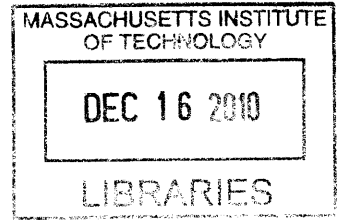


**A Hybrid Data Structure for Dense Keys in  
In-Memory Database Systems**

by

José Alberto Muñiz Navarro



Submitted to the Department of Electrical Engineering and Computer  
Science

in partial fulfillment of the requirements for the degree of

**ARCHIVES**

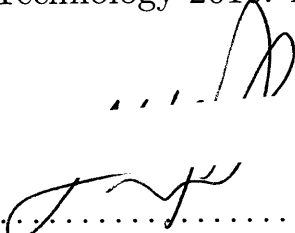
Master of Engineering in Electrical Engineering and Computer Science



at the

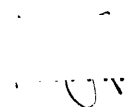
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010

© Massachusetts Institute of Technology 2010. All rights reserved.

Author .....   
Department of Electrical Engineering and Computer Science  
August 20, 2010

Certified by .....  .....   
Samuel Madden  
Associate Professor  
Thesis Supervisor

Accepted by .....  .....  
Christopher J. Terman  
Chairman, Department Committee on Graduate Theses



# A Hybrid Data Structure for Dense Keys in In-Memory Database Systems

by

José Alberto Muñiz Navarro

Submitted to the Department of Electrical Engineering and Computer Science  
on August 20, 2010, in partial fulfillment of the  
requirements for the degree of  
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

This thesis presents a data structure which performs well for in-memory indexing of keys that are unevenly distributed into clusters with a high density of keys. This pattern is prevalent, for example, in systems that use tables with keys where one field is auto-incremented. These types of tables are widely used.

The proposed data structure consists of a B+ Tree with intervals as keys, and arrays as values. Each array holds a cluster of values, while the clusters themselves are managed by the B+ Tree for space and cache efficiency. Using the H-Tree as an in-memory indexing structure for an implementation of the TPC-C benchmark sped up the transaction processing time by up to 50% compared to an implementation based on B+ Trees, and showed even more dramatic performance gains in the presence of few and large clusters of data.

Thesis Supervisor: Samuel Madden  
Title: Associate Professor



## Acknowledgments

First and foremost, I wish to thank my advisor, Prof. Madden, for all his advice and support on this project.

I'd also like to thank Evan Jones for his continued support and guidance during the elaboration of this thesis. Evan was always available to help me, and his advice and comments were essential to the completion of this thesis.

Thanks to my parents, Sócrates and Ruth, for their constant encouragement and inspiration. Their hard work and intelligence have set an example that I'll always strive to follow in my life.

Finally, many thanks to my friends and my girlfriend, Natalie, for their patience, understanding and invaluable feedback throughout the year.

The inspiration and insights I received from all of you go well beyond what is reflected in these pages.



# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
1.1	Index Structures in Main Memory Database Systems . . . . .	13
1.2	Indices with densely-populated keys . . . . .	14
1.3	Practical applications . . . . .	16
<b>2</b>	<b>Previous Work</b>	<b>19</b>
2.1	Multi-level memory model . . . . .	19
2.2	B+ Trees, AVL Trees, Arrays, and Hash Tables as main memory index structures . . . . .	20
2.3	T-Trees . . . . .	27
2.4	Cache Conscious Trees . . . . .	28
2.5	Summary . . . . .	29
<b>3</b>	<b>The H-Tree Data Structure</b>	<b>31</b>
3.1	Overview . . . . .	31
3.2	The ArrayObj object . . . . .	32
3.3	Query . . . . .	34
3.4	Insertion . . . . .	35
3.5	Deletion . . . . .	38
3.6	Analysis . . . . .	39
3.7	Algorithm Parameters . . . . .	40
3.7.1	$\rho_{min}$ . . . . .	40
3.7.2	$s_0$ . . . . .	41

3.7.3	<code>split_array()</code> . . . . .	41
3.8	Implementation details . . . . .	42
<b>4</b>	<b>Benchmarks</b>	<b>45</b>
4.1	Insertion benchmarks . . . . .	46
4.1.1	General Setup . . . . .	46
4.1.2	Performance under different loads . . . . .	46
4.1.3	Varying across key dimensions . . . . .	53
4.2	Deletion benchmarks . . . . .	60
4.3	The algorithm parameters . . . . .	60
4.3.1	$\rho_{min}$ . . . . .	60
4.4	OLTP Workload: The TPC-C Benchmark . . . . .	61
<b>5</b>	<b>Conclusion</b>	<b>65</b>
<b>A</b>	<b>Benchmark architecture</b>	<b>67</b>
A.1	Single point benchmarks . . . . .	67
A.2	Multi-point benchmarks . . . . .	69



# List of Figures

1-1	A sample small hybrid tree representing the following clusters: [1 to 20], [100 to 125], [750 to 790], [1300 to 1350], [1400 to 1450]. In this example, keys represent the starting value of some <code>ArrayObj</code> instance $a_i$ which holds the values for that interval. . . . .	15
1-2	Histogram of zip code cluster sizes . . . . .	17
1-3	Histogram of cluster sizes on Wikipedia's page table . . . . .	18
2-1	An array index data structure . . . . .	21
2-2	An AVL tree . . . . .	22
2-3	A B+ Tree . . . . .	23
2-4	Cost model comparing different in-memory indices using two levels of memory . . . . .	26
2-5	A T-Tree . . . . .	27
3-1	Components of <code>ArrayObj</code> instance $a_i$ . . . . .	33
3-2	Query of key 110. The value in the array cell marked with red is returned only if the corresponding value <code>valid[110 - 100]</code> is set to <code>true</code> . . . . .	34
3-3	Algorithm for lookup of key $k$ from H-Tree $T$ . . . . .	34
3-4	Sample insertion of key 19. The key fits perfectly in array $a_1$ . The blocks shaded red are read by the H-Tree algorithm. . . . .	36

3-5	Sample insertion of key 30. The key does not fit its closest array $a_1$ , but doubling it would fit it and preserve the density property. We therefore reset array $a_1$ to contain values 1 through 40, twice its original value, and reduce insertion to the first case. . . . .	37
3-6	Sample insertion of key 3000. The key does not fit its closest array $a_5$ , and doubling it would still not make it fit or would violate the density property. We therefore create a new array $a_6$ starting on this key and with size $s_0$ . . . . .	38
3-7	Algorithm for insertion of key $k$ with associated value $v$ into H-Tree $T$ .	39
3-8	Two different array splitting policies. The position marked with $k$ is the key that was just deleted. The left diagram splits the array into two subarrays from the position of the largest sequence of zeroes. The right diagram shows the split from the position of the largest sequence of zeros containing $k$ . . . . .	40
3-9	Algorithm for deletion of key $k$ from H-Tree $T$ . . . . .	41
4-1	Schematic representation of keys in a clustered configuration . . . . .	46
4-2	Keys inserted in order . . . . .	47
4-3	Benchmark for strictly sequential insertions and few (4) large clusters	48
4-4	Benchmark for strictly sequential insertions and several (400) small clusters . . . . .	49
4-5	Keys inserted in random cluster order, but sequential intra-cluster order	49
4-6	Benchmark with keys inserted with random cluster selection and few (4) large clusters . . . . .	50
4-7	Benchmark with keys inserted with random cluster selection and several (400) small clusters . . . . .	51
4-8	Keys inserted in random cluster order, but sequential intra-cluster order	52
4-9	Benchmark with keys inserted with window algorithm. The window size is 15. . . . .	53

4-10	Benchmark with keys inserted with window algorithm. The window size is 1,500. . . . .	54
4-11	Benchmark with keys inserted with window algorithm. The window size is 15,000. . . . .	55
4-12	Density graph with almost sequential insertion and large number of keys ( $8 \times 10^5$ ). . . . .	56
4-13	Density graph with almost sequential insertion and small number of keys (80). . . . .	57
4-14	Density graph with almost random insertion and large number of keys ( $8 \times 10^5$ ). . . . .	58
4-15	Density graph with almost random insertion and small number of keys (80 . . . . .	59
4-16	Number of arrays created as a function of the clustering factor in an array for a random insertion . . . . .	61
4-17	Result of TPC-C Benchmark with modified Items table. . . . .	63
4-18	Result of TPC-C Benchmark with all tables indexed by H-Trees and all tables indexed by B+Trees . . . . .	63



# Chapter 1

## Introduction

### 1.1 Index Structures in Main Memory Database Systems

Traditional database systems store most of their data on hard disk drives. An in-memory buffer stores pages from the hard disk to provide faster access to information that is accessed frequently [9]. However, the cost and availability of main memory has reached a point where it is feasible to hold entire databases in it. According to [20], the majority of OLTP databases are at most 1 TB in size. In a few years, servers that hold this amount of memory should not be atypical.

For this reason, database systems that store their entire content in memory are of rising importance. The data for some applications can feasibly be held by an in-memory database system, such as enterprise applications that need to hold a few thousand bytes per employee or customer, where the amount of information to be held is small and grows more slowly than the rate at which memory capacities improve [10]. Several commercial and open-source database products already mainly store data in-memory. Some examples include H-Store ([12], [20]), Oracle's TimesTen, McObject's eXtremeDB, and MonetDB.

Main memory is significantly faster than hard disks, and also behaves differently under random accesses. In particular, whereas hard disks have large seek times –

times used to physically move the read/write heads to the correct position in non-sequential accesses, main memory incurs in no such delay. For that reason, the block arrangement of B-Trees, for example, may be less attractive in main memory [10]. Section 2.1 describes a more complete model of the behavior of main memory.

There have been some previous attempts to develop data structures that are based on trees and hash tables ([17], [18], [14], [10]). These structures are explained briefly in Sections 2.2, 2.3, 2.4, along with a summary of their performance, cache utilization, and storage efficiency, using the results from these papers. In this thesis, we present a data structure based on arrays and B+ Trees which performs better than other trees whenever the inserted keys consist of a series of regions of contiguous keys. This workload is explained in more detail in Section 1.2.

## 1.2 Indices with densely-populated keys

A particularly interesting workload is one where key values are grouped into several densely-packed clusters. These clusters are sets of contiguous or almost contiguous keys with perhaps large gaps between different clusters. This pattern emerges, for example, when an application assigns a key automatically and incrementally, and then performs batch deletions of intervals of keys. The pattern may also arise when batches of keys are assigned to different parties so that they perform sequential insertions independently.

One way in which we may store a table with keys that can be grouped in large clusters is an array of pointers, where the  $i$ th position is a pointer to the structure associated with key  $i$ . The advantage of this storage is that structures associated with a given key can be found in constant time. However, the array cells corresponding to inter-cluster gaps would be empty, resulting in very large empty spaces whenever there are few, small clusters separated by large gaps. Moreover, since the data structure must be flexible enough to allow arbitrary insertions and deletions, an array implementation could potentially require a very large number of expensive resizing operations.

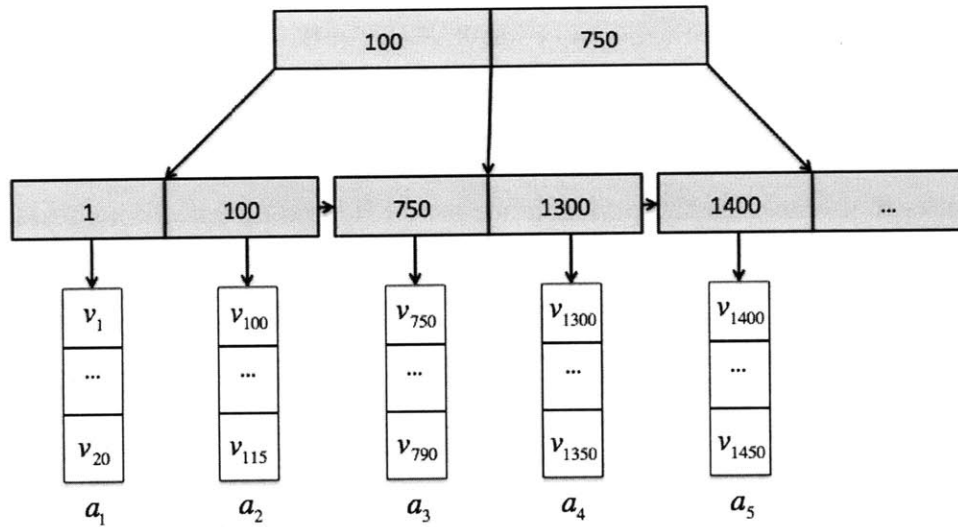


Figure 1-1: A sample small hybrid tree representing the following clusters: [1 to 20], [100 to 125], [750 to 790], [1300 to 1350], [1400 to 1450]. In this example, keys represent the starting value of some `ArrayObj` instance  $a_i$  which holds the values for that interval.

Other data structures, such as tree structures or hash tables do not necessarily have this wasted storage problem, but they do not exploit the clustering structure of the keys, providing worse performance than arrays. Chapter 2 provides a cost-model for previous data structures considering two levels of main memory (cache and RAM) and compares and contrasts the differences in performance and storage between them.

Chapter 3 describes the *H-Tree*, a tree-like data structure for in-memory storage of keys that follow the densely-populated cluster structure described in Section 1.2. The H-Tree exploits the performance advantage of arrays, but minimizes the amount of empty spaces. At a high level, the H-Tree is simply a tree of arrays. The arrays hold the values corresponding to contiguous keys. In order to insert or access the value associated with a key, we first find the node that contains the array that holds the key, and then perform the operations in that array. Figure 1-1 shows a sample H-Tree.

The H-Tree's performance is bounded between the performance of an array and the performance of a tree. Whenever the data is inserted sequentially with few, large

clusters, the H-Tree performance is comparable to an array. On the other hand, when the data is inserted in random order, and the data set itself consists of several very small clusters, then the H-Tree performance is comparable to the performance of a B+Tree plus a small constant overhead. Chapter 4 shows the results of several benchmarks that showcase the performance of the H-Tree under various tests.

### 1.3 Practical applications

An example of a data set that shows a clustering behavior is a table keyed by zip code. Zip codes uniquely identify regions of the US, and therefore a table with all zip codes may be used by an application that wishes to identify a customer's approximate address based only on their zip code.

Zip codes have an internal structure, whereby the most significant digits represent a group of states, the next digits identify cities, and progressively less significant digits demarcate smaller regions. However, not all the possible zip codes corresponding to a particular state and city are used, resulting in a series of clusters of contiguous zip codes. Figure 1-2 shows a histogram of the cluster sizes. The average size of a cluster of contiguous keys is 15.7, with the most popular cities containing clusters of hundreds of contiguous zip codes. For example, the region of zip codes starting with form  $100xx$  consist of 50 contiguous keys representing regions in the New York City borough of Manhattan.

There are 29,470 different zip codes [1], ranging from zip codes 01001 and 99950. If we represent a pointer to the row associated with a zip code in memory as a 4 byte integer, then storing the complete table of zip code pointers requires  $29,470 \times 4$  bytes = 115 kb. An array storing this data in the way described above would require a size of roughly 100,000 elements, which would take 390 kb, three times more than the required size. Other tree-based data structures provide better storage characteristics, but at the expense of worse access performance.

The underlying database that persists the articles behind the online encyclopedia *Wikipedia* also shows clustering behavior. Each page on Wikipedia has an entry on



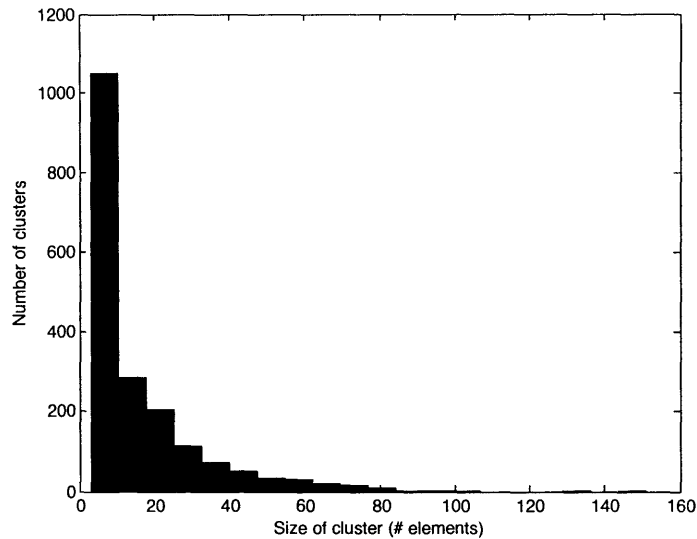


Figure 1-2: Histogram of zip code cluster sizes

the page table, containing meta-data such as the page title, the type of page, etc [4]. The key for this table is `page_id`, assigned auto-incrementally. Since some pages are deleted, the resulting distribution of page ids has a clustered pattern. Figure 1-3 shows a histogram of the cluster sizes. The average size of a cluster of contiguous keys is 134.59 elements.

There were 1,163,319 unique page ids as of the database dump from April 2010, ranging from ids 1 to 1,499,144. Representing a page id as a 4 byte integer, storing the complete table of page ids requires 4.4 Mb, whereas an array as described above would require 5.71 MB – about 30% more space than the necessary space.

A third system that exhibits a clustered key set distribution is the TPC-C benchmark. The TPC-C benchmark is an online transaction processing (OLTP) workload. It consists of a mixture of insert and update-intensive transactions that simulate the typical execution of a wholesale company system that processes orders and manages stock. This company operates with a variable number of warehouses, specified at the start of the benchmark. Each warehouse, in turn, covers a fixed number of districts, each of which in turn covers a fixed number of customers. A database with nine different tables stores the information regarding order and stock information. A number

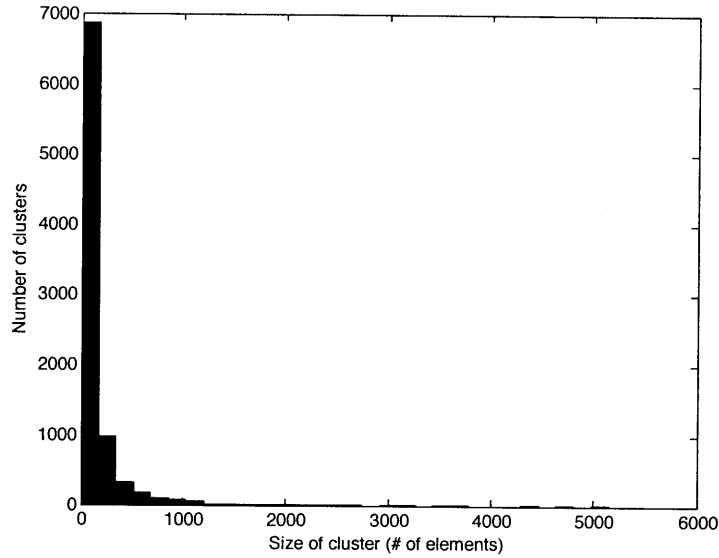


Figure 1-3: Histogram of cluster sizes on Wikipedia's `page` table

of realistic transactions are then run on these tables, such as attempting to fulfill a new order of several items from a warehouse. [2] provides a good general overview of the TPC-C benchmark, while [3] describes the full specification for the benchmark.

Several tables within the benchmark contain sequential keys, such as the `Stock` table, which stores information about the stock on each warehouse, and the table of `Items`, which contains information about items such as its price and name. Section 4.4 explains the behavior of several data structures, including the newly-developed H-Tree [15]. The *New Order Transaction* in the benchmark inserts a sequence of orders in the `Orders` relation with increasing order ids.

# Chapter 2

## Previous Work

### 2.1 Multi-level memory model

Typically, access time for data in main memory is assumed to be constant. However, as main memory becomes the principal access method in some database systems, it is useful to expand the single-access time model to account for the existence of heterogeneous access times in modern day memory system architectures [19].

In particular, modern systems include *caches* in small fast-speed memories. Caches are mappings of ranges of addresses to the bytes in memory corresponding to those ranges. These bytes are referred to as a *line*. The cache line is thus the unit of transfer between main memory and cache.

Since these caches are expensive, they are small compared to the total size of RAM. Modern architectures have at least two levels of cache with varying sizes and speeds. On typical Intel Core2 systems, an L1 cache miss creates a delay of 10 CPU operations (or about 4 ns on a 2.27 GHz CPU), while an L2 cache miss creates a delay of 200 CPU cycles (or about 90 ns on a 2.27 GHz CPU) [7], [11]. Virtual to physical address translation can add additional latency to this process.

When information cannot be found in cache, it is then retrieved from the address translator or main memory, respectively. This new information, along with possibly more, is then inserted into the cache in accordance with the *cache fetch algorithm*. For example, nearby addresses may be cached in addition to the requested address

in expectation that they are likely to be used in the future. If the cache was already full, a pair is deleted according to the *cache replacement policy* such as FIFO, LRU, or random eviction.

As implied above, the existence of cache is justified by the existence of temporal and spatial *locality of reference*. Temporal locality of reference assumes that the probability that a given address is used at a given time is larger when this address has been accessed before than when it has not. This is true, for example, in the case of a loop, when the same instructions and variables are read several times. Spatial locality of reference assumes that the probability that a given address issued is larger when addresses nearby have been accessed. This is true, for example, in the case where instructions are fetched sequentially or when structures that are used at a given time are allocated in contiguous memory locations and accessed regularly.

## 2.2 B+ Trees, AVL Trees, Arrays, and Hash Tables as main memory index structures

Several traditional data structures can be used as in-memory indices. [5], [6], [14], and [13] discuss the performance in terms of characteristics and access time of each of the structures.

**Arrays** Arrays are the simplest data structures for storing values. Here, the  $i$ th position stores the value with key  $i$ . This provides the advantage that keys do not need to be explicitly stored, thus saving space. Plus, when an array is situated in memory, the constant time in random access allows a fast insertion and retrieval time, since any value can be obtained and modified after one offset calculation and one memory access, in constant time.

Figure 2-1 shows a diagram of an array index data structure.

The usefulness of the cache in this case is dictated by the access pattern of the application: if it uses values that are close to each other, a cache that uses locality of reference will be effective in minimizing access costs.

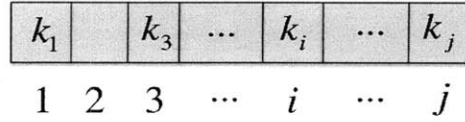


Figure 2-1: An array index data structure

The memory footprint of this structure is as large as the maximum value of the key  $k_j$  multiplied by the size of each value. This is optimal only if the structure holds the keys  $1\dots j$ . For this reason, sparsely populated sets of values are not suitable for arrays.

Using a similar argument as [13], an estimate for the access time in a two-level memory model is given for random access by:

$$C_{array} = Z \times \left(1 - \frac{|M||P|}{L|R|}\right) \quad (2.1)$$

Here,

- $Z$  is the relative cost of memory access with respect to cache access
- $|M|$  is the size of the cache (number of cache lines).
- $|R|$  is the number of tuples in the relation
- $|P|$  is the size of a cache line

The analysis assumes a random eviction policy, random querying and a fully associative cache. Also, we do not store the full tuple into the array, but rather store a pointer to a memory address where we can find it. In the coming data structures, we store the full tuple data inside of the structure. An alternative to this behavior would be to store only the pointer to the structure with all the tuple data.

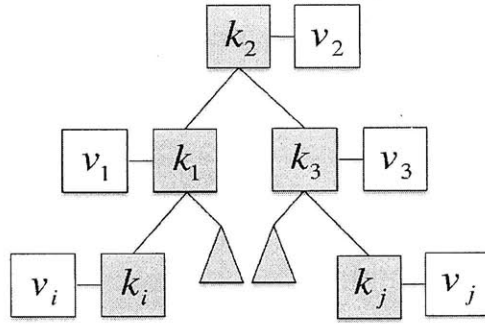


Figure 2-2: An AVL tree

**AVL Trees** AVL Trees are a type of self balancing Binary Search Tree (BST). In order to maintain their balance, AVL Trees perform rotations whenever an insertion or deletion causes any of a node to stop satisfying the condition that the height of its children should differ by at most one. Figure 2-2 shows a diagram of an AVL Tree.

Insertion and querying occur in  $O(\ln n)$  time, where  $n$  is the number of nodes. As developed in [13], a tight estimate for the access time in a two-level memory model is given by:

$$C_{AVL} = Z \times C \times \left(1 - \frac{|M|}{S}\right) + Y \times C \quad (2.2)$$

Here,

- $Z$  is the relative cost of reading to memory with respect to an AVL node comparison.
- $C$  is the number of comparisons needed to find a tuple, where  $C = \log_2 ||R||$  approximately.
- $M$  is the size of the cache (number of cache lines).

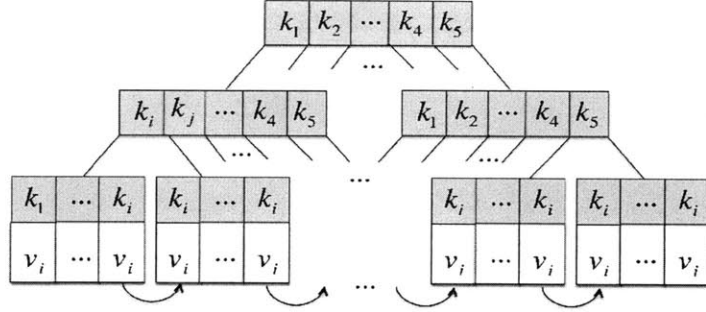


Figure 2-3: A B+ Tree

- $S$  is the amount of space in bytes occupied by the tree, where  $S = ||R|| \times (L + 2p)$ , with pointers of size  $p$  bytes and  $L$  bytes per tuple and  $||R||$  is the number of tuples in the relation,
- $Y < 1$  is some constant that states the relative cost of an AVL comparison with respect to a B+ Tree comparison.  $Y < 1$  because AVL requires less comparisons in a node, whereas B+ Trees require binary search on the node to locate the key and the next link to follow.

As stated in [6], AVL Trees have a low ratio of data to pointers, since each node holds only one value and has two pointers, so cache utilization is relatively ineffective.

**B+Trees** Similar to AVL trees, B+Trees are self balancing trees. Each node stores a sequence of keys  $k_1, k_2, \dots, k_b$ , and between each node  $k_i, k_j$  a pointer to a subtree that have all key values  $k_m \in [k_i, k_j]$ . Each node must have a number of keys and children in some predefined interval at any given point, and when insertions or deletions would violate any of these properties, new nodes are created and merged, and the tree rebalanced in order to continue satisfying the previous properties. Figure 2-3 shows a diagram of a B+Tree.

As before, insertion and querying occur in  $O(\ln n)$  time, where  $n$  is the number of nodes. However, the cost for key access in this case is in given in [6] by:

$$C_{BTree} = Z \times (\text{height} + 1) \left(1 - \frac{|M|}{S'}\right) + C' \quad (2.3)$$

Now,

- $Z$  is the relative cost of reading to memory with respect to an AVL node comparison,
- $|M|$  is the size of the cache (number of cache lines),
- $S'$  is the amount of space in bytes occupied by the tree, where  $S = D \times \frac{A}{A-1}$ ,
- $D$  is the number of leaves, estimated as  $\|R\| \frac{L}{69P}$ ,
- $A$  is the fan-out, given by  $69 \frac{P}{K+p}$ ,
- $C'$  is the number of comparisons needed to find a tuple. For simplicity, assume  $C = C'$ .

Using these models, the prediction is that AVL Trees will outperform trees when cache misses are not a factor (when the cache size is very large with respect to the amount of data in the tree). When this is not the case, then using Equations 2.2 and 2.3 we obtain that B-Trees will be preferred dependent on the values of  $Z$ ,  $H$ , and  $Y = \frac{\text{height}+1}{\log_2 \|R\|}$ . [6] concludes that for reasonable residency factors, close to 80 or 90 per cent of the table must fit in the cache for AVL to outperform B+ Trees.

However, [14] results seemingly contradict these theoretical expectations, by showing a benchmark of insertions and deletions on B+ Trees and AVL Trees and reporting that B+ Trees are slightly faster than AVL Trees. In fact, Lehman and Carey explain this by pointing out a very low value of  $Y$ , the relative cost of an AVL node comparison to a B+ Tree comparison. They also note that the AVL Tree, however, requires more regular allocation calls and constant tree rebalances, which partially offsets the gain in performance. The effects



Variable	Description	Formula
$\ R\ $	Number of tuples in the relation	$n$
$p$	Pointer size	4 bytes
$Z$	Ratio of memory access cost to register access	200
$ M $	Size of the L2 cache (number of lines)	$\frac{2\text{MB}}{64\text{bytes}}$
$ P $	Size of cache L2 line	64 bytes
$L$	Bytes per tuple	128 bytes
$Y$	Ratio of AVL to B+Tree node comparison time	0.8
$C$	Comparisons to find a tuple in AVL Tree	$\log_2 n$
$C'$	Comparisons to find a tuple in B+Tree	$\log_2 n$
$D$	Number of leaves in B+Tree	$0.03n$
$A$	Fan out of B+Tree	552
height	Height of B+Tree	$-0.55 + 0.11 \log_2 n$

Table 2.1: Description of **variables** along with some typical values, as shown in [7]. We assume a key size of 32 bits (4 bytes) and a tuple size of 128 bytes. Cache values for an Intel Core2 processor.

of caching are not discussed, and the results seem to imply (when evaluated through the previous theoretical model) that the results were evaluated in a system with a very small cache.

**Hash Tables** Hash Tables are mappings from keys to values. In order to insert and delete elements from the hash table, a function  $f : K \rightarrow Z$  is required, where  $K$  is the set of all possible keys. Extendible Hashing [8] and Linear Hashing [16] are two well-known techniques for growing the size of hash-functions dynamically as elements are inserted and deleted.

Random access queries, like arrays, allow for constant time insertions and deletions. However, there are several disadvantages associated with using a hash table as an index data structure:

1. Since  $f$  is not monotonic, elements are not stored in order, and therefore any range query requires a linear scan of the table.
2. Since contiguous keys are not stored close to each other, there are no cache benefits of looking-up or storing contiguous keys.

Data Structure	Random access cost
Array	$Z \times \left(1 - \frac{ M  P }{L R }\right)$
AVL Tree	$Z \times C \times \left(1 - \frac{ M }{  R   \times (L+2p)}\right) + Y \times C$
B+Tree	$Z \times (\text{height} + 1) \left(1 - \frac{ M }{D \times \frac{A}{A-1}}\right) + C'$

Table 2.2: Table for **Costs** for random access on different data structures according to a two-level memory hierarchy model. The value of the literals is explained in Table 2.1

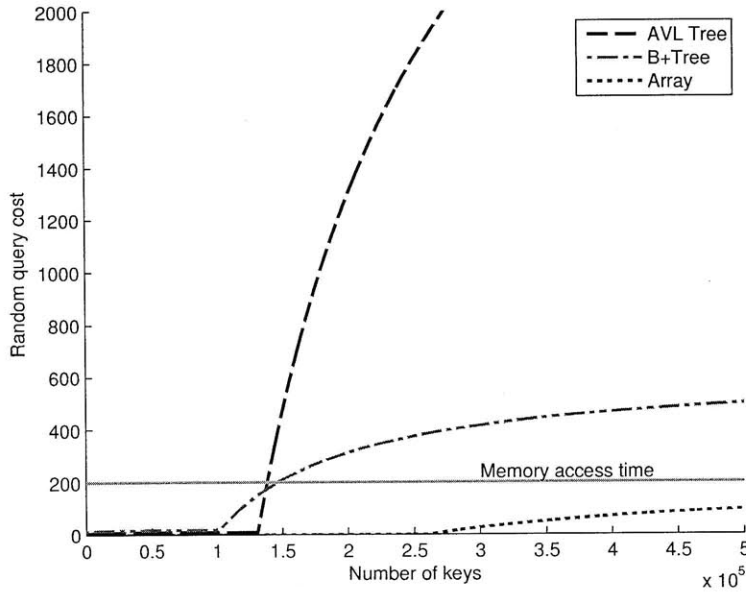


Figure 2-4: Cost model comparing different in-memory indices using two levels of memory

Table 2.2 summarizes the cost model developed in this section. Figure 2-4 shows a graph of the random query cost varying the number of keys in the data structure, and assuming some reasonable values for memory access time and cache sizes as documented in [7]. Table 2.1 summarizes these variable values. We observe the array outperforming every other structure in this model. Similarly, whenever the whole data structure fits in cache, AVL Trees outperform B+Trees, since it performs roughly the same amount of operations, but the operations on AVL Tree nodes are expected to be cheaper than the binary search within a large B+Tree node. However, the B+Tree exhibits much better cache performance in this model, resulting in a

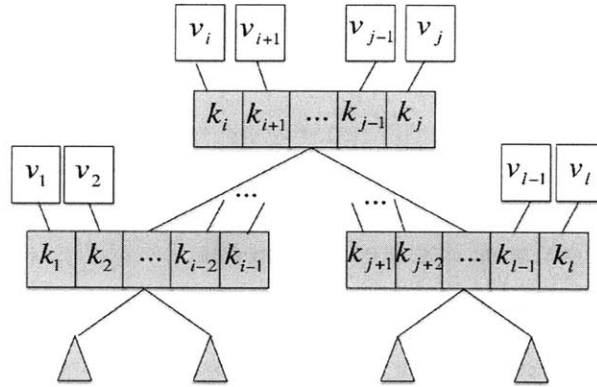


Figure 2-5: A T-Tree

preferable structure whenever the number of keys is large with respect to the size of the cache.

## 2.3 T-Trees

T-Trees are self-balancing trees that are hybrids between B+ Trees and AVL Trees that exploit the fast comparison times of AVL Trees along with the low pointer density of the B+ Trees, in an attempt to provide a structure that is as space efficient as a B+Tree and as fast as an AVL Tree. A T-Tree node represents an interval of keys  $k_i, \dots, k_j$ . Each node has only two pointers: the left pointer points to a subtree where all keys  $k$  satisfy  $k \leq k_i$ ; the right pointer points to a subtree where all keys  $k$  satisfy  $k_j \leq k$ . All nodes have an equal maximum capacity, and a guarantee of minimum occupancy of each node keeps the tree balanced. Whenever this guarantee is violated, nodes are added and removed in order to enforce the self-balancing property. Figure 2-5 shows a diagram of a T-Tree.

Like B+ Trees, T-Trees have a high density of values, which makes this structure cache-friendly in terms of key density. T-Trees gain some cache efficiency over a binary tree, since at the bottom of the tree, the keys are packed densely together. However, at all other levels of the tree, a search only makes a comparison by reading only two keys, resulting in a worse cache utilization compared to a B+Tree. In particular,

cache misses for T-Trees are on the order of  $\log_2 n$  whereas B+ Trees have a miss rate of  $\log_{\frac{m}{2}} n((\log_2 \frac{mL}{c}) + \frac{c}{mL})$  [17], where  $m$  is the number of keys,  $L$  is the number of bytes per key (as before),  $n$  is the number of records indexed, and  $c$  is the size of a cache line. For this reason, B+ Trees will in general have less cache misses than T-Trees in the 2 memory level architecture model.

Additionally, unlike the B+ Tree which requires expensive binary search in the node in order to do comparisons, this tree requires to do at most two comparisons to determine which subtree should be picked, and then binary search is necessary only at the last level. The performance of the T-Tree relative to other data structures is analyzed by [14]. However, all the benchmarks are performed on VAX-11/750 machines, with cycle times of 300 ns and 2 MB MOS memory with a latency of 100 ns, which implies a much faster memory with respect to CPU cycle times [18]. Under these CPU and memory speed characteristics, the results show that T-Trees are faster than B+ Trees and AVL Trees [14]. Insertion is thought to be faster than either tree under these conditions because it performs less node allocations and rebalances by relying on intra-node key movement. Similarly, the results show that T-Trees are faster than B+ Trees when querying, and only slightly slower than AVL Trees. The reasoning given in the paper for this behavior is that AVL Trees need not perform any binary search, T-Trees require to do it only at the last node, and B+ Trees need to do it always. A benchmark with more current CPU and memory speeds is developed in [17] and explained in Section 2.4.

In terms of space usage, T-Trees are very similar in footprint as B-Trees because of their key to value ratio. Benchmarks in [14] showed space utilization improvements over AVL Trees, which have a large amount of pointers.

## 2.4 Cache Conscious Trees

**CSS Trees** In an effort to solve the relative cache-inefficiency of T-Trees, [17] develops CSS Trees, an index structure very similar to B+ Trees that attempts to minimize cache misses. This structure makes a trade-off between computational

cost of comparing between nodes and key/pointer ratio. By removing pointers, cache lines can be used more efficiently and therefore generate less cache misses. In particular, cache trees do not require pointers by placing the tree nodes contiguously, so that the children of a node  $n$  are all located between  $b(m + 1) + 1$  and  $b(m + 1) + (m + 1)$ , where  $m$  is the number of keys at each node.

Queries on this type of indexing structure are very fast when  $m$  is chosen suitably, because most of the binary search inside a node is done with the help of the cache, and cache utilization is maximized in the absence of pointers, allowing the cache to hold more levels of the tree.

However, insertions and deletions from this tree are very expensive, because they require the nodes to be stored contiguously and as such need to be restructured several times when a new key is inserted. For this reason, this structure is only useful in OLAP databases, where insertions are rare in comparison with queries, and are performed in batches.

**CSB+ Trees** As a solution to the very expensive insertion operations developed above, [18] introduces a B+ Tree variant that does contain pointers, but with a much lower density than regular B+ Trees. Nodes of a given node belong to a *node group*; nodes inside a given node group are placed contiguously, so that they can be accessed in a manner similar to CSS Trees. The node then needs a single pointer, which points to the beginning of the node group. In this way, new levels can be added at locations that are not contiguous to their parents, since the parent will hold a pointer to them, decreasing the overhead for inserting new nodes.

## 2.5 Summary

According to the theoretical model and previous benchmarks explained above, arrays are ideal data structures for insertion and lookup of values, since they require a

constant time to perform these operations on a single key, regardless of the number of keys already inside the structure. Hash tables provide similar guarantees, but require a good hash function. Furthermore, they do not maintain the keys in order, making operations such as range scans very computationally expensive.

Trees lack the ability to perform constant time operations, but can adapt their structure sizes much more easily than arrays. Amongst trees, AVL trees have worse cache performance than other trees such as B+ Trees and cache conscious trees, in part because of its large number of pointers which make the structure bigger. B+ Trees provide an overall good cache performance, further enhanced by the CSB+ Trees. Although these last structures provide better cache performance than AVL trees, the amount of operations per node is larger on the former structures, making AVL trees better suited to deal with key sets with very few keys.

# Chapter 3

## The H-Tree Data Structure

### 3.1 Overview

In Section 2.2, we discussed the array data structure as an indexing structure that provides extremely fast lookup compared to using a tree. However, the applicability of arrays is limited, since they can only store keys efficiently when keys are contiguous and when the first key is 0. Similarly, adding or deleting an unbounded number of keys is complicated, and could potentially require making copies of the whole array.

We can think of an H-Tree as a B+Tree with integers as keys and array-like structures as values. Every key in the B+Tree is the starting value of some array-like structure. We refer to these array-like structures as `ArrayObj` objects; these contain the array of values associated with contiguous keys, plus some extra information such as density and node validity of the array of values. Section 3.2 describes these objects in more detail. The `ArrayObj` objects are shown in Figure 1-1 as  $a_i$ , and these can store values associated with a contiguous range of keys. In particular, the tree from Figure 1-1 can currently hold, without modification, all keys that lie in the intervals [1 to 20], [81 to 100], [350 to 370], [1001 to 1100], and [1350 to 1700].

Notice that unlike other trees like the B+Tree or BSTs, we require that the stored keys be integers, and not just comparable objects, since the H-Tree stores contiguous keys next to each other. However, unlike cache-conscious trees, we attempt to exploit the contiguousness of certain keys for faster lookups, instead of requiring expensive

binary search within all nodes of the tree.

The main challenge of the insertion and deletion algorithms is to maintain a tree structure with a small amount of wasted space, but that creates as few arrays as possible. Long, densely packed arrays provide fast lookups. However, (key, value) pairs are potentially inserted in any order, and the data structure needs to decide when to create clusters, when to merge clusters, and when to split them dynamically. Sections 3.3, 3.4, 3.5 explain the algorithms for query, insertion, and deletion.

## 3.2 The ArrayObj object

Logically, these objects hold the values corresponding to contiguous keys in some interval  $k_{min} \dots k_{max}$ , plus some metadata. In particular, each `ArrayObj` object contains the following information:

**Start value**  $k_{min}$  This is the smallest key the array can hold, and is used to offset insertions and queries.

**End value**  $k_{max}$  This is the largest key the array can hold.

**Array of values** `value[n]` The actual array of size  $k_{max} - k_{min} + 1$  that holds the contiguous values. The value `value[n]` holds the value associated with key  $k_{min} + n$

**Validity bit array** `valid[n]` Not all the cells in an `ArrayObj` instances necessarily represent valid values. For example, some key in an interval may not have been inserted, or it may have been deleted. In that case, the `value[n]` array will hold some value in that position, but this value will be invalid. The `valid[n]` array is an array of bits, where the  $k$ th bit is equal to one if and only if the value at `value[k]` is valid.

**Array density**  $\rho$  We also maintain the percentage of the array that contains values associated with actual inserted keys. In other words, this is the sum of the



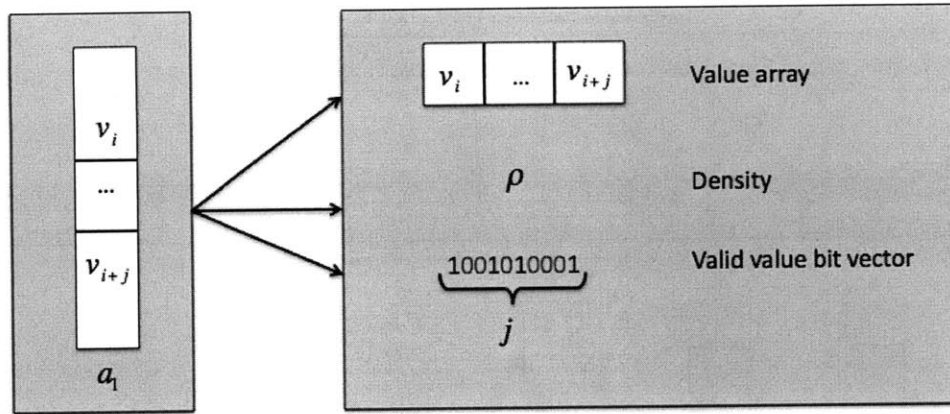


Figure 3-1: Components of ArrayObj instance  $a_i$ .

number of bits with value one in the validity array, divided by the length of this array ( $|k_{max} - k_{min} + 1|$ ).

We use shorthand notation to represent an ArrayObj instance  $a$  as  $[k_i, k_{i+j}]$ , which represents the array of length  $|j|$  and such that  $a.k_{min} = k_i$ . Figure 3-1 shows a detailed view of one of the arrays  $a_i$  shown in Figure 1-1. In particular, it shows its actual array of values, its density, and its validity bit array, represented as a binary stream of size  $j + 1$ . When we have an array  $a_n = [k_i, k_j]$ , we refer to its attributes as  $[k_i, k_j].valid[n]$ ,  $[k_i, k_j].value[n]$ , and  $[k_i, k_j].\rho$ , or  $a_n.valid[x]$ ,  $a_n.value[x]$ , and  $a_n.\rho$ .

In general, we keep two invariants on ArrayObj objects along the tree.

**Disjoint array property** Two different array objects cannot represent overlapping keys. In other words, no two arrays in the H-Tree can hold the same key. This is necessary for correctness of the query, insertion, and deletion algorithms.

**Density property** An array cannot have a density  $\rho < \rho_{min}$ . In Section 4.3.1 we provide a discussion of performance under different values of  $\rho_{min}$ . Although this value does not affect the correctness of the insertion, query, and deletion algorithms, it does have a performance impact.

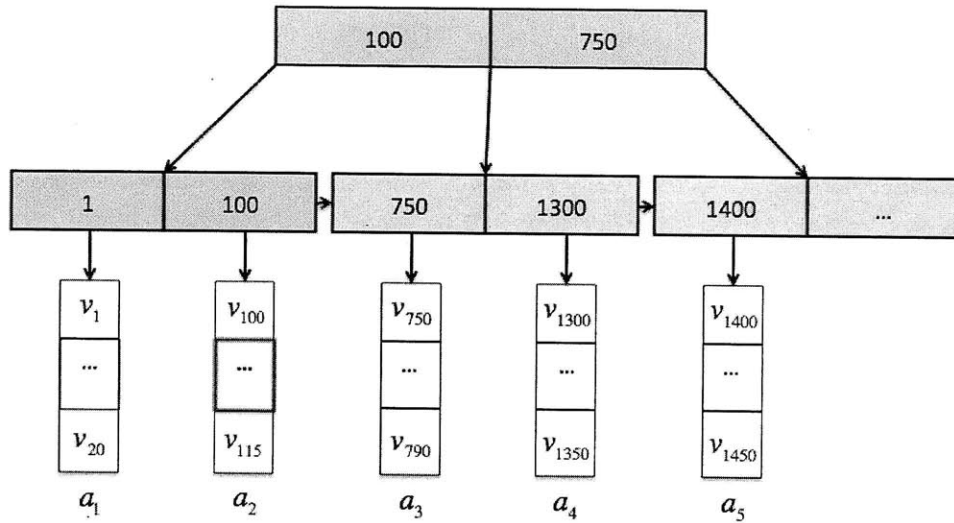


Figure 3-2: Query of key 110. The value in the array cell marked with red is returned only if the corresponding value `valid[110 - 100]` is set to true.

### 3.3 Query

In order to retrieve the value  $V$  associated with a key  $k$ , we first find the `ArrayObj` array  $a = [k_i, k_j]$  that would contain the value associated with this key if it existed in the H-Tree. We then verify if  $k$  is indeed in the range of the array (that is, if  $k$  is between  $k_i$  and  $k_j$ ). If so, we finally verify the `valid[n]` array to make sure that the value for key  $k$  is valid. Figure 3-2 shows a sample query for key 110 on a given H-Tree. The shaded boxes correspond to nodes traversed by the H-Tree algorithm.

```

QUERY( $T, k$ )
1  ▷ Check the candidate ArrayObj instance
2   $[a, b] \leftarrow$  Largest array such that  $k \geq a$ 
3  if  $k$  in  $[a, b]$  and  $[a, b].\text{valid}[k - a]$ 
4    then
5      return  $[a, b].\text{values}[k - a]$ 
6    else
7      return Not found

```

Figure 3-3: Algorithm for lookup of key  $k$  from H-Tree  $T$ .

Figure 3-3 shows a more detailed pseudo-code for querying an H-Tree. Line 2

obtains the largest array  $[a, b]$  such that  $k \geq a$ . This is the candidate `ArrayObj` since if any instance is to hold the key, it would be this candidate instance. This operation can be implemented by finding a nearby lower bound and then scanning the leaves of the tree in ascending key order until we find the desired value. The particular implementation of this operation is discussed in Section 3.8. If no candidate array exists, then the key is not found. Notice that no other array associated with the key could be a candidate array because of the disjoint array property. In particular, if another array  $b = [a', b']$  existed such that  $k \in [a', b']$ , then we'd have that  $a' < a$  (by the choice of  $[a, b]$  as the largest lower bound), and that  $b' < b$  (by the disjoint array property). Therefore, we'd have  $a' \leq b' < a \leq b \leq k$ , so  $k$  cannot be in the interval  $[a', b']$ . Lines 4 through 7 simply return the appropriate value, assuming the value corresponds to a valid key in the range of array  $[a, b]$ . The position of the value in array  $[a, b]$  is  $[a, b].\text{values}[k - a]$ , since the first element in the array is key  $a$ .

### 3.4 Insertion

When a key  $k$  is to be inserted, there are three possible cases that need to be considered:

**Key fits** In this case, one of the arrays of values already contains the slot that would fit the key. In this case, no further modification to the tree is needed. The value is inserted in the correct position, and the validity bit array and density values are updated as needed. Figure 3-4 shows a sample insertion of key 19 into an H-Tree. The metadata update is shown in red, with the value of the bit array updated to 1 to indicate a valid inserted key.

**Near miss** This occurs whenever no existing array holds the key that needs to be inserted, but there is an array that holds nearby keys. If the array  $a$  can be resized to twice its original size so that it can hold the new key without violating the density property, then we replace  $a$  with an array of twice the original size, such that it can fit the value to be inserted. When doubling the original array

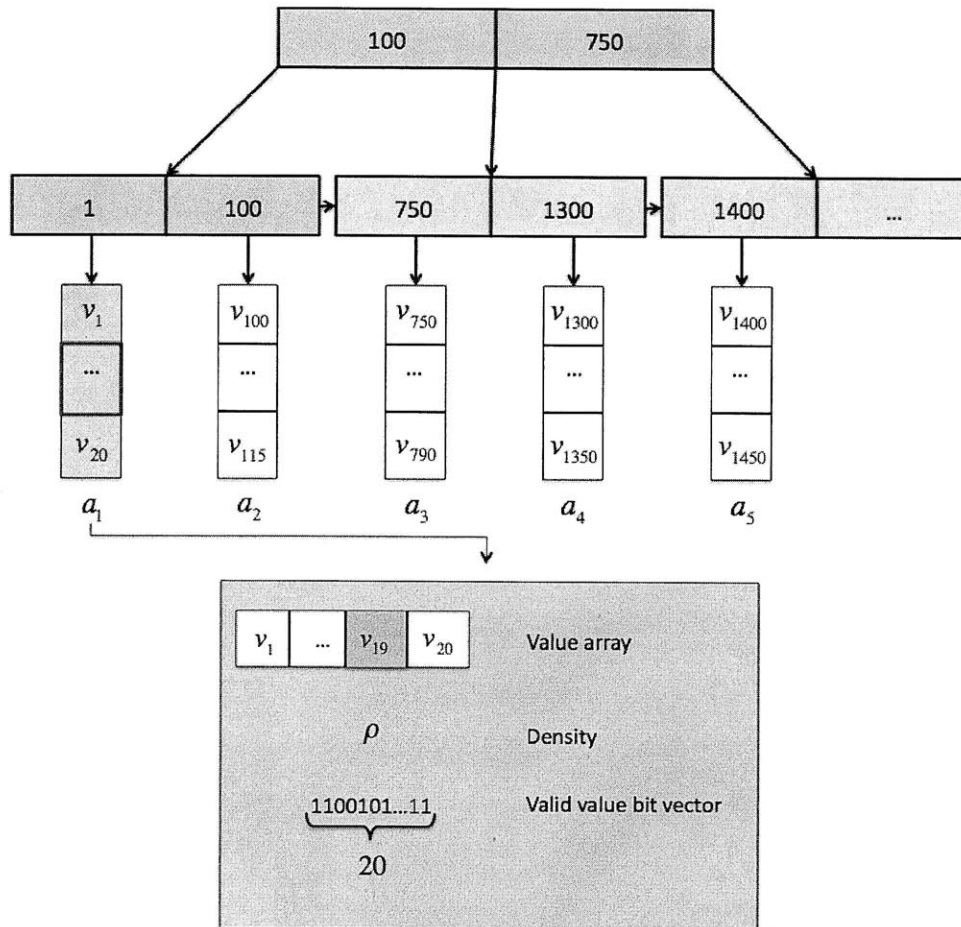


Figure 3-4: Sample insertion of key 19. The key fits perfectly in array  $a_1$ . The blocks shaded red are read by the H-Tree algorithm.

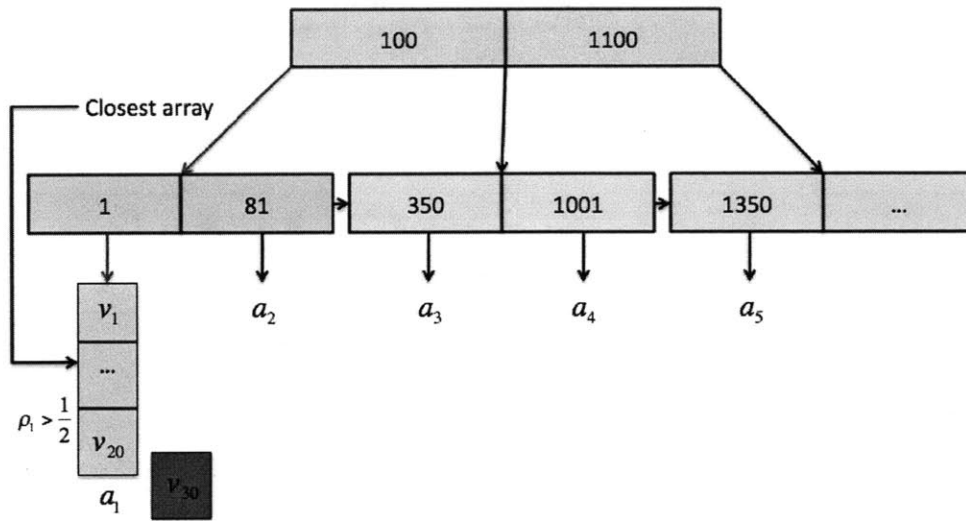


Figure 3-5: Sample insertion of key 30. The key does not fit its closest array  $a_1$ , but doubling it would fit it and preserve the density property. We therefore reset array  $a_1$  to contain values 1 through 40, twice its original value, and reduce insertion to the first case.

$a_i$ , the new end key may overlap with the next array. In that case, we merge both arrays into a single array. Figure 3-5 depicts a simple insertion without merging. In this case, we wish to insert key 30, which is not part of either  $a_1$  or  $a_2$ . However, doubling this array would fit the key while satisfying the density property. Therefore, the array is doubled and its data is updated accordingly. Since the next array  $a_2$  starts further than 40, these arrays do not need to be merged.

We must note that we may need to merge more than one array, if the array that is being doubled is very large such that the new size covers a number of other (smaller) nearby arrays.

We can also see that merging these arrays does not violate the density property, since both arrays satisfy it, the number of valid keys after merging equals the sum of the valid keys of the merged arrays, and that total size of the array is at most the sum of the sizes of the subarrays.

**Far miss** Whenever we wish to insert a key that is far away from every array and

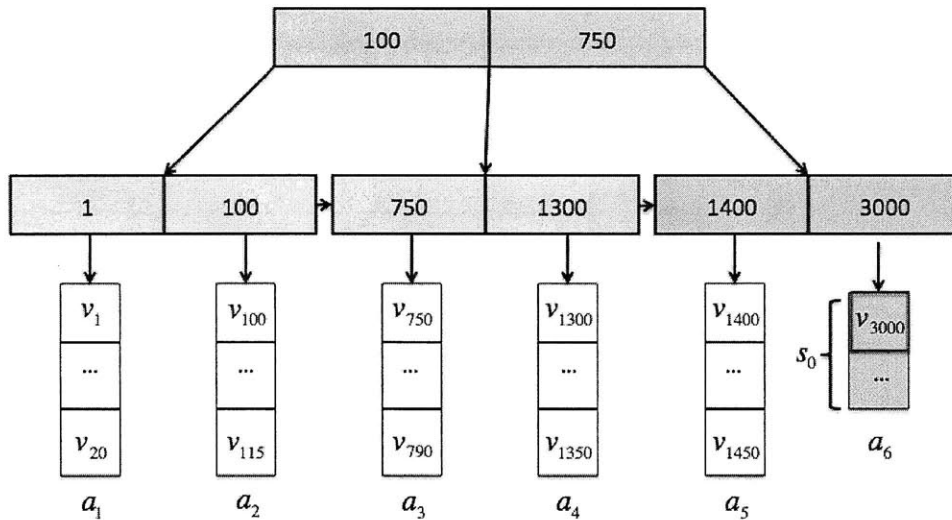


Figure 3-6: Sample insertion of key 3000. The key does not fit its closest array  $a_5$ , and doubling it would still not make it fit or would violate the density property. We therefore create a new array  $a_6$  starting on this key and with size  $s_0$ .

thus does not qualify for the previous two cases, then we build a new array of size  $s_0$ , the default initial array size. As with  $\rho_{min}$ , this is a parameter that can be adjusted for performance reasons, and its impact is discussed later. Figure 3-6 shows the same tree, after key 3000 has been inserted.

Figure 3-7 details the pseudo-code for the insert operation. Lines 4 and 5 deal with the first case, lines 7 to 19 deal with the second case, and lines 21 to 23 deal with the third case.

### 3.5 Deletion

In the case of deletion, we simply update the validity array to zero out the position corresponding to the key we wish to delete. Upon performing this operation, we may violate the density property, which we then need to restore by splitting the array into parts. To do this, we use a function `split_array()` that takes in the `ArrayObj` that just violated the density property upon deletion of an element, and outputs a set of subarrays, each satisfying the density property.

```

INSERT( $T, k, v$ )
1  ▷ Check if  $k$  is in some ArrayObj instance  $[a, b]$ 
2  if  $k$  in  $[a, b]$ 
3      then
4           $[a, b].values[k - a] \leftarrow v$ 
5          update  $\rho$ 
6      else
7          ▷ Check if nearby arrays can fit in  $k$  by doubling them
8           $[a, b] \leftarrow$  Largest array such that  $k \geq a$ 
9          if  $k \in [a; a + 2 \cdot (b - a)]$ 
10             then
11                 Reallocate  $[a, b]$  twice the size  $[a, a + 2 \cdot (b - a)]$ 
12                  $[a, a + 2 \cdot (b - a)].values[k - a] \leftarrow v$ 
13                 ▷ Check if we need to merge an array
14                  $[c, d] \leftarrow$  next array after  $[a, b]$ 
15                 if  $c < a + 2 \cdot (b - a) + 1$ 
16                     then
17                         Merge arrays  $[a, 2 \cdot (b - a)]$  and  $[c, d]$ .
18
19                 update  $\rho$ 
20             else
21                 Create new array  $[k, k + s_0 - 1]$ 
22                  $a[0] \leftarrow v$ 
23                 update  $\rho$ 
24

```

Figure 3-7: Algorithm for insertion of key  $k$  with associated value  $v$  into H-Tree  $T$ .

Figure 3-9 details the pseudo-code for deleting elements. Lines 6 and 7 replace the original array  $[a, b]$  with the result of splitting arrays.

### 3.6 Analysis

There are many possible ways to define optimality of an H-Tree for a given key set, such as an instance with the smallest number of arrays without violating the tree density property. However, an arbitrary insertion of values does not inevitably converge to an optimal H-Tree. For example, inserting values 1, 3, 5, ... will result in a H-Tree with as many arrays as values, instead of a single array with density  $\rho = \frac{1}{2}$ .

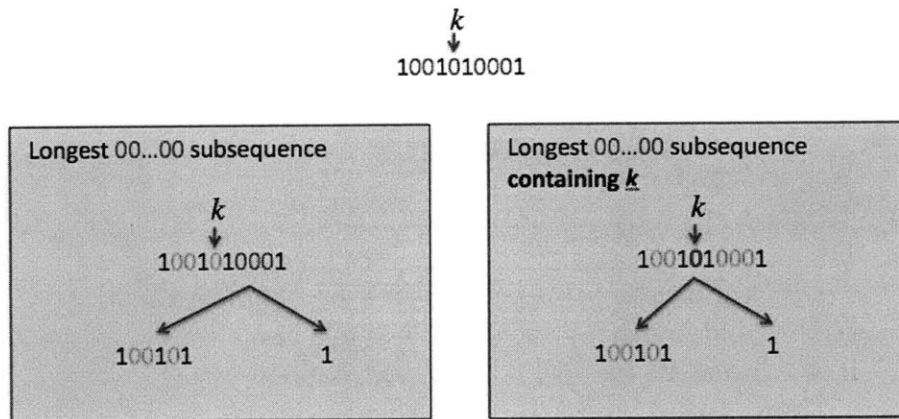


Figure 3-8: Two different array splitting policies. The position marked with  $k$  is the key that was just deleted. The left diagram splits the array into two subarrays from the position of the largest sequence of zeroes. The right diagram shows the split from the position of the largest sequence of zeros containing  $k$

However, in the worst case, the data structure will behave similarly to a B+Tree, since it will consist of several single-valued arrays. Apart from the added overhead at lookup caused by the extra level of indirection created by the array, and the insertion overhead of inspecting nearby arrays to see if doubling can fit the value, the B+Tree bounds our worst-case performance. This overhead is a constant cost above the B+Tree, and occurs when the structure is used to store non-contiguous ranges of keys. Similarly, due to the density property of arrays, the wasted space is at most a constant multiple of the number of keys present in the data structure. This guarantees that our data structure will never exhibit an extremely bad processing or memory behavior.

## 3.7 Algorithm Parameters

### 3.7.1 $\rho_{min}$

In some sense, increasing the value of  $\rho_{min}$  artificially creates a key set with a higher clustering. However, it also amplifies the negative performance effect of randomness



```

DELETE( $T, k$ )
1  ▷ Check the candidate ArrayObj instance
2   $[a, b] \leftarrow$  Largest array such that  $k \leq a$ 
3   $[a, b].\text{valid}[k - a] \leftarrow \text{false}$ 
4  if  $[a, b].\rho < \rho_{min}$ 
5      then
6          T.remove( $[a, b]$ )
7          T.add_all[split_array( $[a, b]$ )]
8

```

Figure 3-9: Algorithm for deletion of key  $k$  from H-Tree  $T$ .

in the insertion pattern. This is because the amount of doubling causes more frequent merges. Additionally, In terms of space, the lower the minimum threshold density, the more potential for unoccupied space. Chapter 4 describes some benchmarks to measure the effects of different values of  $\rho_{min}$ .

### 3.7.2 $s_0$

This specifies the initial size of the array. This size cannot be picked in a way that would violate the density property. For example, if the density were chosen as  $\rho_{min} = \frac{1}{4}$ , then the maximum value of  $s_0$  would be 4, since the original array would only have a single element. Chapter 4 describes some benchmarks that measure the effect of  $s_0$ .

### 3.7.3 split\_array()

There are several candidate functions that can be used to split arrays. For example, we considered the following two splitting functions:

1. Split the array into two, by finding the longest subsequence of invalid keys, and then by creating two subarrays: one of them out of the values to the left of the longest subsequence, and another one of them out of the values to the right of the longest subsequence.
2. Split the array into two, as before, by finding the longest subsequence of invalid

keys that include the key that was just deleted. This policy prevents us from having to visit the whole array to detect where to split, but does not guarantee the removal of the largest possible amount of zeroes.

The first strategy will delete the largest number of zeroes when splitting into two arrays. However, it requires an entire scan to the array before deciding how to split. On the contrary, the second strategy only requires to scan keys that will actually be deleted, but may not necessarily get rid of a large number of zeros. The justification for this heuristic is that when deletions also occur clustered, then several nearby zeros are likely to be found when attempting to split the array.

Figure 3-8 shows an example of these two array splitting policies in action, by depicting the validity bit array of some `ArrayObj` object. The letter  $k$  marks the key that was just deleted and that caused the violation of the density property. In the first case, we find the last run of three zeros and create two sub-arrays from the arrays in both sides of the run of invalid keys. In the second case, only one cell is removed, corresponding to the value that was just deleted. Once more, we discuss the impact of each of the parameters below.

## 3.8 Implementation details

**Choice of representation** We implemented the structure detailed above in C++.

The H-Tree contains an `stx::btree` with integers as keys and `ArrayObj` objects as values. The value array in the `ArrayObj` class is actually an array of pointers to values. We chose to implement the array as an array to pointers with the expectation that values could potentially have large sizes, and could make the operations of merging and splitting even more expensive. Keeping these objects separated sacrifices locality of reference when accessing contiguous values, but ensures that only address values get copied during array operations.

**next\_smallest()** In the pseudocode from Sections 3.3, 3.4, 3.5, we needed to obtain the array starting at the largest key smaller or equal to a key  $k$ . The function

that retrieves such a key is the procedure `next_smallest()`. In order to implement this function, we use the tree's `lower_bound` function. This function takes in a key and returns a pointer to the first key equal to or greater than the specified key. After appropriate boundary conditions, the `next_smallest()` element is the element preceding the `lower_bound` element.

**Store value start, value end, pop\_count** Instead of storing the density as a floating point number, we preferred to store the actual number of valid elements in the array. This avoids using floating point arithmetic, while still making it possible to calculate the density whenever needed. The end value is stored, even when it is redundant and can be calculated from the array size. This allows us to perform one operation less every time we want to check if a key is contained in some array.

**Representation of valid array entries** Initially, we implemented an array of pairs (`V value, bool is_valid`). However, storing the arrays separately allows us to compress the array of booleans and store them contiguously into integers, accessing their values by shifting and masking bits, and even resulted in speed improvements of up to 10%. There are other reasonable ways to encode valid array entries. For example, in the case of a highly dense range, we could simply store an array of positions that are invalid.



# Chapter 4

## Benchmarks

We now test the performance of the data structure under different sequences of key insertions and compare it to other data structures.

Section 4.1 details the two kinds of benchmarks to measure insertion behavior. Section 4.1.2 discusses the performance under different insertion orders and varying key sizes. Section 4.1.3 then characterizes the structure of a key insertion sequence by three dimensions: the amount of clustering, the number of keys inserted, and how sequentially keys are inserted. We report and compare the performance of the H-Tree and other structures along different points in the dimensions. Section 4.2 then modifies the insertion benchmarks to incorporate deletions after keys have been inserted. Section 4.3 discusses the results of running the insertion benchmarks with different parameter values of the insertion algorithm discussed in Section 3.7. Finally, Section 4.4 discusses the performance gains of the H-Tree under a more complete system, the TPC-C benchmark.

Appendix A summarizes the design and implementation details of the developed benchmarking system.

## 4.1 Insertion benchmarks

### 4.1.1 General Setup

The benchmarks in this section proceed in two phases:

**Insertion Phase** In this phase, we insert the keys according to the particular insertion pattern outlined below.

**Query Phase** After the data structure contains all the keys to be inserted, we perform several query operations to lookup keys at random.

### 4.1.2 Performance under different loads

We first tested the performance on the H-Tree under three different benchmarks, one in a condition where it should excel and another in a situation where its behavior should be not be much worse than other data structures. For these cases, we considered insertion of keys that form densely-packed clusters. Figure 4-1 shows a series of clusters of keys. A *cluster* is simply a sequence of contiguous keys. Notice that although the keys themselves are contiguous, the order in which these keys are inserted into the data structure can vary. On each of the benchmarks described below, we first generate a set of keys of a given size, and then run and time both phases of the benchmark on each data structure, and then generate a new set of keys with a larger size. We then graph the insertion and query time as a function of the number of keys inserted. We built three insertion scenarios for clustered keys, depending on the order of insertion of the keys:

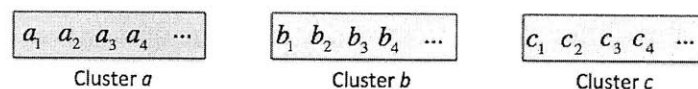


Figure 4-1: Schematic representation of keys in a clustered configuration

### Strictly sequential insertion

In this case, keys were inserted, one cluster at a time, and in increasing order within the cluster. This type of insertion could correspond to an insertion into a table with an auto incrementing key. Figure 4-3 shows a depiction of the insertion order of the keys in the cluster.

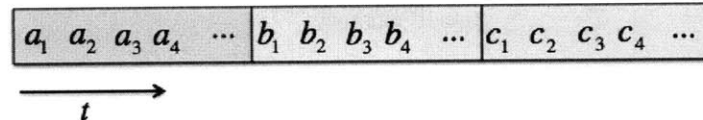


Figure 4-2: Keys inserted in order

Figures 4-3 and 4-4 show the results of the sequential insertion benchmark with 4 and 400 clusters. The top diagram on each figure shows the average time it takes to query a key for a varying number of inserted keys. The bottom diagram of each figure shows the average time it takes to insert a key on a varying number of inserted keys.

In terms of insertion time, the data structures other than the H-Tree are unaffected by the number of clusters. In particular, since the insertion sequence is monotonically increasing, the state of these trees should be the same under both benchmarks. The array presents the best possible performance, with no growing insertion or query time as the number of keys increases. On the other hand, both the B+Tree and the AVL Tree show a logarithmic increase in the time it takes to insert keys, with the B+Tree being strictly faster than the AVL Tree regardless of the number of keys. The insertion times for the H-Tree, however, change dramatically depending on the number of clusters present. Whenever there are few, large clusters, insertion is fast and similar in performance to the array, since the arrays are large in comparison to the tree portion of the data structure. However, for several, small clusters the operations on the tree part of the H-Tree begin to dominate, and the performance progressively resembles the B+Tree.

A similar situation occurs in the performance of queries. When there are few,

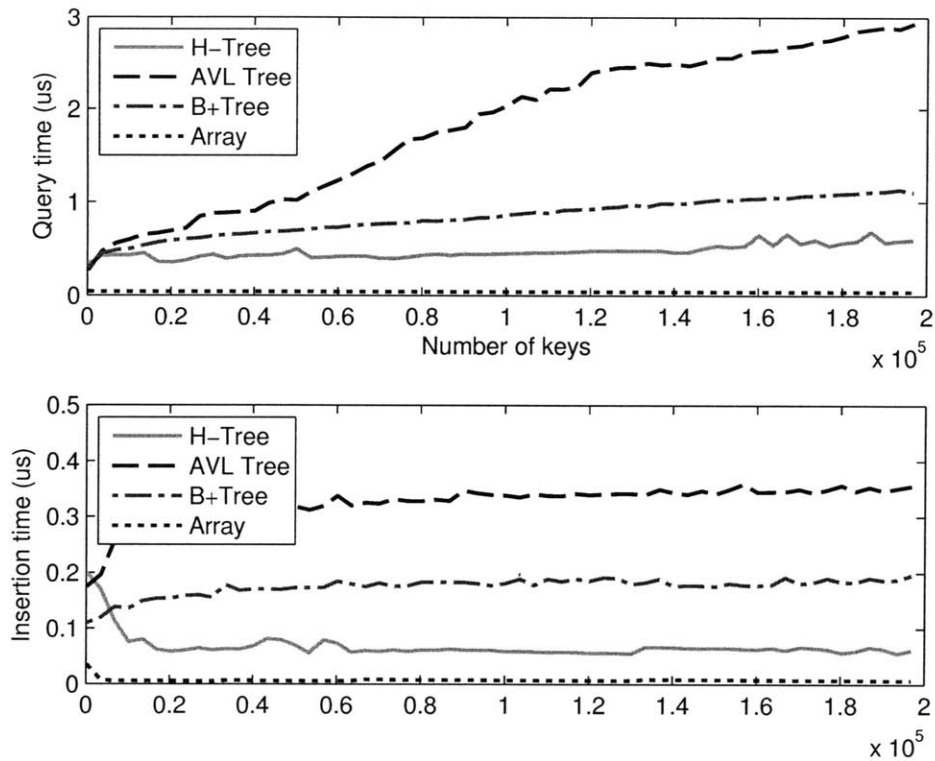


Figure 4-3: Benchmark for strictly sequential insertions and few (4) large clusters

large clusters, the H-Tree easily outperforms B+Trees even when few keys are present. However, when there are many more small clusters, the query performance starts off as worse than B+Trees (and even AVL trees), although it eventually catches on, since its tree component does not grow as fast as the other structures because of the small amount of clustering still present.

### Alternating cluster, sequential intra-cluster insertion

In this case, we pick a cluster randomly, and then pick the key to insert within that cluster sequentially. In other words, the only guarantee on ordering is that if we consider the subsequence of inserted keys corresponding to a single cluster, this subsequence is monotonically increasing. This type of pattern could emerge, for example, when different entities insert data sequentially into the table, each using a different range of pre-allocated keys. Figure 4-6 shows one sample insertion scenario with this property.



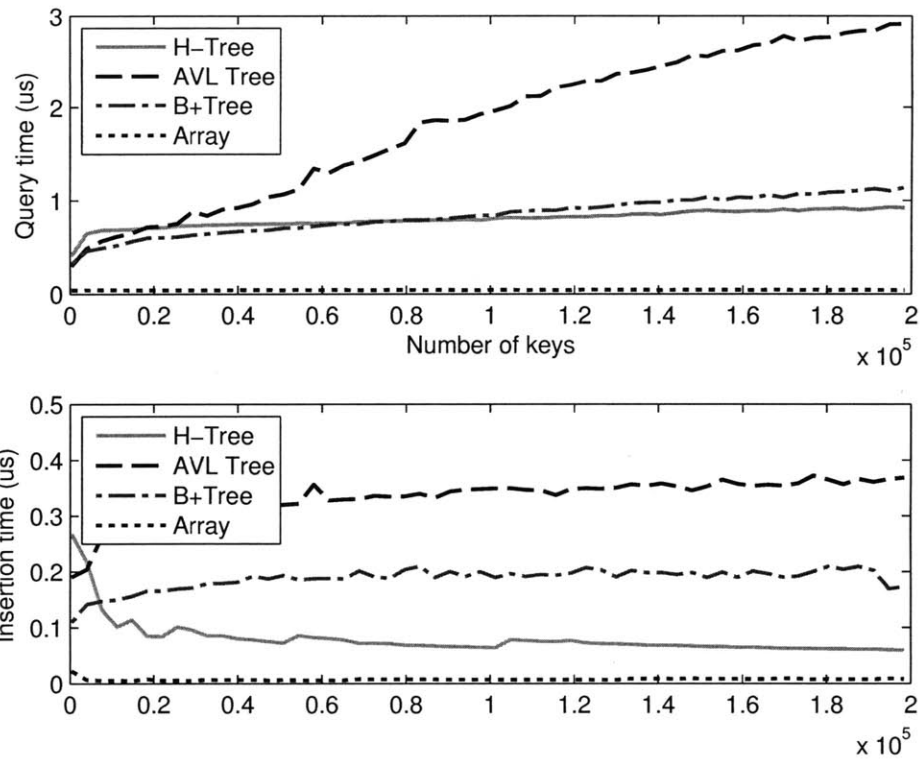


Figure 4-4: Benchmark for strictly sequential insertions and several (400) small clusters

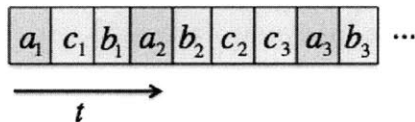


Figure 4-5: Keys inserted in random cluster order, but sequential intra-cluster order

Figures 4-6 and 4-7 show the insertion and query time results of benchmarking the H-Tree with the alternating sequence insertion pattern, again with 4 and 400 clusters, respectively.

Compared to the strictly sequential insertion pattern shown on Figures 4-3 and 4-4, the structures other than H-Tree show differences, since the relative key ordering has now changed. However, the performance is still similar when comparing the case with few large clusters and the case with many small clusters. For the H-Tree, the insertion time is larger on average on alternating insertions than strictly sequential

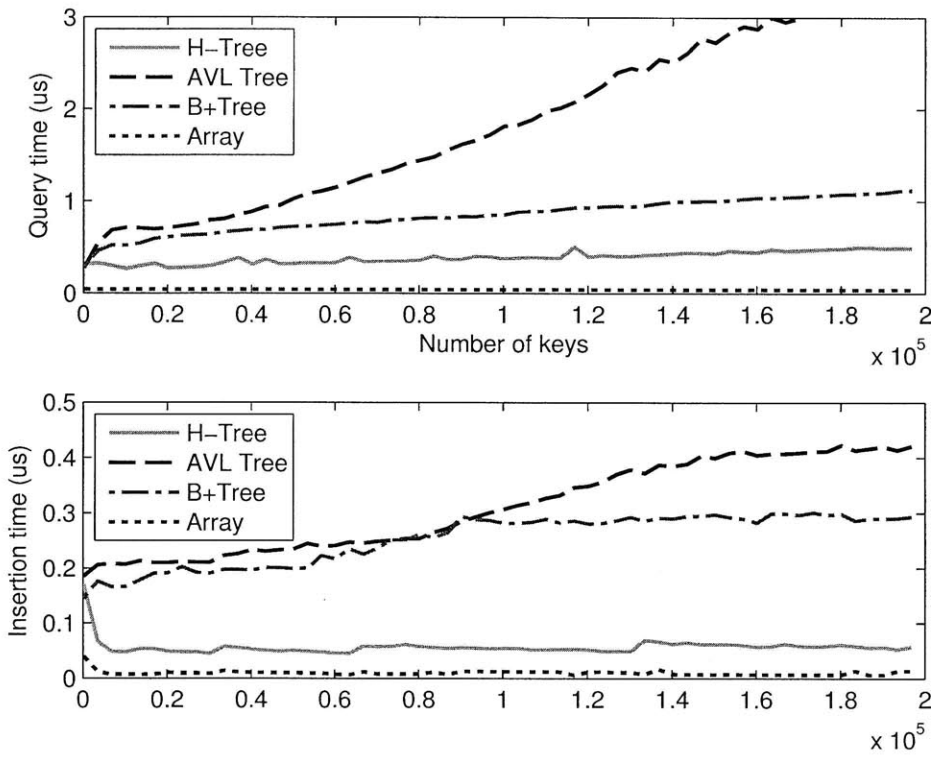


Figure 4-6: Benchmark with keys inserted with random cluster selection and few (4) large clusters

insertions, presumably because arrays are accessed with less locality, hurting cache performance.

On the other hand, querying time did not seem to vary compared to the strictly sequential insertion. This suggests that the structure of the H-Tree (tree component and array configuration) after the two types of insertions was the same. As before, querying performance when the key set had few, large clusters resembled array performance, moving more towards B+Tree like performance as the clusters became smaller and increased in number.

### Windowed randomization

The previous two insertion patterns showcase some idealized insertion patterns that could be performed on the H-Tree, but are potentially hard to find in practice. In particular, although the data structure is designed as an index structure for clusters

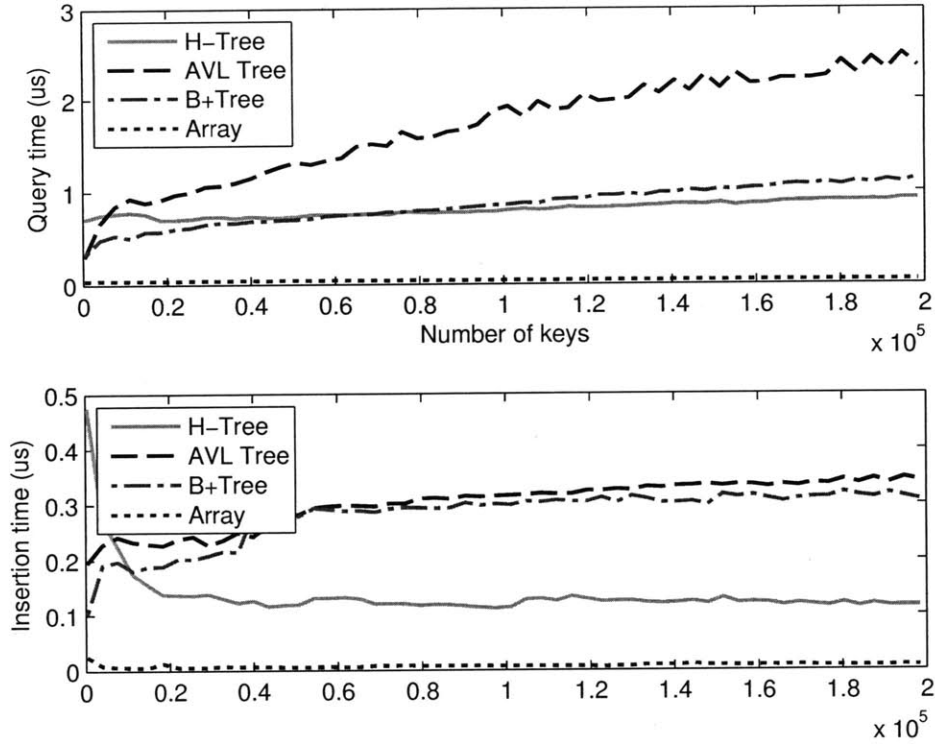


Figure 4-7: Benchmark with keys inserted with random cluster selection and several (400) small clusters

of data when keys are inserted almost sequentially intra-cluster, this insertion pattern may not be perfectly sequential. For example, the insertion may proceed with some keys missing, or inserted much later. Figure 4-8 summarizes a procedure to obtain an insertion pattern that follows this idea.

We define a window of size  $k$ ; the larger the value of  $k$ , the larger amount of randomization that we have. We start with the sequence of insertions from sequential intra- cluster insertion, and the window positioned over the first  $k$  elements of the sequence. Elements under the window are shuffled, such that every permutation is equally likely. The window is then moved forward by one element, and the procedure is repeated. The resulting sequence is the randomized insertion pattern. Essentially, this procedure creates a localized randomization, where keys are unlikely to move too far from their original insertion point. However, it is still theoretically possible (although unlikely) that an early key be moved to the end of the insertion sequence.

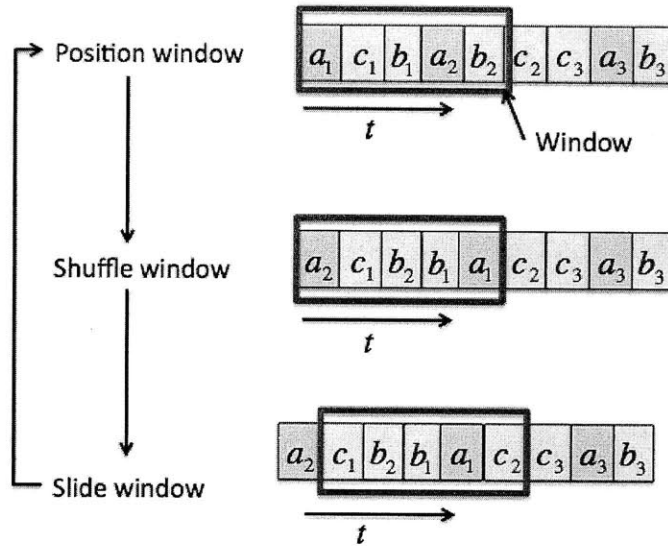


Figure 4-8: Keys inserted in random cluster order, but sequential intra-cluster order

From Figure 4-8, we can see that a choice of window size  $k = 1$  reduces to the random cluster, sequential intra-cluster insertion pattern, whereas choosing  $k$  to be the total size of the keys reduces to a uniformly random shuffling of the keys within the cluster.

Figures 4-9, 4-10, and 4-11 show the benchmark executed on key sets with the same number of clusters, but with varying randomization by increasing the size of the window. In particular, we used window sizes of 15, 1500, and 15000 with 4 clusters.

The insertion times for data structures other than the H-Tree are not noticeably affected by the variation of window size in these graphs. However, as more variation is added, the performance of the H-Tree becomes gradually worse, especially for few keys. At this stage, the number of array creations is very large, and does not get compensated by insertions into array portions. This is a similar effect as exhibited on Figures 4-4 and 4-7, with a slightly decreasing insertion time curve.

Query time, on the other hand, gradually increases with the randomization. The query performance under very small randomness in this benchmark creates the tree structure with less amount of arrays possible, and therefore provides good query time. The larger amount of randomization creates a larger tree with more, smaller arrays,

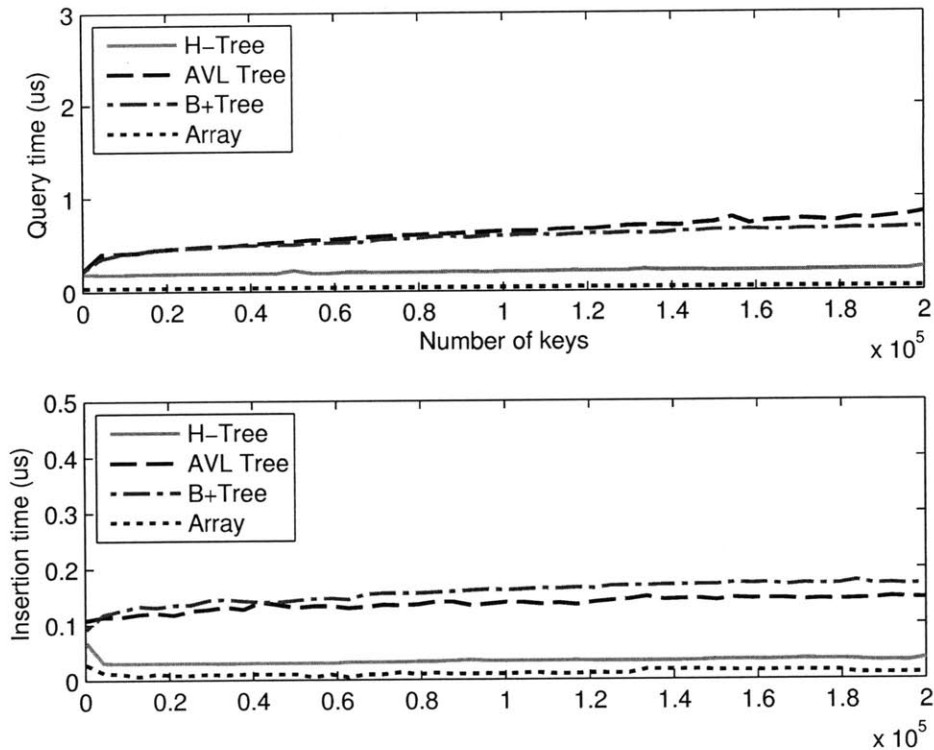


Figure 4-9: Benchmark with keys inserted with window algorithm. The window size is 15.

that increase query time. However, perhaps surprisingly, the performance penalty under a very large randomization does not make the performance as bad as that of a B+Tree.

### 4.1.3 Varying across key dimensions

For this section, we determined the key insertion sequence by varying three different dimensions that characterize the sequence of inserted keys:

**Clustering key set** This refers to how close to each other the set of keys to be inserted are. This measure is completely independent of the actual order in which these keys are inserted. A set of contiguous keys has as large as possible clustering of keys, whereas a set of keys where every key is very far away from each other has a very low clustering.

We expect that, all else being equal, a higher amount of clustering will make

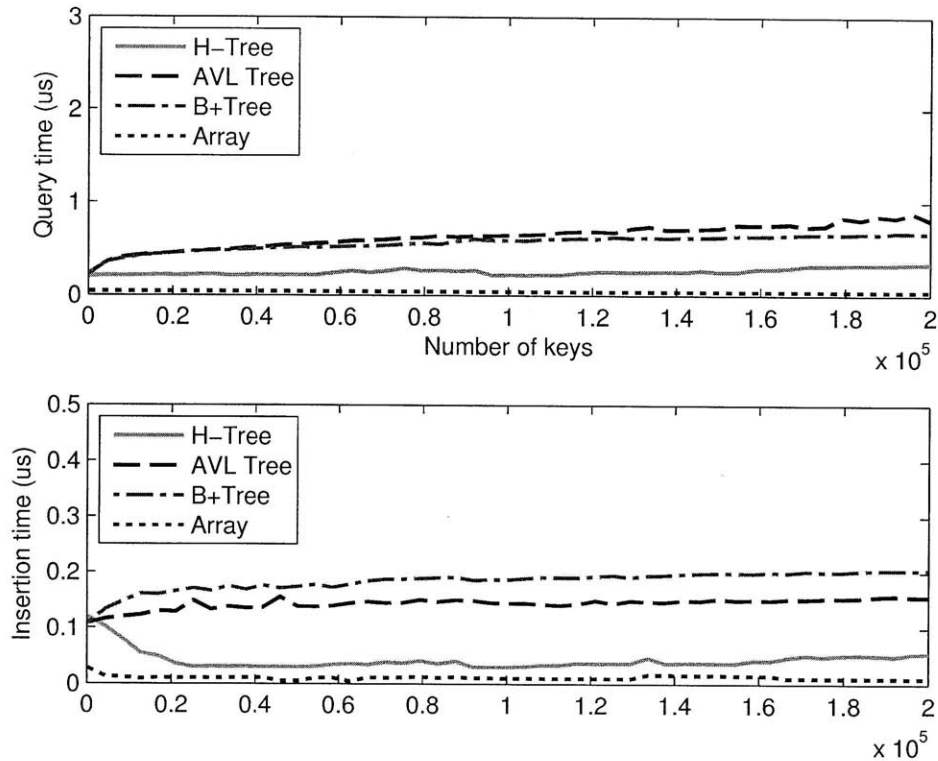


Figure 4-10: Benchmark with keys inserted with window algorithm. The window size is 1,500.

the H-Tree perform better, since its performance will resemble that of an array, with a large number of arrays with respect to the size of the tree portion. On the other hand, the smaller the clustering, the more performance will resemble that of a B+Tree, since the array portions will hold a low number of values, and the time spent traversing the tree will dominate the performance.

**Key insertion randomness** This refers to the actual order in which the keys get inserted into the data structure. For a given key set that gets inserted, whenever it is inserted in order, we say that the key clustering is very high.

Ideally, the H-Tree would generate an optimal structure based exclusively on the key set, and regardless of the actual insertion pattern. However, in practice this is not true. However, the hope is that the structure is somewhat resilient to randomness in the key insertion.

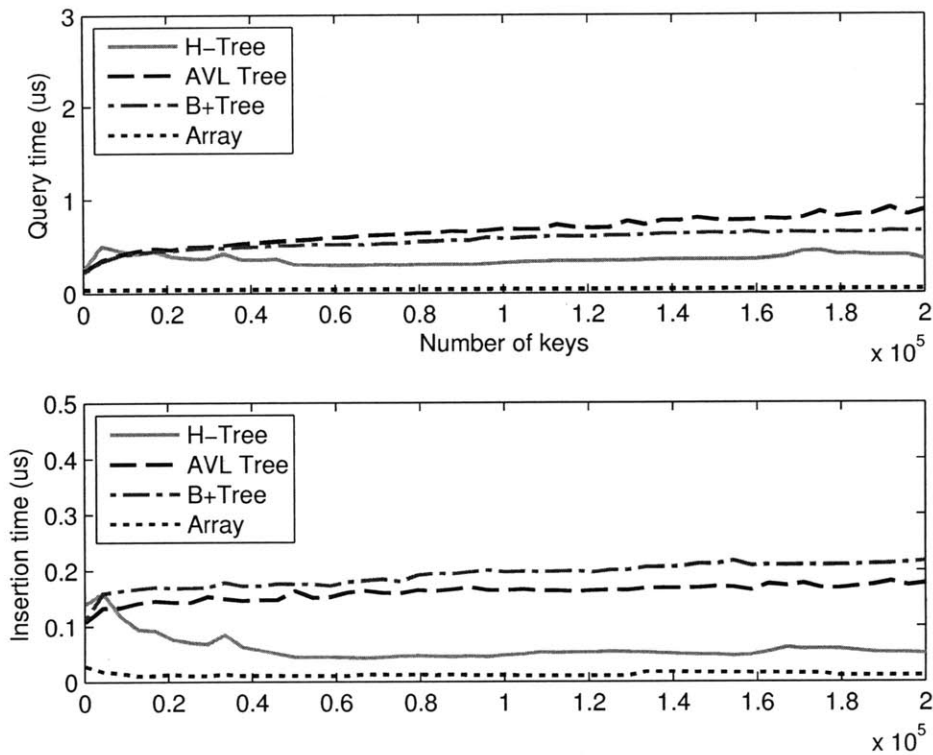


Figure 4-11: Benchmark with keys inserted with window algorithm. The window size is 15,000.

**Key set size** This refers to the actual absolute number of keys held by the H-Tree.

If we fix the key insertion randomness and key set size, we can generate a plot of insertion and query times as a function of the amount of clustering (ranging from *many, small clusters* to *few, large clusters*). Based on the previous results, the expectation is that the B+Tree and the array will be constant along the graph, since their insertion and queries depend on the relative ordering between keys, and not how spread out they are. However, the H+Tree should have a similar performance to an array for few large clusters, and resemble a B+Tree for many small clusters, plus some overhead.

We chose four extreme points regarding size and randomness in its insertion of keys, to get a picture for how the performance characteristics of the H+Tree varied depending on how many keys it was holding, and how they were inserted. We thus came up with four graphs that are representative of the whole input space, and

should provide some insight on the performance of the H-Tree on very different types of loads. For each of these loads, we plot the insertion and query times as a function of the amount of clustering (which ranges from *many, small clusters* to *few and large clusters* ).

1. **Almost sequential insertion. Large  $n$ .** Figure 4-12 shows the outcome from this benchmark.

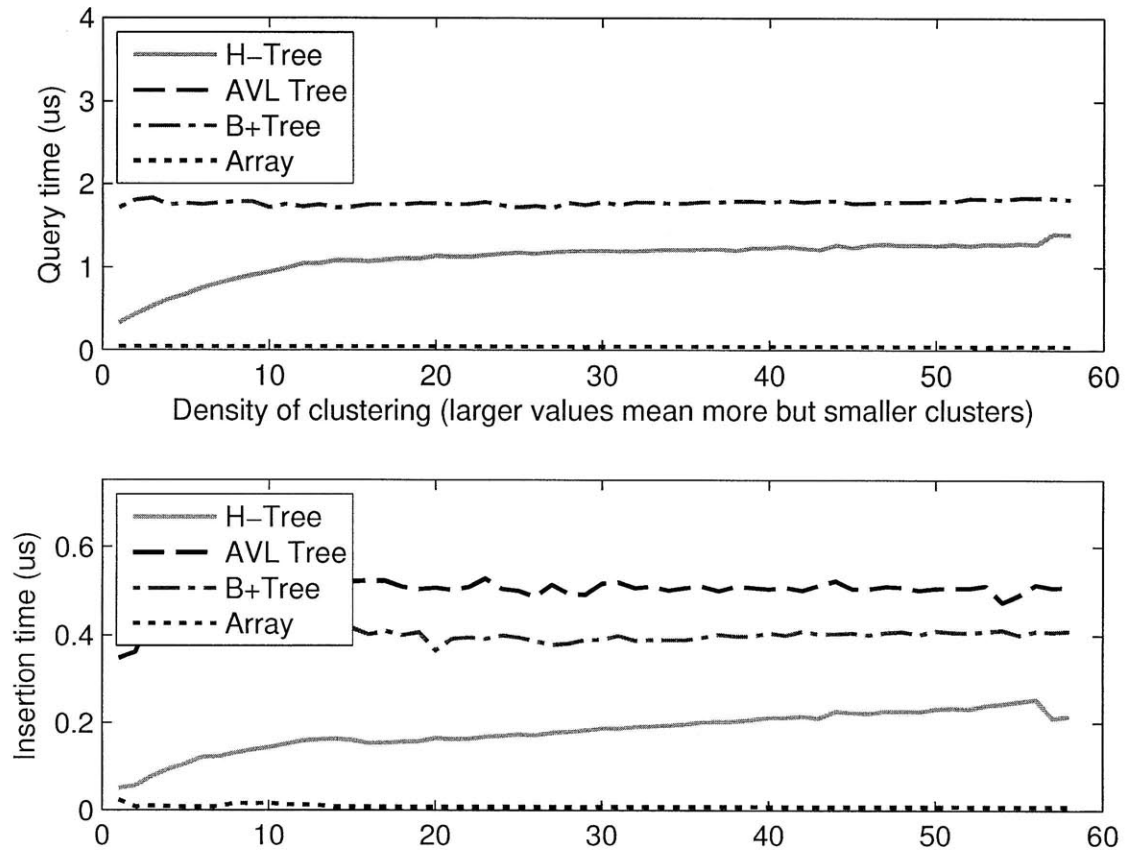


Figure 4-12: Density graph with almost sequential insertion and large number of keys ( $8 \times 10^5$ ).

2. **Almost sequential insertion. Small  $n$ .** Figure 4-14 shows the outcome from this benchmark.
3. **Almost random insertion. Large  $n$ .** Figure 4-13 shows the outcome from this benchmark.



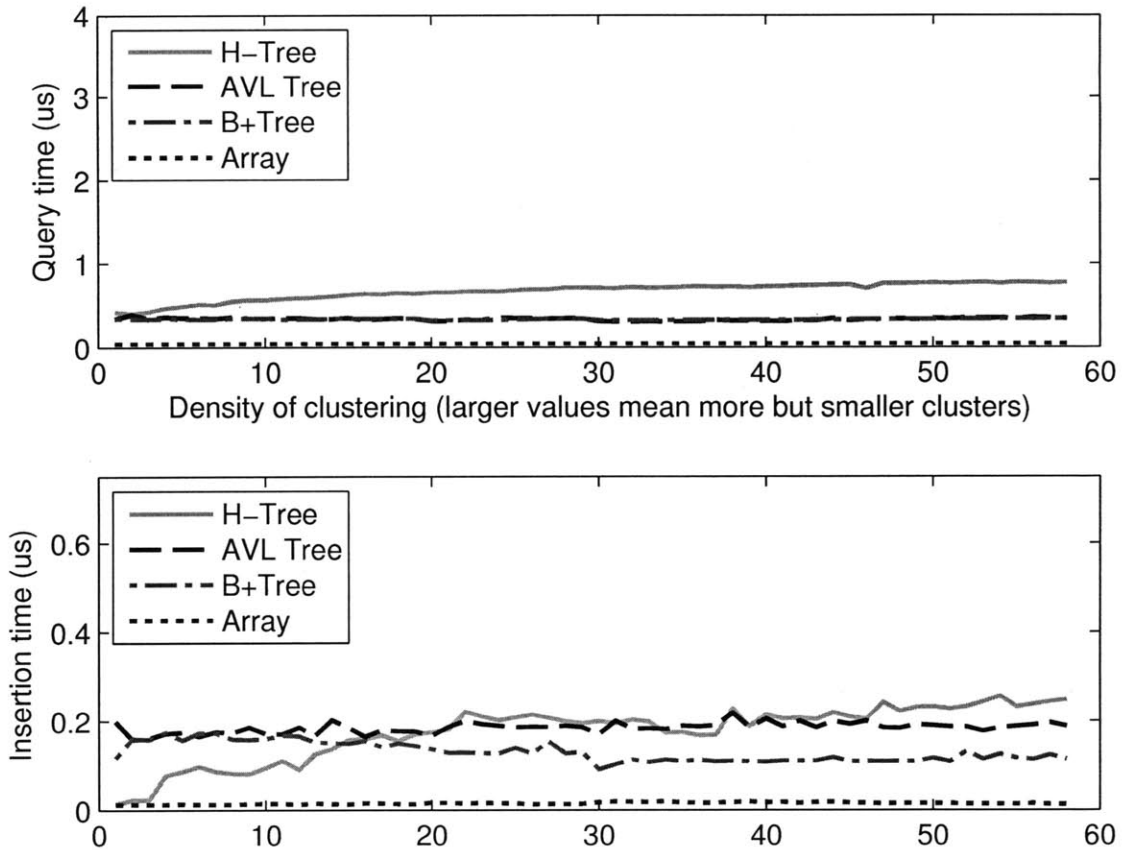


Figure 4-13: Density graph with almost sequential insertion and small number of keys (80).

4. **Almost random insertion. Small  $n$ .** Figure 4-15 shows the outcome from this benchmark.

The different key sets for a fixed size  $n$  and insertion pattern are generated for a given amount of clustering by increasing the number of clusters and then choosing the number of keys per cluster that create  $n$  keys total. As Figures 4-14, 4-12, 4-15, and 4-13 demonstrate, the H-Tree begins with a performance comparable to a B+Tree and becomes more array-like as the key set consists of fewer, large clusters.

When the value of  $n$  is small the query time decreases for all data structures, are expected from a smaller data structure. However, the query performance for the H-Tree should be very similar when there are fewer keys than when there are more keys, but a fixed clustering since the number of keys per cluster is smaller, but the size of

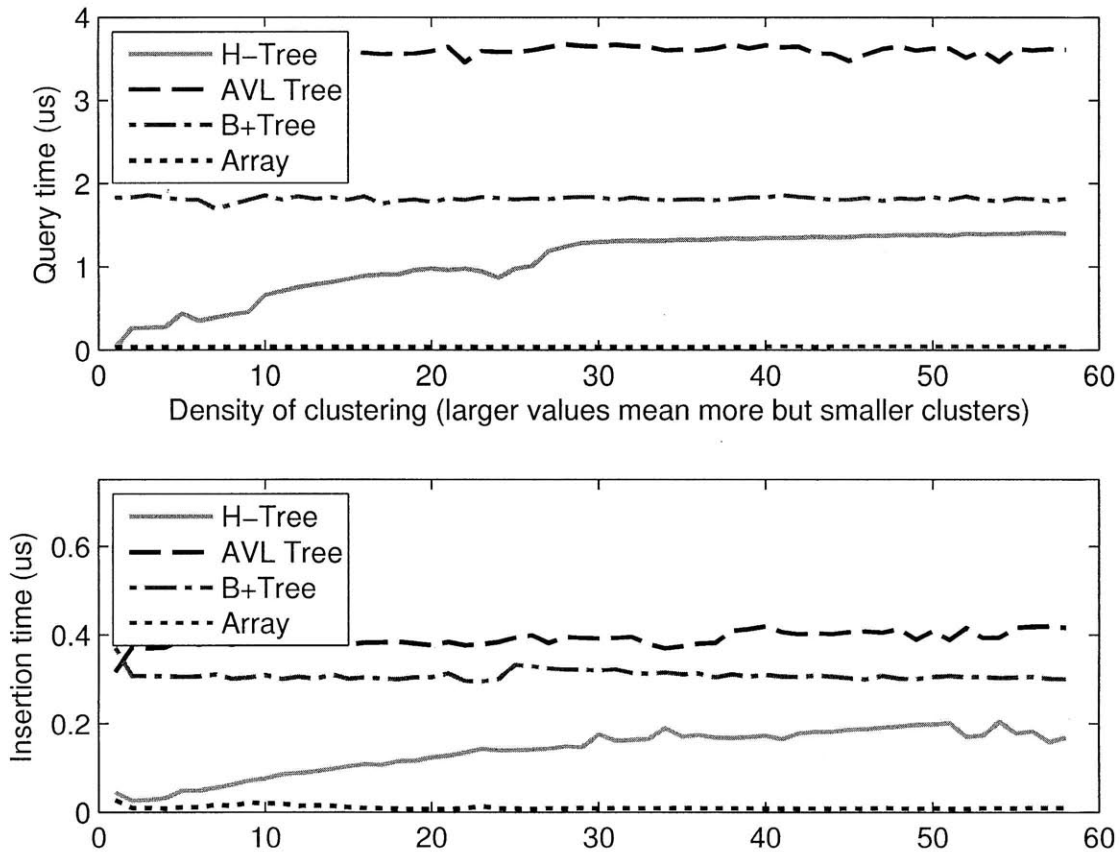


Figure 4-14: Density graph with almost random insertion and large number of keys ( $8 \times 10^5$ ).

the tree stays constant. Figures 4-12 and 4-15 share the same amount of keys being inserted, and present similar query behavior. The same is true of Figures 4-13 and 4-14, except this effect is less apparent, since because the presence of very random insertions causes the tree with more keys more inefficient by creating clusters more inefficiently. Notice that there are, however, small differences that make a larger tree slightly worse than the smaller tree counterparts in both cases. This can be attributed to different cache performance, since lookups in a large array may presumably exhibit worse cache performance than lookups in a smaller array.

However, in the case of insertions, the same amount of nodes is created, but the cost of creating the array is distributed between less keys, increasing the average insertion time per key. Figures 4-12 and 4-15 show this effect, which is clearer with

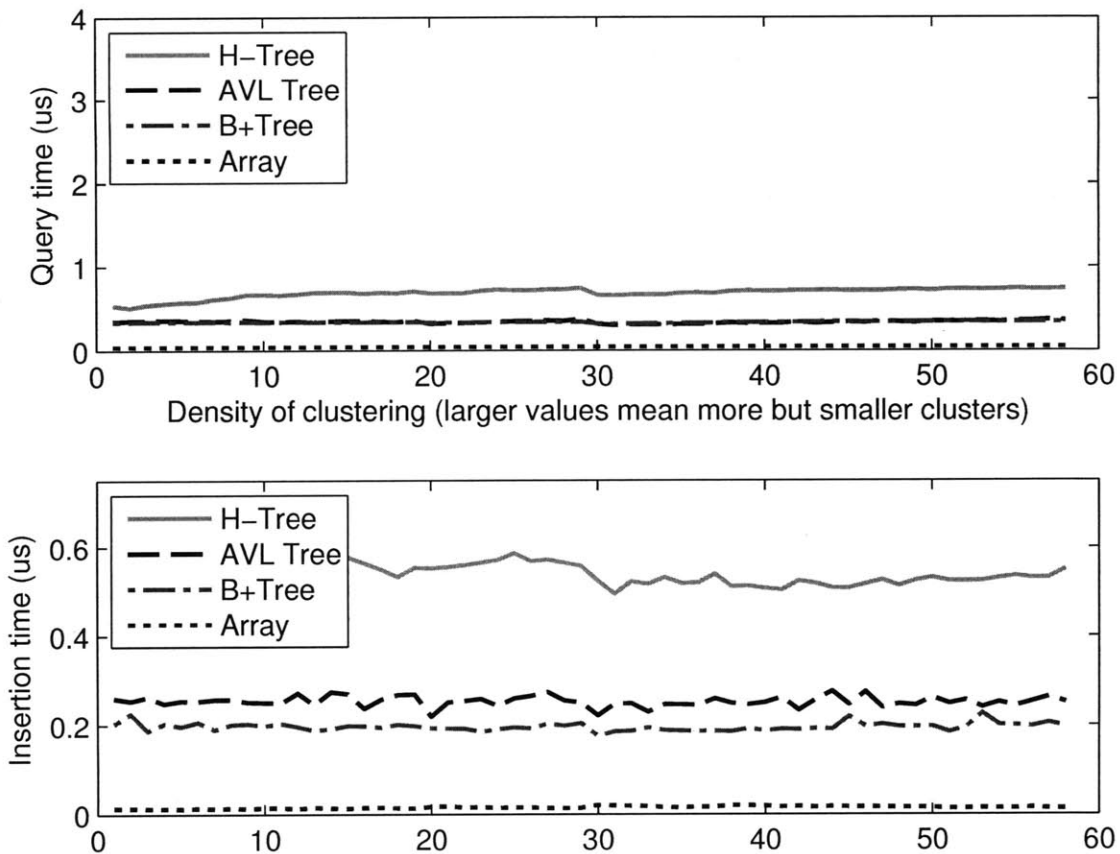


Figure 4-15: Density graph with almost random insertion and small number of keys (80)

few keys. Notice that, in the case of Figures 4-13 and 4-14, the effect is not as apparent because each cluster contains more keys per cluster than the previous pair.

The less randomization, the faster the H-Tree starts behaving like an array in terms of query performance. This is caused by a relatively poorer tree configuration whenever insertions are random, which in turns counters some of the performance gains as the clustering of the keys increases. The query overhead with respect to the B+Tree does not change, since the performance worst case is still the creation of really small arrays as leaves. This effect can be seen clearly by comparing the query performance between Figures 4-12 and 4-14.

However, even in the case of a completely random insertion, there are still gains in terms of query performance with respect to the B+Tree for a densely enough packed

key set. Insertion is slower when there is more randomness, presumably because more key copying is required than when the keys are inserted randomly. This is especially noticeable with few keys, since the average costs of merging and splitting are distributed amongst less keys, which is even more apparent when comparing Figures 4-15 and 4-13 where – as discussed above – the small amount of keys distributes the extra merging costs amongst less keys and increases the average insertion time per key.

## 4.2 Deletion benchmarks

We considered two different scenarios for deletions. The first, random deletions, removes keys from the data structure with equal likelihood. After performing the same routine of insertions described in the section above, we then performed a series of deletions and then queried the structure as before. Secondly, we performed batch deletions, removing large chunks of contiguous keys in order. The query time remained mostly unaffected on both benchmarks, since the H-Tree remained with a similar tree configuration. The average deletion time of the structure is comparable to the average deletion time of the other data structures.

## 4.3 The algorithm parameters

### 4.3.1 $\rho_{min}$

Figure 4-16 shows the result of running a benchmark with some randomization, and varying the threshold  $\rho_{mix}$ . A high value  $\rho_{mix}$  makes the algorithm less eager to create new arrays, causing less merge operations to be performed. This is especially true for key sets with few, large clusters. However, it also causes the algorithm to create a larger number of arrays. We can see that varying the threshold invariably causes more arrays to be doubled.

The other two parameters,  $s_0$  and `split_array()` showed no visible different in the benchmark results. Intuitively, although large values of  $s_0$  create larger initial values,

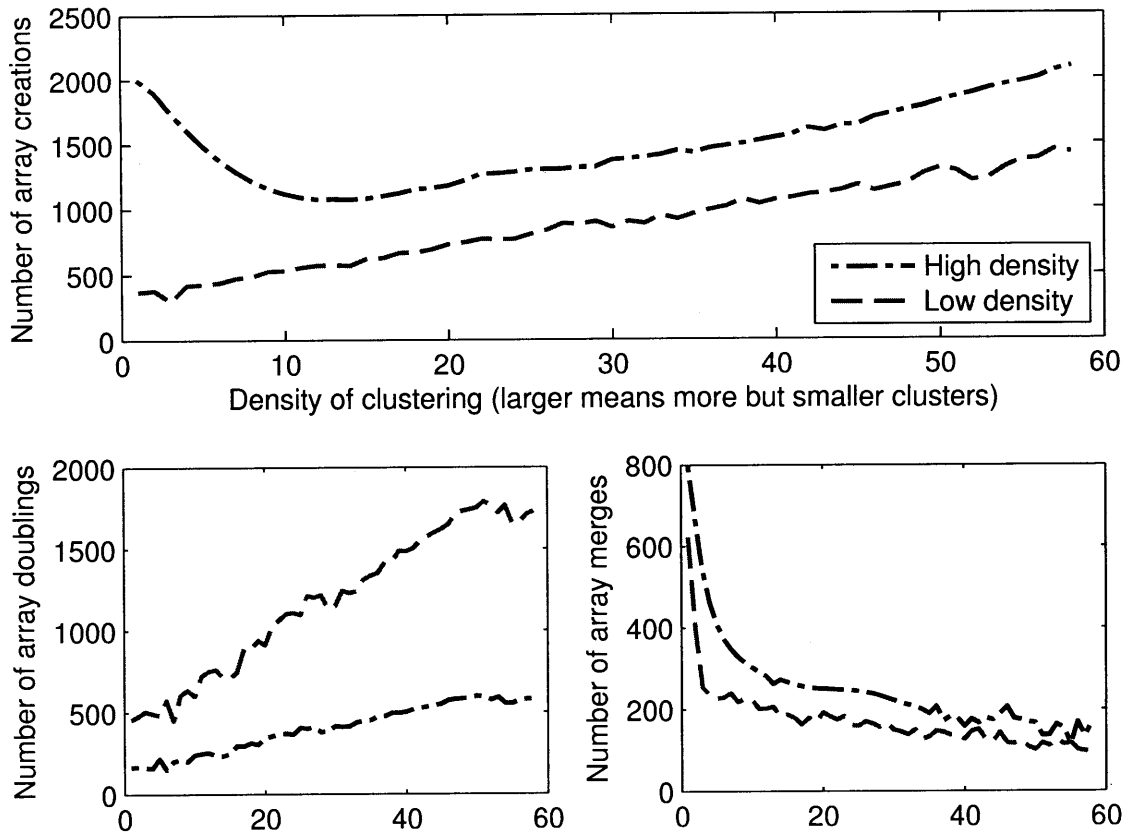


Figure 4-16: Number of arrays created as a function of the clustering factor in an array for a random insertion

the density  $\rho_{min}$  actually dominates the performance of the algorithm in terms of array creations and merges and thus is responsible for the performance differences for the ranges of keys analyzed in this section. Similarly, the `split_array()` split policies showed few differences in the deletion times and subsequent query times. This occurs, presumably, because the deletion time is dominated by the copying of arrays, and not by the lookup of large chunks of empty values.

## 4.4 OLTP Workload: The TPC-C Benchmark

We then tested the performance of H-Tree on a more realistic benchmark with mixed operations: the TPC-C benchmark. We used an implementation of an in-memory database system, which stores each table in memory completely, using different data

structures to hold each table. We then ran the benchmark using different types of data structures to store the different tables.

Some of these tables, such as the table of `Stock` is keyed by a composite key (`stock_id`, `warehouse_id`). Since the H-Tree holds values that are keyed by integers, and not composite keys, we used a mapping from composite keys to integers. For example, we use the function  $f[(\text{stock\_id}, \text{warehouse\_id})] = (\text{warehouse\_id} \times \text{NUM\_STOCK\_PER\_WAREHOUSE}) + \text{stock\_id}$ . Here, `NUM_STOCK_PER_WAREHOUSE` is the maximum amount of stock entries there can be for a given warehouse.

However, some of these tables do not require the existence of such a function. For example, the table of `Items` is keyed simply by the `item_id`. Figure 4-17 shows the benchmark execution times whenever the tables that do not contain composite keys are stored using the H-Tree, compared with other data structures. Figure 4-18 shows the benchmark execution times whenever all tables are stored using the H-Tree, compared with storing the same values using exclusively arrays, and B+Trees. Note that, in order to store the values using an array, we need very tight upper bounds for each of the maximum values of the sub-keys.

Figure 4-17 shows the amount of time required to complete 200,000 new order transactions in the TPC-C benchmark. The H-Tree performs about 5% better than the B+Tree because of the clustering of keys. However, the H-Tree performs about 5% worse than the hashmap and about 15% worse than the array on average.

Figure 4-18 shows the same benchmark, but with all tables indexed by H-Trees, versus an identical version indexed by B+Trees. The H-Tree performs constantly better than the B+Trees for all the number of warehouses tested, with a performance gain of between 20% and 30%.

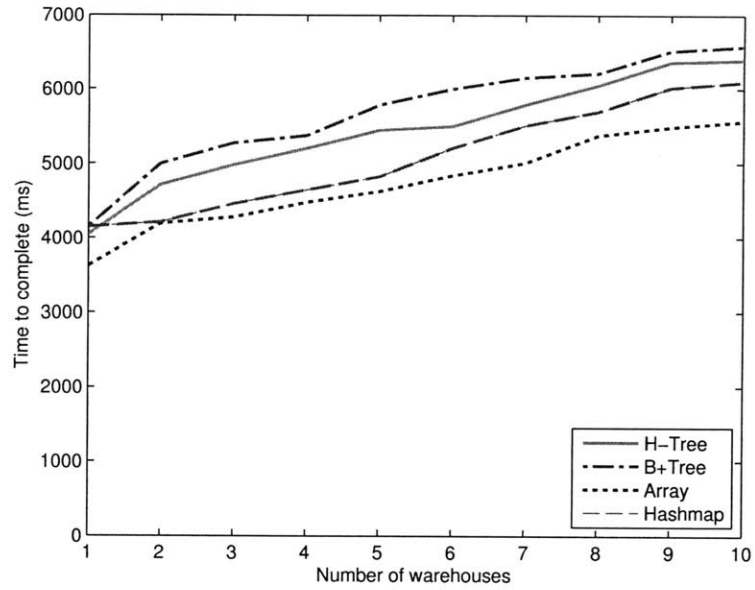


Figure 4-17: Result of TPC-C Benchmark with modified Items table.

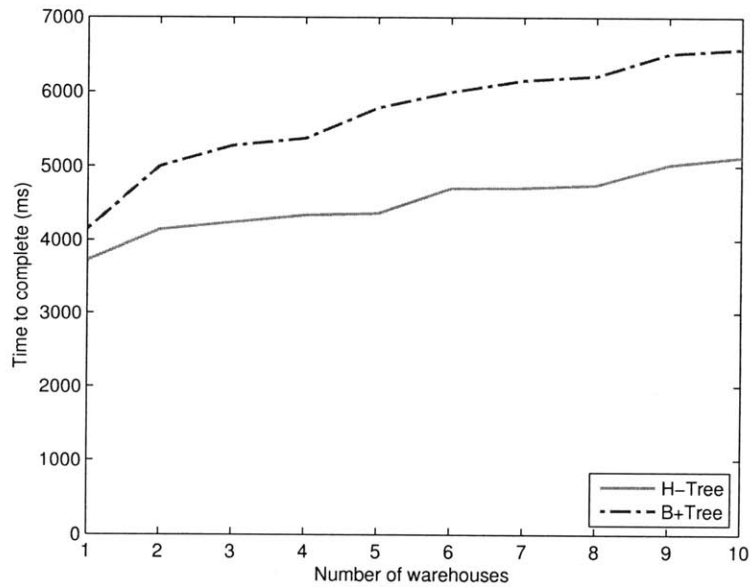


Figure 4-18: Result of TPC-C Benchmark with all tables indexed by H-Trees and all tables indexed by B+Trees





# Chapter 5

## Conclusion

We presented the H-Tree, a data structure for storing values with integer keys. This data structure exhibits its best behavior when it stores a large number of keys that form clusters (intervals of contiguous keys). Its behavior in these idealized conditions is that of an array. When the stored keys are more spread out (and thus form less clusters), the behavior of the data structure is similar to a B+Tree.

The H-Tree performs best when clusters are inserted sequentially or almost sequentially. However, it still performs well when the keys are not inserted in order, outperforming B+Trees even when there are clusters consisting of a few keys.

The data structure performs well under both random and batch deletions, dynamically adjusting its size and ensuring that the allocated size for the data structure is at most a tunable constant factor.

Using the H-Tree as an in-memory indexing structure for an implementation of the TPC-C benchmark sped up the transaction processing time by up to 50% compared to an implementation based on B+Trees. Although other data structures, such as hash tables, provide similar performance on this benchmark, the H-Tree allows for range scan operations, which are not possible on hash tables.

H-Trees are ideal data structures for main memory databases with keys inserted close to sequential order. They can be added to existing systems easily, without significant degradation in performance when used on data sets that are not ideal.



# Appendix A

## Benchmark architecture

For all data structures, we used 32-bit signed integers, and 32-bit pointers to structures. The tests were performed on a GNU/Linux machine with 2 Intel Pentium(R) 4 CPUs at 3.06 GHz and a cache size of 512 MB. The test computer had 2 GB in RAM. We benchmarked four different data structures: H-Tree, an AVL Tree (`std::map`), and a B+Tree (`stx::btree` version 0.8.2). In addition, we created an array with a perfect hash-function. In this case, calculating the array offset for a key consists of a multiplication and an addition. This data structure provides the best-case performance of our data structure.

For each test, we ensure that we test a large enough range of keys so the differences between data structures are visible. We found that showing the range of  $10^5$  keys already highlights the differences between each data structure.

### A.1 Single point benchmarks

In order to support the different combinations of clusters, operation patterns and data structures, we built a generic benchmarking tool for easily adding benchmarks and obtaining the results. The benchmark is a templated C++ function `do_work()` which takes in the following arguments:

1. **List of operations to be performed.** The  $i$ th operation tells the benchmark

how to use the  $i$ th key provided to the benchmark. Example operations are `KEY_INSERT`, `KEY_DELETE`, and `KEY_QUERY`.

2. **List of keys on which operations are to be performed.** These keys are read in sequential order.

As mentioned before, the function is parameterized by two types: a `MapType` and a `BenchType`.

1. A `MapType` specifies the type of data structure to initialize. In order to use the benchmark, for each `MapType`, the user must implement functions that act as wrappers that translate the particular syntax of the data structure to a homogenous set of modifiers for use by the benchmark. These are:

(a) `insert(MapType& struct, Key k, Value v)`

(b) `V find(MapType& struct, Key k)`

(c) `delete(MapType& struct, Key k)`

2. A `BenchType` specifies the type of data structure to initialize. In this case, the user must implement a number of functions that do pre-processing and post-processing on the data structure as required.
3. `init(MapType& struct, List& keys)` . This method initializes the structure. This is useful, for example, when we want to pre-populate the values so that we only measure the query times, and not the time it takes to insert the values into the data structure.
4. `finalize(MapType& struct, List& keys)` This method is called once the benchmark has been performed. The method can be used to collect additional data required by the benchmark, such as the size of the final structure or, in the case of the H-Tree, the number of merges and array creations that happened during the benchmark.

The `do_work` function times the time that passes between the preprocessing and the post-processing stage, and returns it as the return value. All functions are marked as `inline`, to indicate to the compiler that we wish to generate particular benchmarks for each data structure, each with the correct modifiers called directly from the body of the benchmark.

## A.2 Multi-point benchmarks

In order to generate graphs that measure performance varying the set of keys (say, by increasing its size gradually or by increasing some property of the set, like its clustering factor), we can create different `BenchType` classes, each of which calls `do_work()` using a different key set. We created a generic `TwoPhaseBench`, which creates the two-phase benchmarks described above. The first phase consists of timing the insertions of the key set, while the second phase consists of timing the query time once the insertion has been completed. The `TwoPhaseBench` type is parametrized by a key generating function. This function receives a number, from a minimum range to a maximum range (specified at construction of the `TwoPhaseBench` object), and outputs a list of keys and operations. The number can, for example, specify the amount of keys to be generated, or the amount of clustering they should exhibit. By calling this function with successively increasing values as arguments, the `TwoPhaseBench` object can generate a graph consisting of several distinct points.



# Bibliography

- [1] ZIP code statistics. United States Census Bureau. <http://www.census.gov/tiger/tms/gazetteer/zips.txt>, 2002.
- [2] Benchmark overview TPC-C. White paper, Fujitsu. [https://globalsp.ts.fujitsu.com/dmsp/docs/benchmark\\_overview\\_tpc-c\\_jp.pdf](https://globalsp.ts.fujitsu.com/dmsp/docs/benchmark_overview_tpc-c_jp.pdf), October 2003.
- [3] TPC benchmark C standard specification. Standard Specification, Transaction Processing Performance Council. [http://www.tpc.org/tpcc/spec/tpcc\\_current.pdf](http://www.tpc.org/tpcc/spec/tpcc_current.pdf), February 2010.
- [4] Wikipedia database layout. MediaWiki. [http://www.mediawiki.org/wiki/Manual:Database\\_layout](http://www.mediawiki.org/wiki/Manual:Database_layout), 2010.
- [5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
- [6] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 1–8, New York, NY, USA, 1984. ACM.
- [7] Ulrich Drepper. What every programmer should know about memory, 2007.
- [8] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.
- [9] E. B. Fernández, T. Lang, and C. Wood. Effect of replacement algorithms on a paged buffer database system. *IBM J. Res. Dev.*, 22(2):185–196, 1978.
- [10] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, 1992.
- [11] Rodrigo Gonzalez, Szymon Grabowski, Veli Mkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA05) (Greece)*, pages 27–38, 2005.

- [12] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.
- [13] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.
- [14] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.
- [15] Scott T. Leutenegger and Daniel Dias. A modeling study of the TPC-C benchmark. *SIGMOD Rec.*, 22(2):22–31, 1993.
- [16] Witold Litwin. Linear hashing: a new tool for file and table addressing. pages 570–581, 1988.
- [17] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
- [18] Jun Rao and Kenneth A. Ross. Making B+-trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486, 2000.
- [19] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.
- [20] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1150–1160. VLDB Endowment, 2007.