Global Optimization for Multi-Channel Wireless Data Broadcast with AH-Tree Indexing Scheme

Xiaofeng Gao, Yongtian Yang, Guihai Chen, Xin Lu, and Jiaofei Zhong

Abstract—Multi-channel wireless data broadcast is an appropriate approach to disseminate data to a mass number of mobile clients. In this paper, we present a global optimization for multi-channel wireless data broadcast with Alphabetic Huffman Tree (AH-Tree) Indexing scheme, which can deal with skewed access frequencies well. We present three novel designs to reduce the access latency and tuning time of the broadcast system. Firstly, we design a dynamic programming for AH-Tree construction with low time complexity. Compared with traditional Hu-Tucker algorithm in $O(n^k)$, our algorithm can build a k-ary AH-Tree index in $O(n^2)$. Next, we depict a new control table design, which eliminates up to 50 percent redundant entries while keeps the searching efficiency. We also theoretically prove that an optimal alphabetic tree has the minimum average tuning time among all tree-based index structures for skewed data broadcast. Thirdly, we design the new index and data allocation algorithms to further reduce the tuning time and access latency. The simulation results validate the effectiveness of our algorithms. In all, our global optimization mechanism can greatly improve the system performance and time efficiency for wireless data broadcast applications.

Index Terms—Data broadcast, index, Huffman Tree

1 Introduction

Wireless data broadcast is an efficient data dissemination technology to a mass number of mobile clients with battery-constraint portable wireless devices. Instead of point-to-point query-reply mode, a server broadcasts public information like traffic conditions, TV streams, 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.

Each mobile device has two modes: *active mode* and *doze mode*. Its energy consumption in active mode is far greater than that in doze mode. Intuitively, the criteria to evaluate the performance of a wireless data broadcast system are the downloading time and energy consumption of mobile devices. Correspondingly, *access latency (AL)* and *tuning time (TT)* 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 and data/index allocation methods are the most effective methods to reduce the tuning time and access latency. On one hand, indices help to reduce the active time of a mobile device significantly. Clients can follow the direction of indices, turn off during the waiting period, and turn on again right before the required datum appears. On the other hand, proper allocation methods can assign relevant data items and indices as closely as possible, and thus reduce the clients' waiting time. Hence, a broadcast service provider often considers both of the two techniques to improve the system performance. It first constructs indices according to the data set features. Then it linearizes the indices and allocates both indices and data onto the broadcasting channels.

Many works discussed efficient indexing schemes, which can be classified into three categories: hashing-based indexing [1], [2], tree-based indexing [3], [4], [5], and table-based indexing [6]. Among them, tree-based indices are more popular according to their easy-searching and fast-constructing characteristics. Additionally, based on the investigation of website popularity [7], say, few data items have high popularity while majority of data items actually get low preference, we choose the *Alphabet Huffman Tree* (AH-Tree) to build indices, which deals with skewed data well. In a typical AH-Tree, the higher the frequency of a datum, the shorter the path from the root to the corresponding leaf index.

After constructing an index tree, researchers tended to modify the index tree into a distributed index sequence to further improve the system performance [3], [8], [9]. This scheme relies on a *control table* structure, which is attached to some index nodes and directs the clients the nearest proper index node.

The next step is to linearize the index tree and allocate both the data and indices onto the broadcasting channels. For index allocation, several methods can be found in [5], [10], [11], [12], [13]. A typical example is [5], in which the authors proposed a heuristic linearization method to maintain the order of the index tree while keep the relative index nodes as close as possible. For data allocation, researchers

- *X. Gao, Y. Yang, and G. Chen are with the Shanghai Key Laboratory of Scalable Computing and Systems, Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. E-mail: {gao-xf, gchen}@cs.sjtu.edu.cn, yangyongtian@yahoo.com.*
- *X. Lu is with the Department of Computer Science, Columbia University, New York, NY. E-mail: xinlu@cs.columbia.edu.*
- *J. Zhong is with the Department of Computer Science, California State University East Bay, Hayward, CA. E-mail: jiaofei.zhong@csueastbay.edu.*

Manuscript received 17 Jan. 2015; revised 12 July 2015; accepted 3 Aug. 2015. Date of publication 16 Sept. 2015; date of current version 15 June 2016. Recommended for acceptance by A. Chandra. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2015.2479603

hope to arrange the data items according to their popularity, so that the average client waiting time is minimized. Yee et al. [14] gave an $O(t^2m)$ dynamic programming called DP to find an optimal schedule for data set with uniform length, where m is the number of data channels. Ardizzoni et al. [15] improved DP and proposed *Dichotomic* with $O(mt \log t)$-time. For *non-uniform* data lengths, the problem has been proved to be strong NP-hard for arbitrary m [15]. In this case, several greedy strategies are proposed in [14], [15], [16].

Although many literatures discussed almost every aspect to construct a broadcast system, none of them have considered this system globally and analyzed the inherent relationship among these aspects. Actually, each intermediate step does influence the overall system performance, and there still exist many optimization problems. For instance:

For tree construction, many literatures have studied AH-Tree. In [17], Hu and Tucker first proposed a binary AH-Tree algorithm in $O(t \log t)$, where t is the size of data items. Their design relies on a complex queue technology, which cannot be directly extended to arbitrary k-ary AH-Tree [18]. Later, Shivakumar et al. [19] extended Hu-Tucker Algorithm into k-ary AH-Tree, and first implemented it for data broadcast. However, 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 their design would be high up to $O(t^k)$ without adopting any particular queue structure. Similarly, [5], [8], and [20] discussed AH-Tree index for data broadcast respectively, none of which provided complete tree-construction process with time complexity analysis. How to construct an arbitrary k-ary AH-Tree efficiently remains an open question.

For control table generation, we found that enrolling control table results in two problems. Firstly, it brings additional space overhead, which is a non-negligible factor since we found that a large number of items in the control table are redundant. Secondly, it is hard to predict the performance of the index structure since with the help of control tables, clients will not always search the tree from the root.

For index allocation, we found that all the previous works have two impractical assumptions. The first assumption is that clients can get any of the next m broadcasted items, so researchers allocated relevant indices as closely as possible for better performance. However, their designs may perform poorly because when the client hops to a channel, there are potential synchronization costs [21], which is not small compared to the time downloading an index node. Thus clients may miss some intermediate directions and fail to get the correct index location. The second assumption is that clients always search the index tree from the root. After introducing control tables into the system, this is not necessary.

For data allocation, all the previous methods did not take the advantage of data size.

In this paper we consider a skewed data broadcast system with single query requests and provide a global optimization with the following improvements:

1) We propose an efficient dynamic programming to build a k-ary AH-Tree index in $O(t^2)$ time, which outperforms the previous $O(t^k)$ greatly.
2) We provide novel scheme to eliminate redundant entries in control tables using tolerable time, which saves up to 50 percent index size while keeps the searching efficiency. We also analyze the effect of control table and prove that an optimal alphabetic tree has the minimum average tuning time among all tree-based indices for skewed data broadcast.
3) We design the new index and data allocation methods to better improve the system performance. Our index allocation methods consider the effects of hopping and data size.

Our paper is organized as follows. Section 2 introduces the related works. Section 3 illustrates the problem formulation and system architecture. In Section 4 we describe index construction, including a dynamic programming for k-ary AH-Tree and distributed index sequence with simplified control tables. Sections 5 and 6 describe the data and index allocation methods respectively. In Section 7 we discuss the dynamic index update issue. In Section 8 we compare our allocation methods with previous works. Finally, Section 9 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 and index onto channels, in order to reduce access latency and tuning time.

Traditional disk-based indexing techniques have been modified to meet the requirement of data broadcast systems, which can be classified into three categories: (1). Hashing-based schemes [1], [2], which use hash functions to distribute data onto channels. E.g., Yao et al. [1] proposed MHash to facilitate skewed access probabilities and reduce access latency. (2). Table-based schemes, like exponential index [6], which shares links in different search tables and allows users to start searching at any index node. However, it may not perform well under non-uniform access probabilities. (3). Tree-based schemes, e.g., B tree [3], [4], Huffman tree [5]. They 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. [9] redesigned BTD and built a complete multi-channel broadcasting system with non-uniform data access probabilities and unequal data sizes.

Alphabetical Huffman tree is a skewed index tree that takes into account the data access probability, where more popular data have shorter search paths [19]. It is more practical since many types of data in social sciences can be approximated with a Zipfian distribution [7]. Zhong et al. [20] proposed a uniform AH-Tree indexing scheme to satisfy all possible environments. Lu et al. [22] built a multi-channel broadcasting system using AH-Tree index technology. In Section 4, we prove that the average tuning time of an optimal alphabetic tree is minimum among all index trees with control tables. Thus an AH-Tree, an optimal alphabetic tree, can handle various data set features without any impractical assumptions or constraints.

When it comes to multi-channel data broadcasting, the way to allocate index and data will produce heavy impact on the performance. Several index allocation methods [5], [10], [11], [12], [13] could be helpful to a specific index

TABLE 1
Symbol Description

Symbol	Description
D	Data set $D = \{d_1, \ldots, d_t\}$, totally t items
s_i, p_i	Size and access probability of d_i
s_{max}	$s_{max} = \max\{s_1, s_2, \ldots, s_t\}$
m, n	Number of data and index channels
T	An AH-Tree
S_i, P_i	$\sum_{d_j} s_j$, $\sum_{d_j} p_j$ (d_j is assigned to channel i).
L, l	Height and the cut level of T
k	Maximum branch no. for T
h_i^j	The jth index at the ith level of T
Δ_i	The ith sub-tree at level $l+1$
DFS(Δ_i)	Use Deep-First Search to linearize subtree Δ_i.
PATH($h_{i_1}^{j_1}, h_{i_2}^{j_2}$)	A path from $h_{i_1}^{j_1}$ to $h_{i_2}^{j_2}$, excluding $h_{i_2}^{j_2}$
MAX(h_i^j)	Maximum key h_i^j domains

structure, but meanwhile it might reduce the efficiency of another index scheme. Wang and Chen [10] proposed an index allocation method named TMBT, which creates a virtual BTD for each data channel and multiplexes them on the index channel. Lo and Chen [12] proposed an optimal index and data allocation, which minimizes the access latency, by representing all the possible allocations as a tree. However, most of these works do not consider the effects of hopping time, which makes them impractical. In [21], Yee and Navathe pointed out that each hop takes milliseconds, which should be added to the access time for clients.

Several works [8], [14], [15], [16], [21] deal with data allocation for multi-channel data broadcast. For the data set of uniform lengths, [14] proposed an $O(t^2m)$ dynamic programming to find the optimal schedule and a near optimal greedy algorithm to reduce the time complexity. In [15], an improved $O(mt \log t)$ algorithm named *Dichotomic* was proposed. For *non-uniform* lengths case, the problem has been proved to be *strong NP*-hard for arbitrary m [15], so researchers tend to design algorithms based on greedy and heuristic strategy [15], [16], [23]. While most of the researches dealt with the data allocation problem from server side, several works [24], [25], [26], [27], [28] discussed data retrieval scheduling problem from the client side.

3 System Architecture

3.1 Symbols and Definitions

Let $D = \{d_1, \ldots, d_t\}$ be the data set to broadcast, where t is the number of data items. A *Bucket* is the smallest broadcast unit and a datum may be split into several buckets. Let each datum d_i has bucket size s_i and access frequency q_i. Define $s_{max} = \max\{s_1, s_2, \ldots, s_t\}$. In this system, we assign m channels to the data and n channels to the index.

Clients request a single datum d_i each time. The *Access Latency* is the time from when clients send the request to the time they receive the server response. The *Tuning Times* is the number of times that clients tune into the channel. The system aims at minimizing these two measurement.

Table 1 summarizes the symbols used in this paper. Some of them will be described later.

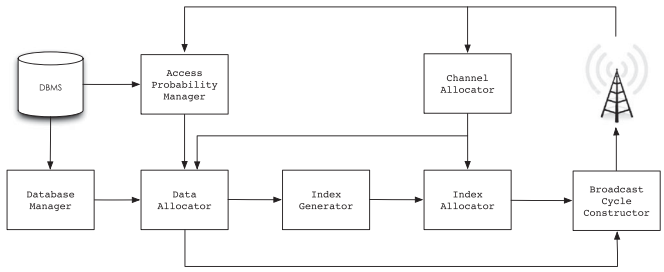


Fig. 1. System architecture.

3.2 System Architecture

The system architecture is illustrated in Fig. 1. The flow of the data broadcast system can be described as follows: Firstly, *Database Manager* collects the data from DBMS and each datum has an access probability generated by *Access Probability Manager* from historical record. Next, *Channel Allocator* assigns index channels and data channels. Then *Index Generator* will generate index sequences from the data set D and *Index Allocator* assigns these sequences into the index channels, while at the same time, *Data Allocator* allocates data bucket onto data channels. Finally, *Broadcast Cycle Constructor* combines these together to form a complete broadcast program. In this paper, we proposed methods to optimize the key parts of this system.

4 Index Generator

In this section, we describe the *Index Generator*, which constructs index based on the data set. In this modular, we use AH-Tree as index and propose an efficient AH-Tree construction algorithm that can build arbitrary k-ary AH-Tree index in $O(t^2)$, which outperforms the previous $O(t^k)$ greatly [5], [8], [19], [20].

4.1 AH-Tree Construction

First, let us introduce the two-stage construction process of the k-ary AH-Tree [17], [19]. In the first stage, we build an optimal k-ary Huffman tree without 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 Algorithm 1.

Stage 1 (Line 4 to 10) contains several iterations. During each iteration, we select k nodes to merge into one new parent node and then insert it into the data sequence. The frequency of the new node is the frequency sum of its k children. Then mark the k nodes as processed leaf nodes, and delete them from the sequence. After that, we start a new iteration and repeat 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.

Note that k is the number of tree branches. The selected k nodes should satisfy three conditions: (1). There are no leaf nodes among them; (2). The sum of their frequencies is minimum among all k candidate groups; and (3). They should be the leftmost nodes.

Stage 2 (Line 11 to 17) adjusts T' in a bottom-up, left-right approach such that every k consecutive nodes at

TABLE 2
Example Data Set

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

the same level have a same parent. Finally, an AH-Tree T is constructed. The correctness proof is provided in [29].

Algorithm 1. AH-Tree Construction

1: **Input:** D, P;
2: **Output:** Node set H of AH-Tree T.
3: Create t leave nodes for $d_i \in D$ and push them into N. Set $I = \{1, \ldots, t\}$.
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 $n_1, \ldots, n_k$ **and** $n_1, \ldots, n_k$ are the leftmost k nodes **then**
6: Merge $n_1, \ldots, n_k$ as n' with $r_{n'} = \sum_{i=1}^{k} r_{n_i}$ (n' is parent of $n_1, \ldots, n_k$) ;
7: Insert n' into N, mark $n_1, \ldots, n_k$ 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** $l = L \rightarrow 2$ **do**
13: Find the leftmost index node p on the $(l-1)$th level and the leftmost k nodes $n_1, \ldots, n_k$ on the lth level, and then mark them;
14: Record "p" into field new_parent of $n_1, \ldots, n_k$ and "$n_1, \ldots, n_k$" 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 $new\_children$.

Example 1. Throughout the paper, we use a data set in Table 2 as an example. The freq. means the access frequency of each datum from historical record.

We apply Algorithm 1 to this data set to construct the AH-Tree. After that, we generate T' and T as shown in Fig. 2, respectively. T is the H-tree after Stage 1, while T' is the AH-tree after Stage 2. Data items are labeled by gray circles with access frequencies and the numbers below the circle are their keys. Note that data items are ordered by keys after Stage 2.

4.2 Dynamic Programming for Data Selection

It is time-consuming to choose k nodes from the data sequence in stage 1. Without any special data structure, we need $O(i^k)$ to select each group, where i is the number of nodes in the current iteration and k is the number of branches in the tree, so totally the time complexity should be $O(t^k)$. In this section, we design a novel dynamic programming to reduce the time complexity to $O(tk)$. 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 $[d_1, d_2, \ldots, d_i]$. There are two cases to consider.

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.

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

1) If the ith 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 ith 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 one.

Then the recursive relation for f is shown below:

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

In Case 2, there are also two subcases:

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

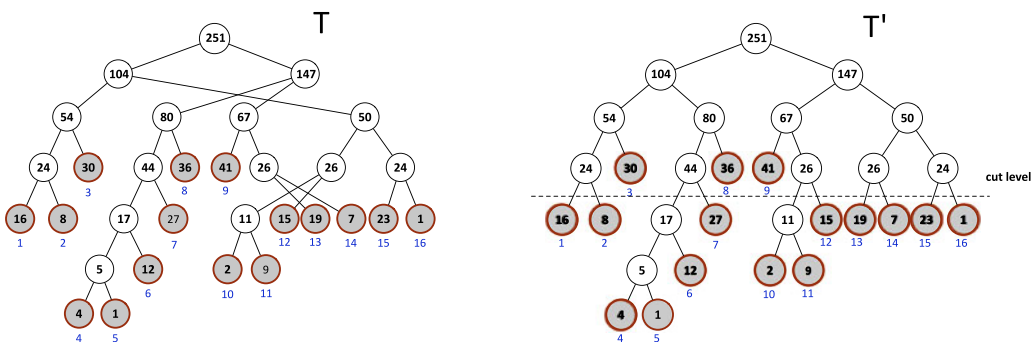


Fig. 2. T (AH-tree construction after stage 1) and T' (AH-tree construction after stage 2).

Hence, the recursive relation for g is as following:

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

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.

Lemma 1. *The time complexity of this subroutine dynamic programming is $O(tk)$.*

Proof. Since the time complexity of computing each $f(i,j)$ and $g(i,j)$ is $O(1)$ and there are nk such $f(i,j)$ and $g(i,j)$ and $n \leq t$, the time complexity of this subroutine is $O(tk)$. □

Theorem 1. *The time complexity of Algorithm 1 is $O(t^2)$.*

Proof. By Algorithm 1, this subroutine will run at most $\frac{t}{k-1}$ times. Therefore, by Lemma 1, the time complexity of Algorithm 1 is $O(t^2)$. □

4.3 Correctness of the Dynamic Programming

We now verify the correctness of our dynamic programming. The first step is to prove that the problem has an *optimal substructure*, then we prove that it also has overlapping subproblems. In the following, we show that both f and g have optimal substructures. Let's start with g, which is independent of f. For detailed proofs please refer our preliminary work [30].

Lemma 2 (Optimal Substructure of g). *Let $D_i = [d_1, \ldots, d_i]$ 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 D_i). Then:*

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

Lemma 3 shows that solving recursion f in Case 1 also contains an *optimal substructure*.

Lemma 3 (Optimal substructure of f). *Let $D'_i = [d_1, \ldots, d_i]$ be the sequence, $Z_j = [d_{i_1}, \ldots, d_{i_j}]$ be any optimal solution satisfying Case 1 (we refer it as an optimal solution of D'_i), then Z_j is an optimal solution of D'_{i-1}.*

Lemma 2 and Lemma 3 imply that both f and g have optimal substructures. The next two lemmas will complete the final conclusion, in which Lemma 4 can be directly derived from Lemma 2 and Lemma 3.

Lemma 4. *The problem of solving $\min(f(n,k), g(n,k))$ has optimal substructure and overlapping problems.*

Form Lemma 4, we have Theorem 2.

Theorem 2. *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 selections.*
3) *The k nodes should be the leftmost candidates.*

TABLE 3
Format of the Control Table

1	$\text{MAX}(\Delta_{g-1})$	$h_1^{1[1]}$
2	$\text{MAX}(h_2^{j_2})$	$h_1^{1[x_1+1]}$
...	...	...
r	$\text{MAX}(h_r^{j_r})$	$h_{r-1}^{j_{r-1}[x_r+1]}$
...	...	...
i	$\text{MAX}(h_i^{j_i})$	$h_{i-1}^{j_{i-1}[x_i+1]}$

4.4 Distributed AH-Tree Construction

To avoid searching from the root every time, we employ the distributed index technique called *control tables* [3]. The tree is split into *replicated* 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 in to the right position. However, there are two problems for control tables. The first is that control tables have redundancy. When the tree becomes larger, the redundant entries will be as many as half of the whole control index, which wastes lots of resource. The second is that it is hard to analyze the performance of an index tree because clients do not always search from the root. In this section, we propose an efficient algorithm to eliminate all the redundancy entries and give theoretical analysis for distributed index trees.

4.4.1 Control Table Construction

Let l be the *cut level* for index tree T. The nodes in replicated part are called *control index*, while the remaining nodes are named *search index*. Let h_i^j denote the jth index node at level i. Note that if the jth node of level i is a data node, then h_i^j does not exist.

Let Δ_i denote the ith subtree below the cut level l, rooted at h_{l+1}^i. To generate the distributed index sequence, we use $\text{PATH}(h_i^j)$ to represent a path from the root to h_i^j excluding h_i^j. For instance, $\text{PATH}(h_4^4)$ in Fig. 2 is $[h_1^1, h_2^2, h_3^3]$. In order to broadcast data, we linearize the tree by depth-first search. For replicate-part nodes, we use $h_i^{j[x]}$ to denote the xth appearance of h_i^j during the process of depth-first search.

For each control index $h_i^{j[x]}$, suppose that $\text{PATH}(h_i^{j[x]}) = [h_1^{1[x_1]}, h_2^{j_2[x_2]}, \ldots, h_{i-1}^{j_{i-1}[x_{i-1}]}]$, then the control table of $h_i^{j[x]}$ is shown in Table 3.

Each entry of the control table contains two elements: the key value and the control index it hops to. $\text{MAX}(\Delta_{g-1})$ in the first entry gives a lower bound of the dominating range of $h_i^{j[x]}$. If the required key is less than $\text{MAX}(\Delta_{g-1})$, it means that the key has been broadcast, so we have to wait for the next broadcast cycle. In this case, we jump to $h_1^{1[1]}$. The key value in the rth entry gives an upper bound of the dominating range of $h_r^{j_r}$. The second element gives a hint of which subtree to hop to in the next step. When the client retrieves a datum with key greater than this element, it should hop to the control index in this entry. Note that a control index

TABLE 4
The Control Table of the Example Data Set

$h_1^{1[1]}$	no no	$h_2^{1[1]}$	no no 8 $h_1^{1[2]}$	$h_3^{1[1]}$	no no 8 $h_1^{1[2]}$ 3 $h_2^{1[2]}$	$h_4^{1[1]}$	no no 8 $h_1^{1[2]}$ 3 $h_2^{1[2]}$ 2 $h_3^{1[2]}$	$h_4^{1[2]}$	1 $h_1^{1[1]}$ 8 $h_1^{1[2]}$ 3 $h_2^{1[2]}$ 2 $h_3^{1[2]}$	$h_3^{1[2]}$	2 $h_1^{1[1]}$ 8 $h_1^{1[2]}$ 3 $h_2^{1[2]}$	$h_2^{1[2]}$	3 $h_1^{1[1]}$ 8 $h_1^{1[2]}$	$h_3^{2[1]}$	3 $h_1^{1[1]}$ 8 $h_1^{1[2]}$ 8 $h_1^{1[2]}$
$h_4^{3[1]}$	3 $h_1^{1[1]}$ 8 $h_1^{1[2]}$ 8 $h_1^{1[2]}$ 7 $h_3^{2[2]}$	$h_4^{3[2]}$	6 $h_1^{1[1]}$ 8 $h_1^{1[2]}$ 8 $h_1^{1[2]}$ 7 $h_3^{2[2]}$	$h_3^{2[2]}$	7 $h_1^{1[1]}$ 8 $h_1^{1[2]}$ 8 $h_1^{1[2]}$	$h_1^{1[2]}$	8 $h_1^{1[1]}$	$h_2^{2[1]}$	8 $h_1^{1[1]}$ 16 no	$h_3^{3[1]}$	8 $h_1^{1[1]}$ 16 no 12 $h_2^{2[2]}$	$h_3^{3[2]}$	9 $h_1^{1[1]}$ 16 $h_1^{1[2]}$ 12 $h_2^{2[2]}$	$h_4^{6[1]}$	9 $h_1^{1[1]}$ 16 no 12 $h_2^{2[2]}$ 12 $h_2^{2[2]}$
$h_4^{6[2]}$	11 $h_1^{1[1]}$ 16 no 12 $h_2^{2[2]}$ 12 $h_2^{2[2]}$	$h_2^{2[2]}$	12 $h_1^{1[1]}$ 16 no	$h_3^{4[1]}$	12 $h_1^{1[1]}$ 16 no 16 no	$h_4^{7[1]}$	12 $h_1^{1[1]}$ 16 no 16 no 14 $h_3^{4[2]}$	$h_4^{7[2]}$	16 $h_1^{1[1]}$ 16 no 16 no 14 $h_3^{4[2]}$	$h_3^{4[2]}$	16 $h_1^{1[1]}$ 16 no 16 no	$h_4^{8[1]}$	14 $h_1^{1[1]}$ 16 no 16 no 16 no	$h_4^{8[1]}$	15 $h_1^{1[1]}$ 16 no 16 no 16 no

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. 2 is shown in Table 4. Look at the control table of $h_4^{6[2]}$ in the lower left corner of the table. Since it is the second appearance of h_4^6, all the keys less than or equal to 11 have been broadcast. Thus if the searching key is less than 11, we jump to $h_1^{1[1]}$. The second entry means that, if the search key is larger than 16, then there is nowhere to go since the largest key is 16. Suppose that we want to search the key value of 14, then the next hop is $h_2^{2[2]}$, which is exactly the nearest ancestor dominating 14.

From the control tables in Table 4, we find that some control tables, marked as gray, have redundancy. For example, in the control table of $h_3^{4[1]}$, there are two entries "*16 no*". As the next section shows, almost $\frac{1}{k}$ of the control tables contain redundancy.

4.4.2 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 tree-based indices employing distributed technique. For the balance k-ary tree, the scheme can save $\frac{1}{k}$ of the space for storing the control tables.

Having redundancy in the control table means that for two entries x and y in the control table, the key values of them are identical. Formally speaking, suppose the ith and jth entries are redundant in $h_i^{j[x]}$'s control table. It means that the upper bound of the dominating range of ith and jth ancestors of $h_i^{j[x]}$ 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. Thus 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. The upper bounds of the dominating range of the control indices along this path are all the same. Consider the left part of Fig. 3. Since y is the rightmost child of x, we have $\mathrm{MAX}(x) = \mathrm{MAX}(y)$. Thus the control table of z 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. 3. However, all those causes can be combined into Theorem 3. Before that, we first give two properties of the control table.

Property 1. *If index node x and y are ancestors of z, and x is ancestor of y, then $\mathrm{MAX}(x) \geq \mathrm{MAX}(y) \geq \mathrm{MAX}(z)$, and in the control table of z, the entry of x is in front of that of y.*

Property 2. *If $\mathrm{MAX}(x) = \mathrm{MAX}(y)$ and y is a child of x, then y is the rightmost child of x and vice versa.*

Theorem 3. *Let $\mathrm{PATH}'(h_i^{j[x]}) = [h_2^{j_2}, h_3^{j_3}, \ldots, h_i^{j_i}]$, which is $\mathrm{PATH}(h_i^{j[x]})$ plus $h_i^{j[x]}$ excluding the root. The control table of $h_i^{j[x]}$ contains redundancy iff there exist consecutive index nodes $h_{a-1}^{j_{a-1}}$, $h_a^{j_a}$ in $\mathrm{PATH}'(h_i^{j[x]})$, such that $h_a^{j_a}$ is the rightmost child of $h_{a-1}^{j_{a-1}}$.*

Proof. Please refer to our previous work [30]. □

By traversing T, we can find and eliminate all redundant entries by Theorem 3. Finally we get "light-weight" control tables.

Theorem 4. *The time complexity of the redundancy elimination algorithm is $O(t)$.*

Proof. To eliminate all the redundancy, we only need to traverse the index tree, say by DFS, and check possible redundancy by Theorem 3, which costs $O(1)$ time. This is tolerable compared to the benefit the elimination procedure brings. □

4.4.3 Control Table Analysis

In this section, we give theoretical analysis of the distributed index trees. Note that there are some properties when clients visit the index channel:

Property 3. *Clients need to visit the index channels more than once.*

Property 4. *With the help of control tables, clients need not search the tree from the root.*

These two properties make it difficult to analyze the performance of index scheme. In this section, we will provide a reasonable measurement to evaluate the performance of index sequences.

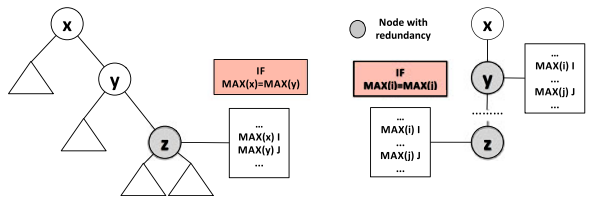


Fig. 3. Example of redundancy in control table.

Recall that the system has n index channels. Because the index nodes may be attached by control tables, the indices' sizes may vary. We first give some definitions.

Definition 1. *Extend the definition of* PATH *in Section 4.4. Let* $\mathrm{PATH}(h_i^j, d_x)$ *denote the path from index node* h_i^j *to data node* d_x*, excluding* d_x*, that is,* $\mathrm{PATH}(h_i^j, d_x) = [h_i^j, h_{i+1}^{j_1}, h_{i+2}^{j_2} \ldots]$*. Additionally, if* h_i^j *is the root of the tree, we write* $\mathrm{PATH}(d_x)$ *for simplity.*

We next define $\mathrm{PATHC}(h_i^j, d_x)$ w.r.t. $\mathrm{PATH}(h_i^j, d_x)$.

Definition 2. *PATHC* $(h_i^j, d_x) = [c(h_i^j), c(h_{i+1}^{j_1}), \ldots]$*, where* $c(h_i^j)$ *is the index channel that* h_i^j *is assigned to.*

Note that some consecutive indices in $\mathrm{PATH}(h_i^j, d_x)$ may be assigned to the same channel y, thus in $\mathrm{PATHC}(h_i^j, d_x)$, there will be some pattern like $\ldots, y, y, \ldots$. We define the following compressing operation on $\mathrm{PATHC}(h_i^j, d_x)$.

Definition 3. *The compression of* $\mathrm{PATHC}(h_i^j, d_x)$*, denoted as* $\mathrm{cPATHC}(h_i^j, d_x)$*, is a sequence of index channels, where if there are consecutive* y*s in* $\mathrm{PATHC}(h_i^j, d_x)$*, we only write one* y *in* $\mathrm{cPATHC}(h_i^j, d_x)$*.*

Example 3. Suppose $\mathrm{PATH}(d_x) = [h_1^1, h_2^2, h_3^4, h_4^7, h_5^{12}]$ and after allocation, h_1^1 and h_2^2 are assigned to channel c_1, h_3^4 is assigned to channel c_2 and h_4^7, h_5^{12} are assigned to channel c_3. Then

1) $\mathrm{PATHC}(d_x) = [c_1, c_1, c_2, c_3, c_3]$,
2) $\mathrm{cPATHC}(d_x) = [c_1, c_2, c_3]$.

To analyze the effects of control tables, we make three assumptions to simplify the analysis:

1) When visiting index channel c_i, we assume on average $\frac{1}{2}S_i$ logical time is needed to get all the desired indices in this channel, where S_i is the total size of index nodes assigned to channel i.
2) When searching the index tree for datum d_x, after the first hopping, each index on $\mathrm{PATH}(d_x)$ will be accessed with equal propabality.
3) The cost of hopping from one channel to another channel is defined as *cost*.

The first assumption says that on the average we will stay at channel c_i for about $\frac{1}{2}S_i$ logical time. The second assumption is based on the fact that if the clients tune to the channels randomly, each index node in the path will be accessed equally likely.

Using these definitions and assumptions, we derive the measurements of average expected delays (AEDs) and average tuning times for tuning the index channels.

Lemma 5 (Average Expected Delay for Index Channels involving Control Table (AEDCT)). *For index trees involving control tables, the average expected delay for any allocation algorithm is*

$$\sum_{d_x \in D} \frac{p_x}{|PATH(d_x)|} \sum_{\substack{h_i^j \in PATHC(d_x) \\ c_y \in cPATHC(h_i^j, d_x)}} \left(\frac{S_y}{2} + cost \right). \tag{1}$$

AEDCT is computed as follows. For each datum d_x in D, after the first tuning, the client will jump to index h_i^j on $\mathrm{PATH}(d_x)$ with probability $1/|\mathrm{PATH}(d_x)|$. And then, the client begins to search the tree from h_i^j. The channels that the client hops to are in $\mathrm{cPATHC}(h_i^j, d_x)$. For each channel y in $\mathrm{cPATHC}(h_i^j, d_x)$, $\frac{1}{2}S_y + cost$ is needed to get all the indices needed by Assumption (1), (3). Finally, we get Eq. (1) by taking the expectation.

Next, we give the definition of *average tuning time* for index trees involving control tables.

Definition 4 (Average Tuning Time for Index Tree involving Control Table (ATTCT)). *The average tuning time for an index tree involving control table is*

$$\mathrm{ATTCT} = \sum_{i=1}^{t} p_i l_i, \tag{2}$$

where l_i *is the number of indices needed to visit when searching* d_i*, and* p_i *is the access probability of* d_i*.*

Because of control tables, most of the time clients do not need to search from the root. Thus l_i in Eq. (2) is not the depth of d_i in the tree. However, Lemma 6 shows that according to Assumption (2), there is a strong relation between l_i and the depth of d_i.

Lemma 6. *On average clients need to visit* $\frac{|PATH(d_i)|}{2} + \frac{1}{2}$ *indices to find* d_i*.*

Proof. By Assumption (2), the number of indices needed visited is

$$\frac{\sum_{j=0}^{|\mathrm{PATH}(d_i)|-1} |\mathrm{PATH}(d_i)| - j}{|\mathrm{PATH}(d_i)|} = \frac{|\mathrm{PATH}(d_i)|}{2} + \frac{1}{2}.$$

Therefore the lemma holds. □

Theorem 5. *The average tuning time of an optimal alphabetic tree is minimum among all index tree involving control tables.*

Proof. By Lemma 6, $l_i = \frac{|\mathrm{PATH}(d_i)|}{2} + \frac{1}{2}$. Then the average tuning time is

$$\begin{aligned} \mathrm{ATTCT} &= \sum_{i=1}^{t} p_i l_i = \sum_{i=1}^{t} p_i \left(\frac{|\mathrm{PATH}(d_i)|}{2} + \frac{1}{2} \right) \\ &= \frac{1}{2} \sum_{i=1}^{t} p_i \times |\mathrm{PATH}(d_i)| + \frac{1}{2}. \end{aligned}$$

Since an optimal alphabetic tree has minimum $\sum_{i=1}^{t} p_i \times |\mathrm{PATH}(d_i)|$, the theorem holds. □

Thus we have proved that optimal alphabetic trees have the minimum ATTCT. For $k = 2$, [31] proved that AH-Tree is actually the optimal alphabetic tree. For other cases, [32] provided an algorithm to construct an general optimal alphabetic tree.

5 Index Allocator

In this section, we propose the index allocation methods used in the *Index Allocator*. Previous works assumed that the hopping time can be ignored. However, when the client

hops to a channel, there are potential synchronization costs [21]. These costs are on the order of milliseconds, which is not small compared to the time to download an index node. Thus clients may miss some intermediate directions and fail to get the correct index location. In the following sections, two algorithms considering hopping cost are presented. The first algorithm tries to minimize the number of hopping times while it may enlarge the length of the broadcast cycle. The second algorithm improves the previous algorithm and tries to balance the length of broadcast cycle and the hopping times.

5.1 Minimum Hopping Allocation

The simplest way to avoid hopping is to assign all the indices on the same path into one channel. Thus the basic idea of this algorithm is to treat all the indices on $\mathrm{PATH}(d_i)$ as a unit and then run the algorithm for data allocation of non-uniform lengths. The detailed description is shown in Algorithm 2.

Algorithm 2. Minimum-Hopping Allocation

1: For each d_i, define a new datum d'_i, which contains all the indices on $\mathrm{PATH}(d_i)$. The size z'_i of d'_i is defined as the total size of indices in $\mathrm{PATH}(d_i)$ and the probability p'_i of d'_i is defined as the probability p_i of d_i. Define $D' = \{d'_1, d'_2, \ldots, d'_t\}$.
2: Run one of the data allocation algorithm on D', for example, GREEDY proposed in [14].
3: In the final schedule, replace d'_i with the indices of $\mathrm{PATH}(d_i)$.

Theorem 6. *The hopping number of Algorithm 2 is 1.*

Proof. Since we treat each path as a unit, all the indices along the path locate on the same channel. Therefore, clients just need to hop to one channel to get all the desired indices. □

The main disadvantage of Algorithm 2 is that each index is broadcast more than once, making the broadcast cycle too long. This is shown in Theorem 7.

Theorem 7. *There are $\Omega(t \log t)$ index nodes in the broadcast cycle in total.*

Proof. Let T represent the AH-Tree constructed from data set D. By Algorithm 2, the number of index nodes broadcasted is $\sum_{i=1}^{t} \mathrm{PATH}(d_i)$. Define another data set $Q = \{q_1, q_2, \ldots, q_t\}$ and each q_i has the same access probability. To prove that $\sum_{i=1}^{t} \mathrm{PATH}(d_i) = \Omega(t \log t)$, we need the following property: □

Property 5. *The AH-Tree T' constructed from Q is a complete k-ary tree.*

Reassign the probability p_i of the leaf node d_i in T such that $p_i = \frac{1}{t}$. Since each leaf node in T' has depth $O(\log t)$, by the property of AH-Tree,

$$\sum_{i=1}^{t} \frac{1}{t} O(\log t) \le \sum_{i=1}^{t} \frac{1}{t} |\mathrm{PATH}(d_i)| = \frac{1}{t} \sum_{i=1}^{t} \mathrm{PATH}(d_i).$$

Thus, $\sum_{i=1}^{t} \mathrm{PATH}(d_i) = \Omega(t \log t)$.

5.2 Balanced Allocation

Algorithm 2 gives a minimum hopping allocation with additional cost arising from duplicating several kernel searching nodes by a factor of $O(\log t)$. To improve it, instead of treating a path as a single meta-datum, we treat the path to a group of data items as a whole. In this way, all the index nodes on the same path will be assigned to the same channel, which keeps the minimum hopping time, meanwhile only small number of index nodes are broadcast more than once.

Suppose that l' is an integer between 1 and the depth of the AH-Tree. Recall that Δ_i denotes the ith subtree at level $l'+1$, counting from left to right. Let $\mathrm{DFS}(\Delta_i)$ be the list of indices getting by doing the DFS on Δ_i. Algorithm 3 shows the detailed improvement.

Algorithm 3. Balanced Allocation

1: For each subtree Δ_i rooted at $h^i_{l'+1}$, define a new datum d'_i for $[\mathrm{PATH}(h^1_1, h^i_{l'+1}), \mathrm{DFS}(\Delta_i)]$. Define the size s'_i of d'_i as the total size of indices in $[\mathrm{PATH}(h^1_1, h^i_{l'+1}), \mathrm{DFS}(\Delta_i)]$ and the probability p'_i of d'_i as the sum of the probabilities of data items in the subtree Δ_i. Define $D' = \{d'_1, d'_2, \ldots, \}$.
2: Run one of the data allocation algorithm on D', for example, GREEDY proposed in [14].
3: Replace d'_i by $[\mathrm{PATH}(h^1_1, h^i_{l'+1}), \mathrm{DFS}(\Delta_i)]$ in the finial schedule.

Algorithm 3 has nice properties shown below.

Theorem 8. *The hopping times in Algorithm 3 is 1.*

Proof. By Algorithm 3, all the indices in the same path are in the same channel. Therefore, clients just need to hop to one channel to get all the desired indices, which means that only one hopping is needed. □

Theorem 9. *If $l' = O(1)$, there are $O(t)$ index nodes in the broadcast cycle in total.*

Proof. Suppose that $|\mathrm{DFS}(\Delta_i)|$ is the number of indices in the subtree Δ_i. It is easy to see that there are at most $k^{l'}$ subtrees. Then the number of index nodes in the broadcast cycle is less than

$$\sum_{i=1}^{k^{l'}} (l' + |\mathrm{DFS}(\Delta_i)|) = \sum_{i=1}^{k^{l'}} l' + \sum_{i=1}^{k^{l'}} |\mathrm{DFS}(\Delta_i)| = O(t),$$

since the tree has at most $2t-1$ index nodes. □

It seems that Algorithm 3 is better than Algorithm 2 because they have the same hopping time but Algorithm 3 has shorter broadcast cycle. However, AEDCT does not only depend on these two factors. In Section 8, we will further study the relation between AEDCT and the hopping times and other factors.

6 Data Allocator

Now let us optimize the *Data Allocator*. We first introduce a baseline method named DPYRAMID [9]. Then we design a greedy algorithm for *non-uniform* data sets, taking advantage of data sizes. For uniform length data set, our algorithm always results in an optimal solution, while GREEDY [14] cannot guarantee this.

6.1 Problem Definition

Recall that there are m identical data channels to broadcast data set $D = \{d_1, d_2, \ldots, d_t\}$. Each d_i has an access probability p_i and a length s_i. A *valid* data allocation $V = \{D_1, D_2, \ldots, D_m\}$ is an m-partition for D. Each D_i contains data items assigned to data channel i. Define the channel length $S_i = \sum_{d_k \in D_i} s_k$, as the total data length of D_i. The *average expected delay* (AED) over all data channels is defined as [14]:

$$\text{AED} = \frac{1}{2}\sum_{j=1}^{m}\left(S_j \sum_{d_i \in D_j} p_i\right). \tag{3}$$

Data Allocation Problem is to find a valid schedule V to minimize AED in Eq. (3).

6.2 Dynamic Pyramid Allocation (DPYRAMID)

Algorithm 4 shows DPYRAMID [9], which can be viewed as a baseline for comparison.

Algorithm 4. Dynamic Pyramid Allocation (DPYRAMID)

1: **Input:** D, m;
2: **Output:** A valid partition $V = \{D_1, D_2, \ldots, D_m\}$.
3: Sort D by $\frac{p_i}{s_i}$ descendingly as $D' = \{d'_1, d'_2, \ldots, d'_t\}$;
4: Set $ave = 0$; $p = \sum_{i=1}^{t} p_i$; $thre = \frac{p}{m}$; $D_i = \emptyset$; $j = 1$;
5: **for** $i = 1$ to t **do**
6: **if** $ave \leq thre$ **then**
7: $ave = ave + p'_i$; $D_j = D_j \cup \{d'_i\}$;
8: **else**
9: $p = p - ave$; $ave = 0$; $thre = \frac{p}{m-j}$; j++; i−−;
10: **end if**
11: **end for**

There is a dynamic threshold for each data channel. Suppose that the access frequency sum is p, then the threshold of the first channel is $\frac{p}{m}$. We assign data items to the first data channel one by one until the total weight exceeds this threshold. We then assign the remaining indices to the next channel with an updated threshold as follows for fairness.

$$thre = \frac{\text{total frequency of the remaining data items}}{\text{number of remaining data channels}}.$$

We repeat the above process until no datum is left.

Note that we can apply the same method to allocate index, which we simply define the weight of index as the probability of this index.

6.3 GDPDA Algorithm

Lemma 7 [15] inspires the design of our *grouped dynamic programming data allocation* (GDPDA) algorithm.

Lemma 7. *Let $D^i = [d^i_1, d^i_2, \ldots]$ be the data set of size i with non-increasing order on access probability (each d^i_j has the same size). There is an optimal solution for partitioning the items of D into m groups $D_1, D_2, \ldots, D_m$ such that, if $a < b < c$ and $d^i_a, d^i_c \in D_j$, then $d^i_b \in D_j$.(proved in [15])*

In other words, there exists an optimal solution in the form:

$$D^1 : \underbrace{d^1_1, \ldots, d^1_{S^{(1)}_1}}_{\in \text{Channel } c_1}, \underbrace{d^1_{S^{(1)}_1+1}, \ldots, d^1_{S^{(1)}_2}}_{\in \text{Channel } c_2}, \ldots, \underbrace{d^1_{S^{(1)}_{m-1}+1}, \ldots, d^1_{|D^1|}}_{\in \text{Channel } c_m}$$

$$D^2 : \underbrace{d^2_1, \ldots, d^2_{S^{(2)}_1}}_{\in \text{Channel } c_1}, \underbrace{d^2_{S^{(2)}_1+1}, \ldots, d^2_{S^{(2)}_2}}_{\in \text{Channel } c_2}, \ldots, \underbrace{d^2_{S^{(2)}_{m-1}+1}, \ldots, d^2_{|D^2|}}_{\in \text{Channel } c_m}$$

$$\vdots$$

$$D^z : \underbrace{d^z_1, \ldots, d^z_{S^{(z)}_1}}_{\in \text{Channel } c_1}, \underbrace{d^z_{S^{(z)}_1+1}, \ldots, d^z_{S^{(z)}_2}}_{\in \text{Channel } c_2}, \ldots, \underbrace{d^z_{S^{(z)}_{m-1}+1}, \ldots, d^z_{|D^z|}}_{\in \text{Channel } c_m}$$

We give our data allocation algorithm in Algorithm 5. We first group data items by data sizes, then for each group with data size i, we try to find an partition for it to reduce the AED as much as possible.

Algorithm 5. Grouped Dynamic Programming Data Allocation

1: Group data items D by data size s_i's.
2: For data group D^i with size i, sort it decreasingly by access frequency and run a dynamic programming to find the sub-optimal assignment based on the assignments of groups with sizes $s_{max}, s_{max} - 1, \ldots, i + 1$.

Specifically, Algorithm 5 contains two steps:

Step 1: Group data items according to their lengths:

In the first step, we group the data items by lengths. There are at most t different data lengths in D. Therefore, t link lists are enough to store all groups and the total space needed is $O(t)$. It is easy to see that the time complexity of this step is $O(t)$.

Step 2: Group assignment:

In this step, we deal with D^i's in turn. Suppose that the groups $D^{s_{max}}, D^{s_{max}-1}, \ldots, D^{i+1}$ have been assigned onto channels. Define AED^{i+1}_j corresponding to channel j as

$$\text{AED}^{i+1}_j = \frac{1}{2} P^{i+1}_j S^{i+1}_j,$$

where P^{i+1}_j and S^{i+1}_j are the sum of frequencies and lengths of the data items assigned to channel j. That is, AED^{i+1}_j is the *temporal* AED of channel j after we finish allocating $D^{s_{max}}, D^{s_{max}-1}, \ldots, D^{i+1}$.

Given AED^{i+1}_j for $j = 1, 2, \ldots, m$, our algorithm finds a partition of D^i of the form

$$\underbrace{d^i_1, \ldots, d^i_{S^{(i)}_1}}_{\text{Channel } c_1}, \underbrace{d^i_{S^{(i)}_1+1}, \ldots, d^i_{S^{(i)}_2}}_{\text{Channel } c_2}, \ldots, \underbrace{d^i_{S^{(i)}_{m-1}+1}, \ldots, d^i_{|D^i|}}_{\text{Channel } c_m}$$

such that $\sum_j \text{AED}^i_j$ is minimum. To do this, we use the following dynamic programming strategy. Note that the initial stage of the algorithm is the assignment of $D^{s_{max}}, D^{s_{max}-1}, \ldots, D^{i+1}$.

Define $M[|D^i|][m]$ to be the minimum cost of assigning $d^i_1, \ldots, d^i_{|D^i|}$ to channel $1, 2, \ldots, m$. We use the following recurrence to compute $M[|D^i|][m]$,

$$M[|D^i|][m] = \min_{1<x<|D^i|}(M[x][m-1] + C(x+1, |D^i|, m), \quad (4)$$

where $C(x+1, |D^i|, m)$ is the cost of assigning data items $d^i_{x+1}, \ldots, d^i_{|D^i|}$ to channel m. Note that when computing $C(l_1, l_2, m)$, we need to consider the former items that have been assigned to channel m. Thus we should use Eq. (5) to compute $C(l_1, l_2, m)$:

$$C(l_1, l_2, m) = \frac{1}{2}\left(P^{i+1}_m + \sum_{x=l_1}^{l_2} p^i_x\right)(S^{i+1}_m + i(l_2 - l_1 + 1)). \quad (5)$$

Given $\text{AED}^{i+1}_m, P^{i+1}_m$ and S^{i+1}_m, $C(l_1, l_2, m)$ can be computed independently. By Eq. (4) and Eq. (5), we can implement this idea easily.

Theorem 10. *The time complexity of Algorithm 5 is $O(t^2 m)$.*

Proof. For group of size i, the most time-consuming part is the dynamic programming of Step 2, whose time complexity is $O(|D^i|^2 m)$. Summing for all i, we get the complexity of Algorithm 5 is $O(t^2 m)$. □

The most important consideration made in Algorithm 5 is that we first deal with groups of larger size. Intuitively, the allocation for data group with larger data size in the optimal solution tends to be similar to the optimal allocation if we only consider this group. This is because moving a large datum leads to large cost. By allocating groups of larger data size first, Algorithm 5 tries to match the optimal solution as close as possible, which results in good performance.

To further evaluate our design, Section 8 compares Algorithm 5 with the traditional greedy algorithm *GREEDY* [14] and the dynamic pyramid algorithm DPYRAMID [9] on the number of data items, channels and the skew of data distribution, respectively.

7 Dynamic Update

We dynamically update our indices when the broadcasting data change. The dynamic updating process could be classified into two categories: *Simple Update*, and *Multipart Update*. The *Database Manager* checks the DBMS for potential update periodically, since the *Access Probability Manager* can update the access probability of each datum. All update requests will be sent to a queue Q, maintained by the *Database Manager*, and will be processed according to the order they are received. When each update request is being processed, it may be forwarded to the *Data Allocator*, *Index Generator*, *Index Allocator*, and *Broadcast Cycle Constructor*, to check any possible changes of the broadcast sequence.

7.1 Simple Update

If the update results in no change to any datum's *key*, size, and access probability, or the update does not affect the relative position of any datum in the data set, then the index structure remains the same. Based on the Theorem in [33], we observe that only updates in the left-most or right-most nodes can violate the key range of a local AH-tree. In this case, the server will simply update the datum and the corresponding index node(s) and control table(s), and then broadcast the updated sequence from the beginning of the next broadcast cycle. After the completion of this process, this request will be deleted from the queue Q.

TABLE 5
Parameter Setting for Index Comparisons

Parameter	Default	Range	Meaning
t	10,000	-	Num of data items
r	20,000	-	Num of requests
$m+n$	10	-	Num of channels
k	3	2 to 20	k-ary AH-Tree
l	3	1 to $(L-1)$	cut level
m	3	1 to 9	Num of index channels

7.2 Multipart Update

If the update results in actual change of data key, size, and/or access probability, then at least some part of the index structure and/or broadcast sequence need to be updated. The change in the data set is completed first, then the corresponding index node at the lowest level is checked for potential update, then its parent node, etc., using the bottom-up approach, until we reach a node which need not to be updated. Updating the entire broadcast sequence is usually time consuming, but it is still feasible. We can repeat those steps according to our proposed algorithms to construct a new AH-tree in the worst case, and then allocate index and data correspondingly.

If the client initiates the update requests, for example, updating a certain datum, such request will be sent to the queue Q on the server side as well. The STUBindex approach can be applied to our system, which supports energy efficiency and concurrency control in wireless data broadcast systems with dynamic updates [34]. Initially, the server appends a timestamp to each datum in the data set, and broadcasts the timestamp with the datum. When a datum is updated, the server updates the timestamp as well. The system maintains Timestamp Array (TSA) for each transaction, and a Conflict Array (CFA) to record all conflicts; server appends Update Broadcast (Ucast) as control information into the broadcast sequence.

8 Performance Evaluation

We divide our simulation into three parts. In the first part, we compare two index schemes, which are AH-Tree and B^+ tree. In the second part, we analyze the index allocation methods for different hopping cost and we compare different index allocation methods. In the last part, we compare the performance of our data allocation method with the GREEDY [35] and DPYRAMID [9] under various parameters. Note that all the data and index allocation methods are implemented upon the AH-Tree.

8.1 Performance of AH-Tree *versus* B^+ Tree

We compare the performance of AH-Tree with B^+ tree. Firstly, we model different system conditions by setting 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. The performance of our system is mainly measured by two metrics:

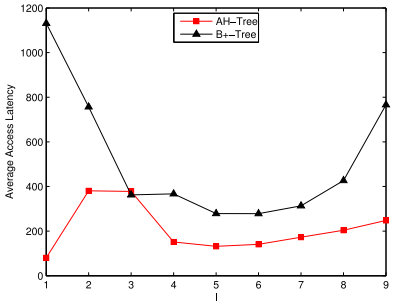


Fig. 4. Change of AAL under zipf distribution w.r.t. l ($k = 3, m = 3$).

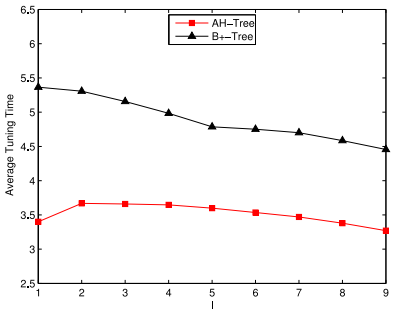


Fig. 5. Change of ATT under zipf distribution w.r.t. l ($k = 3, m = 3$).

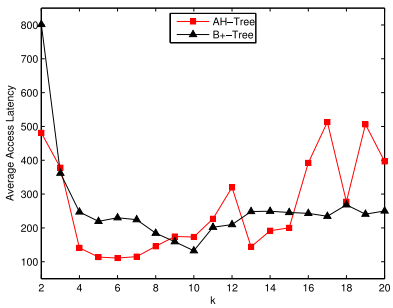


Fig. 6. Change of AAL under zipf distribution w.r.t. k ($l = 3, m = 3$).

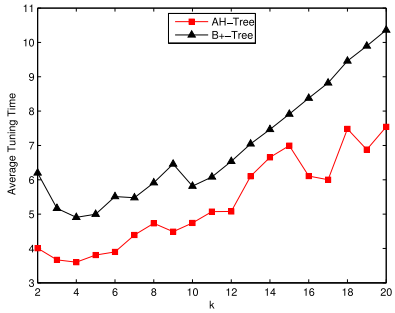


Fig. 7. Change of ATT under zipf distribution w.r.t. k ($l = 3, m = 3$).

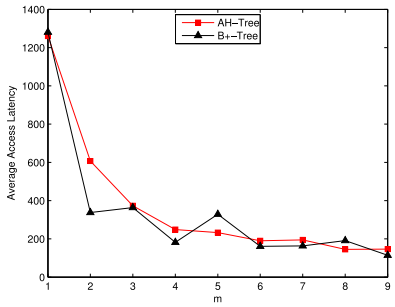


Fig. 8. Change of AAL under zipf distribution w.r.t. m ($k = 3, l = 3$).

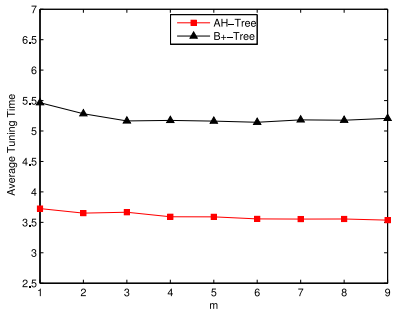


Fig. 9. Change of ATT under zipf distribution w.r.t. m ($k = 3, l = 3$).

Average Access Latency (AAL) and *Average Tuning Time* (ATT), both of which are counted in *logical time units*. For index bucket with 1 head segment, k children pointers and 1 default pointer, it requires $(k + 2) * 0.1$ time units to visit.

Both of the two systems adopt DPYRAMID index allocation algorithm and neglect the effects of hopping times. The results are shown in Figs. 4, 5, 6, 7, 8, and 9. Generally, Figs. 4 , 5, 6, 7, 8, and 9 reveal that AH-Tree performs better than B$^+$-Tree on both access latency and tuning time for data following Zipf distribution. In Fig. 5, the ATT declines with the increase of l since more control indices contribute to faster hopping process. The AAL in Fig. 4 firstly drops then rises again since too many control indices lengthen the index sequences, and thus increase the access latency. Fig. 6 shows that large k has negative effects on AH-Tree's performance and the AAL fluctuates severely with the variation of k. Thus, choosing a proper k for AH-Tree 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. 8. 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. 9. Note that if we deal with uniform distribution, B$^+$-Tree would guarantee a better expected searching step due to its feature.

8.2 Performance of Index Allocation Methods

In this section, we first analyze the effects of the hopping cost for different index allocation methods, and then compare the performance of index allocation methods. We neglect the size of control tables and assume that all the index nodes are of the same size.

Hopping effects. Although index nodes are small, the hopping time may take on the order of milliseconds or more. Thus in logical time, *cost* may equal to as several tens of or even several hundreds of logical time. In this experiment, we set *cost* ranges from 0 to 500. We use DPYRAMID to allocate the indices onto the channels because it does not optimize *cost*, which means that it can reflect the effects of *cost* well. The other parameters are setting as shown in Table 6.

Suppose that AEDCT(x) is the AEDCT (defined in Eq. (1) of Section 4.4.3) for the case $cost = x$, thus AEDCT(20) means the AEDCT when we set the hopping cost to 20. In this experiment we use the difference AEDCT(*cost*)-AEDCT (0) as the measurement.

TABLE 6
Parameter Setting for Allocation Algorithms

Parameter	Range	Default	Meaning
t	500-5,000	3,000	Num of data items
m	5-30	20	Num of data channels
s_{max}	5-50	20	Range of data size
θ	0.1-0.9	0.5	skew of data set
$cost$	0-500	50	Cost of hopping

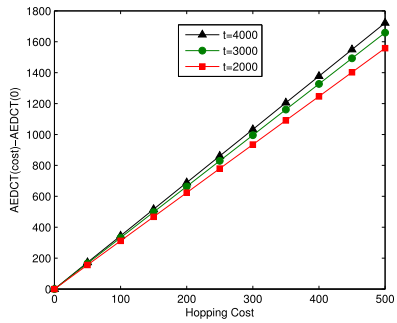


Fig. 10. Effects of hopping cost w.r.t. c ($m = 20, s_{max} = 20, \theta = 0.5$).

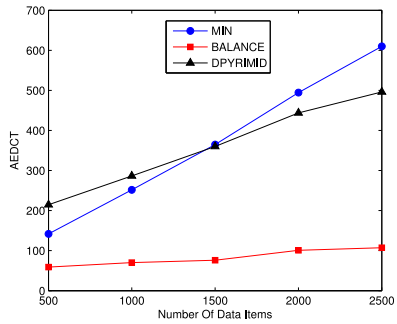


Fig. 11. Index allocation comparasion w.r.t. t ($m = 20$,c$= 50, \theta = 0.5$).

The results are shown in Fig 10. Three cases are shown, which are $t = 2{,}000, 3{,}000, 4{,}000$. For a fixed t, we see that as $cost$ increases, AEDCT($cost$)-AEDCT(0) increases as expected. Note that the increasing speed is almost fixed. The most interesting point is that for different t, for example, when $t = 2{,}000$ and $t = 4{,}000$ (100 percent in difference), the differences of $cost$ are only about 10 percent, which shows that the effects of hopping cost depends little on the number of data items, but mainly on the allocation algorithms.

Index allocation methods. In this part, we compare the performances of three index allocation algorithms. The first two are the minimum hopping (MIN in Algorithm 2) and balanced allocation method (BALANCE in Algorithm 3) described in Section 5. The rest one is DPYRAMID [9]. Note that DPYRAMID can be used in both data allocation and index allocation.

We set the number of data items t range from 500 to 2,500 and other parameters are described in Table 6. Note that we set the hopping cost to be 50, which is not very large. AEDCT defined in Eq. (1) of Section 4.4.3 is the measurement. The results are shown in Fig. 11. It can be seen that BALANCE has much better performance than other two methods. For small data set, hopping cost is the dominating factor. Therefore, MIN and BALANCE have better

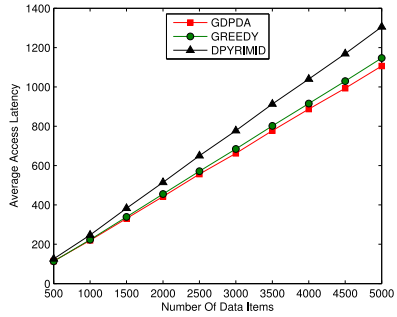


Fig. 12. Change of AAL w.r.t. t ($m = 20, s_{max} = 20, \theta = 0.5$).

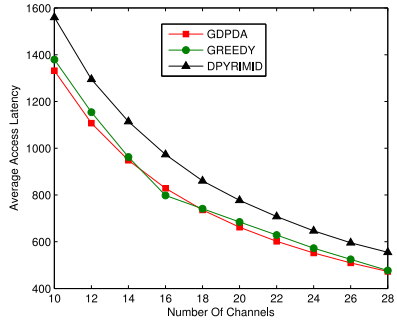


Fig. 13. Change of AAL w.r.t. m ($t = 3{,}000, s_{max} = 20, \theta = 0.5$).

performance than DPYRIMID. However, as the data set grows larger, the effects of hopping becomes smaller. In this case, the performance of MIN becomes poor because of the reduntant broadcast cycle. BALANCE does not suffer this, therefore its performance remains good.

8.3 Performance of Data Allocation Methods

In this section, we compare our data allocation method (GDPDA in Algorithm 5) with DPYRAMID [9] and GREEDY [35]. We list the used parameters in Table 6. We let the frequencies of data follow the Zipf distribution with parameter θ. It becomes increasingly skewed as θ increases. The number of data items is in the range [500, 5,000] with default value 3,000. Performance of the data allocation methods are mainly measured by *AED* defined in Eq. (3), which is counted in *logical time units*.

Effects of t. To show how the number of data items affect the performance of the three methods, we set t range from 500 to 5,000 and fix other parameters as the default values. The results are shown in Fig. 12, where GDPDA performs slightly better than GREEDY and much better than DPYRAMID. As the number of data items increases, the AED of all three methods are increasing stably. However, DPYRAMID increases much faster than GDPDA and GREEDY, so the difference of AED between DPYRAMID and the other methods is getting larger and larger.

Effects of m. We set the number of data channels m range from 10 to 30 and other parameters as default. The results are shown in Fig. 13. In the figure, we can see that as the number of data channels increases, the AED of the three methods decreases, which means that all the three methods can effectively use the data channels. Meanwhile, GDPDA and GREEDY are 15 to 20 percent better than DPYRAMID. It is hard to tell which is better among GDPDA and GREEDY.

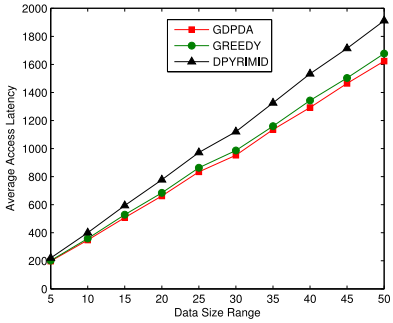


Fig. 14. Change of AAL w.r.t. S ($t = 3{,}000, m = 20, \theta = 0.5$).

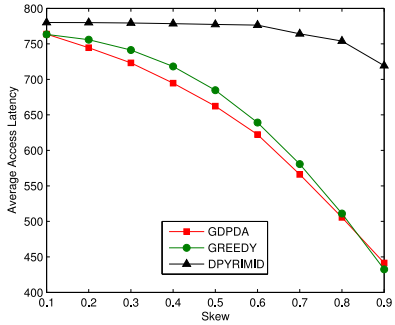


Fig. 15. Change of AAL w.r.t. θ ($t = 3{,}000, s_{max} = 20, m = 20$).

Effects of s_{max}. The size of the data is also a key factor that affects the performance. we set data size, which is an integer in our implementation, distributes uniformly on $[1, s_{max}]$. s_{max} ranges from 5 to 50 and other parameters as default. The results are shown in Fig. 14. Since the expectation of the data size is $0.5s_{max}$, the AED will increase as s_{max} increases, as shown in the figure. Also, the AED of all the three method are increasing stably, which means that different data sizes do not cause trouble for all the methods. It also can be seen that GDPDA is slightly better than GREEDY and about 20 percent better than DPYRAMID. What is more, the difference between them is enlarged.

Effects of θ. Now we discuss how well those methods deal with skew data. As θ increases, the skew of data increases. We set θ range from 0.1 to 0.9 and all other parameters to the default values. The results are shown in Fig. 15. On the one hand, as θ increases, the AED of DPYRAMID remains almost the same, which means that DPYRAMID do not suit well with the distribution of the data. While GDPDA and GREEDY are decreasing rigidly. On the other hand, we see that GDPDA is about 10 percent better than GREEDY when $0.1 < \theta < 0.8$, which is the typical range of θ.

Above all, we can conclude that GDPDA is better than GREEDY especially when the data is skew, and always at least 15 percent better than DPYRAMID.

9 Conclusion

In this paper, we present a global optimization for multi-channel wireless data broadcast system with AH-Tree indexing scheme. We provide three novel designs to improve the performance of the system. Firstly, given that the traditional Hu-Tucker algorithm takes $O(t^k)$ time to construct an arbitrary k-ary AH-Tree, we design a dynamic programming to build a tree in $O(t^2)$. Next, we improve the control table design, which eliminates up to 50 percent redundant entries while keeps the searching efficiency. We also theoretically prove that an optimal alphabetic tree has the minimum average tuning time among all tree based index structures. Finally, we design new index and data allocation algorithms to further reduce the average tuning time and access latency. Our index allocation algorithms consider hopping cost, which has better performance than the traditional ones. Simulation results validate the effectiveness of our algorithms.

In all, we propose a fast distributed AH-Tree index construction and effective index and data allocation algorithms to improve the performance of multi-channel wireless data broadcast system. Our design can be scalable to asynchronous environment with difference data size and arbitrary available channel numbers, which is time and energy efficient to a mass number of mobile clients.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (61202024, 61472252, 61133006), China, 973 project (2014CB340303, 2012CB316201), the Natural Science Foundation of Shanghai (12ZR1445000), Shanghai Educational Development Foundation (Chenguang Grant 12CG09), and Shanghai Pujiang Program 13PJ1403900.

References

[1] Y. Yao, X. Tang, E. Lim, and A. Sun, "An energy-efficient and access latency optimized indexing scheme for wireless data broadcast," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 8, pp. 1111–1124, Aug. 2006.

[2] J. Zhong, Z. Gao, W. Wu, W. Chen, X. Gao, and X. Yue, "High performance energy efficient multi-channel wireless data broadcasting system," in *Proc. Wireless Commun. Netw. Conf.*, 2013, pp. 4346–4351.

[3] T. Imielinski, S. Viswanathan, and B. Badrinath, "Data on air: Organization and access," *IEEE Trans. Knowl. Data Eng.*, vol. 9, no. 3, pp. 353–372, May/Jun. 1997.

[4] Y. Chang, Y. Chang, G. Wu, and S. Wu, "B*-trees: A new representation for non-slicing floorplans," in *Proc. ACM Des. Autom. Conf.*, 2000, pp. 458–463.

[5] S. Jung, B. Lee, and S. Pramanik, "A tree-structured index allocation method with replication over multiple broadcast channels in wireless environments," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 3, pp. 311–325, Mar. 2005.

[6] J. Xu, W. Lee, X. Tang, Q. Gao, and S. Li, "An error-resilient and tunable distributed indexing scheme for wireless data broadcast," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 3, pp. 392–404, Mar. 2006.

[7] L. Adamic and B. Huberman, "Zipf's law and the internet," *Glottometrics*, vol. 3, no. 1, pp. 143–150, 2002.

[8] J. Zhong, Z. Gao, W. Wu, W. Chen, and L. Wang, "Multi-channel energy-efficient hash scheme broadcasting," in *Proc. 21st Int. Conf. Softw. Eng. Data Eng.*, 2012, pp. 115–120.

[9] X. Gao, Y. Shi, J. Zhong, X. Zhang, and W. Wu, "Sambox: A smart asynchronous multi-channel blackbox for B+-tree based data broadcast system under wireless communication environment," in *Proc. Int. Conf. Softw. Eng. Data Eng. 2012*, pp. 1–7.

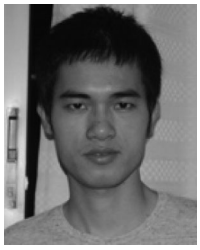
[10] S. Wang and H. Chen, "TMBT: An efficient index allocation method for multi-channel data broadcast," in *Proc. 21st Int. Conf. Adv. Inf. Netw. Appl. Workshops*, 2007, vol. 2, pp. 236–242.

[11] M.-S. Chen, K.-L. Wu, and P. S. Yu, "Optimizing index allocation for sequential data broadcasting in wireless mobile computing," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 1, pp. 161–173, Jan./Feb. 2003.

[12] S. Lo and A. Chen, "Optimal index and data allocation in multiple broadcast channels," in *Proc. IEEE 16th Int. Conf. Data Eng.*, 2000, pp. 293–302.

[13] B. Lee and S. Jung, "An efficient tree-structure index allocation method over multiple broadcast channels in mobile environments," in *Proc. 14th Int. Conf. Database Expert Syst. Appl.*, 2003, pp. 433–443.
[14] W. G. Yee, S. B. Navathe, E. Omiecinski, and C. Jermaine, "Efficient data allocation over multiple channels at broadcast servers," *IEEE Trans. Comput.*, vol. 51, no. 10, pp. 1231–1236, Oct. 2002.
[15] E. Ardizzoni, A. Bertossi, M. Pinotti, S. Ramaprasad, R. Rizzi, and M. Shashanka, "Optimal skewed data allocation on multiple channels with flat broadcast per channel," *IEEE Trans. Comput.*, vol. 54, no. 5, pp. 558–572, May 2005.
[16] S. Anticaglia, F. Barsi, A. A. Bertossi, L. Iamele, and M. Pinotti, "Efficient heuristics for data broadcasting on multiple channels," *Wireless Netw.*, vol. 14, no. 2, pp. 219–231, 2008.
[17] T. Huand A. Tucker, "Optimal computer search trees and variable-length alphabetical codes," *SIAM J. Appl. Math.*, vol. 21, no. 4, pp. 514–532, 1971.
[18] J. Zhong, W. Wu, X. Gao, Y. Shi, and X. Yue, "Evaluation and comparison of various indexing schemes in single-channel broadcast communication environment," *Knowl. Inf. Syst.*, vol. 40, no. 2, pp. 375–409, 2014.
[19] N. Shivakumar and S. Venkatasubramanian, "Efficient indexing for broadcast based wireless systems," *ACM J. Mobile Netw. Appl.*, vol. 1, no. 4, pp. 433–446, 1996.
[20] J. Zhong, W. Wu, Y. Shi, and X. Gao, "Energy-efficient tree-based indexing schemes for information retrieval in wireless data broadcast," in *Proc. 16th Int. Conf. Database Syst. Adv. Appl.*, 2011, pp. 335–351.
[21] W. Yee and S. Navathe, "Efficient data access to multi-channel broadcast programs," in *Proc. ACM Conf. Inf. Knowl. Manage.*, 2003, pp. 153–160.
[22] X. Lu, X. Gao, and Y. Yang, "SETMES: A scalable and efficient tree-based mechanical scheme for multi-channel wireless data broadcast," in *Proc. ACM 7th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2013, pp. 176–192.
[23] C. Kenyon and N. Schabanel, "The data broadcast problem with non-uniform transmission times," in *Proc. 10th Annu. ACM-SIAM Symp. Discrete Algorithm*, 1999, pp. 547–556.
[24] Z. Lu, W. Wu, and B. Fu, "Optimal data retrieval scheduling in the multichannel wireless broadcast environments," *IEEE Trans. Comput.*, vol. 62, no. 12, pp. 2427–2439, Dec. 2013.
[25] X. Gao, Z. Lu, W. Wu, and B. Fu, "Algebraic data retrieval algorithms for multi-channel wireless data broadcast," *Theoretical Comput. Sci.*, vol. 497, pp. 123–130, 2013.
[26] Y. Shi, X. Gao, J. Zhong, and W. Wu, "Efficient parallel data retrieval protocols with mimo antennae for data broadcast in 4g wireless communications," in *Proc. 21st Int. Conf. Database Expert Syst. Appl.*, 2010, pp. 80–95.
[27] Z. Lu, Y. Shi, W. Wu, and B. Fu, "Efficient data retrieval scheduling for multi-channel wireless data broadcast," in *Proc. IEEE INFOCOM*, 2012, pp. 891–899.
[28] Z. Lu, Y. Shi, W. Wu, and B. Fu, "Data retrieval scheduling for multi-item requests in multi-channel wireless broadcast environments," *IEEE Trans. Mobile Comput.*, vol. 13, no. 4, pp. 752–765, Apr. 2014.
[29] T. Hu, "A new proof of the TC algorithm," *SIAM J. Appl. Math.*, vol. 25, no. 1, pp. 83–94, 1973.
[30] Y. Yang, X. Gao, X. Lu, J. Zhong, and G. Chen, "Distributed Ah-tree based index technology for multi-channel wireless data broadcast," in *DASFAA 2013*, pp. 176–192.
[31] T. C. Hu, D. J. Kleitman, and J. K. Tamaki, "Binary trees optimum under various criteria," *SIAM J. Appl. Math.*, vol. 37, no. 2, pp. 246–256, 1979.
[32] A. Itai, "Optimal alphabetic trees," *SIAM J. Comput.*, vol. 5, no. 1, pp. 9–18, 1976.
[33] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu, "Efficient B-tree based indexing for cloud data processing," *Proc. VLDB Endowment*, vol. 3, nos. 1/2, pp. 1207–1218, 2010.
[34] Y. Huang and Y.-H. Lee, "An efficient indexing method for wireless data broadcast with dynamic updates," in *Proc. IEEE Int. Conf. Commun., Circuits Syst. West Sino Expo.*, 2002, vol. 1, pp. 358–362.
[35] W. Yee, "Efficient data allocation for broadcast disk arrays," Georgia Inst. Technol., Atlanta, GA, USA, Tech. Rep. GIT-CC-02-20, 2002.

Xiaofeng Gao received the BS degree in information and computational science from Nankai University, China, in 2004, the MS degree in operations research and control theory from Tsinghua University, China, in 2006, and the PhD degree in computer science from The University of Texas at Dallas, in 2010. She is currently an associate professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. Her research interests include data engineering, wireless communications, and combinatorial optimizations. She has published more than 80 peer-reviewed papers in the related area, and has served as the PCs and peer reviewers for a number of international conferences and journals.



Yongtian Yang received the BS degree in computer science from Southeast University, China, in 2012, and the MS degree from Department of Computer Science and Engineering at Shanghai Jiao Tong University, China, in 2015. His research interests include data management and engineering, especially the indexing design and data retrieval protocols.

Guihai Chen received the BS degree from Nanjing University in 1984, the ME degree from Southeast University in 1987, and the PhD degree from the University of Hong Kong in 1997. He is a distinguished professor of Shanghai Jiao Tong University, China. He had been invited as a visiting professor by many universities including Kyushu Inst. of Tech., Japan, University of Queensland, Australia, and Wayne State Univ. He has a wide range of research interests with focus on sensor networks, peer-to-peer computing, high-performance computer architecture and combinatorics. He has published more than 200 peer-reviewed papers, and more than 120 of them are in well-archived international journals such as *TPDS*, *TMC*, *TON*, *TC*, *TWC*, *JPDC*, and also in well-known conference proceedings such as HPCA, MOBIHOC, INFOCOM, ICNP, ICPP, IPDPS, and ICDCS.

Xin Lu received the BS degree in software engineering from Shanghai Jiao Tong University in 2013. He is currently working toward the PhD degree in the Department of Computer Science at Columbia University. His research interests include operating systems, network systems, and data management. He completed this work when he was a student at Shanghai Jiao Tong University.

Jiaofei Zhong received the PhD degree in computer science in 2012 and the MS degree in 2010, both from the University of Texas at Dallas. She is currently an assistant professor in the Department of Mathematics and Computer Science, California State University, East Bay. She has served as a peer reviewer for a number of international conferences and journals, and has been the Publicity chair, Financial chair, and OCS co-chair in the organizing committees of several international conferences. Her research interests are in the areas of data engineering and information management, especially in Wireless Communication Environment.