# Efficient index and data allocation for wireless broadcast services

Shou-Chih Lo [a,*], Arbee L.P. Chen [b]

[a] *Department of Computer Science and Information Engineering, National Dong Hwa University, No. 1, Sec. 2, Da Hsueh Road, Shoufeng, Hualien 974, Taiwan, ROC*

[b] *Department of Computer Science, National Chengchi University, Taipei 116, Taiwan, ROC*

Received 29 January 2006; accepted 3 February 2006
Available online 7 March 2006

**Abstract**

The periodic broadcasting of frequently requested data can reduce the workload of uplink channels and improve data access for users in a wireless network. Since mobile devices have limited energy capacities associated with their reliance on battery power, it is important to minimize the time and energy spent on accessing broadcast data. The indexing and scheduling of broadcast data play a key role in this problem. In this paper, we formulate the index and data allocation problem and propose a solution that can adapt to any number of broadcast channels. We first restrict the considered problem to a scenario with no index/data replication, and introduce an optimal solution and a heuristic solution to the single-channel and multichannel cases, respectively. Then, we discuss how to replicate indexes on the allocation to further improve the performance. The results from some experiments demonstrate the superiority of our proposed approach.


*Keywords:* Mobile computing; Data broadcast; Broadcast channels; Data allocation; Index replication

## 1. Introduction

The rapid advances in computer software, computer hardware, and wireless network technologies have led to the widespread implementation of mobile computing. In this environment, users can retrieve information from wireless channels (with generally narrow bandwidth) anytime and anywhere. How to disseminate data efficiently to a large number of users in the mobile computing environment is challenging due to the necessity to consider time and energy efficiencies, given that mobile devices have limited energy capacities associated with their reliance on battery power. There are two general methods to disseminate data through wireless channels: (1) *broadcast* (push-based), which enables users to retrieve data by simply listening to a particular channel, and (2) *on-demand* (pull-based), in which users send requests to get data.

* Corresponding author. Tel.: +886 3 8634029; fax: +886 3 8634010.
*E-mail addresses:* sclo@mail.ndhu.edu.tw (S.-C. Lo), alpchen@cs.nccu.edu.tw (A.L.P. Chen).

0169-023X/$ - see front matter 
doi:10.1016/j.datak.2006.02.002

The broadcast mechanism allows users to retrieve data with a lower power requirement (due to no user transmission being required) and with a cost that is independent of the number of users. However, the data broadcast has the drawback that data can only be accessed sequentially, and hence users must wait for desired data items to appear in the broadcast channel. Building an index of the broadcast data helps users to decide when and where their desired data items will be available, which allows the mobile device to be turned into a power-saving mode whilst waiting for desired data items. One crucial aspect of index broadcast is how to mix the index with data items on the broadcast channel so as to achieve the largest savings in time and energy for users.

A common metric to estimate the cost of data access for a particular index and data allocation scheme was introduced in [13]. The *access time* (the time elapsed from the moment a user sends a request to the moment the result is downloaded by the user) and *tuning time* (the time spent by the user downloading packets from the broadcast channel) are used to estimate the response time and power consumption for a data request, respectively. Here we assume that a single data item is retrieved in a data request. Using a tree-like index for the data items to be broadcast, the tuning time for a data request is directly proportional to the level of the desired data item in the index tree. Minimizing the average tuning time for a data request is expected to require a skewed index tree that has more popular data items located at higher levels (where the root is located at the highest level).

The construction of a skewed index tree is not analogous to constructing a Huffman tree by simply summing up access frequencies of data items. The index tree so constructed would lose the important sorted property, preventing it from functioning as a search tree. The methods proposed in [6] exhibit this problem. The type of Huffman tree termed an *alphabetic Huffman tree* proposed in [12] can normally function as a binary search tree, and the extension to an $n$-ary search tree was provided in [25].

A different allocation of nodes of the index tree into the broadcast channel would result in a different average access time for data requests. However, the average tuning time is unaffected by the broadcast allocation. The order and replication of nodes are the main considerations when generating different broadcast allocations. The replication can provide a fault-tolerant capability and can even speed up data searching, but a possible side effect is an increase in the average access time due to tremendous redundant data being broadcast [14]. The average access time can be shortened considerably by increasing the number of broadcast channels: ideally, the access time increases $k$-fold by using $k$ broadcast channels. Moreover, multichannel environments are quite popular in modern wireless networks for deceasing channel interference in overlapping coverage areas.

In this paper, we consider the problem of index and data allocation for any number of broadcast channels with the goal of minimizing the average access time. We follow the method proposed in [25] to construct an $n$-ary alphabetic Huffman tree over the data items to be broadcast, which helps achieve a lower average tuning time. We then develop our allocation method in two phases. In the first phase, we assume that there is no index/data replication in the allocation. We transform our problem under a single broadcast channel into the *directed optimal linear ordering* (*DOLO*) *problem* [2], which has a polynomial-time solution. We then use a heuristic approach to extend the result to the case with multiple broadcast channels. In the second phase, we improve our allocation method by using index replication.

The remainder of this paper is organized as follows. Section 2 briefly surveys related work, and Section 3 describes the broadcast structure and formally defines the problem under consideration. Sections 4 and 5 present the broadcast allocations without and with replication, respectively. Section 6 presents the results of a performance evaluation to demonstrate the superiority of our proposed scheme. Finally, we present conclusions and describe future work in Section 7.

## 2. Related work

Data broadcasting has become an attractive solution for data dissemination in wireless and mobile environments, which have been realized via different broadcast media such as TV signals, cables, and satellites. Since a broadcast channel may have a low bandwidth, we have to properly determine the broadcast data so as to maximize the broadcast efficiency. Ideally, only the most frequently accessed data items would be broadcast, but it is usually difficult to acquire the access frequencies of data items being broadcast since the users usually retrieve data items passively. A commonly used technique to determine the access frequencies of data items is to randomly drop them from the broadcast channel, which will force the users to explicitly send data

requests for them. The broadcast server can then estimate the access frequencies by the number of associated data requests [7,24,28].

The efficiency of a broadcast system is usually quantified by the amount of time and energy consumed in accessing broadcast data. When time is the major concern, a pure schedule of data items without any index information is normally performed on the broadcast channel. This is called the *pure data scheduling problem*. Periodic schedules were studied in [1,15,27], where the same data item has equal spacing on the broadcast channel, and data items with higher access frequencies are broadcast more frequently. This problem in a multichannel environment was considered in [10,21,30].

The retrieval of several data items in any order in the same data request complicates the pure data scheduling problem [5,18]. The solution relies on clustering those data items on the broadcast channel that appear in the same data request more frequently. The scheduling problem becomes more challenging when data items on the broadcast channel are accessed in a particular order. We can model the order relationships into an access graph where nodes and edges represent objects to be broadcast and their order, respectively. Moreover, we can associate each edge with a weight that indicates the access frequency of following that particular access order. The access graph can model anchor relationships of web pages, reference integrity constraints, or is-a relationships in the database; the broadcast objects can be web pages, relations, or classes in the database.

The allocation of objects modeled in an access graph to the broadcast channel was discussed in [3,9,16,19,23]. The use of relations or classes as broadcast objects was presented in [23], where the optimal allocation is found by a branch-and-bound searching algorithm. The use of an acyclic access graph under the single-channel and multichannel environments was discussed in [3] and [9], respectively, using a set of heuristic rules. The use of a general access graph under a single-channel environment was discussed in [19]. This type of problem can be generalized to the allocation of $n$ dependent objects on $m$ broadcast channels, and can be mapped to the task scheduling problem within a multiprocessor environment, which has been proven to be NP-hard [8].

Providing indexes on broadcast channels is mostly performed when the energy consumption of the mobile devices is the major concern in broadcast data access. This is called the *index and data scheduling problem*. The indexing techniques of tree-based indexing, hashing, and signatures were applied to this problem with a single broadcast channel in [11,13], and [14], respectively. The impact of inheritance and aggregation relationships on the index placement for an object-oriented database was studied in [4]. The issue of fault tolerance in the presence of channel errors was discussed in [17,26].

The index and data scheduling problem in a multichannel environment was discussed in [16,20,22,25,29] based on the adoption of a tree-based index. The broadcast channels were separated into index channels and data channels in [20,25], where the number of index channels was at least one in [20] and equal to the depth of the index tree in [25]. However, these methods lack flexibility when using broadcast channels. The data and index were scheduled separately in [22,29]. The data were mostly scheduled using techniques similar to those for the pure data scheduling problem, with the index information inserted into broadcast channels. In [22], only the local index to an individual broadcast channel was provided so that the users had to sequentially tune into broadcast channels to locate their desired data items. In [29], the index and data were uniformly stripped such that all broadcast channels would alternate between a data period (during which only the data are transmitted) and an index period. This method would nullify the benefit of having more channels due to certain transmission periods ending simultaneously.

Our proposed method adapts well to any number of broadcast channels. Most importantly, we schedule the index and data at the same time so as to maximize channel utilization and minimize the time requirement for data access. In our previous work [16], we developed a pruning algorithm that still had high time complexity and, moreover, had no index/data replication in the allocation.

## 3. Preliminary

### 3.1. Broadcast server

Fig. 1 shows the general architecture of a broadcast server. The database stores all the data items that mobile users are interested in. If we assume that these data items are identified by their key values (e.g., using

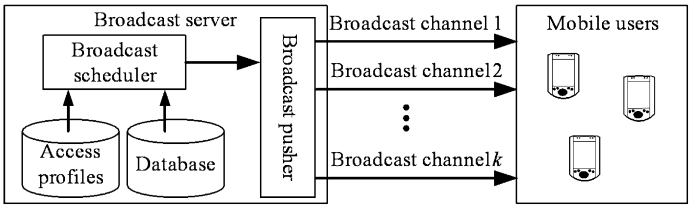


Fig. 1. Structure of a broadcast server.

numerical identification), the access profiles can provide statistical data about the access frequency of each data item in the database. The maintenance of access profiles is beyond the scope of this paper, and the reader is referred to [28]. The broadcast scheduler periodically looks up the access profiles and selects those data items that have been accessed frequently during the recent past (e.g., days, weeks, or months). The broadcast scheduler then constructs an index tree based on the key values of these selected data items. The next job is to generate a broadcast program, which can be viewed as a process to place the nodes in the index tree into several data streams. Each data stream will be cyclically pushed into each dedicated broadcast channel by the broadcast pusher. One complete transmission of a broadcast program is also called a broadcast cycle.

Here we assume that there are $k$ physical broadcast channels with equal bandwidths for the wireless broadcast services. These broadcast channels can be uniformly divided in the frequency domain from the available bandwidth of the particular certain wireless system (e.g., a satellite or cellular mobile system). Any mobile user can monitor the index information on a broadcast channel in order to search for a desired data item.

### *3.2. Broadcast program*

The scheduling to generate a broadcast program is our major concern. Suppose that a set of data items that are of different sizes are to be broadcast. We construct a skewed index tree for these data items whilst taking their access frequencies into account. This type of index tree can be built using the alphabetic Huffman tree, as mentioned in Section 1. For example, suppose that we have five data items denoted by keys $A$–$E$. We construct a binary alphabetic Huffman tree as shown in Fig. 2, in which the leaf node is the data node containing one data item with its access frequency labeled below. The nonleaf node is the index node. Our goal is to allocate these index and data nodes into broadcast channels such that mobile users can most efficiently access their desired data items.

The index and data nodes are transported using buckets that form the basic transmission units on broadcast channels. Each bucket has the same size (in bytes). We assume that a bucket can carry at most one index node of the maximal size among all index nodes in the index tree. That is, each index node is transported using one bucket, with tail padding where necessary to ensure that all buckets are of uniform size. A data node that cannot be transported using a single bucket is fragmented into several buckets.

The internal structure of a bucket is shown in Fig. 3. The first field is the preamble for channel synchronization. The second field records the sequence number of a bucket and is used to distinguish the buckets within

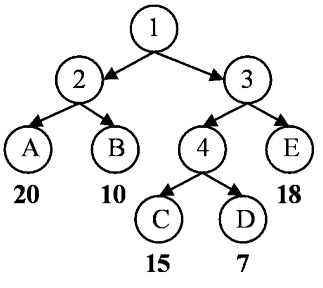


Fig. 2. An example index tree.

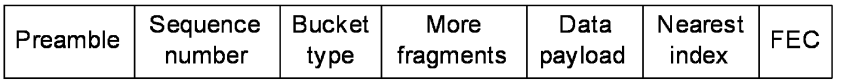


Fig. 3. Internal structure of a bucket.

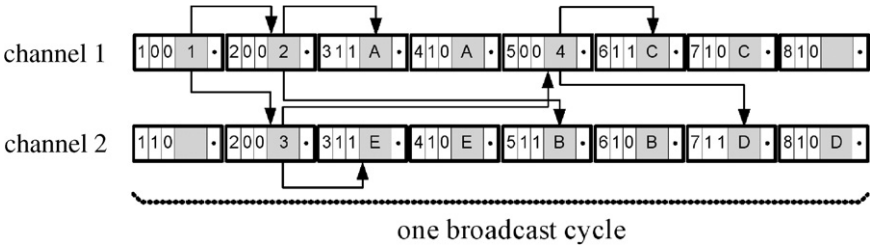


Fig. 4. Broadcast program.

a single broadcast cycle on a given broadcast channel. The third field indicates the type of node (either index or data) that the bucket holds. The fourth field indicates whether the current bucket holds the last fragment of a certain data node (a value of 0 indicates the last fragment). The fifth field can hold one index node or one fragment of a larger data node. The sixth field is a pointer that indicates the next nearest bucket that contains the root node of the index tree (named *index root*). The last field is the forward error correction (FEC) code for recovering possible bit errors in the bucket.

In our example, we assume that every data node can be fragmented into two buckets, and use B($C$, $S$) to denote the bucket with sequence number $S$ on broadcast channel $C$. Fig. 4 shows a broadcast program to the index tree of Fig. 2 on two broadcast channels that each contain eight buckets within a broadcast cycle. Broadcast channel 1 contains index nodes 1, 2, and 4, and data nodes $A$ and $C$; and broadcast channel 2 contains index node 3, and data nodes $E$, $B$, and $D$. Note that two of the buckets carry null data (called *null buckets*) on the broadcast channels: B(1, 8) and B(2, 1). The purpose of null buckets is explained in Section 4.2.

In the original index tree, each index node has index pointers pointing to its child nodes; these index pointers must be maintained in our broadcast program. Let us assume that an index node is located at B($C_i$, $S_i$), and that one of its index pointers is pointing to a child node located at B($C_j$, $S_j$). We use the pair $\langle C_j, S_j\text{–}S_i\text{–}1\rangle$ to represent this index pointer, which denotes that the child node can be accessed by waiting for $S_j\text{–}S_i\text{–}1$ buckets after switching to broadcast channel $C_j$. For example, index node 1 at B(1, 1) has two child nodes (index nodes 2 and 3) that are located at B(1, 2) and B(2, 2), respectively. The corresponding index pointers are represented by the pairs $\langle 1, 0\rangle$ and $\langle 2, 0\rangle$. We depict index pointers in Fig. 4 using directed lines. An index pointer pointing to a data node will be guided to the first fragment of the data node on the broadcast channel.

The field with a dot symbol inside a bucket indicates the location of the nearest index root. In our example, for a bucket with sequence number $S$, this field stores the pair $\langle 1, 8\text{–}S\rangle$, which denotes that the index root can be accessed by waiting for 8–$S$ buckets after switching to broadcast channel 1.

### 3.3. Broadcast data access

We now explain the steps involved in locating a desired data item in a data request. We suppose that a mobile device is equipped with a single receiver, so those buckets with the same sequence number on different broadcast channels cannot be accessed simultaneously by the same user.

If a user wants to retrieve a data item with key $K$, the broadcast data access proceeds as follows:

(1) Tune to one of the broadcast channels and download one bucket. Check if the downloaded bucket contains the index root: if it does, proceed to the next step; otherwise, follow the nearest-index information in the bucket to retrieve the index root.

(2) Compare key $K$ with the key values of the current bucket and follow a sequence of index pointers to retrieve the desired data item.

During the broadcast access, we assume that the user knows which frequency band to switch to in order to access a given channel number. The time required to perform channel switching might mean that the user is not be able to locate the next relevant bucket in time, particularly for an immediate-neighbor bucket in another channel. This problem can be avoided by ensuring that the preamble in the bucket is of sufficient length.

### 3.4. Problem definition

The access time for a data request on broadcast channels can be divided into two parts: *probe wait* and *data wait*. The probe wait is the time taken to reach a bucket containing the index root after the first probe, whereas the data wait is the time from just after the probe wait to the moment that the desired data item is downloaded. An example scenario explaining this is shown in Fig. 5.

We assume that the access time is measured in units of buckets. We denote the average access time for any data request to a particular broadcast program by $AT$. Let the average probe wait and average data wait be denoted by $PW$ and $DW$, respectively; hence

$$AT = PW + DW. \tag{1}$$

$PW$ is the same for any data request, and $DW$ can be further represented by

$$DW = \left( \sum_{D_i \in D} w_{D_i} \times g_{I_1 \to D_i} \right) \Big/ \sum_{D_i \in D} w_{D_i}, \tag{2}$$

where $g_{I_1 \to D_i}$ denotes the distance in buckets between the bucket containing index node $I_1$ (i.e., an index root) and the bucket containing the last fragment of data node $D_i$ within the same broadcast cycle. For example, $g_{1 \to B}$ is 6 (the subtraction of the sequence numbers of these two buckets, plus 1) in Fig. 4.

Our example problem is formally defined as follows:

*Index and data allocation problem* (*abbreviated as IDA*): We are given $k$ ($k \geqslant 1$) broadcast channels and an index tree that is a type of alphabetic Huffman tree, and includes index and data nodes. Let the broadcast channels be $C = \{C_1, C_2, C_3, \ldots, C_k\}$, the set of index nodes be $I = \{I_1, I_2, I_3, \ldots, I_m\}$, and the set of data nodes be $D = \{D_1, D_2, D_3, \ldots, D_n\}$. Each data node $D_i$ containing one data item is associated with a weight $w_{D_i}$ that represents its average access frequency. The IDA (or $k$-IDA with $k$ broadcast channels) is to allocate the nodes in $I \cup D$ on broadcast channels in $C$ such that the average access time is minimized.

In Sections 4 and 5 we discuss two categories of IDAs based on the enforcement of replication: (1) *IDA without replication*, where we enforce that a complete index or data node can only appear once within a broad-

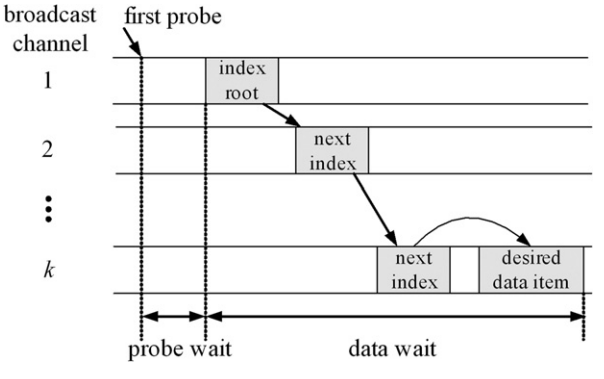


Fig. 5. Time spent on accessing a desired data item.

cast cycle (the broadcast program shown in Fig. 4 belongs to this category); and (2) *IDA with replication*, where a complete index or data node can appear more than once within a broadcast cycle.

## 4. IDA without replication

Our solution to the IDA without replication can be divided into two parts: (1) derive an optimal allocation for single-channel cases and (2) then derive a heuristic allocation for multichannel cases.

### 4.1. Optimal solution to single-channel cases

In a 1-IDA without replication, the leading bucket of each broadcast cycle contains the index root. Also, the length of a broadcast cycle is directly related to the total number of nodes in the index tree. Hence, $PW$ is equal to half the length of a broadcast cycle and is independent of the method of allocation. Consequently, the minimization of $AT$ becomes the minimization of $DW$.

We tackle this problem by introducing another problem, the DOLO problem [2], as defined below.

*DOLO problem*: We are given a directed graph G(V, E), where V is the set of vertices and E is the set of edges. Each edge $(u, v) \in$ E is associated with weight $c_v$ ($c_v > 0$). The DOLO problem involves finding an optimal linear order of these vertices in V, such that vertices $u$ and $v$ located at positions $f(u)$ and $f(v)$ obey the constraint $f(u) < f(v)$ whenever $(u, v) \in$ E, and such that $\sum_{(u,v)\in E} c_v \times h_{u\to v}$ is minimum (where $h_{u\to v} = f(v) - f(u)$). This problem is NP-complete, but is solvable in polynomial time if G is a tree.

For example, one possible linear order of the graph of Fig. 6a is shown in Fig. 6b with the following cost: $2(2-1) + 5(3-1) + 3(4-1) + 4(4-3) = 25$. In the following, we show that a given index tree in the 1-IDA can be exactly transformed to a graph in the DOLO problem. The transformation is performed using the following steps:

1. Let the edges in the index tree be directed from the parent to the child.
2. Associate each index node from the bottom upwards with a weight equal to the sum of all weights of its children.
3. Let each edge in the index tree have a weight equal to that of the node pointed to by the edge.

One example of this transformation is shown in Fig. 7. Accordingly, we can rewrite $DW$ as follows:

$$
\begin{aligned}
DW &= \sum_{D_i\in D} w_{D_i} \times g_{I_1\to D_i} \Bigg/ \sum_{D_i\in D} w_{D_i} = [w_A \times g_{1\to A} + w_B \times g_{1\to B} + w_C \times g_{1\to C} + w_D \times g_{1\to D} + w_E \times g_{1\to E}] \Bigg/ \sum_{D_i\in D} w_{D_i} \\
&= [w_A \times (g_{1\to 2} + g_{2\to A}) + w_B \times (g_{1\to 2} + g_{2\to B}) + w_C \times (g_{1\to 3} + g_{3\to 4} + g_{4\to C}) \\
&\quad + w_D \times (g_{1\to 3} + g_{3\to 4} + g_{4\to D}) + w_E \times (g_{1\to 3} + g_{3\to E})] \Bigg/ \sum_{D_i\in D} w_{D_i} \\
&= [(w_A + w_B) \times g_{1\to 2} + w_A \times g_{2\to A} + w_B \times g_{2\to B} + (w_C + w_D + w_E) \times g_{1\to 3} \\
&\quad + (w_C + w_D) \times g_{3\to 4} + w_C \times g_{4\to C} + w_D \times g_{4\to D} + w_E \times g_{3\to E}] \Bigg/ \sum_{D_i\in D} w_{D_i} \\
&= [w_2 \times g_{1\to 2} + w_A \times g_{2\to A} + w_B \times g_{2\to B} + w_3 \times g_{1\to 3} + w_4 \times g_{3\to 4} \\
&\quad + w_C \times g_{4\to C} + w_D \times g_{4\to D} + w_E \times g_{3\to E}] \Bigg/ \sum_{D_i\in D} w_{D_i} \\
&= \sum_{(u,v)\in E} w_v \times g_{u\to v} \Bigg/ \sum_{D_i\in D} w_{D_i}
\end{aligned}
$$

We can make the following observations: The quantity $\sum_{D_i\in D} w_{D_i}$ is a constant for a given index tree. The notation $g_{u\to v}$ is equivalent to the notation $h_{u\to v}$ if all index and data nodes are of size 1. Under this assumption, we

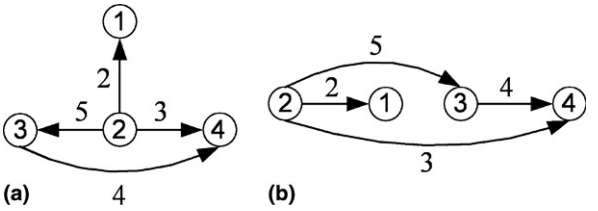


Fig. 6. An example of linear order.

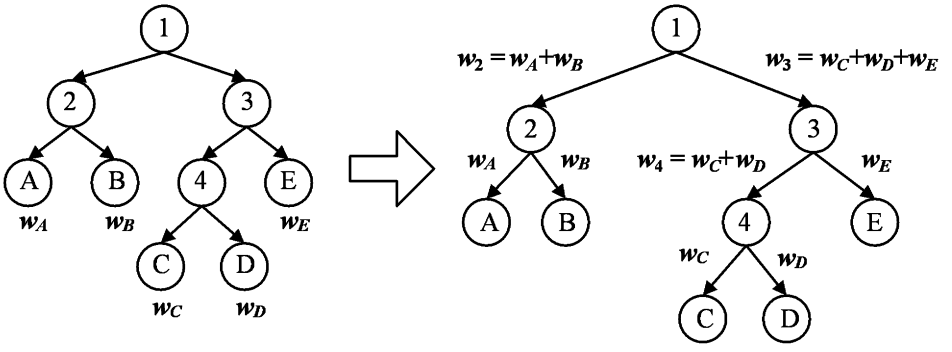


Fig. 7. Transformation of an index tree.

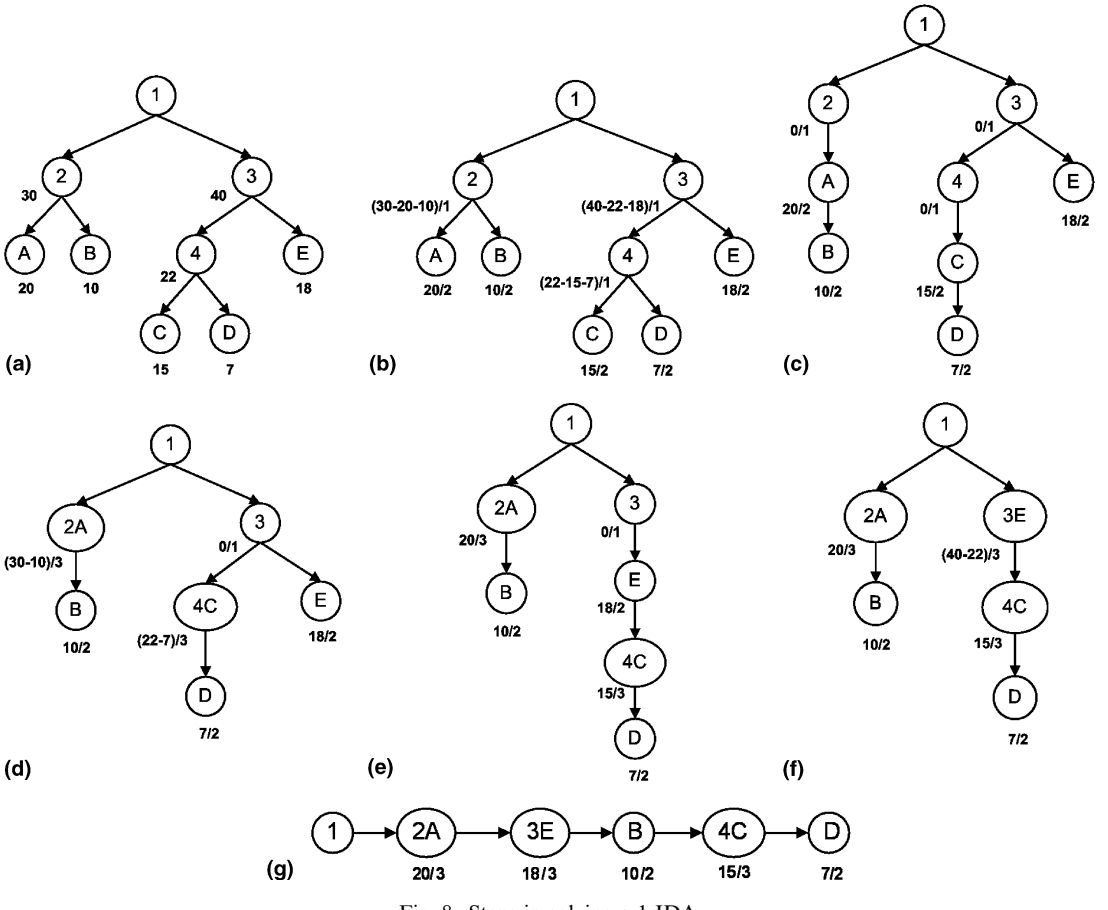


Fig. 8. Steps in solving a 1-IDA.

get $\sum_{(u,v)\in E} w_v \times g_{u\to v} = \sum_{(u,v)\in E} c_v \times h_{u\to v}$ by setting $c_v = w_v$. As a result, the optimal linear order of vertices of the graph is just the same as the optimal index and data allocation of nodes of the corresponding index tree.

Fortunately, our constructed index has a tree-based structure, so we can use the polynomial-time algorithm proposed in [2] to solve our problem. More importantly, this algorithm can naturally deal with cases where the nodes are of different sizes. In Fig. 8, we use an example to describe how to apply this algorithm to our problem.

First, we transform the index tree into a directed and weighted graph by following the steps mentioned above. Whether weight $w_v$ is associated with the edge leading to node $v$ or with node $v$ in the graph has the same effect. For convenience, we associate each node $v$ with weight $w_v$. An example graph is shown in Fig. 8a after the transformation. As stated in [2], the nodes in the graph should be adjusted first so that the node weights decrease monotonically from top to bottom. This property holds since the weights in our index tree are obtained by a cumulative summation from the bottom upwards.

Second, we calculate the *difference ratio* of each node in the graph, which is defined for node $v$ as $R(v) = (w_v - \overline{w}_v)/l_v$, where $\overline{w}_v$ is the sum of all weights of the children of node $v$ in the graph and $l_v$ is the size of node $v$. $\overline{w}_v$ is zero if node $v$ is a leaf node (e.g., $R(2) = (w_2 - (w_A + w_B))/l_2$). The calculation of the difference ratio of each node is shown in Fig. 8b. This algorithm is implemented by gradually merging the nodes from the bottom upwards into a chain according to the descending order of their difference ratios. All the descendants of a subroot with more than one child are first merged into a chain. This process starts from the *terminal subroot* that has no other subroots in its descendants. In Fig. 8b, we have two terminal subroots of nodes 2 and 4. Hence, we merge nodes $A$ and $B$, and nodes $C$ and $D$ into two individual chains according to the descending order of difference ratios in Fig. 8c.

After the merge, we have to confirm that the difference ratios of nodes in the chain monotonically decrease from top to bottom. For example, nodes 2 and $A$, and nodes 4 and $C$ in Fig. 8c violate this rule. In this case, two neighboring nodes in the chain that violate the descending rule would be condensed into a single node (called a *condensed node* hereafter). If two nodes $A$ and $B$ (where $B$ is a child of $A$) are condensed, we define the *condensed order* to be $AB$. If a condensed node $P$ contains nodes $p_1, p_2, \ldots, p_k$ (which follow the condensed order), the difference ratio of $P$ is defined as $R(P) = (w_{p_1} - \sum_{i=1\sim k} \overline{w}_{p_i})/\sum_{i=1\sim k} l_{p_i}$, where $\sum_{i=1\sim k} \overline{w}_{p_i}$ is the sum of all weights of nodes in the set $\cup_{i=1\sim k} Child(p_i) - \cup_{i=1\sim k} p_i$ (i.e., the set of nodes that are in the union of children of $p_i$ but not in the union of $p_i$). For example, if $P$ containing nodes 2 and $A$ ($p_1 = 2$, $p_2 = A$) is a condensed node, we have $\cup_{i=1\sim 2} Child(p_i) - \cup_{i=1\sim 2} p_i = \{A, B\} - \{2, A\} = \{B\}$. Note that the parent–child relationship mentioned here is referred to as the initial graph (i.e., Fig. 8a). Hence, $R(2A) = (w_2 - w_B)/(l_2 + l_A)$.

Consequently, we condense nodes 2 and $A$, and nodes 4 and $C$ into two condensed nodes, and recalculate their difference ratios in Fig. 8d. The condensing process is performed repeatedly within a chain so as to maintain the descending rule. Next, we merge the descendants of terminal subroot 3 into a chain in Fig. 8e. Again, nodes 3 and $E$ violate the descending rule and are condensed in Fig. 8f. Finally, we obtain a single chain in Fig. 8g. The optimal index and data allocation is now simply the order of nodes in the chain from the head to the tail: $12A3EB4CD$.

To efficiently maintain the chains during the process, we need a data structure that supports the following operations: insert a node, delete a node with the largest weight, and combine two chains into one. The max heap (the top node is always the one with the largest weight) is one such data structure that supports these operations in logarithmic time. The complete algorithm is listed as Algorithm 1.

**Algorithm 1** (*The optimal solution to the* 1*-IDA without replication*).

(1) Let the parent have a weight equal to the sum of all weights of its children.
(2) Calculate the difference ratio of each node from the bottom upwards. If the difference ratio of a parent is strictly less than that of its only child, condense these two nodes into one.
(3) Start from a terminal subroot to form a max heap from all its descendants. Condense the terminal subroot and the top node of the max heap into one if the descending order of the difference ratio is violated. Continually check the condensed node with the new top node of the max heap, and condense these two nodes into one if necessary. Finally, insert the condensed node into the max heap. Do this to all terminal subroots until the entire graph becomes a max heap.

(4) Delete the node from the top of the max heap and repeat this step until all nodes are deleted. Sequentially traverse the internal nodes in these deleted nodes and output the optimal index and data allocation.

The complexity of this algorithm is dominated by heap operations and is equal to $O(n\log(n))$, where $n$ is the total number of nodes in the index tree.

### 4.2. Heuristic solution to multichannel cases

In determining the optimal index and data allocation on multiple broadcast channels, we have to consider both the time consumed in accessing broadcast data and the energy consumed in switching channels. Here we propose a heuristic on the allocation by using the following rules:

**Rule 1.** A node with an earlier order in the 1-IDA solution has a higher allocation precedence into the broadcast channel.
**Rule 2.** A node with a larger weight has a higher allocation precedence into the same broadcast channel of its parent than its sibling nodes.
**Rule 3.** A broadcast channel with the shortest length of already allocated nodes is used first.
**Rule 4.** We pad sufficient null buckets before a node that is currently allocated, if needed, so that this node will just follow its parent.

Rule 1 aims at time efficiency by considering that the optimal allocation of 1-IDA has a certain influence on the allocation of $k$-IDA. Rule 2 reduces the average number of channel switches by putting those nodes along the path from the index root to the data node with a larger weight in the index tree into the same broadcast channel. Rule 3 increases time efficiency by balancing the load (number of allocated nodes) among broadcast channels. If a node can be allocated so that it is directly adjacent to its parent on the same broadcast channel, Rule 2 has a higher priority than Rule 3. Rule 4 is based on the principle that we cannot access a child node until its parent node has been accessed.

In the following, we use the output of Fig. 8g as an example to generate a heuristic answer to 2-IDA without replication. Table 1 summarizes the node information to be allocated. The processing steps are shown in Fig. 9. We maintain a stack structure to record the nodes that are free to be allocated. We first push the index root into the stack, and then we pop at most two nodes (corresponding to the available number of broadcast channels) each time from the stack and allocate them into broadcast channels. In this case, we allocate node 1 to the first broadcast channel. Next, all the children of the nodes just allocated will be pushed into the stack, because these children become free for allocation. Here we push nodes 2 and 3 into the stack in the order determined by Rule 1 (i.e., the node with earlier optimal order would be pushed later). Hence, we maintain the order $(2,3)$ in the stack from the top downwards.

Again, we pop at most two nodes (nodes 2 and 3) from the stack. The order of allocation of these nodes to the broadcast channels is determined by Rule 2. In this case ($w_3 > w_2$), the allocation order is $(3,2)$. Thus, we allocate node 3 to be adjacent to its parent node 1. According to Rule 3, node 2 will be allocated at

Table 1
Example reference table

| Node | Optimal order | Weight |
|---|---|---|
| 1 | 1 | 70 |
| 2 | 2 | 30 |
| 3 | 5 | 40 |
| 4 | 7 | 22 |
| A | 3 | 20 |
| B | 4 | 10 |
| C | 8 | 15 |
| D | 9 | 7 |
| E | 6 | 18 |

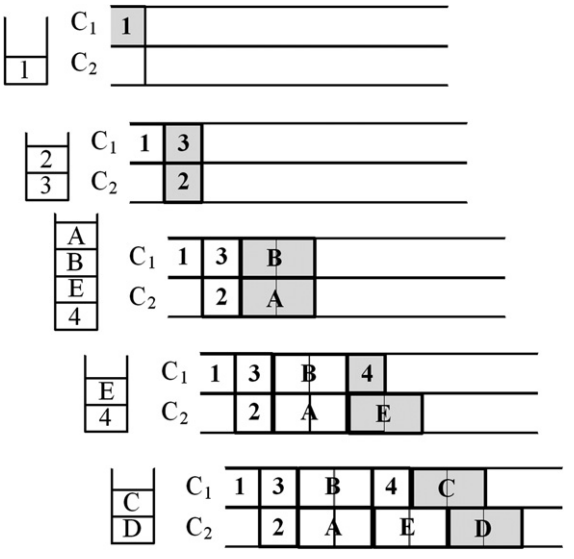


Fig. 9. Steps for generating the result of a 2-IDA.

B(2, 1) but will overlap its parent node 1. Therefore, we pad one null bucket at B(2, 1) and allocate node 2 at B(2, 2).

Next, we push nodes 4, *E*, *B*, and *A* into the stack according to the order guided by Rule 1. Then nodes *A* and *B* with allocation order (*A*, *B*) are allocated. We put node *A* into the same broadcast channel as its parent node 2. We allocate the remaining nodes in a similar way. The complete algorithm is described as follows:

**Algorithm 2** (*The heuristic to the k-IDA without replication*).

(1) Use Algorithm 1 to obtain the optimal linear order of nodes of the index tree.
(2) Push the index root into the stack.
(3) While the stack is not empty:
(4) Pop at most $k$ nodes from the stack. The allocation order of these nodes to broadcast channels is determined by Rule 2. The channel selection for a being-allocated node is decided by Rule 3. In the actual allocation, we will pad sufficient null buckets if Rule 4 is triggered.
(5) Push all the children of the nodes allocated in step 4 into the stack according to the order guided by Rule 1.

Step 1 costs $O(n\log(n))$, as stated earlier. In the While loop, the operations are dominated by the sorting with bound $O(n\log(n))$, and the number of loops is about $n/k$. Therefore, the complexity of this algorithm is $O((n^2/k)\log(n))$.

## 5. IDA with replication

To retrieve any broadcast data, the user first has to access a bucket that contains the index root on the broadcast channel. Our solution so far has only one such bucket within a broadcast cycle. To reduce the probe wait after tuning into the broadcast channel, we can replicate the index root on the broadcast channel. Indeed, the replication can increase node availability so as to reduce access time.

The replication comprises data replication and index replication. Data replication has been widely discussed with reference to the pure data scheduling problem, and can be performed after index replication. We do not discuss data replication in this paper. Index replication differs from data replication, since index nodes have order relationships. In the following, we present an enhanced solution to the IDA problem by using index replication.

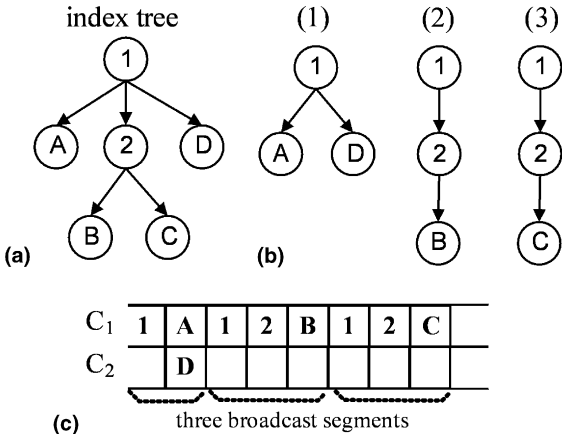


Fig. 10. Tree split.

### 5.1. Index replication

We model index replication as a process to split an index tree. An index tree can be split into several pieces, each of which contains some data nodes and relevant index nodes. We call each piece an *induced tree*. For example, we can split the index tree of Fig. 10a into the three induced trees of Fig. 10b. Then we can allocate each of these induced trees[1] into broadcast channels using Algorithm 1 or 2. Fig. 10c shows an example where index nodes 1 and 2 become replicated. Actually, we can determine the degree of replication by controlling the number of induced trees that are generated.

We call the broadcast data allocated from an induced tree a *broadcast segment* (see Fig. 10c). A broadcast cycle comprises broadcast segments that are each generated from each of the induced trees. In the following, we introduce a tree-split algorithm that can generate a set of induced trees with the goal of minimizing the average access time.

We first introduce the following terminology:

$ancestor(x)$ set of nodes that are ancestors of node $x$
$descendant(x)$ set of nodes that are descendants of node $x$
$LN_r$ set of nodes at the $r$th level
$w_x$ weight of node $x$ (the setting is the same as that in Algorithm 1)
$l_x$ size of node $x$ in buckets
$|S|$ sum of the sizes of nodes in set $S$ (i.e., $|S| = \sum_{x \in S} l_x$)
$L_x$ cumulative sizes of node $x$ (i.e., $L_x = l_x + |descendant(x)|$)
$SEG_k$ average length in buckets of a broadcast segment under $k$ broadcast channels
$BCAST_k$ length in buckets of a broadcast cycle under $k$ broadcast channels

### 5.2. Level split

Our proposed tree split is performed by level splits from the top downwards in the index tree. In this subsection, we introduce the basic level-split procedure at a certain level (say $r$) of an index tree. The procedure is outlined as follows: We first collect the nodes at the $r$th level of the index tree and denote them by $LN_r$. An induced tree is then either generated by some data nodes (Case 1) or by one of the index nodes (Case 2) in $LN_r$:

**Case 1**: We group the data nodes with common parents in $LN_r$. For a data group $d_i$, we can generate an induced tree that contains the following nodes:

[1] Note that the order of allocating these induced trees can be arbitrary, since they have become independent components.

$$\bigcup_{x_i \in d_i} ancestor(x_i) \cup \{x_i\}.$$

The maximal number of induced trees generated from the data nodes in $LN_r$ is equal to the number of data groups.

**Case 2**: We denote the children of an index node in $LN_r$ by set $G$. For a subset of $G$ (denoted by $H$), we can generate an induced tree that contains the following nodes:

$$\bigcup_{y_i \in H} ancestor(y_i) \cup \{y_i\} \cup descendant(y_i).$$

Basically, each disjoined subset of $G$ can generate an induced tree. Suppose that an index node has $m$ children and we need to generate $n$ induced trees from this node. This process can be viewed as the distribution of these $m$ children into $n$ nondistinct groups so that no group is empty. For each group, we combine the nodes in the group with their ancestors and descendants into an induced tree. The maximal number of induced trees generated by this index node is $m$ (i.e., $n = m$).

For example, we can split the index tree of Fig. 10a by nodes in $LN_2 = \{A, 2, D\}$ into three induced trees. Induced tree 1 is generated from the data group of nodes $A$ and $D$, and induced trees 2 and 3 are generated from index node 2.

We can derive the following lemmas from the level split:

**Lemma 1.** *An index node $x$ and all its ancestors will appear in all induced trees generated by this index node. That is, the replicated nodes of these induced trees are in the set $ancestor(x) \cup \{x\}$.*

**Lemma 2.** *Suppose that an index node $x$ is located at the $r$th level of the index tree. If $n$ induced trees are generated by this index node, the total size of nodes in these induced trees is $n \times |ancestor(x) \cup \{x\}| + |descendant(x)|$ regardless of the method of generation. If we denote the average size of an index node by $I_{\mathrm{SIZE}}$, this equation can be rewritten as $n \times r \times I_{\mathrm{SIZE}} + (L_x - l_x)$.*

**Proof.** Since every induced tree contains the replicated nodes in $ancestor(x) \cup \{x\}$ according to Lemma 1, the first added term ($n \times |ancestor(x) \cup \{x\}|$) counts the total sizes of replicated nodes in these induced trees. The second added term ($|descendant(x)|$) counts the total sizes of nonreplicated nodes. □

**Lemma 3.** *If we perform a level split from all index nodes in $LN_r$ and generate a total of $N_r$ induced trees, the total size of nodes in these induced trees is*

$$N_r \times r \times I_{\mathrm{SIZE}} + \sum_{x \in LN_r \text{ and } x \text{ is an index node}} (L_x - l_x).$$

**Proof.** Suppose that each index node $x$ generates $n_x$ induced trees ($n_x > 0$ and $N_r = \sum_{x \in LN_r} n_x$). According to Lemma 2, the total size of nodes can be approximated as

$$\sum_{x \in LN_r \text{ and } x \text{ is an index node}} n_x \times r \times I_{\mathrm{SIZE}} + (L_x - l_x) = N_r \times r \times I_{\mathrm{SIZE}} + \sum_{x \in LN_r \text{ and } x \text{ is an index node}} (L_x - l_x). \qquad \square$$

### 5.3. Tree split

The tree split is performed by each level split from the top downwards. We use the following pseudocode to describe our proposed algorithm:

1. Put the index tree into *TreeSet*.
2. For $r = 1$ to the depth of the index tree.
3. Collect $LN_r$ from all the trees in *TreeSet*.
4. Generate induced trees from all the data nodes in $LN_r$.

5. Generate a maximum of $N_r$ induced trees from the index nodes in $LN_r$.
6. Replace all the trees in *TreeSet* by newly generated induced trees.

In this algorithm, we have to decide how to choose the $N_r$ value and how to generate these $N_r$ induced trees.

*5.3.1. Number of induced trees*

We compute $N_r$ so as to minimize the average access time. The average access time is computed using (1). We observe the following:

1. The beginning of each broadcast segment contains only the index root, so $PW$ is equal to half the average length of a broadcast segment.
2. If we use the average value of $g_{I_1 \to D_i}$ to approximate each individual value of $g_{I_1 \to D_i}$ in (2), $DW$ will be equal to the average value of $g_{I_1 \to D_i}$, which is equal to half the length of a broadcast cycle.

Therefore, we have $AT = PW + DW = SEG_k/2 + BCAST_k/2$. We can approximate $SEG_k$ and $BCAST_k$ as $SEG_1/k$ and $BCAST_1/k$, respectively. Furthermore, $SEG_1$ can be approximated as $BCAST_1/n$ if there are a total of $n$ broadcast segments within a broadcast cycle. If we generate $N_r$ induced trees from the index nodes in $LN_r$ and allocate them into $k$ broadcast channels, we will have (from Lemma 3)

$$BCAST_1 = N_r \times r \times I_{\text{SIZE}} + \sum_{x \in LN_r \text{ and } x \text{ is an index node}} (L_x - l_x).$$

Therefore,

$$AT = \frac{SEG_k}{2} + \frac{BCAST_k}{2} = \frac{BCAST_1}{2k \times N_r} + \frac{BCAST_1}{2k}.$$

$$\frac{\mathrm{d}_{AT}}{\mathrm{d}_{N_r}} = \frac{r \times I_{\text{SIZE}}}{2k} - \frac{2k \times \sum_{x \in LN_r \text{ and } x \text{ is an index node}} (L_x - l_x)}{(2k \times N_r)^2}.$$

The average access time is a minimum when $\frac{\mathrm{d}_{AT}}{\mathrm{d}_{N_r}} = 0$, so we have

$$N_r = \left\lfloor \frac{\sum_{x \in LN_r \text{ and } x \text{ is an index node}} (L_x - l_x)}{r \times I_{\text{SIZE}}} \right\rfloor. \tag{3}$$

Given $N_r$, we have to determine the number of induced trees that will be generated from each index node in $LN_r$. Here we use a weighted function defined as

$$f(x) = L_x^{1-\alpha} \times w_x^{\alpha}. \tag{4}$$

We compute $f(x)$ for each index node $x$ in $LN_r$ and use the fraction of $f(x)$ to distribute the $N_r$ value to node $x$. The basic idea is attempting to split a larger subtree in size or in weight into more induced trees. Procedure Distribute_Num performs this operation.

**Procedure** Distribute_Num( )
**Input**: $LN_r$, $N_r$
**Output**: $n_x$ (number of induced trees that will be generated by node $x$)
(1) For each index node $x$ in $LN_r$, compute $f(x)$.
(2) Sort these $x$'s in $LN_r$ according to the descending order of $f(x)$.
(3) $Total\_F = \sum_{x \in LN_r \text{ and } x \text{ is an index node}} f(x)$.
(4) For each index node $x$ in $LN_r$:
(5) $\quad c_x =$ number of children of node $x$.
(6) $\quad n_x = \min(c_x, \lfloor N_r \times f(x)/Total\_F \rfloor)$.
(7) $\quad N_r = N_r - n_x$.
(8) $\quad Total\_F = Total\_F - f(x)$.

*5.3.2. Generation of induced trees*

Given the number of induced trees that an index node should generate, we have to determine the real generation of these induced trees. As discussed in Section 5.2, this generation is equivalent to distributing the children into groups. Again, we use the same weighted function in (4) to perform this distribution.

We first compute $f(x)$ for each of the children of an index node, and then we distribute these children into a given number of groups such that the summation of the $f(x)$ values is similar in each group. The basic idea is attempting to merge several small subtrees into a large induced tree. This problem is similar to the bin packing problem, and a possible heuristic is listed in procedure Gen_InducedTree.

**Procedure** Gen_InducedTree( )
**Input**: index node $x$, $n$
**Output**: $n$ induced trees
(1) Compute $f(x_i)$ for each child node $x_i$ of node $x$.
(2) Compute $avg\_f = \sum_{x_i \text{ is a child of node } x} f(x_i)/n$.
(3) Sort these child nodes in descending order of the $f(x_i)$ values.
(4) Let $y_i$ denote the $i$th sorted child.
(5) $j = 1$.
(6) For each $y_i$:
(7) If the sum of $f(y_i)$ values of allocated child nodes in group $j < avg\_f$,
(8) put $y_i$ into group $j$.
(9) Else $j = j + 1$.
(10) For each group $j$:
(11) Generate an induced tree that contains the nodes in the group and their ancestors and descendants.

*5.3.3. Proposed algorithm and running example*

Algorithm 3 lists the entire process of tree split. The tree split stops when no new entry is created in *TreeSet*.

**Algorithm 3** (*The heuristic to the k-IDA with index replication*).

(1) $r = 1$.
(2) Put the index tree into *TreeSet*.
(3) While $r \leqslant$ the depth of the index tree:
(4) Collect $LN_r$ from all the trees in *TreeSet*.
(5) Generate induced trees from all the data nodes in $LN_r$.
(6) Compute $N_r$ according to (3).
(7) Call procedure Distribute_Num to determine the number of separate induced trees.
(8) Call procedure Gen_InducedTree to generate the given number of induced trees.
(9) Modify the trees in *TreeSet* by pruning the nodes, which have generated some induced trees in steps 5 and 8, and their descendants.
(10) Insert the new generated induced trees into *TreeSet*.
(11) If no new tree is generated in the current loop then end the loop, else $r = r + 1$.
(12) Input each tree in *TreeSet* to Algorithm 1 as $k = 1$ or Algorithm 2 as $k > 1$.

Let $n$ denote the total number of nodes in the index tree. During the While loop (about $\log(n)$ iterations), the node counting (for cumulative weights and sizes, etc.) dominates the cost in computing $N_r$ and Distribute_Num. The counting of each node will visit at most $n$ nodes, so the total counting cost is $O(n\log(n))$. The sorting dominates the cost of Gen_InducedTree, and the cumulative cost during the loop is $O(n\log(n))$. Including the analysis of Algorithms 1 and 2, the complexity of this algorithm is $O(n\log(n))$ for $k = 1$ and is $O((n^2/k)\log(n))$ for $k > 1$.

We now use a running example to explain the tree split. Taking the index tree of Fig. 11a as the input, we have $LN_1 = \{1\}$ and $N_1 = \lfloor\sqrt{13/1}\rfloor = 3$. Since an index node can generate at most as many induced trees as the number of its children, we generate two induced trees 1–1 and 1–2 from index node 1 as shown in Fig. 11b.

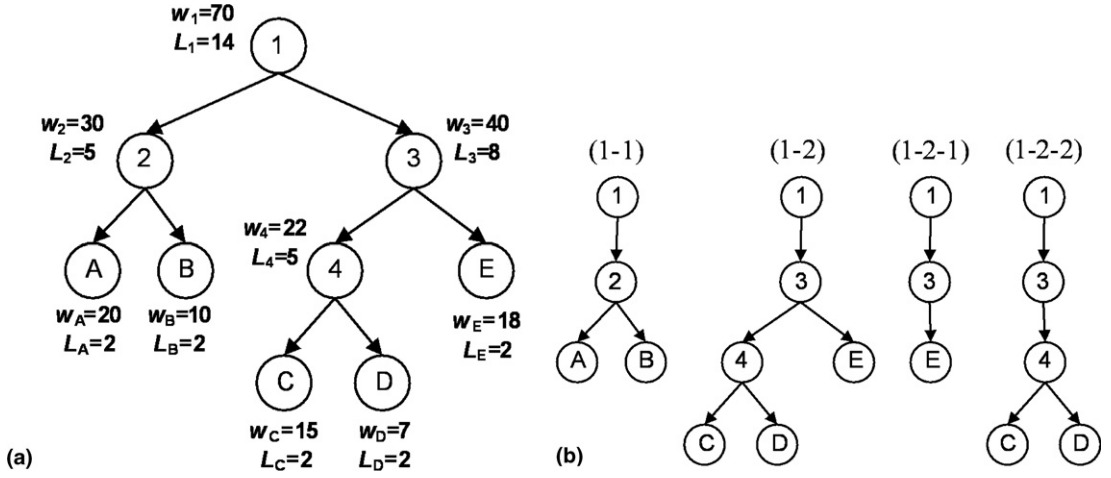


Fig. 11. A running example of tree split.

Then we consider $LN_2 = \{2, 3\}$ and compute $N_2 = \lfloor\sqrt{11/2}\rfloor = 2$. Next, we distribute $N_2$ using procedure Distribute_Num, and get $n_2 = 0$ and $n_3 = 2$. Hence, induced tree 1–1 remains unchanged and we further split induced tree 1–2 into two induced trees 1–2–1 and 1–2–2 from index node 3. Next, we consider $LN_3 = \{A, B, 4, E\}$ and compute $N_3 = \lfloor\sqrt{4/3}\rfloor = 1$. The data nodes in $LN_3$ have been located in their corresponding induced trees. Therefore, we obtain the final three induced trees 1–1, 1–2–1, and 1–2–2.

## 6. Performance evaluation

We used simulations to quantify the performances of Algorithms 1–3. We first generate a set of data nodes whose access frequencies follow the Zipf($\mu, \theta$) distribution [31], where $\mu$ is the mean and $\theta$ increases with the skewness of the data. The size of a data node in terms of buckets is randomly selected from the range 10–20. We construct an index tree with a given fanout (the number of index pointers of an index node) over these data nodes using the same construction method as for the Huffman tree. All index nodes are of the same size for one bucket.

We evaluate the time and energy efficiency by measuring the average access time and average number of channel switches in locating a data node on broadcast channels. Actually, the tuning time is a metric to measure the energy consumption on broadcast data access. In multichannel environments, the tuning time includes the cost of downloading all relevant packets and the cost of switching channels. The number of downloaded packets for locating a data node is fully dependent on the index structure. If we use the same index tree, this download cost is independent of the method used for broadcast data allocation. Hence we measure the number of channel switches as an index of energy consumption for a particular broadcast scheme. More channel switches will consume more energy. The parameter settings in our simulation are listed in Table 2. We ignore any channel error in our simulation.

Table 2
Parameter settings

| Parameter | Value |
|---|---|
| Number of channels | 1–5 (default, 2) |
| Number of data nodes | 1000–1800 (default, 1200) |
| Index fanout | 4–12 (default, 8) |
| Size of a data node | 10–20 buckets |
| Size of an index node | 1 bucket |
| Data access frequency | Zipf(100, $\theta$), $\theta = 0.1$–0.9 (default, 0.7) |
| $\alpha$ in $f(x)$ | 0.0–1.0 (default, 0.2) |

We first evaluate the $k$-IDA algorithm without replication and compare our approach with the two proposed in [9] and [22], which we refer to as Data_Bin and Access_Graph, respectively. In Data_Bin, data nodes are allocated to broadcast channels first using the bin packing technique. Every local index is then inserted in local data nodes in every broadcast channel. In Access_Graph, the constructed index tree would be viewed as an access graph. Then a weighted function based on the connectivity of the graph is used to evaluate the precedence of a node to be allocated.

We performed the simulation by changing the available number of broadcast channels, which produced the results shown in Figs. 12 and 13. Data_Bin sequentially scans the broadcast channels, and has the longest access time and the smallest number of channel switches. The difference between Data_Bin and the other two approaches becomes more significant as the number of channels increases. Both Access_Graph and $k$-IDA, which might switch to another channel after visiting an index node, use more channel switches than Data_Bin. Access_Graph uses more channel switches and has a slightly higher access time (further explored below) than $k$-IDA. Indeed, if the number of broadcast channels is larger than the largest number of siblings among all levels of the index tree, both Access_Graph and $k$-IDA can obtain the optimal allocation in terms of access time. The optimal allocation is to assign each of the nodes at the same level into each bucket with the same sequence number of each broadcast channel. This is why the performance of these two approaches becomes more similar with an increasing number of broadcast channels.

We further compare Access_Graph and $k$-IDA by changing other parameters. Due to space limitations, we show the major results in Figs. 14–17. We found that $k$-IDA outperforms Access_Graph in all cases, with the superiority being almost independent of parameter settings. This reveals that the allocation order in the single broadcast channel still has a significant impact on that in an environment with multiple broadcast channels. The improvement over Access_Graph will become large when we use index replication, since there is no replication scheme provided in Access_Graph.

We next evaluate the performance when using index replication. Since we use the same allocation method regardless of the method of replication, it is sufficient to evaluate the effect of replication only in a single broadcast channel. Figs. 18 and 19 show comparisons of $k$-IDA approaches with and without replication,

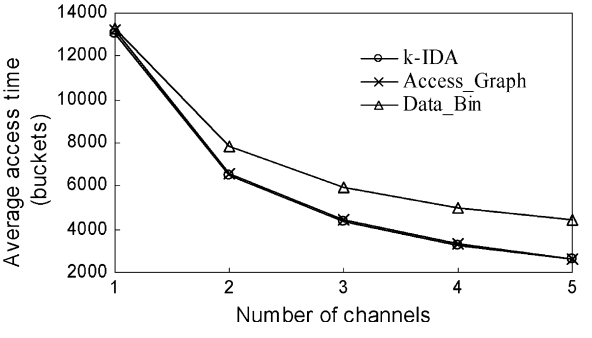


Fig. 12. Access time vs. number of channels.

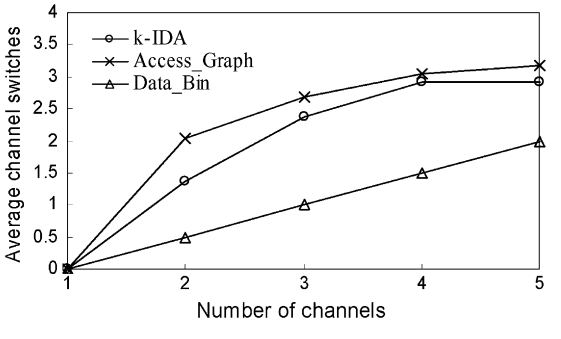


Fig. 13. Number of channel switches vs. number of channels.

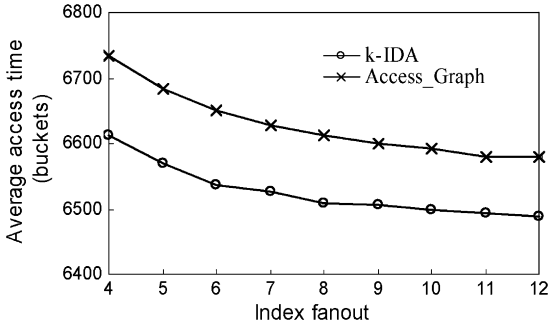


Fig. 14. Access time vs. index fanout.

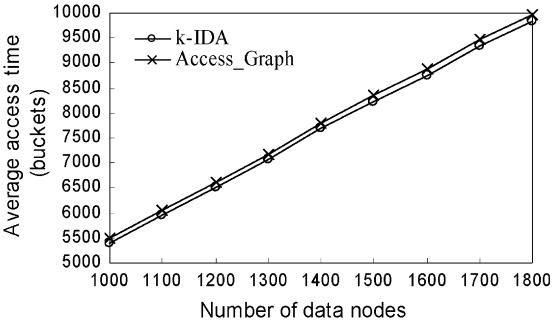


Fig. 15. Access time vs. number of data nodes.

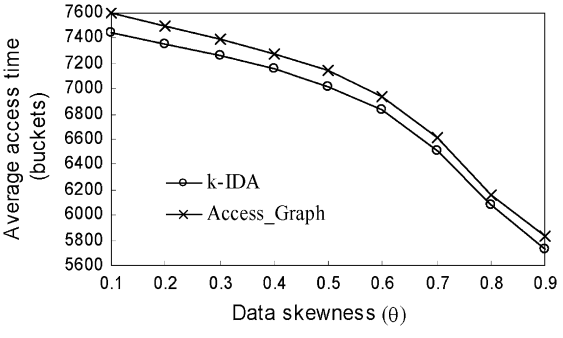


Fig. 16. Access time vs. data skewness.

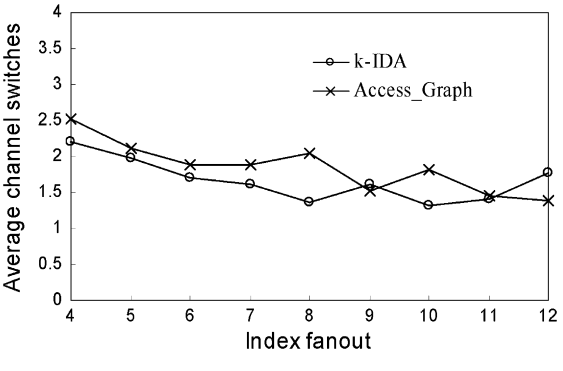


Fig. 17. Number of channel switches vs. index fanout.

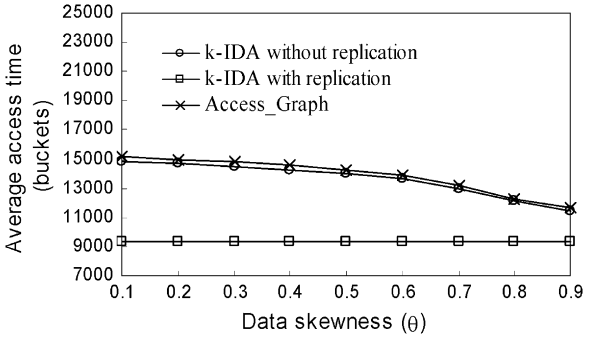


Fig. 18. Replication vs. data skewness.

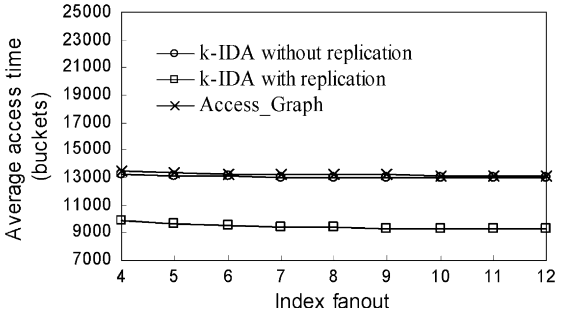


Fig. 19. Replication vs. index fanout.

Table 3
Optimal $\alpha$ value

| (fanout, optimal $\alpha$) | (data nodes, optimal $\alpha$) | ($\theta$, optimal $\alpha$) |
|---|---|---|
| (4,0.2) | (1000,0.2) | (0.1,0.3) |
| (5,0.2) | (1100,0.2) | (0.2,0.3) |
| (6,0.4) | (1200,0) | (0.3,0) |
| (7,0.3) | (1300,0.4) | (0.4,0) |
| (8,0) | (1400,0.4) | (0.5,0) |
| (9,0.3) | (1500,0.2) | (0.6,0) |
| (10,0.3) | (1600,0.4) | (0.7,0) |
| (11,0.1) | (1700,0.5) | (0.8,0) |
| (12,0.1) | (1800,0.3) | (0.9,0) |

which indicate that index replication can reduce the access time by 28%. The worse performance of the non-replication approach is due to a long probe wait. As can be seen in the figures, our proposed approach when using index replication will largely outperform Access_Graph.

The $\alpha$ value in weighted function $f(x)$ can be used to adjust the performance of tree split. In practice, we can use a simple iterative evaluation to find the optimal value of $\alpha$. We summarize the optimal settings in Table 3 for various values of index fanout, number of data nodes, and data skewness. Only the data skewness is clearly related to $\alpha$: $\alpha$ tends to zero with increasing skewness. This means that the value of $w_x$ in $f(x)$ becomes less important in a skewed index tree.

## 7. Conclusion

Data broadcast is a powerful tool for data dissemination in mobile and wireless environments. In this paper, we provide an efficient method for generating a broadcast program that minimizes the average access

time for the user. Our solution adapts well to any number of broadcast channels. We first consider the problem by restricting it to the case where there is no replication, and derive the optimal solution for the single-channel case. To deal with multiple broadcast channels, we provide a heuristic using a set of rules on the allocation. To further reduce the average access time on retrieving index information, an index replication scheme is developed. We propose a tree-split algorithm to split an index tree into several small induced trees. Each of these induced trees can be separately allocated to broadcast channels in the same way as for no replication. The performance evaluation shows that index replication can significantly reduce the average access time.

Our major contributions are a mapping solution to the index and data allocation problem, deriving a multichannel allocation method that can reduce the number of channel switches, and the provision of an index replication scheme. There are several issues that could be assessed in future work. The first is dynamically changing the access frequencies of data items. If such changes are frequent, an efficient on-line algorithm to immediately reflect the current user's access patterns is needed. The second is considering the allocation over multiple broadcast channels with different bandwidths and channel error rates. Finally, the impact on the allocation of multiple receivers in the mobile device would also be an interesting topic to investigate.

## References


[1] S. Acharya, R. Alonso, M. Franklin, S. Zdonik, Broadcast disks: data management for asymmetric communication environments, in: Proceedings of ACM SIGMOD Conference, May 1995, pp. 199–210.

[2] D. Adolphson, T.C. Hu, Optimal linear ordering, SIAM Journal on Applied Mathematics 25 (3) (1973) 403–423.

[3] Y.C. Chehadeh, A.R. Hurson, M. Kavehrad, Object organization on a single broadcast channel in the mobile computing environment, Multimedia Tools and Applications 9 (1) (1999) 69–92.

[4] Y.C. Chehadeh, A.R. Hurson, L.L. Miller, Energy-efficient indexing on a broadcast channel in a mobile database access system, in: Proceedings of IEEE International Conference on Information Technology: Coding and Computing, 2000, pp. 368–374.

[5] Y.D. Chung, M.H. Kim, Effective data placement for wireless broadcast, Distributed and Parallel Database (9) (2001) 133–150.

[6] M.S. Chen, K.L. Wu, P.S. Yu, Optimizing index allocation for sequential data broadcasting in wireless mobile computing, IEEE Transactions on Knowledge and Data Engineering 15 (1) (2003).

[7] A. Datta, D.E. VanderMeer, A. Celik, V. Kumar, Broadcast protocols to support efficient retrieval from databases by mobile users, ACM Transactions on Database Systems 24 (1) (1999) 1–79.

[8] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman Publishing Company, 1976.

[9] A.R. Hurson, Y.C. Chehadeh, J. Hannan, Object organization on parallel broadcast channels in a global information sharing environment, in: Proceedings of IEEE International Conference on Performance, Computing, and Communications, 2000, pp. 347–353.

[10] C.H. Hsu, G. Lee, A.L.P. Chen, A near optimal algorithm for generating broadcast programs on multiple channels, in: ACM Conference on Information and Knowledge Management, November 2001, pp. 303–309.

[11] Q. Hu, W.C. Lee, D.L. Lee, Power conservative multi-attribute queries on data broadcast, in: Proceedings of the International Conference on Data Engineering, February 2000, pp. 157–166.

[12] T.C. Hu, A.C. Tucker, Optimal computer search trees and variable-length alphabetic codes, SIAM Journal on Applied Mathematics 21 (4) (1971) 514–532.

[13] T. Imielinski, S. Viswanathan, B.R. Badrinath, Energy efficient indexing on air, in: Proceedings of ACM SIGMOD Conference, May 1994, pp. 25–36.

[14] T. Imielinski, S. Viswanathan, B.R. Badrinath, Data on air: organization and access, IEEE Transactions on Knowledge and Data Engineering 9 (3) (1997) 353–372.

[15] S. Jiang, N.H. Vaidya, Satellite-based information services: response time in data broadcast systems: mean, variance and tradeoff, Mobile Networks and Applications 7 (1) (2002) 37–47.

[16] S.C. Lo, A.L.P. Chen, Optimal index and data allocation in multiple broadcast channels, in: Proceedings of the 16th IEEE International Conference on Data Engineering, February 2000, pp. 293–302.

[17] S.C. Lo, A.L.P. Chen, An adaptive access method for broadcast data under an error-prone mobile environment, IEEE Transactions on Knowledge and Data Engineering 12 (4) (2000) 609–620.

[18] G. Lee, S.C. Lo, Broadcast data allocation for efficient access of multiple data items in mobile environments, Mobile Networks and Applications 8 (4) (2003) 365–375.

[19] G. Lee, S.C. Lo, A.L.P. Chen, Data allocation on the wireless broadcast channel for efficient query processing, IEEE Transactions on Computers 51 (10) (2002) 1237–1252.

[20] B. Lee, S. Jung, An efficient tree-structure index allocation method over multiple broadcast channels in mobile environments, Database and Expert Systems Applications (2003) 433–443.

[21] W.C. Peng, J.L. Huang, M.S. Chen, Dynamic leveling: adaptive data broadcasting in a mobile computing environment, ACM Mobile Networks and Applications 8 (4) (2003) 355–364.
[22] K. Prabhakara, K.A. Hua, J.H. Oh, Multi-level multi-channel air cache designs for broadcasting in a mobile environment, in: Proceedings of the 16th IEEE International Conference on Data Engineering, February 2000, pp. 167–176.
[23] A. Si, H.V. Leong, Query optimization for broadcast database, Data and Knowledge Engineering 29 (3) (1999) 351–380.
[24] K. Stathatos, N. Roussopoulos, J.S. Baras, Adaptive data broadcast in hybrid networks, in: Proceedings of the 23rd VLDB Conference, August 1997, pp. 326–335.
[25] N. Shivakumar, S. Venkatasubramanian, Energy-efficient indexing for information dissemination in wireless systems, ACM, Journal of Wireless and Nomadic Application (1996).
[26] K.L. Tan, B.C. Ooi, On selective tuning in unreliable wireless channels, Data and Knowledge Engineering 28 (2) (1998) 209–231.
[27] N.H. Vaidya, S. Hameed, Scheduling data broadcast in asymmetric communication environments, Wireless Networks 5 (3) (1999) 171–182.
[28] J.X. Yu, T. Sakata, K.L. Tan, Statistical estimation of access frequencies in data broadcasting environments, Wireless Networks 6 (2) (2000) 89–98.
[29] W.G. Yee, S.B. Navathe, Efficient data access to multi-channel broadcast programs, in: ACM Conference on Information and Knowledge Management, November 2003, pp. 153–160.
[30] W.G. Yee, S.B. Navathe, E. Omiecinski, C. Jermaine, Efficient data allocation over multiple channels at broadcast servers, IEEE Transactions on Computers 51 (10) (2002) 1231–1236.
[31] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, P.J. Weinberger, Quickly generating billion-record synthetic databases, in: Proceedings ACM SIGMOD Conference, Minneapolis, MN, May 1994, pp. 243–252.

**Shou-Chih Lo** received the B.S. degree in computer science from National Chiao Tung University, Taiwan, in 1993, and the Ph.D. degree in computer science from National Tsing Hua University, Taiwan, in 2000. He joined the Computer & Communication Research Center at National Tsing Hua University in 2000 as a Postdoctoral Fellow. He has been with National Dong Hwa University, Taiwan, since 2004 and is now an assistant professor in the Department of Computer Science and Information Engineering. His current research interests are in the area of mobile and wireless networks with emphasis on mobility management, MAC protocols, and broadcast services designs.

**Arbee L.P. Chen** received the B.S. degree in computer science from National Chiao-Tung University, Taiwan, in 1977, and the Ph.D. degree in computer engineering from the University of Southern California in 1984. He was a member of technical staff at Bell Communications Research, New Jersey, from 1986 to 1990, an adjunct associate professor in the Department of Electrical Engineering and Computer Science, Polytechnic University, New York, and a research scientist at Unisys, California, from 1984 to 1986. He joined National Tsing Hua University (NTHU), Taiwan, as a National Science Council (NSC) sponsored visiting specialist in August 1990, and became a professor of the Department of Computer Science, NTHU, in 1991. In August 2001 he took a leave from NTHU and assumed the position of the chairman of the Department of Computer Science and Information Engineering at National Dong Hwa University, Taiwan. Since August 2004, he became Chengchi University Chair Professor at National Chengchi University, Taiwan. His current research interests include data stream management, multimedia databases and data mining. Dr. Chen organized 1995 IEEE Data Engineering Conference and 1999 International Conference on Database Systems for Advanced Applications in Taiwan, and served as a conference chair or program chair/committee member for many international conferences. He has been a visiting scholar at Kyoto University, Beijing Tsinghua University, Stanford University and King's College London. He was invited to deliver a speech at the NSF sponsored Inaugural International Symposium on Music Information Retrieval at Plymouth, USA, 2000, and a speech in the IEEE Shannon Lecture Series at Stanford University, 2005. He has published more than 170 papers in international journals and conference proceedings, and has been a recipient of the NSC Distinguished Research Award since 1996.