key 7, so MC follows pointer_3 and tunes into c3. The pointer_2 in c3 leads MC to the data bucket with key 7.

3 Constant-size I-node Distributed Indexing

In this section, we are presenting our new indexing scheme termed *Constant-size I-node Distributed Indexing* (CI). CI is a variant of DI and hence, it follows the same tree structure formation as per DI. Given the number of data items *N*, we come up with a tree of order n^k, where *n* is the number of children for each node and *k* is the number of

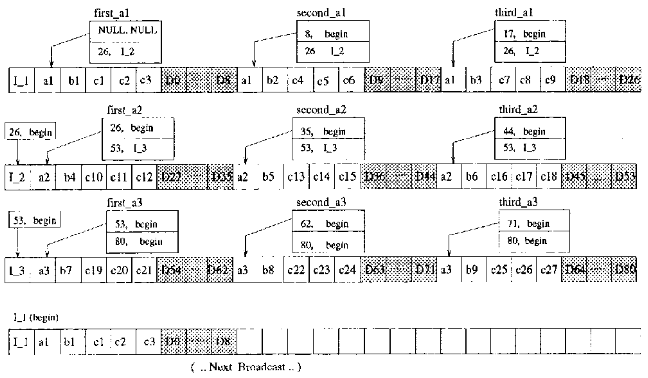


Figure 2. Distributed Indexing: linear broadcast with control information

levels of the tree. As in DI, the broadcast can be expressed as a sequence of triples, $< Rep(b_i), Ind(b_i), Data(b_i) >$. Thus, if we use the same example, select the same value of n and k for the tree and make the same simplified assumptions as for DI in the previous section, the layout of the broadcast cycle for CI is similar to that of DI as shown in Figure 2 with one significant difference, namely, the absence of the variable-size *control information*. In CI, no control information is attached to any node.

The second difference is that, in CI, the size of index nodes in the tree is constant. An index node in CI has a pair of range keys defined by *max key* and *min key* which defines the range of data indexed by the node and one *pointer* which points at the next bucket to tune into. This is in contrast to DI in which the size of index node is variable and contains n such pairs of range and pointer as the number of children nodes. Figure 3 shows for both the DI and CI schemes, the content of the first four index nodes in the bcast of our example index tree in Figure 1. In both schemes, the header of each node is of the same size and contains an extra field that in the case of DI, encodes whether or not a node contains control information whereas in the case of CI, it encodes the node level. By eliminating the control information and maintaining index nodes of constant size equal to the minimum size of the index node in DI, CI results in smaller size bcast and hence requires less access time.

3.1 Access Protocol

The ingenuity of the CI scheme lies in the way the single pointer in index nodes is interpreted at each level of the tree. The protocol can be understood by keeping in view the tree structure and its linear layout on bcast. If the search key lies within the range of an index node, say *IN*, (i.e., it is a *hit*), then the data item lies in the subtree for which *IN* is the root. On linear bcast, this subtree starts just after *IN* and hence there is no need for an explicit pointer. An MC just needs to read the next index node on bcast. This process goes on recursively for each subsequent hit. For the index nodes which are at the last level (k^{th}), intuitively, in case of a hit, the pointer must point to the data item. Our access protocol complies with this observation and the pointers for k^{th} level nodes points to the start of the group of n data items which is indexed by that k^{th} level index node.

In the case that the search key does not lie in the range given by *IN* (i.e., it is a *miss*), it means that the data item is not in the subtree for which *IN* is the root and the next subtree needs to be searched, for which the root is the next subsequent index node at same level. In case of replicated nodes, in order to ensure that the next index node at same level is not a replication, but a different node, the pointer is made to point at the index node which is one level higher. The same strategy is used for index nodes which are NRR. The pointer points at one higher level node, so that, comparisons can be made at a higher level and in case of *miss*, a larger portion of the tree can be skipped. In case of other non-replicated nodes, the pointer simply points at the next subsequent node at the same level.

In the case of nodes which are at the lowest level (k^{th}), as stated above, the pointer would be needed to point at data items. It would seem, another pointer is needed to point to index nodes in case of *miss*. But, in linear bcast, k^{th} level nodes are placed one after the other in group of n (number of children). This observation made it possible to have only one pointer at k^{th} level too. In case of a miss, an MC just reads the next node on the bcast which is of same level.

The steps of access protocol are given below:

- For nodes which belong to levels 1 to r+1 (which is the level of NRR), if the search key does not lie in between *min* and *max* key specified in the index node, the client uses the pointer to read the next higher level node (in

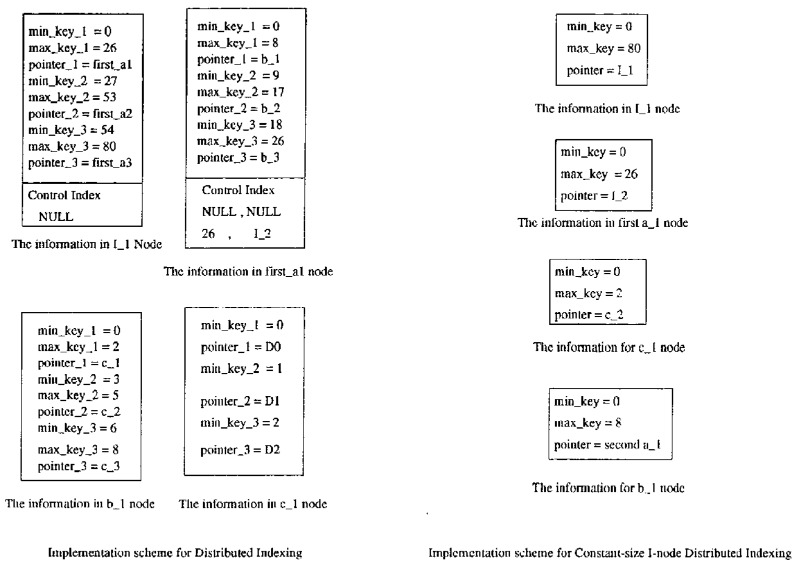


Figure 3. DI and CI index nodes

case of level 1, this is the next bcast). Otherwise, if the search key is within the range, the client reads the next index node on broadcast (which is always one level lower).

- For nodes which belong to the level between r+2 to k-1, which are all non-replicated index nodes, if the search key is not within range, the client uses the pointer to read the next index node at the same level. Otherwise, if the search key is within range, the client reads the next index node which again, is one level lower.

- In the case of k^{th} level index (last index level) if the search key is within the range, the client follows the pointer which points at the start of the data segment indexed by the k^{th} level node. Then, the client tries to match the search key with key in data items one by one. Since the data items are sorted, the client stops if it encounters a data item with a key having greater value than search key or it has checked n data items (which is the number of its children). If the MC cannot find the required key, it concludes that the data item is not on the bcast.

- At k^{th} level if the search key is not within range, then an MC continues reading the next index node on bcast which is of same level, until it hits a data segment. At this point, if the client has not just tuned in the k^{th} level index segment, then it means that the data item is not in the bcast. Otherwise, if the client has just tuned in the k^{th} level, the client has to start its search from the next index bucket pointed by the next bucket on bcast which is a data bucket.

The protocol will become more clear with the following example which is the same used above for DI: Consider the broadcast structure given for 81 data items in Figures 1 and 4. Level 1 is the root(I) and its min_key and max_key covers the entire data set being broadcasted. Lower indexes cover their part of data. Now, consider a case when a mobile client *MC* is searching for key 7 and tuned into the data bucket with key 8. In Figure 4, the shaded boxes are the ones MC tunes into. Since all data buckets have a pointer which points to the start of the next index segment, MC reads the next index node second_a_1. This is a *hit* since a_1 has a range from 0 - 26. So, MC reads the next adjacent lower level index node b_2. This is a *miss* since b_2 indexes data items between 9 - 17. According to the protocol, MC follows the pointer to read the next higher level index node third_a_1. Again, this is a hit, and so, MC reads the adjacent index node b_3 which is a miss. Hence, MC follows the pointer to read the next higher level index node first_a2 which again is a miss, so MC reads the next higher level node I_3. Since I_3 is a hit, MC reads the next (one level lower) index node first_a3 which is a miss. Thus, MC needs to follow again the pointer to read the next higher level node I_1 which is the start of the next broadcast cycle. Now, MC gets a hit and reads the next adjacent node, first_a1 which is also a hit, so it reads the next node b_1 which is again, a hit, so MC reads the next index node c_1. c_1 is a miss be-

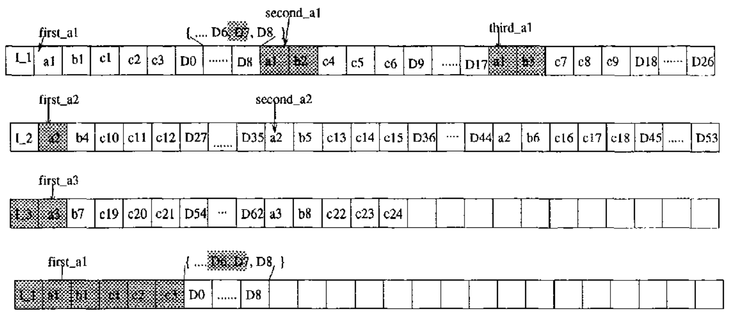


Figure 4. CI: linear broadcast and tuning in search of value 7

cause it indexes data 0 - 2. Thus, following the protocol on a miss between level r+1 and k-1, MC reads the adjacent same level index nodes until it reads a data bucket (which implies the end of n last level index nodes). In our example, MC reads c_2 which is a miss and then c_3 which is a hit. Given the hit, MC follows the pointer in c_3 which leads it to D6. D6 is a miss, so MC reads the adjacent data bucket D7 which is a hit and MC finds its data.

4 Performance Evaluation

Although it is not difficult to show analytically that CI always supports better average access time than DI, this is not the case with tuning time [2]. Hence, we evaluated CI and DI using simulation.

4.1 Experimental Testbed

Our simulation model is a discrete-event simulation using the unit-time approach to advance the simulation clock [9]. We implemented our simulation model in C language. For experimentation, we implemented No-Indexing, DI and CI. No-Indexing is a good reference since it offers optimal access time and an upper bound for tuning time.

Our broadcast model consists of a Server and a Mobile Client (MC). The server broadcasts data periodically by constructing the *bcast* according to a selectable indexing scheme for each search. For both DI and CI, given the number of data items N on the bcast, the simulator generates a tree of order n^k, where n is the number of children for each node and k is the number of levels of the tree, such that N $\leq min(n^k)$. The replicated part is the top r levels of the tree. Our experimental results have indicated that the $\lfloor k/2 \rfloor$ is the best choice for r.

Recall that when index nodes and data are arranged linearly in a bcast, the contiguous set of either index nodes or data items form index and data segments, respectively.

Servers	1
Clients	1
Bcast size	10-18,000 data items
bucket size	32 bytes
Bcast pointer	2 bytes
Data size	30 bytes
Data key	2 bytes
Key searches	5,000

Table 1. Simulation Parameters

These index and data segments are then divided to fit into *buckets*, the physical units of bcast, which are of fixed size. In our experiments, the buckets are 32 bytes long. This is reasonable if we are focusing, for example, on stock information broadcast. One bucket holds either a part of data segment or a part of index segment, but not both, i.e., index segments and data segments are not mixed. As discussed earlier, to keep things simpler, we are assuming that one data item fits into one bucket. This does not affect the results, as the policy is uniform across all indexing schemes discussed in this paper. Above, we have also assumed that each index node fits in a separate bucket. However, in our simulation, multiple index nodes are fitted in one bucket, or spread across multiple buckets, depending on the size of bucket and index node. This leads to some fragmentation of buckets and the unused space is padded with spaces.

MC is provided as input a list of keys of data items which it has to search. In each run the access set consists of 5,000 data items to be searched. Averages are taken over these 5,000 searches for any result. This access data set is prepared by taking Zipf distribution function [5]. Using Zipf, 5,000 random numbers are generated ranging between 1 to *number of data items on bcast* in each case. MC randomly tunes in between the broadcast and starts its search for a data item by comparing the *keys*.

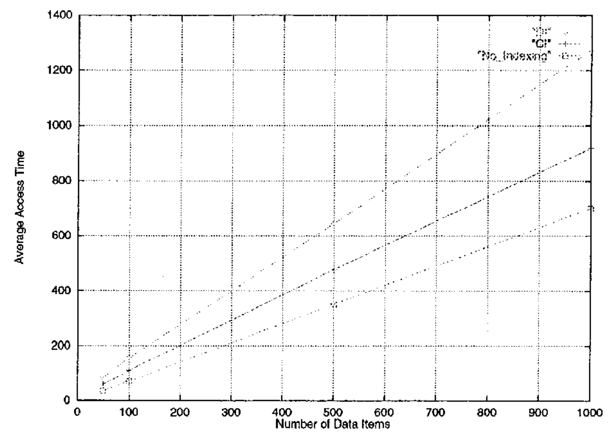


Figure 5. Access Time for 10-1000 data items

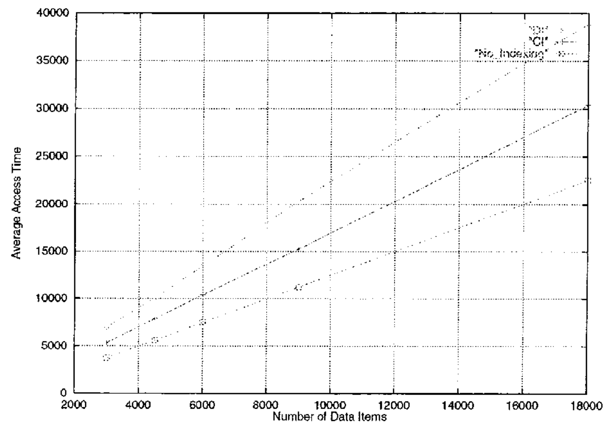


Figure 6. Access Time for 3,000-18,000 data items

In our experiments, the bcast ranges from 10 to 18000 data items. The size of each data item is set to 30 bytes and the size of the key field to 2 bytes. The size of a broadcast pointer is 2 bytes. Each of these experiments are repeated 30 times to take confidence intervals. The parameters of our simulator are summarized in Table 1.

4.2 Access Time

Figures 5 and 6 show the average access times for different numbers of data items on a single channel, for each of the indexing schemes in our experiments. Figure 5 depicts the graphs for 10 to 1000 data items and Figure 6 depicts the graphs for 3000 to 18000 data items. Clearly, as expected, the increase is linear for all of the schemes. Further it is no surprise that No-Indexing has better access time for all

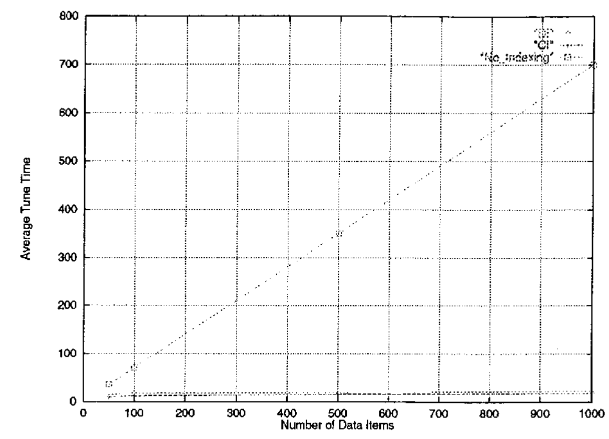


Figure 7. Tuning time for 10-1000 data items

ranges of data items, since it has the smallest bcast size.

CI exhibits the second best performance for all data ranges, including the DI as predicted by our analysis. CI reduces the average access time as much as 25% of the average access time of DI for broadcast sizes of 12,000 data items, which is considered to be an upper bound in practical applications. Compared to No-Indexing, the average performance of CI is up to 20% better.

4.3 Tuning time

Figure 7 depicts the average tuning time for No-indexing, CI and DI for broadcasts of 10-1000 data items. The general observation is that both CI and DI significantly reduce the tuning time compared to No-Indexing which as expected, exhibits the worst average tuning time since No-Indexing is always in active mode.

To enhance the readability of our results and facilitate their analysis, we show in Figure 8, the average tuning time for only DI and CI. In the left graph of Figure 8, for a very small number of data items ($<$ 15) DI is performing better than CI. But for such a small set of data using either DI or CI is not practical. In fact, for such small broadcast sizes, No-Indexing exhibits the best performance.

For higher ranges of data items, CI exhibits as much as 15% better average tuning time than DI. As shown in the right graph of Figure 8, this trend continues until the size of the data set becomes 12000. At this point, a crossover can be seen and DI becomes a better approach. This can be explained by the fact that (recall section 3.1) in CI a MC has to perform a linear search once it reaches the beginning of a data segment consisting of n data items by following the pointer at k^{th} level index node. For larger values of data items n becomes large and hence the MC has to perform a longer linear search at the data level. Since data items are larger in size, this increases the tuning time.

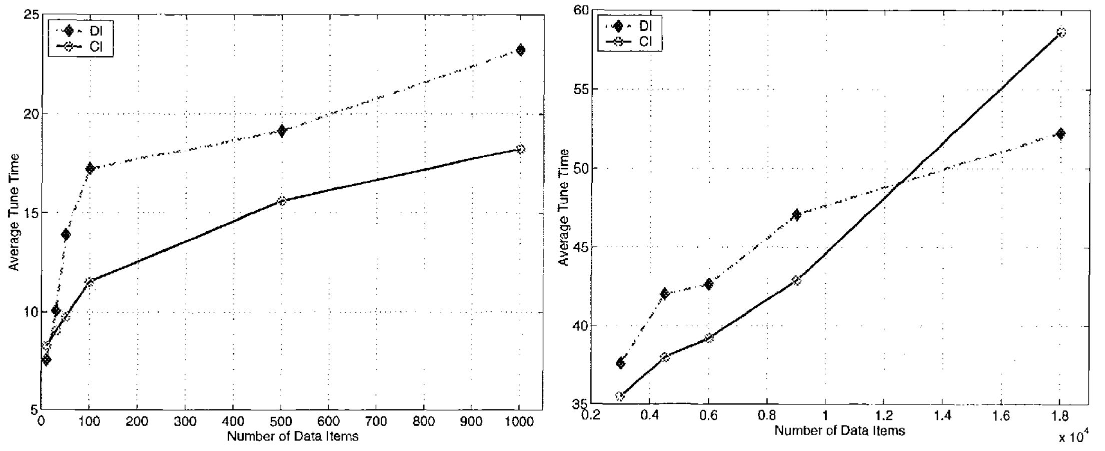


Figure 8. DI and CI Tuning Times

5 Conclusions and Future Work

Push-based data dissemination using broadcasting has attracted considerable attention both in E-Commerce and mobile environments. This is a result of the need for a scalable and efficient way of delivering information to a large number of clients. For mobile clients, an important factor for efficiency is the reduction on energy consumption. Towards this, this paper proposed a new indexing protocol, termed *Constant-size I-node Distributed Indexing* (CI) that outperforms the currently best performing Distributed Indexing (DI) in both access time and tuning time for broadcast sizes of 12,000 or fewer data items found in practical applications. Our experiments showed that CI reduces up to 25% the access time and 15% the turning time of DI, representing significant savings in energy. For example, the power savings can be up to 40% in the case of a client equipped with Hobbit Chip (AT&T) connected with a 19.6 Kbps broadcast channel [2].

Our experiments suggested that the effectiveness of the different indexing schemes is dependent on the broadcast size. Currently, we are evaluating all the existing indexing schemes including CI for single and multiple channels to identify the bounds of the applicability of each scheme. We are also investigating several optimizations to CI.

Acknowledgments: The authors thank Sujata Banerjee and Mohamed Sharaf for their helpful comments on this work.

References

[1] Acharya S., R. Alonso, M. J. Franklin, and S. B. Zdonik. Broadcast disks: Data management for asymmetric communication environments. *Proc. of the SIGMOD Conference*, pp. 199–210, 1995.

[2] Agrawal R and P. K. Chrysanthis. Improving Access Time to Air Caches While Saving Energy. *Technical Report TR-01-07*, University of Pittsburgh, Feb. 2001.

[3] Datta A. et al. Adaptive Broadcast Protocols to Support Efficient and Energy Conserving Retrieval from Databases in Mobile Computing Environments. *Proc. of the Int'l Conf. on Data Engg.*, pp. 124–133, 1997.

[4] Franklin M. J. and S. B. Zdonik. A framework for scalable dissemination-based systems. *Proc. of the OOPSLA Conference*, pp. 94 – 105, 1997.

[5] Gray J. et al. Quickly generating billion-record synthetic databases. *Proc. of the SIGMOD Conference*, pp. 243–252, 1994.

[6] Imielinski T., S. Viswanathan, and B.R. Badrinath. Energy efficient indexing on air. *Proc. of the SIGMOD Conference*, pp. 25–36, 1994.

[7] Imielinski T., S. Viswanathan, and B.R. Badrinath. Power efficient filtering of data on air. *Proc. of the Int'l Conf. on Extending Database Technology*, 1994.

[8] Imielinski T., S. Viswanathan, and B. R. Badrinanth. Data on Air: Organization and Access. *IEEE Trans. on Knowledge and Data Engg.*, 9(3):353–372, 1997.

[9] Jain R.. *The Art of Computer Systems Performance Analysis*. John Wiley and sons, 1991.

[10] Lee W.C. and D. L. Lee. Using signature tecniques for information filtering in wireless and mobile environments. *Distributed and Parallel Databases*, pp. 205–227, 1996.

[11] Sheng S., A. Chandrasekharan, R. W. Broderson. A portable multimedia terminal for personal communications. *Communication Magazine*, pp. 64–75, 1992.

[12] Stathatos K., N. Roussopoulos, and J. S. Baras. Adaptive data broadcast in hybrid networks. *Proc. of the VLDB Conference*, pp. 326–335, 1997.