Distributed AH-Tree Based Index Technology for Multi-channel Wireless Data Broadcast*

Yongtian Yang[1], Xiaofeng Gao[1,**], Xin Lu[2], Jiaofei Zhong[3], and Guihai Chen[1]

[1] Dept. of Computer Science and Engineering, Shanghai Jiao Tong Univ., China
[2] School of Software, Shanghai Jiao Tong Univ., China
[3] Dept. of Mathematics and Computer Science, Univ. of Central Missouri, USA

Abstract. Alphabetic Huffman Tree (AH-Tree) is an appropriate data structure to index data set with skewed access frequencies, which fits the feature of web-based wireless data broadcast service to a mass number of mobile clients. In this paper we solve a long-time open question to construct an arbitrary k-ary AH-Tree with Hu-Tucker algorithm [1] by dynamic programming, whose time complexity is $O(kn^2)$, where n is cardinality of the data set. We then build a distributed AH-Tree index sequence with a special control-table shrinking technique. Next, we introduce a pyramid index allocation method, which is scalable to any available broadcast channel. We prove the correctness of our algorithm, analyze the time complexity of tree-construction process, and compare our design with B^+-Tree index by numerical experiments. Both mathematical analysis and simulation results prove the efficiency of our design. To the best of our knowledge, we are the first to propose a detailed, fast, and distributed k-ary AH-Tree index with allocation protocol, which has both theoretical and practical significance in this area.

Keywords: Huffman Tree, Distributed Index, Data Broadcast.

1 Introduction

Wireless data broadcast is an efficient data dissemination technology to a mass number of mobile clients with battery-constraint portable wireless devices (i.e., mobile phones, smart phones, and PDAs). Due to the nature of wireless communication, instead of point-to-point query-reply mode, a server broadcasts public information like traffic conditions, live TV streams, weather forecasts, and tourist services, etc., over multiple channels periodically. Each mobile client can access onto the channel, wait for the required data items, and download its required data packet sequence each at a time slot.

Intuitively, the criteria to evaluate the performance of a wireless data broadcast system are the downloading time and energy consumption of mobile devices.

* This work has been supported in part by the National Natural Science Foundation of China (Grant numbers 61073152, 61133006, and 61202024), China 973 project (2012CB316200), Shanghai Educational Development Foundation (Chenguang Grant No.12CG09), and the Natural Science Foundation of Shanghai (Grant No.12ZR1445000).

** Corresponding author.

Correspondingly, *access latency* and *tuning time* are two widely accepted system evaluation standards. The former denotes the time interval from when a client sends a request to the time when it receives the required datum, while the latter denotes the activating time of the mobile device during data retrieval process.

Indexing technology is one of the most effective methods to reduce tuning time. Each mobile device has two modes: active mode and doze mode. Its energy consumption during active mode is far greater than that in doze mode. Indices help to reduce the active time of a mobile device significantly. Clients can follow the direction of indices on broadcasting channels, turn off during the waiting period, and turn on again right before the required data item appears.

There have been a lot of works discussing efficient indexing schemes, which can be classified into three categories: hashing-based indexing [2], tree-based indexing (*e.g.*, B^+ tree [3], Huffman tree [4]), and table-based indexing (*e.g.*, exponential index [5]). Among them, tree-based indices are the most widely implemented structures according to their easy-searching and fast-constructing characteristics. Additionally, researchers prefer to choose a balanced tree as a base for their index design, since it is easier to control the tree height and bound the index size to avoid large increase of access latency.

However, data broadcast system serves mobile clients with hundreds or thousands of data items, whose visiting frequencies vary hugely according to the empirical statistics of human behavior with preference difference. Based on the investigation of website popularity [6], we find that a few data items have very high popularity; a medium number of data items have middle-of-the-road visiting frequencies; while a huge number of data items actually have very low preference. Such phenomenon implies that balanced tree is probably not an appropriate index structure for web service since each searching path has similar length, bringing longer tuning time on average.

To overcome the aforementioned shortcoming, the *Alphabet Huffman Tree* (AH-Tree) is a good choice. In a typical AH-Tree, the higher the frequency of a data item, the shorter the path from root to the corresponding leaf index. Many literatures have studied AH-Tree index during the past two decades. In [1], Hu and Tucker first proposed a binary AH-Tree algorithm with time complexity $O(n \log n)$, where n is the size of data items. Their design is based on a complex queue technology, which cannot be directly extended to construct k-ary AH-Tree with arbitrary k [7]. Later, Shivakumar et al. [8] extended Hu-Tucker Algorithm into k-ary AH-Tree, and first implemented it as indices to broadcast environment. Nevertheless, they did not describe the algorithm clearly. They only mentioned a skeleton of how to build a k-ary AH-Tree, lack of specification of internal node construction with k branches. The time complexity of such design would be high up to $O(n^3)$ if we directly follow their description without creating any particular queue structure, which is impractical for real-world applications. Similarly, [4], [9], and [10] discussed AH-Tree index for data broadcast problem respectively, none of which provided complete tree-construction process with time complexity analysis. They only gave simple explanation according to the description in [8], almost in the same manner. Therefore, how to construct an arbitrary k-ary AH-Tree by Hu-Tucker algorithm remains an open problem.

Additionally, researchers tended to modify the index tree into a distributed index sequence to improve the performance [3,10,11]. This scheme relies on the use of control table. However, we found that the control table also results in overhead in space, which becomes an unneglectable factor as we found half of the control table is redundant. Moreover, the control table contains as much redundancy as the useful information. How to eliminate these redundancy becomes crucial.

In this paper, we propose an efficient AH-Tree construction with the help of dynamic programming. Our algorithm can build arbitrary k-ary AH-Tree index with bounded tree height in $O(kn^2)$ time. We then modify this tree into a distributed index sequence with control table design to further reduce the searching steps. Considering the influence of control table size, we further propose a general and effective scheme to eliminate redundant entries in control tables to reduce the overall index packet length, which save almost 50% of the storage. Finally, we use the dynamic pyramid scheme for index and data allocation. We prove the correctness and complexity of our design theoretically and illustrate the performance of the broadcast system by numerical experiments. Both theoretical proofs and simulation results validate the efficiency of our design. To the best of our knowledge, we are the first to propose the detailed design for k-ary Hu-Tucker algorithm with time complexity analysis. Our design does not rely on any special data structure and queue design, which can be easily implemented in any practical system. The simulation results show that our design gains significant growth regard to skew distributed data. However, it does not over perform B-Tree with regard to uniform distribution [12].

The rest of this paper is organized as follows. Section 2 summarizes the related work in this research area. Section 3 illustrates the problem formulation and the architecture of broadcast system. In Sec. 4 we introduce the dynamic programming to construct k-ary AH-Tree index and provide the correctness proofs, while in Sec. 5 we complete the process to build a distributed index sequence with control tables. Section 6 describes the index allocation method. In Sec. 7 we compare our design with the latest work in [12] and prove the efficiency of our construction. Finally, Section 8 gives conclusion and future works.

2 Related Works

The key research topics in wireless data broadcast are basically focusing on how to deign index structures and how to allocate data onto channels, in order to reduce access latency and tuning time [13].

A series of research works deal with data scheduling problem to decrease access latency, without implementing indexing technology [14]. As a result, the tuning time is as long as the access latency, which still leads to high power consumption for mobile devices.

Traditional disk-based indexing techniques have been modified to meet the requirement of data broadcast systems, which can be classified into three categories: hashing-based [2], tree-based (*e.g.*, Huffman tree [4]), and table-based (*e.g.*, exponential index [5]) schemes. Hashing-based schemes use hash functions to distribute data onto channels. For instance, Yao *et al.* [2] proposed MHash to facilitate skewed access probabilities and reduce access latency. Table-based schemes include exponential index proposed by Xu *et al.* [5], which shares links in different search tables and allows users to start searching at any index node. However, this scheme may not perform well under non-uniform access probabilities. Tree-based schemes are sometimes faster to design and easier to maintain, thus achieving more attentions. One common tree-based index, *i.e.* B^+-tree distributed index (*BTD*) was extended to satisfy different system requirements. Gao *et al.* [11] redesigned BTD and built a complete multi-channel broadcasting system with non-uniform data access probabilities and unequal data sizes.

When it comes to multi-channel data broadcasting, how to allocate index and data will produce heavy impact on the performance of each scheme. A certain allocation method could be helpful to a specific index structure, but at the same time it might reduce the efficiency of another index scheme. Several works [10,15] deal with data allocation for multi-channel data broadcast. One work [16] proposed an index allocation method named TMBT, which creates a virtual BTD for each data channel and multiplexes them on the index channel.

Huffman tree is a skewed index tree that takes into account the data access probability, where more popular data have shorter search paths from the root of the tree [8]. However, most existing works discussed Huffman tree for a certain data type with special constraints and features. Recently, Zhong *et al.* [9] proposed a uniform AH-Tree indexing scheme to satisfy all possible environments.

Based on the observation that the previous schemes can be further improved, we propose a novel AH-Tree based indexing approach under multi-channel environment, where data items have different access probabilities. Our scheme is further refined to minimize both average access latency and tuning time, while the performance compared to other related schemes is also provided. Simulation results confirm the efficiency of our scheme.

3 Problem Formulation and System Architecture

We consider multi-channel wireless data broadcast with a server and numbers of clients. The server first retrieves data from its local database and then calls the Index Generator modular to index the data. Next, the Channel Allocation modular allocates channels to data and index. We take index-data separation mode in this paper to reduce the possible switches among channels. After that, the server periodically broadcasts data and index sequences in a fixed range over wireless channels. Clients can access the data at anytime by tuning onto the channels. In this paper, we focus on Index Generator and Channel Allocator. We propose distributed AH-Tree based index sequence in Index Generator modular and pyramid index allocation scheme in the Channel Allocation modular.

Let $D = \{d_1, d_2, \ldots, d_t\}$ be the data set to broadcast, where t is the number of data item. Associated with D, $P = \{p_1, p_2, \ldots, p_t\}$ is the access frequency set, where p_i is the access frequency of d_i. Also, since each d_i may have different size, we use s_i, measured by KB, to represent the size of d_i and $S = \{s_1, s_2, \ldots, s_t\}$, There are $A = m + n$ channels in the system. We assign m channels to the data and n channels to the index. The channel set is $\mathbf{C} = \{C_1, C_2, \ldots, C_A\}$.

For clarity, we summarize the symbols with their meanings in Table 1. Some of them will be described in the following sections.

Table 1. Symbol Description

Sym	**Description**	**Sym**	**Description**
D	Data set. $D = \{d_1, \cdots, d_t\}$	$\mathbf{C}$	Channels. $\mathbf{C} = \{C_1, \cdots, C_A\}$
L	Level of T	l	Cut level of T
T	An AH-Tree	t	Number of data items
k	Maximum branch no. for T	B_i^j	The j^{th} index at i^{th} level of T
N	Node set of index tree	D_k	The k^{th} datum in T
A	Available channels $A = m + n$	Δ_i	The i^{th} sub-tree at level $l + 1$
m	Number of data channels	PATH(B_i^j)	A path from B_1^1 to B_i^j
MAX(B_i^j)	Maximum key B_i^j domains	n	Number of index channels

4 Basic Index Technology

In this section, we describe the index technique used in the Index Generator modular and an AH-Tree index technique is proposed.

4.1 Tree Construction

First, let us introduce the two-stage construction process of the k-ary AH-Tree [1,8]. In the first stage, we build an optimal k-ary Huffman tree without

Algorithm 1. AH-Tree Construction

```
1: **Input:** $D$, $P$;
2: **Output:** Node set $N$ of AH-Tree $T$.
3: Create $t$ leave nodes for $d_i \in D$ and push them into $N$. Set $I = \{1, \cdots, t\}$.
4: **while** $|I| > 1$ **do**
5:     **if** $\sum_{i=1}^{k} p_{n_i} = \min(\sum_{i=1}^{k} p_{x_i})$, where $n_i$, $x_i \in I$ **and** no leaves among $n_1, \cdots, n_k$
       **and** $n_1, \cdots, n_k$ are the leftmost $k$ nodes **then**
6:         Merge $n_1, \cdots, n_k$ as $n'$ with $r_{n'} = \sum_{i=1}^{k} r_{n_i}$ ($n'$ is parent of $n_1, \cdots, n_k$) ;
7:         Insert $n'$ into $N$, mark $n_1, \cdots, n_k$ as "processed" and remove them from $I$;
8:     **end if**
9:     $n = n - (k - 1)$;
10: **end while**
11: Traverse $T$, mark each node's level from the root and get max level $L$;
12: **for** $l = L \to 2$ **do**
13:     Find the leftmost index node $p$ on the $(l-1)^{th}$ level and the leftmost $k$ nodes
        $n_1, \cdots, n_k$ on the $l^{th}$ level, and then mark them;
14:     Record "$p$" into field *new_parent* of $n_1, \cdots, n_k$ and "$n_1, \cdots, n_k$" into array
        *new_children* of $p$ without altering their original parent/children;
15:     Keep finding new nodes until no unmarked nodes exist in level $l$;
16: **end for**
17: Replace *parent* and *children* of all nodes with *new_parent* and *new_children*.
```

alphabetic order, where dynamic programming is employed to simplify the algorithm. In the second stage, we adjust the tree in a bottom-up approach to generate a new tree, which preserves the alphabetic order while keeping the same cost. The construction process is shown in Alg. 1.

Stage 1 (Line 4 to 10) contains several iterations. During each iteration, we select k nodes to merge into one new node and then insert it into the original data sequence to form a new sequence. Recall that k is the number of branches of the tree. The selected k nodes should satisfy three conditions: (1). There are no leaf nodes among them; (2). The sum of their frequencies is the minimum among all k candidate groups; and (3). They should be the leftmost nodes.

We create a new index node as the parent of these k nodes with the frequency as the sum of the k nodes. Then we insert the new node into the data sequence, mark the k nodes as processed leaf nodes, and delete them from the sequence. After that, we start a new iteration and continue the process until there is only one node left in the sequence, which is the root of the tree. At the end of Stage 1, we produce a tree T' without alphabetic order.

Stage 2 (Line 11 to 17) adjusts the tree in a bottom-up, left-right approach such that every k consecutive nodes on the same level have the same parent. Finally, an AH-Tree T is constructed. The correctness proof is provided in [17].

Example 1. Throughout the paper, we use a data set in Table 2. The frequency indicates how many times a datum has been accessed from historical record.

Table 2. Example Data Set

Key	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Fre	16	8	30	4	1	12	27	36	41	2	9	15	19	7	23	1

Based on this data set, we apply Alg. 1 to construct the tree. After two stages, we generate T' and T as shown in Fig. 1, respectively.

4.2 Subroutine Dynamic Programming

It is time-consuming to choose k nodes from the data sequence in stage 1. In this subsection, we design a dynamic programming to solve this problem in $O(ki)$, where i is the number of nodes in the current iteration and k is the number of branches in the tree. We first describe the basic idea and derive the recursive relation, and then verify the correctness in the following part.

Consider the problem of selecting j nodes from sequence $[n_1, n_2, \ldots, n_i]$. There are only two cases to be considered.

Case 1: There is at least one unselected leaf node in $[j+1, \ldots, i]$. Let $f(i, j)$ be the minimal weight sum of the j selected nodes in this case.
Case 2: There is no unselected leaf node in $[j+1, \ldots, i]$. Let $g(i, j)$ be the minimal weight sum of the j selected nodes in this case.

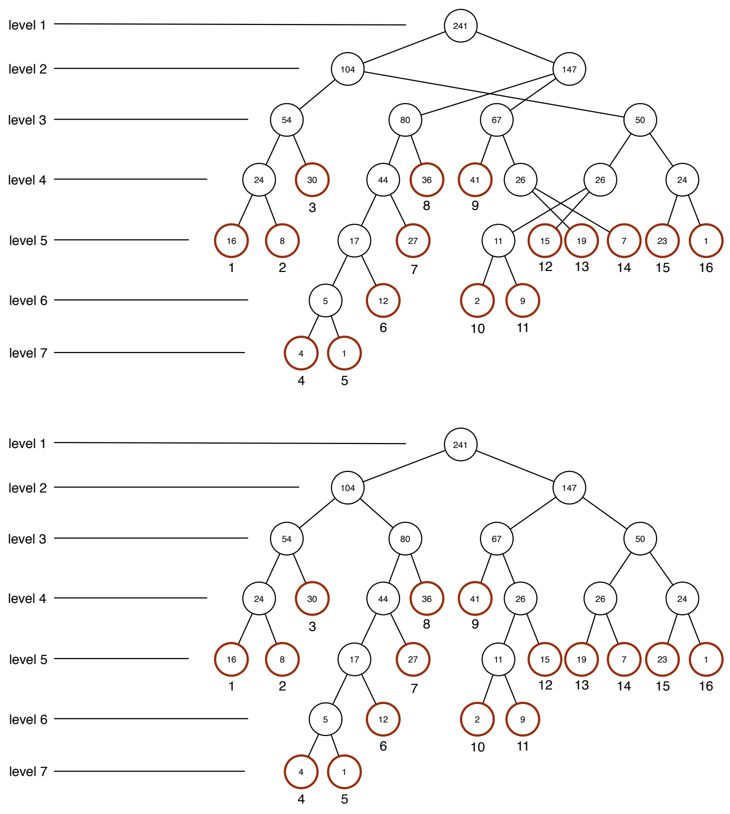

Fig. 1. T' (Tree construction after Stage 1) and T' (Tree construction after Stage 2)

We now derive the recursive relation. In Case 1, the i^{th} node is unselected, otherwise no leaf node exists in $[j+1,\ldots,i-1]$. There are also two subcases:

1. If the i^{th} node is an index node, then there is at least one unselected leaf node in $[j+1,\ldots,i-1]$. We need to solve the subproblem $f(i-1,j)$;
2. If the i^{th} node is a leaf node, then there may be no leaf node in $[j+1,\ldots,i-1]$. We have to consider both $f(i-1,j)$ and $g(i-1,j)$ and choose the minimum.

Then the recursive relation for f is shown below:

$$f(i,j) = \begin{cases} f(i-1,j), & \text{if } i^{th} \text{ node is an index node} \\ \min(f(i-1,j), g(i-1,j)), & \text{if } i^{th} \text{ node is an unmarked leaf} \end{cases}$$

In Case 2, there are also two subcases:

1. If the i^{th} node was an unselected leaf node, it must be selected now, otherwise there will be an unselected leaf node in $[j+1,\ldots,i]$, which violates our assumption of Case 2. So we need to consider the subproblem of $g(i-1,j-1)$.

2. If the i^{th} node is an index node, since whether select it or not will not violate the condition, we have to consider $\min(g(i-1,j-1), g(i-1,j))$.

Hence, the recursive relation for g is as following:

$$g(i,j) = \begin{cases} g(i-1,j-1)+r_i, & \text{if } i^{th} \text{ node is an unmarked leaf} \\ \min(g(i-1,j-1)+r_i, g(i-1,j)), & \text{if } i^{th} \text{ node is an index node} \end{cases}$$

After computing $f(n,k)$ and $g(n,k)$, the final result is $\min(f(n,k), g(n,k))$. We record the choices when generating values in f and g to locate the k nodes.

4.3 Correctness of the Dynamic Programming

We now verify the correctness of our design. The first step is to prove that the problem has *optimal substructure*. In the following, we show that both f and g have optimal substructures. Let's start with g, which is independent to f.

Lemma 1. *(Optimal Substructure of g)* Let $S_i = [d_1, d_2, \ldots, d_i]$ be the data sequence, and $Z_j = [d_{i_1}, \ldots, d_{i_j}]$ be any optimal solution satisfying the condition of Case 2 (we also call it an optimal solution of S_i for convenience). Then:

1. If d_i is an unselected leaf node, then Z_{j-1} is an optimal solution of S_{i-1};
2. If d_i is an index node, then Z_{j-1} is an optimal solution of S_{i-1} if $d_i = d_{i_j}$; otherwise Z_j is an optimal solution of S_{i-1}.

Proof. *Condition 1.* If d_i is a leaf node, then it is surely in Z_j. Suppose $Z'_{j-1} = [d'_{i_1}, \ldots, d'_{i_{j-1}}]$ is an optimal solution of S_{i-1} with cost less than Z_{j-1}, then we replace Z_{j-1} with Z'_{j-1} in Z_j to get Z'_j. Obviously Z'_j is a solution of S_n whose cost is less than Z_j. It contradicts to the condition that Z_j is an optimal solution.

Condition 2. If d_i is an index node, then there are two cases:
(a) if $d_i = d_{i_j}$, that is, we select the i^{th} node. We can use the same method in Condition 1 to prove that Z_{j-1} is an optimal solution of S_{i-1}.
(b) if $d_i \neq d_{i_j}$, that is, the i^{th} node is unselected. Suppose $Z'_j = [d'_{i_1}, d'_{i_2}, \ldots, d'_{i_j}]$, with a cost less than Z_j, is an optimal solution of S_{i-1}. Since the i^{th} node is an index node, it follows that Z'_j is also a solution of S_i. Then we find a solution of S_i with a cost less than Z_j, which contradicts the condition that Z_j is an optimal solution of S_i. Hence, Z_j is an optimal solution of S_{i-1}.

From above, we conclude that g has *optimal substructure*. □

Lemma 2 shows that solving f also contains the *optimal substructure*.

Lemma 2. *(Optimal substructure of f)* Let $S'_i = [d_1, d_2, \ldots, d_i]$ be the sequence, and $Z_j = [d_{i_1}, d_{i_2}, \ldots, d_{i_j}]$ be any optimal solution that satisfies Case 1 (we also refer it an optimal solution of S'_i), then Z_j is an optimal solution of S'_{i-1}.

Proof. The proof is divided into two parts:

Part 1. If d_i is an index node, there is at least one leaf node in $[j+1, \ldots, i-1]$. Thus Z_j is a solution of S'_{i-1}. Let $Z'_j = [d'_{i_1}, d'_{i_2}, \ldots, d'_{i_j}]$ be an optimal solution of S'_{i-1}, which costs less than Z_j. In the case of Z'_j, there must be at least one unselected leaf node in $[j+1,\ldots,i]$ since d_i is an index node. So Z'_j is a solution of S'_i. That is, there is another solution which costs less than Z_j. Contradiction.

Part 2. d_i is an unselected leaf node. Since d_i is unselected, we prove that Z_j is an optimal solution of both S'_{i-1} and S_{i-1}. Suppose Z'_j is an optimal solution of S_{i-1} and S'_{i-1} with a cost less than Z_j. Since d_i is a leaf node, Z'_j is also a solution of S'_i. Thus we find a solution of S'_i that costs less than Z_j, which contradicts the condition. Thus, Z_j is an optimal solution of S_{j-1} and S'_{j-1}. □

Lemma 1 and Lemma 2 imply that both f and g have optimal substructures. The next two lemmas will complete the final conclusion, in which Lemma 3 can be directly derived from Lemma 1 and Lemma 2.

Lemma 3. Solving $\min(f(n,k), g(n,k))$ problem has optimal substructure.

Lemma 4. Solving $\min(f(n,k), g(n,k))$ problem has overlapping subproblem.

Proof. Solving $\min(f(n,k), g(n,k))$ involves solving $f(n-1,k)$ and $f(n-1,k-1)$, which are two overlapping problems. Thus it has overlapping subproblem. □

Theorem 1. If $Z_j = [d_{n_1},\ldots,d_{n_k}]$ is the output of our dynamic programming, then it satisfies three conditions:

1. There are no leaf nodes among these k nodes.
2. The frequency sum of Z_j is the minimum among all possible selection.
3. The k nodes should be the leftmost ones among all the candidates.

Proof. It follows directly from Lemma 3 and Lemma 4. □

5 Distributed AH-Tree Construction

We have constructed an AH-Tree using dynamic programming in Sec. 4. To avoid searching from the tree root every time, we employ the distributed index technique. This technique is first introduced in [3] and applied to B$^+$ tree index. Then it is applied to AH-Tree index in [9]. The tree is split into *replicated* part and *non-replicated* part. The basic idea is to add the dominating range of all ancestors into the replicated-part nodes. Thus, from any replicated-part node, we know which subtree we are looking for, and can directly tune to it.

5.1 Control Table

First we introduce some notations. The index nodes are divided into two parts. Suppose l is the *cut level.* The nodes in replicated part are called *control index*, while the remaining nodes are named *search index.* B_i^j denotes the j^{th} index node of level i, and D_k represents the k^{th} data node in the tree. Note that if the j^{th} node of level i is a data node, then B_i^j does not exist.

Let Δ_i denote the i^{th} subtree below the cut level l, rooted at B_{l+1}^i. To generate the distributed index sequence, we use PATH(B_i^j) to represent a path from the root to B_i^j excluding B_i^j. For instance, PATH(B_4^4) in Fig. 1 is $[B_1^1, B_2^2, B_3^3]$. In order to broadcast data, we linearize the tree by depth-first search. We use $B_i^{j[x]}$ to denote the x^{th} appearance of B_i^j during the process of depth-first search.

For each control index $B_i^{j[x]}$, suppose that $\text{PATH}(B_i^{j[x]}) = \{B_1^{1[x_1]}, B_2^{j_2[x_2]}, \ldots, B_{i-1}^{j_{i-1}[x_{i-1}]}\}$, then the control table of $B_i^{j[x]}$ is shown in Table 3.

Table 3. Format of the Control Table

1	$\text{MAX}(\Delta_{g-1})$	$B_1^{1[1]}$
2	$\text{MAX}(B_2^{j_2})$	$B_1^{1[x_1+1]}$
...	...	...
r	$\text{MAX}(B_r^{j_r})$	$B_{r-1}^{j_{r-1}[x_r+1]}$
...	...	...
i	$\text{MAX}(B_i^{j_i})$	$B_{i-1}^{j_{i-1}[x_i+1]}$

Each entry of the control table contains two elements: the key value and the control index it hops to. The $\text{MAX}(\Delta_{g-1})$ in the first entry gives a lower bound of the dominating range of $B_i^{j[x]}$. If the key k we are looking for is less than $\text{MAX}(\Delta_{g-1})$, it means that k has been broadcast, so we have to wait for another round. In this case, we jump to $B_1^{1[1]}$. The key value in the r^{th} entry gives an upper bound of the dominating range of $B_r^{j_r}$. The second element gives a hint of which subtree to hop to in the next step. When the client wants to retrieve a datum with key greater than this element, it should directly hop to the control index in this entry. Note that a control index cannot appear more than k times in one round, but the value of $(x_r + 1)$ in the computing process may be larger than k. In this case the second element of the entry is set to *no*.

Example 2. The control table of the tree with cut level $l = 4$ in Fig. 1 is showed in Table 4.

Table 4. The Control Table of the Example Data Set

$B_1^{1[1]}$	no no	$B_2^{1[1]}$	no no 8 $B_1^{1[2]}$	$B_3^{1[1]}$	no no 8 $B_1^{1[2]}$ 3 $B_2^{1[2]}$	$B_4^{1[1]}$	no no 8 $B_1^{1[2]}$ 3 $B_2^{1[2]}$ 2 $B_3^{1[2]}$	$B_4^{1[2]}$	1 $B_1^{1[1]}$ 8 $B_1^{1[2]}$ 3 $B_2^{1[2]}$ 2 $B_3^{1[2]}$	$B_3^{1[2]}$	2 $B_1^{1[1]}$ 8 $B_1^{1[2]}$ 3 $B_2^{1[2]}$	$B_2^{1[2]}$	3 $B_1^{1[1]}$ 8 $B_1^{1[2]}$	$B_3^{2[1]}$	3 $B_1^{1[1]}$ 8 $B_1^{1[2]}$ 8 $B_1^{1[2]}$
$B_4^{3[1]}$	3 $B_1^{1[1]}$ 8 $B_1^{1[2]}$ 8 $B_1^{1[2]}$ 7 $B_3^{2[2]}$	$B_4^{3[2]}$	6 $B_1^{1[1]}$ 8 $B_1^{1[2]}$ 8 $B_1^{1[2]}$ 7 $B_3^{2[2]}$	$B_3^{2[2]}$	7 $B_1^{1[1]}$ 8 $B_1^{1[2]}$ 8 $B_1^{1[2]}$	$B_1^{1[2]}$	8 $B_1^{1[1]}$	$B_2^{2[1]}$	8 $B_1^{1[1]}$ 16 no	$B_3^{3[1]}$	8 $B_1^{1[1]}$ 16 no 12 $B_2^{2[2]}$	$B_3^{3[2]}$	9 $B_1^{1[1]}$ 16 $B_1^{1[2]}$ 12 $B_2^{2[2]}$	$B_4^{6[1]}$	9 $B_1^{1[1]}$ 16 no 12 $B_2^{2[2]}$ 12 $B_2^{2[2]}$
$B_4^{6[2]}$	11 $B_1^{1[1]}$ 16 no 12 $B_2^{2[2]}$ 12 $B_2^{2[2]}$	$B_2^{2[2]}$	12 $B_1^{1[1]}$ 16 no	$B_3^{4[1]}$	12 $B_1^{1[1]}$ 16 no 16 no	$B_4^{7[1]}$	12 $B_1^{1[1]}$ 16 no 16 no 14 $B_3^{4[2]}$	$B_4^{7[2]}$	16 $B_1^{1[1]}$ 16 no 16 no 14 $B_3^{4[2]}$	$B_3^{4[2]}$	16 $B_1^{1[1]}$ 16 no 16 no	$B_4^{8[1]}$	14 $B_1^{1[1]}$ 16 no 16 no 16 no	$B_4^{8[1]}$	15 $B_1^{1[1]}$ 16 no 16 no 16 no

Look at the control table of $B_4^{6[2]}$ in the lower left corner of the table. Since it is the second appearance of B_4^6, all the keys less than or equal to 11 have been broadcasted. Thus if the searching key is less than 11, we jump to $B_1^{1[1]}$. The second entry means that, if the search key is larger than 16, then there is no where to go since the largest key is 16. Suppose that we want to search the key value of 14,then the next hop is $B_2^{2[2]}$, which is exactly the nearest ancestor dominating 14.

From the control table in Table 4, we find that some control tables, marked as gray, have redundancy. For example, the control table of $B_3^{4[1]}$, there are two identical entry *16 no*. When the tree becomes large, the redundancy will be as big as half of the whole control index, which will waste lots of resource. As the next section shows, almost *half* of the control tables contains redundancy, so we will save half of the space if we can eliminate those redundancy.

5.2 Redundancy Elimination for Control Table

In this section, we propose a scheme to eliminate redundancy. We claim that our scheme is not only suitable for AH-Tree, but also for any other tree-based indices employing distributed technique. For the balance tree, the scheme can save 50% of the space for storing the control tables.

Having redundancy in the control table means that for two entries A and B in the control table, the key value of them are identical. Formally speaking, suppose the i^{th} and j^{th} entries are redundant in $B_i^{j[x]}$'s control table. It means that the upper bound of the dominating range of i^{th} and j^{th} ancestor of $B_i^{j[x]}$ are the same. Recall that during the process of searching the control table, after we find the key value of some entry less than the searching key, we jump immediately. The following entries with the same key value will never be used. So we can simply discard these following entries. The problem is how to locate the redundant control tables. A simple case is the rightmost path of the tree.

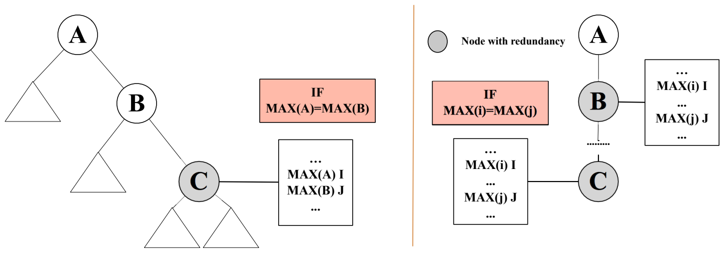

Fig. 2. Example of Redundancy in Control Table

The upper bound of the dominating range of the control index along this path are all the same. Consider the left part of Fig. 2. Since B is the rightmost child of A, we have $\mathrm{MAX}(A) = \mathrm{MAX}(B)$. Thus the control table of C contains redundancy. Another case maybe less obvious. The control table of a control index also contains redundancy if one of its ancestors' control table contains redundancy, as shown in the right part of Fig. 2. However, all those causes can be combined into Theorem 2. Before that, we first give two properties of the control table without proof.

Property 1: If index node A and B are ancestors of C, and A is ancestor of B, then $\mathrm{MAX}(A) \geq \mathrm{MAX}(B) \geq \mathrm{MAX}(C)$, and in the control table of C, the entry of A is in front of that of B.

Property 2: If $\mathrm{MAX}(A) = \mathrm{MAX}(B)$ and B is a child of A, then B must be the rightmost child of A and vice versa.

Theorem 2. Let $\mathrm{PATH}'(B_i^{j[x]}) = [B_2^{j_2}, B_3^{j_3}, \ldots, B_i^{j_i}]$, which is $\mathrm{PATH}(B_i^{j[x]})$ plus $B_i^{j[x]}$ then excludes the root of the tree. The control table of $B_i^{j[x]}$ contains redundancy if and only if there exits consecutive index nodes $B_{a-1}^{j_{a-1}}$ and $B_a^{j_a}$ in $\mathrm{PATH}'(B_i^{j[x]})$, such that $B_a^{j_a}$ is the rightmost child of $B_{a-1}^{j_{a-1}}$.

Proof. $\Rightarrow$. Suppose that $B_{i_1}^{j_{i_1}}$ and $B_{i_2}^{j_{i_2}}$ belong to $\mathrm{PATH}'(B_i^{j[x]})$ and $\mathrm{MAX}(B_{i_1}^{j_{i_1}})$ $= \mathrm{MAX}(B_{i_2}^{j_{i_2}})$ in the control table of $B_i^{j[x]}$. Without loss of generality, we assume $i_1 < i_2$, so $B_{i_1}^{j_{i_1}}$ is an ancestor of $B_{i_2}^{j_{i_2}}$.
(a) If $i_1 + 1 = i_2$, then $B_{i_2}^{j_{i_2}}$ is a child of $B_{i_1}^{j_{i_1}}$. Since $\mathrm{MAX}(B_{i_1}^{j_{i_1}}) = \mathrm{MAX}(B_{i_2}^{j_{i_2}})$, by Property 2 $B_{i_2}^{j_{i_2}}$ is the rightmost child. Then $B_a^{j_a}$ is $B_{i_2}^{j_{i_2}}$, $B_{a-1}^{j_{a-1}}$ is $B_{i_1}^{j_{i_1}}$.
(b) If $i_1+1 < i_2$, then $i_1 < i_1+1 < i_2$, which implies $\mathrm{MAX}(B_{i_1}^{j_{i_1}}) \geq \mathrm{MAX}(B_{i_1+1}^{j_{i_1+1}})$ $\geq \mathrm{MAX}(B_{i_2}^{j_{i_2}})$ by Property 1. However, $\mathrm{MAX}(B_{i_1}^{j_{i_1}}) = \mathrm{MAX}(B_{i_2}^{j_{i_2}})$, so $\mathrm{MAX}(B_{i_1}^{j_{i_1}}) = \mathrm{MAX}(B_{i_1+1}^{j_{i_1+1}})$. We conclude that $B_{i_1+1}^{j_{i_1+1}}$ must be the rightmost child of $B_{i_1}^{j_{i_1}}$ by Property 2. Let $B_a^{j_a}$ be $B_{i_1+1}^{j_{i_1+1}}$ and $B_{a-1}^{j_{a-1}}$ be $B_{i_1}^{j_{i_1}}$.
$\Leftarrow$. Since $B_a^{j_a}$ is the rightmost child of $B_{a-1}^{j_{a-1}}$, we have $\mathrm{MAX}(B_a^{j_a})$=$\mathrm{MAX}(B_{a-1}^{j_{a-1}})$, then there are redundancy in the control table. □

By traveling the tree, we can find all the redundant control tables with the help of the above theorem. Then we eliminate all the redundant entries but the one with least level. Finally we get a "clear" control tables.

6 Index and Data Allocation

After adding control tables to the index nodes, the final step is to allocate them onto channels. In this section, we present the scheme of index and data allocation. There are many index allocation method, such as [4,16,18], but they didn't employ distributed technique. In this paper, we use the simple pyramid scheduling scheme to allocate data and indices onto channels.

We define $W(B_i^{j[x]})$ as the weight of the index node $B_i^{j[x]}$, which denotes the sum of the probability of all its data descendants, while $W(D_i)$ is the access probability of data D_i. Since we apply the same method to allocate index and data, we only describe data allocation in the paper, which is identical to index allocation method. The algorithm of data allocation is shown in Alg. 2.

There is a dynamic threshold for each index channel. In Alg. 2, recall that n is the number of index channels. Suppose that the sum of all the weight is SUM, then the threshold of the first channel is SUM/n. Thus, we assign indices to the first index channel one by one until the total weight of all these assigned

Algorithm 2. Data Allocation on Multiple Channels

```
1: Input: $W$, the weight set of the data; $n$, the number of the data channels;
2: Output:$C = \{C_1, C_2, \ldots, C_n\}$, the generated data channel set.
3:
4: Sort the data set by frequency in ascending order, results in $I = \{D_1, D_2, \ldots, D_t\}$;
5: $SUM = \sum_{i=1}^{t} W(D_i)$;
6: Set $ave = 0$; $p = SUM$; $thre = \frac{SUM}{n}$; $C_i = \emptyset$; $j = 1$;
7: for $i = 1$ to $t$ do
8:     if $ave \leq thre$ then
9:         $ave = ave + W(D_i)$;
10:        $C_j = C_j \cup \{D_i\}$;
11:    else
12:        $p = p - ave$; $ave = 0$; $thre = \frac{p}{n-j}$; $j$++; $i$−−;
13:    end if
14: end for
```

indices exceeds the threshold of the first channel. Next, we begin to assign the remaining indices to the next channel. Instead of having the same threshold for all the channels, we set the threshold of next channel as

$$\frac{\text{total weight of the remaining indices}}{\text{the number of remaining index channels}}$$

for fairness. Then we repeat the above process until no index is left.

Example 3. We use the data set in sec.4 and apply the allocation scheme to it. Suppose there are 4 data channels.

(1) In the first iteration, the threshold $thre = (16 + 8 + 30 + 4 + 1 + 12 \ldots + 23 + 1)/4 = 62$. We assign the data one by one to the first channel until the sum of frequencies of assigned data exceeds 62.This ends the assignment of the first channel and the first 6 data are assigned to it.
(2) We calculate the threshold of the second channel, $thre = (27 + 36 + 41 + \ldots + 23 + 1)/3 = 60$, which the assigned data and channel are not considered any more. We assign the data one by one to the second channel until the sum of frequencies of assigned data exceeds 60.

We apply this process until all the data have been assigned. The final allocation is: $1 : \{1, 2, \ldots, 5, 6\}, 2 : \{7, 8, \}, 3 : \{9, 10, 11, 12\}, 4 : \{13, 14, 15, 16\}$.

7 Simulation

In this section, we evaluate the performance of AH-Tree based system in different conditions. We conduct the performance evaluation on k, l, and m.

Firstly, we model different system conditions by setting all parameters in Table 5. We let the frequency of each data follow Zipf distribution. The number data items are set as 10,000 and we generate 20,000 clients requests in our experiments. Performance of our system is mainly measured by two metrics: *Average Access Latency* (AAT) and *Average Tuning Time* (ATT), both of which are counted in *logical time units*. As proposed in [12], each logical time unit represents the time required to broadcast 1KB data. For index bucket with 1 head segment, k children pointers and 1 default pointer, the size equals to $(k+2)*0.1$KB, that is to say, it requires $(k+2)*0.1$ time units to visit. Our experiments are conducted on a computer with Intel(R) Core(TM) i7-3610QM (2.30GHz) CPU and 4.00 GB memory under Windows 7 version 6.1. The simulator is implemented in Java 1.7.005.

Table 5. Parameters used in our experiments

Parameter	Default value	Range	Meaning
t	10,000		Number of data items
r	20,000		Number of requests
A	10		Number of available channels
k	3	2 to 20	k-ary AH-Tree
l	3	1 to (L-1)	cut level, L is the height of tree
m	3	1 to 9	Number of index channels

Secondly, we verify the effectiveness of our index-allocation algorithm, namely, the pyramid scheduling scheme. We use two different allocation algorithms: the *McNaughton's Wrap-Around* algorithm which simply allocates indices evenly onto index channels, and our pyramid scheduling algorithm, then compare their respective performances. From Fig. 3 to Fig. 8, we can conclude that (1). Pyramid index allocation achieves better performance than simple wrap-around algorithm, and (2). the AAL and ATT of our system have closer relationship with parameter settings of k, l, m than the adopted allocation scheme, because the shapes of the lines look similar with different allocation schemes.

Thirdly, we compare the performance of AH-Tree based system with B$^+$-Tree based one. Both of the two systems adopt pyramid index allocation algorithm. Generally, Fig. 9 to Fig. 14 reveal that AH-Tree performs better than B+-Tree on both access latency and tuning time for data following Zipf distribution. In Fig. 10, the ATT declines with the increase of l since more control indices contribute to faster hopping to targeted data. In the meanwhile, the AAL in Fig. 9 firstly drops then rises again since too much control indices make the index sequences in index channels too long, thus lengthen the access latency. Fig. 11 clearly shows that large k has negative effects on AH-Tree's performance and the AAL of AH-Tree based system fluctuates severely with the variation of k. Therefore, choosing a proper k for AH-Tree based system is crucial for its user experience. For Zipf distribution, the access latency is mainly determined by the waiting time for small number of frequently visited indices, so we find that larger m leads to the decrease of AAL in Fig. 7. On the other hand, the change of m does not change the structure of index sequence, hence it has no effects on ATT as shown in Fig. 8.

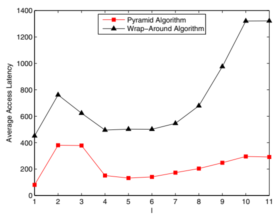

Fig. 3. Change of AAL under Zipf distribution w.r.t. l ($k = 3$, $m = 3$)

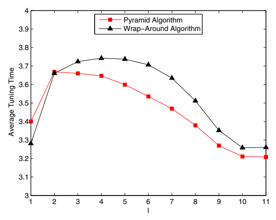

Fig. 4. Change of ATT under Zipf distribution w.r.t. l ($k = 3$, $m = 3$)

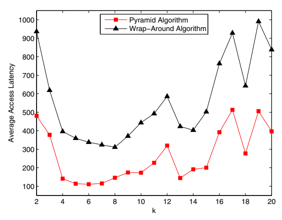

Fig. 5. Change of AAL under Zipf distribution w.r.t. k ($l = 3$, $m = 3$)

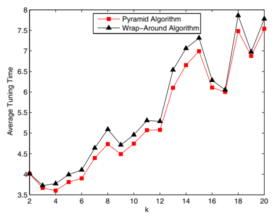

Fig. 6. Change of ATT under Zipf distribution w.r.t. k ($l = 3$,$m = 3$)

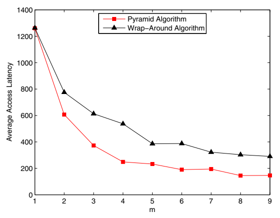

Fig. 7. Change of AAL under Zipf distribution w.r.t.m ($k = 3$, $l = 3$)

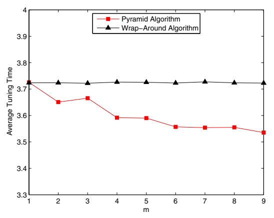

Fig. 8. Change of ATT under Zipf distribution w.r.t.m ($k = 3$, $l = 3$)

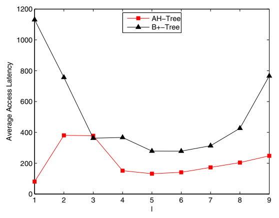

Fig. 9. Change of AAL under Zipf distribution w.r.t. l ($k = 3$,$m = 3$)

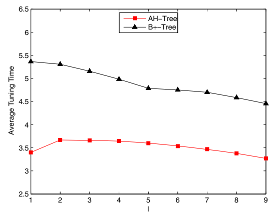

Fig. 10. Change of ATT under Zipf distribution w.r.t. l ($k = 3$,$m = 3$)

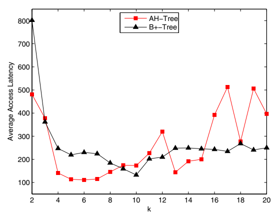

Fig. 11. Change of AAL under Zipf distribution w.r.t. k ($l = 3$,$m = 3$)

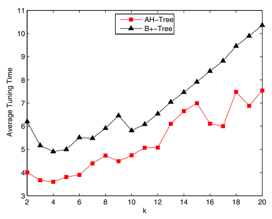

Fig. 12. Change of ATT under Zipf distribution w.r.t. k ($l = 3$,$m = 3$)

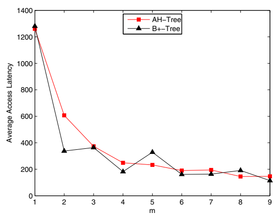

Fig. 13. Change of AAL under Zipf distribution w.r.t.m ($k = 3$, $l = 3$)

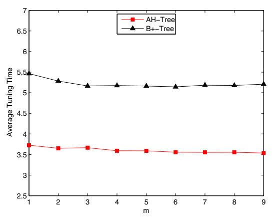

Fig. 14. Change of ATT under Zipf distribution w.r.t.m ($k = 3$, $l = 3$)

8 Conclusion

In this paper, we propose an efficient AH-Tree construction with the help of dynamic programming. Our algorithm can build arbitrary k-ary AH-Tree index with bounded tree height in $O(kn^2)$ time, where n is the number of data items to be broadcast. We then modify this tree into a distributed index sequence with a general and effective control-table shrinking technique to further reduce the index length and the client searching time. We prove the correctness and complexity of our design theoretically and illustrate the performance of the broadcast system by numerical experiments. Both theoretical proofs and simulation results validate the efficiency of our design. To the best of our knowledge, we are the first to propose the detailed design for k-ary Hu-Tucker AH-Tree construction with time complexity analysis, which solves a long-time open question since the beginning of twenty-first's century.

References

1. Hu, T., Tucker, A.: Optimal computer search trees and variable-length alphabetical codes. SIAM Journal on Applied Mathematics 21(4), 514–532 (1971)
2. Yao, Y., Tang, X., Lim, E., Sun, A.: An energy-efficient and access latency optimized indexing scheme for wireless data broadcast. IEEE Trans. on Knowledge & Data Engineering 18(8), 1111–1124 (2006)
3. Imielinski, T., Viswanathan, S., Badrinath, B.: Data on air: Organization and access. IEEE Trans. on Knowledge & Data Engineering 9(3), 353–372 (1997)
4. Jung, S., Lee, B., Pramanik, S.: A tree-structured index allocation method with replication over multiple broadcast channels in wireless environments. IEEE Trans. on Knowledge & Data Engineering 17(3), 311–325 (2005)
5. Xu, J., Lee, W., Tang, X., Gao, Q., Li, S.: An error-resilient and tunable distributed indexing scheme for wireless data broadcast. IEEE Trans. on Knowledge & Data Engineering 18(3), 392–404 (2006)
6. Adamic, L., Huberman, B.: Zipf's law and the internet. Glottometrics 3(1), 143–150 (2002)
7. Zhong, J., Wu, W., Gao, X., Shi, Y., Yue, X.: Efficient redesign and comparison of various indexing schemes for wireless data broadcasting. Submitted to Knowledge and Information Systems (2012)
8. Shivakumar, N., Venkatasubramanian, S.: Efficient indexing for broadcast based wireless systems. ACM Journal of Mobile Networks and Applications 1(4), 433–446 (1996)
9. Zhong, J., Wu, W., Shi, Y., Gao, X.: Energy-Efficient Tree-Based Indexing Schemes for Information Retrieval in Wireless Data Broadcast. In: Yu, J.X., Kim, M.H., Unland, R. (eds.) DASFAA 2011, Part II. LNCS, vol. 6588, pp. 335–351. Springer, Heidelberg (2011)
10. Zhong, J., Gao, Z., Wu, W., Chen, W., Wang, L.: Multi-channel energy-efficient hash scheme broadcasting. SEDE, Los Angeles (2012)
11. Gao, X., Shi, Y., Zhong, J., Zhang, X., Wu, W.: Sambox: A smart asynchronous multi-channel blackbox for b+-tree based data broadcast system under wireless communication environment. SEDE, Los Angeles (2012)

12. Lu, X., Gao, X., Yang, Y.: Setmes:a scalable and efficient tree-based mechanical scheme for multi-channel wireless data broadcast. Submitted to ACM ICUIMC (2013)
13. Sun, W., Liu, P., Wu, J., Qin, Y., Zheng, B.: An automaton-based index scheme for on-demand XML data broadcast. In: Lee, S.-g., Peng, Z., Zhou, X., Moon, Y.-S., Unland, R., Yoo, J. (eds.) DASFAA 2012, Part II. LNCS, vol. 7239, pp. 96–110. Springer, Heidelberg (2012)
14. Vlajic, N., Charalambous, C., Makrakis, D.: Wireless data broadcast in systems of hierarchical cellular organization. In: IEEE International Conference on Communications, ICC 2003, vol. 3, pp. 1863–1869 (2003)
15. Yee, W., Navathe, S.: Efficient data access to multi-channel broadcast programs. In: Proceedings of the 12th International Conference on Information and Knowledge Management, pp. 153–160 (2003)
16. Wang, S., Chen, H.: Tmbt: An efficient index allocation method for multi-channel data broadcast. In: Advanced Information Networking and Applications Workshops, AINAW 2007, vol. 2, pp. 236–242 (2007)
17. Hu, T.: A new proof of the tc algorithm. SIAM Journal on Applied Mathematics 25(1), 83–94 (1973)
18. Lo, S., Chen, A.: Optimal index and data allocation in multiple broadcast channels. In: 16th International Conference on Data Engineering, ICDE 2000, pp. 293–302 (2000)