VAST-Tree: A Vector-Advanced and Compressed Structure for Massive Data Tree Traversal

Takeshi Yamamuro, Makoto Onizuka,Toshio Hitaka, and Masashi Yamamuro
NTT Cyber Space Laboratories, NTT corporation
{yamamuro.takeshi, onizuka.makoto, hitaka.toshio,
yamamuro.masashi}@lab.ntt.co.jp

ABSTRACT

We propose a compact and efficient index structure for massive data sets. Several indexing techniques are widely-used and well-known such as binary trees and B+trees. Unfortunately, we find that these techniques suffer major two shortcomings when applied to massive sets; first, their indices are so large they could overflow regular main memory, and, second, they suffer from a variety of penalties (e.g., conditional branches, low cache hits, and TLB misses), which restricts the number of instructions executed per processor cycle. Our state-of-the-art index structure, called VAST-Tree, classifies branch nodes into multiple layers. It applies existing techniques such as cache-conscious, aligned, and branch-free structures to the top layers of branch nodes in trees. Next, it applies the adaptive compression technique to save space and harness data parallelism with SIMD instructions to the middle and bottom layers of branch nodes. Moreover, a processor-friendly compression technique is applied to leaf nodes. The end result is that trees are much more compact and traversal efficiency is high. We implement a prototype and show its resulting index size and performance as compared to binary trees, and the hardware-conscious technique called FAST which currently offers the highest performance. Compared to current alternatives, VAST-Tree compacts the branch nodes by more than 95%, and the overall index size by 47-84% given that there are 2^{30} keys. With 2^{28} keys, it has roughly 6.0-times and 1.24-times throughput and saves the memory consumption by more than 94.7% and 40.5% as compared to binary trees and FAST, respectively.

Categories and Subject Descriptors

C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)Array and vector processors; E.1 [Data Structures]: Trees

General Terms

Algorithms

EDBT 2012, March 26–30, 2012, Berlin, Germany.

Keywords

Massive Data, Tree Traversal, SIMD, Compression

1. INTRODUCTION

Many recent studies tackle data analysis on massive data sets since the database community is urgently demanding data analysis techniques with high performance. The current data sets being targeted include time series data such as workload information of servers, purchasing logs of shopping sites, and user generated logs (e.g., Twitter and Facebook); they are inclined to be larger than they used to be. With the goal of the high-speed analysis of a particular data set that lies within a massive data set, an efficient approach is to filter the original data and subject the results to query processing. A data structure that can realize this kind of exploration is the tree-shaped index created by the techniques of binary trees and B+trees. Although they offer exact and range scans to support queries on massive data sets, they suffer from two shortcomings: large index sizes and execution inefficiency. Table 1 shows the sizes of these indices for 2^{28} to 2^{32} keys (about billion entries)[1]. The sizes of these indices approach 100GiB with 2^{32} keys, and thus they could overflow the regular memory of recent commodity hardware. Because large indices could degrade performance, index size is an important metric.

log_2(# of keys)	28	30	32
binary trees	5.5 (2.5)	22.0 (10.0)	88.0 (40.0)
B+trees	5.6	23.5	89.7

Table 1: Total sizes (GiB) of binary trees and B+trees. The number of keys is described as log_2(# of keys). The values in parentheses show only the sizes of branch nodes.

The traditional techniques are notorious for being inefficient when executed on modern processors. Figure 1 shows the ratios of execution and the counts of instructions on a Xeon X5260 processor with the exact-match scans of binary trees given 2^{22} to 2^{28} keys. It indicates that the stall time and the conditional branch penalties occupy up 60% to 80% of the total. As a result, their instructions per cycle (IPC), which represents efficiency, are held to low values (less than 0.3 or so). A previous study [11] identified the

[1]We assume that each entry is around 0.1 to 100KiB, and the total number of entries constituting 100TiB of data is about 2^{28} to 2^{32}.

causes of inefficiency as conditional branch penalties, low cache hits, and Translation Look-aside Buffer (TLB) misses. TLB represents a cache mechanism that is expected to hold an often-used translation table that links physical and logical addresses in memory. These translations are executed per page; a page is a physically-consecutive byte sequence in memory. We assume that the cache refers to a L2 one in the same manner as other papers [11], because authors in [1] suggested that the optimization of L2 caches could be better than that of L1 ones empirically.

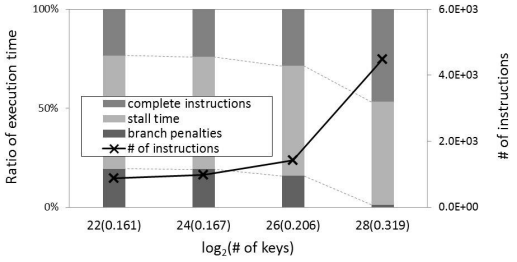

Figure 1: Ratios of execution time and counts of instructions for exact-match scans on a Xeon X5260 processor. The values in parenthesis indicate IPC in the processor.

Many techniques have been proposed to break these bottlenecks on machines with multi-cores and shared memory [3]. One solution, the tree-shaped data structure for indices, called FAST [14], was proposed to solve these problems and to offer improved performance on modern processors. FAST not only holds down cache misses, TLB misses, and conditional branch penalties, but also improves comparison processing on branch nodes with SIMD instructions, which has the ability to subject multiple data streams to the same processes in a single clock cycle. The authors assume that FAST is designed for in-memory structures and read-only workloads, and therefore FAST does not support incremental updates. However, there are three technical issues below;

- Aligning the structure of cache lines and pages for efficiency makes index size much larger, because there are many padding areas inside branch nodes.
- The simultaneous key comparison with SIMD instructions is limited to three, because FAST uses a 32bit space for keys with padding areas and SIMD instructions process 128bit data.
- FAST does not consider the compression of leaf nodes, i.e. uncompressed keys. Therefore, when data set size is large, leaf node size become something of a problem.

Our goal is to design a compact and processor-friendly data structure for massive data sets with the same assumption of FAST: in-memory and read-optimized structures. Our proposed technique, VAST-tree, adaptively applies lossy compression to the keys in branch nodes in order to dramatically reduce the size of branch nodes; VAST-tree classifies branch nodes into multiple layers and applies stronger the lossy compression to the branch nodes as their layer gets deeper in the tree. This adaptive lossy compression approach achieves a good balance of the additional decoding costs during tree traversal and the size of the index. And also, this lossy compression increases the number of the simultaneous key comparisons that can be supported by SIMD operations. Although there are many benefits to the adaptive lossy compression, it causes incorrect key comparisons during tree traversal, so VAST-tree needs to correct the errors by scanning at leaf nodes sequentially after the tree traversal. For confirmation, we model these errors, and evaluate the resulting model with actual errors. Moreover, a processor-friendly compression technique is applied to leaf nodes to be compact. As a result of these optimizations, assuming the target of 2^{30} keys, VAST-Tree compacts branch nodes by more than 95%, and overall index size by 47-84% compared to current alternatives. For 2^{28} keys, its throughput are roughly 6.0-times and 1.24-times those of binary trees and FAST, respectively.

Given the popularity of multi-cores/many-cores, the memory consumption of algorithms used is one of key factors determining scalability [5][14][15], that is, the upper limit of memory bandwidth could restrict the throughput directly when increasing increasing the number of cores. Our compression technique significantly reduces memory consumption; an experiment shows that our technique saves the memory consumption with 2^{28} keys by more than 94.7% and 40.5% as compared to binary trees and FAST, respectively. Our paper makes the following contributions;

- We propose a state-of-the-art index structure called VAST-Tree; it realizes a compact and efficient data structure for indices when applied to massive data sets.
- We implement a prototype and verify its efficiency and effectiveness by experiments on synthetic and realistic data sets. Public Timeline data in Twitter were collected via the Twitter API from May, 2010 to Apr., 2011, and used as realistic data.
- A error caused by the lossy compression in VAST-Tree is modeled, and we evaluate its accuracy as compared to the actual penalties.
- A efficient technique is proposed to handle the worst errors by using the error model, and it saves the processing time by 7.1-18.5%.

This paper is organized as follows; Section 2 describes the handling of comparisons on branch nodes with SIMD instructions, and proposes a state-of-the-art compact and processor-friendly structure called VAST-Tree (Section 3) for massive data sets. In Section 4, we prototype VAST-Tree, and evaluate its size and performance as a comparison to other techniques. Finally, we model the errors caused by the use of adaptive compression, and analyze the consumption of memory bandwidth during tree traversal. Section 6 discusses related works and Section 7 concludes our findings.

2. TREE TRAVERSAL WITH SIMD

Our approach employs SIMD instructions to compare multiple keys on branch nodes so as to exclude conditional branches ('if-then' paths) and thus suppress their penalties along with FAST. This section starts with Figure 2 to explain the usage of SIMD instructions on tree traversal. First, binary trees are segmented into sub-trees (depicted by triangles). The sub-trees are called SIMD blocks, and a single SIMD instruction can compare all keys in a SIMD block in a single clock cycle. FAST assumes that the sizes of SIMD

registers and comparison keys are 128bits and 32bits, respectively, and recent mainstream processors incorporate 128bit SIMD registers (SSE on Intel's processors, and 3DNOW! on AMD's processors). Instead of 'if-then' paths, addition and multiplication instructions are used to traverse trees based on the results of SIMD instructions to exclude branch penalties. We explain how this traversal works in detail given the layout of comparison keys in memory. SIMD blocks (multiple values in a parenthesis represent a SIMD block) in Figure 2 are arranged in breadth first order on physically-consecutive memory as shown below.

[34, 78, 91], [2, 11, 23], [35, 39, 49], [80, 87, 88], ...

The underlined values above are the traversal traces (bold triangles in Figure 2), and 79 is the search key in this example. First, the leftmost values (34, 78, and 91) are loaded into an SIMD register (34, 78, 91, x), compared to the search key in a other SIMD register (79, 79, 79, x) as shown in the top-left part in the example of Figure 2, which yields the output of returned values (1, 1, 0). This SIMD instruction returns 1 if its search key is greater than or equal to the comparison keys, otherwise 0. It determines the next SIMD block to be compared based on this returned value using a lookup table. The lookup table (the top-right table in Figure 2) holds the relationship between returned values and offsets to next SIMD blocks, and is predefined for tree traversal. Therefore, the offset block obtained from the lookup table corresponds to the third SIMD block (80, 87, and 88) from the left on the second level under the root as the next position for comparison. Because the size of each SIMD block is 12B (three 32bit keys), the offset byte to the next position is 36 (12B multiplied by 3).

As explained above, FAST is limited by three key comparisons in a single SIMD instruction. Our approach is that a compression technique applied to branch nodes makes each key size small, and enables the more number of keys to be loaded into a SIMD register. As a result, it increases the number of simultaneous comparisons.

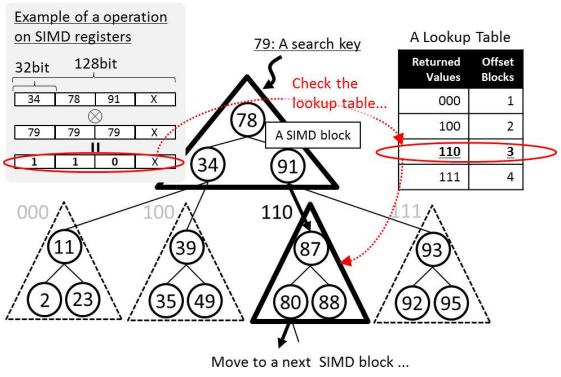

Figure 2: An example of tree traversal with SIMD instructions. This instance shows that 128bit SIMD registers can load four 32bit integers, and three keys can be compared simultaneously.

3. PROPOSAL: VAST-TREE

3.1 Designing Data Structure

We present a tree-like data structure for indices, called VAST-Tree, which is compact and processor-friendly. VAST-Tree compresses comparison keys inside branch nodes and keys on leaf nodes; lossy compression is applied to the comparison keys to strongly reduce the bit number, while keys on leaf nodes undergo lossless compression. The comparison keys need to be integers of certain size such as 32bits and 64bits, and each block consists of multiple comparison keys. In what follows, we assume that the size of comparison keys is 32bits. Figure 3 shows an architectural overview of VAST-Tree. The labels in Figure 3 such as H, SH, CLH, PH, and CH, mean the heights of each layer or each block. For example, SH represents the height of the SIMD blocks. These are noted as the height of binary trees, and for example, the height of the SIMD blocks in Figure 2 is 2.

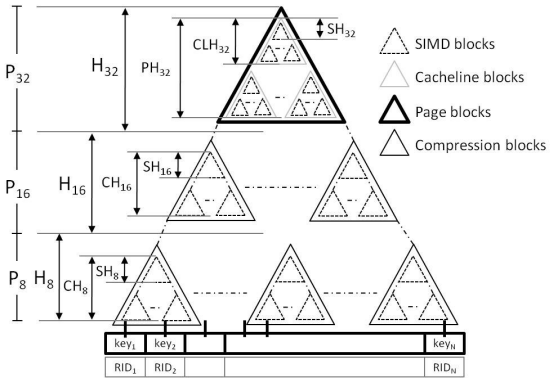

Figure 3: An overview of VAST-Tree. The top layer (P_{32}) of VAST-Tree uses FAST techniques, and VAST-Tree compresses the middle and the bottom layers (P_{16} and P_8) of the trees. The heights of these layers are H_{32}, H_{16} and H_8, respectively. Moreover, keys in leaf nodes (key_1, key_2, ..., key_N) are compressed by using lossless compression.

3.1.1 Branch Nodes

First, we overview the design of branch nodes in VAST-Tree. VAST-Tree compares multiple keys in branch nodes by using SIMD instructions. Moreover, our proposed lossy compression technique in the bottom of trees enables more keys to be compared simultaneously, which improves data-level parallelism. In particular, the branch nodes in VAST-Tree consist of three layers as follows[2];

- P_{32}, it applies FAST without any change, and compares ($2^{SH_{32}}$-1) keys simultaneously.
- P_{16}, it compresses 32bits to 16bits and ($2^{SH_{16}}-1$) keys are compared simultaneously.
- P_8, it compresses 32bits to 8bits and ($2^{SH_8}-1$) keys are compared simultaneously.

Lossy compression is applied to each compression block, and block heights (CH_{16} and CH_8) depend on the size of cache lines and pages in a way similar to FAST. However, this lossy technique in branch nodes could cause incorrect

[2]If we assume that key size is 64bits, there would be four layers: P_{64}, P_{32}, P_{16}, and P_8.

comparison through branch nodes, and finally the incorrect retrieval of keys at the end of tree traversal. These errors are corrected by using the keys in leaf nodes after traversal in branch nodes. As a result, VAST-Tree is a parametric technique with three variables: H_{32}, H_{16}, and H_8. These values must be assigned according to the size and the performance of VAST-Tree. Section 3.2.1 shows how to compress the keys in P_{16} and P_8, and the following section details tree traversal in these compression blocks. We then show error correction after tree traversal in Section 3.2.3.

Block Alignment

Aligning blocks in memory is important in improving performance [14][25]. Layer (P_{32}) of FAST is aligned recursively with two blocks: cache line blocks and page blocks (heights are CLH_{32} and PH_{32}). Of course, these alignment techniques need many padding areas inside the cache lines and the pages. Therefore, alignment in the middle and the bottom layers of trees, which includes a lot of branch nodes, increases the total index size. Due to this trade-off between performance and index size, our technique aligns cache lines and pages only for the top indices, and in P_{16} and P_8, these elements such as compression blocks and SIMD blocks are SIMD-length aligned (SIMD register length is assumed to be 128bit). Figure 4 overviews this alignment, which can minimize total index size for a small drop in performance.

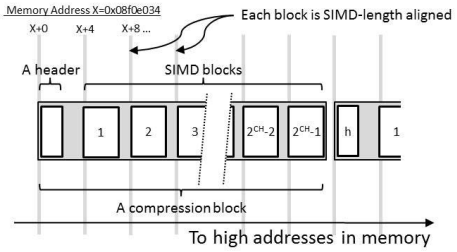

Figure 4: Alignment of compression blocks and SIMD blocks. Each compression block and its internal elements are SIMD-length aligned in memory. A compression block header will be described in Section 3.2.

3.1.2 *Keys on Leaf Nodes*

A lossless compression algorithm is applied to keys on leaf nodes. The keys (key_1, key_2, ..., key_N) on leaf nodes are held as an ascending-order array with physically-consecutive addresses in memory, that is, the array of keys, and the sequence of RIDs shares the same arrangement. VAST-Tree applies a processor-friendly compression method called P4Delta [28] to the sequence of keys. This technique has the ability to pack more keys into a single cache line, while reducing the cost of error correction to hold down the number of required cache lines. Section 3.3 will explain how the keys are compressed using P4Delta in detail.

3.2 Optimization of Branch Nodes

3.2.1 *Lossy Compression*

As described before, compression blocks use lossy compression in a manner similar to early, well-known, techniques called "prefix truncation" and "suffix truncation" in B-trees [2][9], which skip common bits from prefixes and remove bits from suffixes, respectively. Let $V = [V_1, V_2, \ldots, V_n]$ be the keys that are to be assigned to a compression block and V is sorted in ascending order. V is compressed as follows:

1. $W = [0, V_2 - V_1, V_3 - V_1, \ldots, V_n - V_1]$ is obtained by subtracting the minimum value, V_{min}, from each key in V.

2. K-bits are extracted from each value in W, starting from the first bit B where 1 occurs in the bit expression of the maximum value in W. K is equal to 16 in P_{16}, and 8 in P_8.

V_{min} and $(B - K)$, V_{shift}, are recorded in the header of the compression block. The compressed keys are recorded in the body part of the compression block. Figure 5 shows an example of the lossy compression of eight values in P_8.

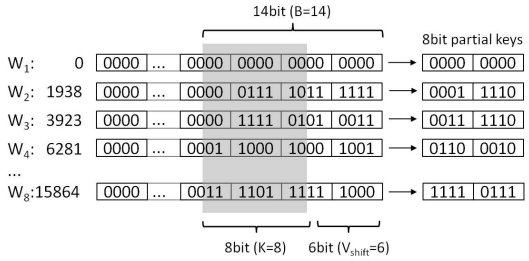

Figure 5: An example of the lossy compression on the eight values (W_1, W_2, ..., W_8) of P_8. The 8bit sequences with gray background are used as comparison keys in VAST-Tree.

Note that this compression method is lossy to obtain higher compression rates. During the search phase, the VAST-Tree algorithm must detect traversal errors caused by the use of the lossy compression. We will describe how to fix the errors in key compression in leaf nodes in Section 3.2.4.

3.2.2 *Tree Traversal on Compression Blocks*

The VAST-Tree algorithm traverses a tree block by block, and then finally checks and corrects keys if needed. This section describes tree traversal in compression blocks, and the following section will explain how to fix key errors. Tree traversal in compression blocks is done via the headers and comparison keys inside block bodies. Because these comparison keys are compressed, search keys are transformed in a similar rule to permit comparison. Search key $skey$ is transformed as follows:

1. VAST-Tree extracts V_{min} and V_{shift} from the compression block header.

2. It subtracts V_{min} from $skey$, and shifts $skey$ to the right by V_{shift}-bits.

This transformation reduces search key bit-length to K-bits. Specifically, Section 3.2.7 will show the pseudo code for tree traversal in P_8.

3.2.3 *Correction of Traversal Errors: Δw*

The lossy compression causes incorrect traversal in trees, which means that incorrect key offsets could be returned at the end of tree traversal. We assume that the 8bit prefix of comparison key "3220 (1100 1001 0100 in binary digit)" is

treated as compressed key "201". If the search key is "3219 (1100 1001 0011 in binary digit)", a traversal error occurs due to the lossy compression. The offset of correct keys from the top is defined as w. And a traversal error is defined as Δw; the offset from the correct key to the incorrect key. For example, the correct search key is $key[w]$, so the incorrect key is $key[w + \Delta w]$ right after tree traversal. The error is detected by the difference between the search key and the keys in leaf nodes. If they are different, VAST-Tree scans the leaf nodes sequentially until Δw becomes 0. Section 3.3.3 shows the pseudo code for error correction.

3.2.4 Compression by Using Error Correction

This above error correction allows VAST-Tree to be more compact; Its technique for optimization is to remove bottom compression blocks whose comparison key height is under SH_8. This optimization could cause additional errors, that is, SH_8 at most. However, our various experiments showed that this had virtually little impact on performance. As a result, this optimization is applied to all patterns of VAST-Tree in the evaluation of Section 4.

3.2.5 A Pseudo Code for Building VAST-Tree

A pseudo code for building VAST-Tree is shown in Algorithm 1. First, the top layer of VAST-Tree is constructed according to FAST (line 9) [14]. Next, the compressed layers (P_{16} and P_8) of VAST-Tree are made up of three loop structures; the outer loop is repeated in the number of compressed layers $rnum$ (line 11-22), the middle loop is to construct compression blocks (line 12-21), and the inner loop is to extract sub-trees of compression blocks according to binary trees and store these blocks on disk (line 13-19). As for extracting the sub-trees of compression blocks explained in Section 3.1, it gets (2^{CH_n}-1) keys by using $extract\_keys()$ (line 14), and then permutes them (line 14). SIMD blocks in compression blocks are arranged in breadth first order on physically-consecutive memory as explained in Section 2. Finally, it extracts the minimum value, V_{min}, and the amount of shift, V_{shift}, and stores the header (V_{min} and V_{shift}) and compressed keys (line 17-18) as described in Section 3.2.

Algorithm 1 Pseudo code for building VAST-Tree

```
 1: /*
 2: key[]: Array of keys
 3: rnum: Number of compressed layers: P16 and P8
 4: Hn: Height of a n-th compressed layer
 5: CHn: Height of a n-th compression block
 6: SHn: Height of a SIMD blocks in CHn
 7: */
 8: height = 0;
 9: build_FAST(key, H32);
10: height = height + H32;
11: for i ← 1 to rnum do
12:   for j ← 1 to Hi/CHi do
13:     for k ← 1 to 2^height do
14:       s[] = extract_keys(key, H32, CHi, SHi, j, k);
15:       Vmin = extract_Vmin(s[]);
16:       Vshift = extract_Vshift(s[]);
17:       lossy_compress_keys(i, s[]);
18:       store_VAST(Vmin, Vshit, s[]);
19:     end for
20:     height = height + CHi;
21:   end for
22: end for
```

3.2.6 Pseudo Code for Tree Traversal

A pseudo code for tree traversal of P_8 is shown in Algorithm 2. First of all, the minimum value, V_{min}, and the amount of right shift of search keys, V_{shift}, are extracted from the header of a compression block at current position, $cpos$, and search key $skey$ is transformed into the compressed key, $skey'$, so that it can be compared with the keys of the compression block (line 10-12). Next, VAST-Tree traverses the compression block using SIMD operations (line 14-18). The output gained upon completion of the current block indicates the next position of SIMD blocks or compression blocks (line 16-17, and line 19-20). Here, $sblk\_sz(SH, i)$ returns the total size from the root to i of SIMD blocks whose height is SH, and $cblk\_sz(i)$ returns the total size of root to i of VAST-Tree. Finally, offset $kpos$ from the head of keys is returned after tree traversal as described in Section 3.2.4 (line 22). This offset is expected to include an error, $w + \Delta w$, which must be detected as shown in Section 3.2.4.

3.3 Optimization of Leaf Nodes

In this section, we detail the structures of keys on leaf nodes. The processor-friendly and lossless compression technique known as "P4Delta" is applied to these keys. Later, we show the pseudo code that can correct the inherent errors of VAST-Tree.

3.3.1 P4Delta

We apply the processor-friendly compression algorithm called P4Delta [28] to a d-gap sequence of keys (ascending order) on leaf nodes. Given a sorted list of keys key, a list of d-gaps d is defined as follows: $d[1] = key[1]$, $d[i] = key[i] - key[i-1]$, $i > 1$. These d-gaps like time series data sets are inclined to be much smaller than that of original data. P4Delta is known to be suitable for these small integers, and is able to achieve a good balance between the compression ratio and the decompression speed. This technique classifies integers in a sequence into two groups: coded or exceptions. Most integers are small and are regarded as coded. It finds the smallest b so that most integers are not greater than 2^b; these integers are then stored as b-bit entries. Specifically, if k consecutive integers are compressed with P4Delta, these integers are packed within a list of $\lceil kb \rceil$ bits. On the other hand, bigger integers (exception values) are left uncompressed, and lie on the tail of the list. Details, omitted due to paper limit, can be found in [28].

Figure 6 overviews the compression of keys on leaf nodes. It is necessary to randomly access the offset of keys on compressed leaf nodes, as the offset is returned after tree traversal in branch nodes. Therefore, k consecutive keys are compressed into a single chunk, and each chunk consists of the minimum quantity of these k keys, coded values of d-gaps, and exception values. The minimum value is located at the head of each chunk so as to quickly decompress a part of a key list for error correction. The chunk size needs to be set to a multiple of the size of cache lines; each chunk is aligned with cache lines so as to minimize cache misses. Moreover, k is a compression parameter on keys on leaf nodes, and needs to be determined to make the total leaf size small. We present a new way to choose optimal parameter k in the

Algorithm 2 Pseudo code for tree traversal in P_8

```
 1: /*
 2: spos: SIMD block offset inside a SIMD block
 3: cpos: Compression block offset in a binary tree
 4: kpos: Key offset after tree traversal
 5: chead: Head position of a current compression block
 6: ssz: Single SIMD block size
 7: csz: Single compression block size
 8: */
 9: for i ← 1 to H8/CH8 do
10:    V_min = extract_V_min(cpos);
11:    V_bit = extract_V_bit(cpos);
12:    skey' = lossy_compress_skey8(skey, V_min, V_bit);
13:    spos = 0, chead = cpos;
14:    for j ← 1 to CH8/SH8 do
15:       res = compare_SIMD8(skey', cpos);
16:       spos = spos × 2^SH8 + lookup(res);
17:       cpos = chead + spos × ssz + sblk_sz(SH8, j);
18:    end for
19:    cpos = cpos × 2^CH8 + spos;
20:    cpos = cpos × csz + cblk_sz(H32 + H16 + i);
21: end for
22: kpos = coffset;
23: return kpos;
```

following section.

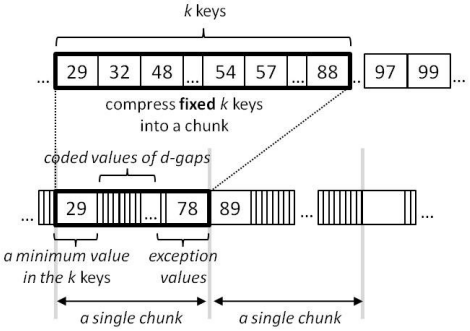

Figure 6: Compression of keys in leaf nodes. Upper array is a uncompressed list of keys, and lower array is a compressed one. VAST-Tree compresses k consecutive keys into a single chunk.

3.3.2 Parameter Decision

We explain here the key difference from the naive P4Delta algorithm. As explained before, we need to choose parameter k so as to minimize its compression ratio. VAST-Tree uses a bisection method to find parameter k. Let chunk size be CS bits, and key size be KS bits. The details are as follows;

1. lk is set to $\lceil CS/KS \rceil$, and rk is set to CS.
2. VAST-Tree checks whether all $\lceil (lk + rk)/2 \rceil$ consecutive keys could be compressed in a single chunk.
3. If not, rk is set to $\lceil (lk + rk)/2 \rceil$, otherwise lk is set to it, then return to 2. Repeat until lk is equal to rk.
4. lk (or rk) is employed as the value of parameter k.

3.3.3 Pseudo Code for Error Correction

A pseudo code for error correction is shown in Algorithm 3. A initial byte position, *cpos*, on compressed leaf nodes is calculated by the return value, *kpos*, of tree traversal (line 9). *kpos* is expected to be w, however, it may be $w + \Delta w$; Δw is caused by the traversal errors as explained Section 3.2.3. It detects whether the current position is correct using the minimum values inside chunks (line 10), and if incorrect, it corrects the position using the loop structure (line 10-13) until the minimum value exceeds *skey*. After error correction, it decompresses a chunk and returns the corresponding RID (line 14-15).

Algorithm 3 Pseudo codes for error correction

```
 1: /*
 2: skey: Search key
 3: RID[]: Array of RIDs
 4: CS: Size of chunks as bytes
 5: k: Number of keys in a single chunk
 6: kpos: Key offset after tree traversal
 7: cpos: Byte position on compressed leaf nodes
 8: */
 9: cpos = ⌈kpos/k⌉ * CS;
10: while val > skey do
11:    val = get_minimum(cpos);
12:    cpos = cpos - CS;
13: end while
14: kpos = decompress_P4Delta(skey, cpos);
15: return RID[kpos];
```

4. EVALUATION

In our experiments, we used synthetic and realistic test data sets to cover different key distributions. First, a sequence of keys that follow a Poisson distribution is employed as a synthetic data set because VAST-Tree is assumed to be suitable for time series data, and these are typically expressed by this distribution. The parameter is noted in $1/\lambda$, and it is set to 16 unless stated. In addition, Public Timeline data in Twitter were collected via the Twitter API from May, 2010 to Apr., 2011, and employed as realistic data. *Ids* and *Timestamps* of the posts in Public Timeline data were extracted from these logs. The number of these entries was 36,068,948 (nearly equal to 2^{25}). Figure 7 shows the distribution of the first 10 smallest d-gaps in these data sets, and it is clear that most parts of the keys are duplicated.

Table 2 shows the parameters of VAST-Tree used throughout our experiments. These were decided based on the following properties of the processors we used; cache line size was 64B, page size was 4KiB (CentOS v5.5 with kernel-2.6.18-194), and SIMD register width was 128bits. The performance evaluation was done on servers with Xeon X5670 processors (6 cores with Intel Hyper Threading and 31.8GiB/s maximum memory bandwidth), and 16GiB of memory. Oprofile was used for processor profile analysis. Oprofile is a processor profiling tool for Linux kernels, and has the ability to observe counters for performance monitoring inside processors. The current version of oprofile is 0.9.6. Because this oprofile version does not support Xeon X5670 processors, profiling actions such as the analysis of execution ratio were executed only on Xeon X5260 processors (2 cores and the 21.2GiB/s maximum memory bandwidth). These codes

(binary trees, FAST, and VAST-Tree) in our experiments were written in C and complied by GNU Compiler Collection v4.1.2 with an option "-O3".

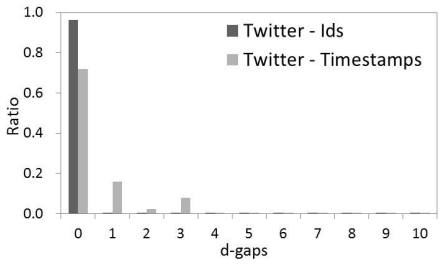

Figure 7: Distributions of the first 10 d-gaps for realistic data used in our experiments. These data were collected from May, 2010 to Apr., 2011 from the Public Timeline of Twitter.

Variables	Height	Variables	Height
SH_{32}	2	SH_{16}	3
CLH_{32}	4	CH_{16}	7
PH_{32}	8	SH_8	4
		CH_8	8

Table 2: Parameters for VAST-Tree in our experiments. These parameters are based on the sizes of cache lines and pages, and the width of SIMD registers.

4.1 Performance Evaluation of VAST-Tree

4.1.1 Compression Ratios

Table 3 and Table 4 show the compression performance of VAST-Tree as compared to the previous techniques such as binary trees and FAST. Table 3 shows that the total size of branch nodes with various VAST-Tree parameters: H_{32}, H_{16}, and H_8. Note that H_8 is calculated by "log_2(# of keys) - (H_{32} + H_{16})", and only synthetic data were used because these sizes do not depend on the key distribution. This table indicates that VAST-Tree has high compression ratios in branch nodes; at the key number of 2^{30}, the branch size of VAST-Tree is less than 5.0% compared to these alternatives[3]. These parameters (H_{32}, H_{16}, and H_8) need to be determined to minimize the probability of errors in tree traversal as explained in Section 3.2.3. On the other hand, the values of H_{16} and H_8 need to be made large from the viewpoint of space complexity. Therefore, subsequent experiments used the parameters which had the smallest errors we found in the experiments. Specifically, with 2^{24} keys these parameters are H_{32}=8 and H_{16}=6, otherwise these are H_{32}=8 and H_{16}=12 in following experiments.

Table 4 shows the compression ratios of leaf nodes and the numbers of keys packed in a single chunk, CK, for various leaf sizes. Note that, if the sizes of keys and chunks are 32bit and 64B, respectively, CK becomes 16 in uncompressed keys. Because, in turn, these compression ratios depend on the distribution of keys, we show the results taken from a variety of test data sets. These results mostly follow a previous work [28], which indicated that decreasing the offsets raises the compression ratio. Moreover, increasing the chunk size improves the compression ratios. As a result of that, VAST-Tree compacts the overall index size (branch and leaf nodes) by 47-84% given that there are 2^{30} keys. However, raising the compression essentially leads to more decoding P4Delta penalties (and thus processing penalties) in leaf nodes as explained in Section 3.3.1. Therefore, there is trade-off between VAST-Tree performance and compression ratios on keys. In the experiments in Section 4.1.2, we used the chunk size of 64B because this yielded the highest VAST-Tree throughput. We compare the results gained from binary trees, FAST, and VAST-Tree; the ratios of execution time, throughput, and the distribution of error. Because VAST-Tree obviously depends on the distribution of keys, we show their results by using the realistic data sets.

[3]Although the rounded sizes of the first and third entry from the top seem to be the same in Table 3, the actual sizes are slightly different. Because the layer of H_{32} is small, these sizes hardly change between H_{32}=0 and H_{32}=6.

log_2(# of keys)	24	26	28	30
VAST-Tree (0, 6)	0.00449	0.00449	0.130	0.130
VAST-Tree (8, 0)	0.00225	0.0186	0.0186	0.519
VAST-Tree (8, 6)	0.00449	0.00449	0.130	0.130
VAST-Tree (8, 12)	0.00248	0.00337	0.00337	0.251
FAST	0.252	1.25	1.25	64.3
binary trees	0.156	0.625	2.50	10.0

Table 3: Total branch node sizes (GiB) of VAST-Tree, binary trees, and FAST. The values in parentheses show H_{32} and H_{16}. The branch nodes of VAST-Tree are obviously highly compressed by using the lossy compression.

Chunk Size	64B	128B	256B
$1/\lambda = 16$	.142(113)	.133(240)	.129(497)
$1/\lambda = 64$	.225(71)	.219(151)	.206(311)
Ids	.219(71)	.211(151)	.199(321)
Timestamps	.500(32)	.320(100)	.285(311)

Table 4: Compression ratios of leaf nodes (VAST-Tree). The values in parenthesis indicate the number of keys, CK, in a single chunk.

4.1.2 Performance Evaluation of VAST-Tree

Figure 8 shows that the ratios of execution and the counts of instructions with exact-match scans of the three techniques: binary trees, FAST, and VAST-Tree. With regard to VAST-Tree, we tested the two settings; uncompressed keys and compressed keys in leaf nodes (noted as "w/o P4Delta" and "w P4Delta", respectively). In all cases of VAST-Tree, the total of stall time and branch penalties caused by binary trees decreased by 72.8% to around 50%, and, as a result, IPC of VAST-Tree is better than that of binary trees. Moreover, the most IPC of VAST-Tree is superior to that of FAST. And also, we found that the distribution of keys and the leaf compression make an impact on IPC and the total of instructions in VAST-Tree, and it tends to make both factors high as compared to "VAST-Tree w/o P4Delta".

VAST-Tree tends to need many instruction counts for tree traversal, because it incurs additional cost when traversing branch blocks, error correction, and decompressing keys in leaf nodes. In particular, the last decompression increases the count as shown in Figure 8. Basically, if memory bandwidth is not restricted by an upper limit, it is better to keep IPC high while suppressing the count of instructions. Hence, the throughput performance is expected to reflect these observations for tree traversal as shown in Figure 9 and Table 5. Although the size of "VAST-Tree w P4Delta" is definitely compact, its throughput performance is slightly inferior to FAST with 2^{24} and 2^{28} keys. "VAST-Tree w/o P4Delta" overcomes FAST with all conditions due to its compact branch representation and no penalty of decoding in leaf nodes. And also, throughput with realistic data sets follow the observation in Figure 8, and the throughput of *Ids* is higher than that of *Timestamps* because of its dense distribution as shown in Figure 7.

Finally, the distribution of the errors Δw are shown in Figure 10 and Figure 11. These figures are for the evaluation conducted with different key distributions and the various numbers of keys, respectively, and the values in parentheses show the averaged and worst error amounts. These errors cause high penalties with regard to the counts of instruction and memory bandwidth, so it is critical to minimize these errors as much as possible. Although the compression in leaf nodes requires the more counts of instructions as explained before, it is expected to minimize the required number of cache lines. For example, if the error correction can be completed in a single chunk, it causes no additional penalty for memory bandwidth. Because the decoding cost is basically lower than the penalty of cache misses in modern infrastructures, the compression in leaf nodes fits VAST-Tree well. In practice, in the setting used for this experiment, as the most errors, Δw, lay within a single chunk, there was no additional cost for their memory bandwidth. As a result, the averaged error amount was kept low compared to CS. On the other hand, the worst errors of these realistic data sets indicate that our lossy compression in branch nodes infrequently makes their error amounts enormously large (ex. 6981 if $Twitter - Ids$, and 4534 if $Twitter - Timestamps$). Section 5.1 provides an analysis of these errors, and then Section 5.2 presents a way to minimize the penalty of the worst errors.

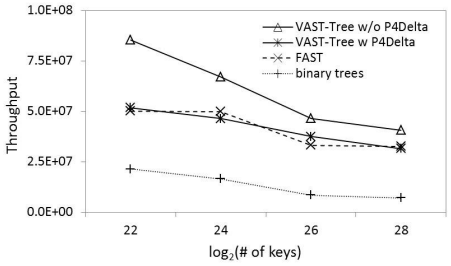

Figure 9: Throughput with exact-match scans as determined from synthetic data sets. "VAST-Tree w/o P4Delta" has the highest throughput. On the other hand, "VAST-Tree w P4Delta" has the higher throughput than FAST only for large key sizes (2^{26} and 2^{28}).

4.1.3 Memory Bandwidth Consumption

Data Sets	*Ids*	*Timestamps*
binary trees	9.05	same as left
FAST	34.3	same as left
VAST-Tree w P4Delta	67.7	31.4
VAST-Tree w/o P4Delta	78.9	41.1

Table 5: Throughput ($\times 10^6$) with exact-match scans by using realistic data sets. The skew of real data sets has little impact on their performance.

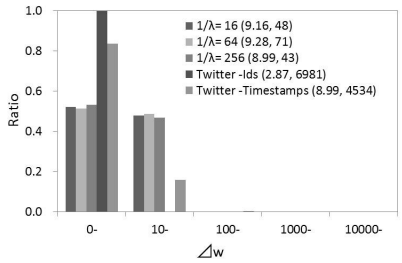

Figure 10: Distributions of the errors caused by the use of our lossy compression. The evaluation used synthetic data with 2^{28} keys. The left values in parenthesis indicate the averaged amounts of errors, and the right values show the worst errors in each condition.

Early studies [14][18][22] report that memory bandwidth is a important factor determining scaling performance as the number of cores increases. Figure 12 shows the averaged amount of memory bandwidth consumed by a single tree search. Many speculative reads in conditional branches in binary trees are believed to increase the memory bandwidth. On the other hand, both versions of VAST-Tree are more efficient than those of binary trees because of the structure without conditional branches. And also, our compression technique leads to the reduction of its consumption as compared to FAST. As a result of that, although VAST-Tree tends to increases instruction count through key searches, it does reduce the memory bandwidth consumed, and more specifically the figure shows that our technique saves the memory consumption by more than 94.7% and 40.5% as compared to binary trees and FAST, respectively, given that

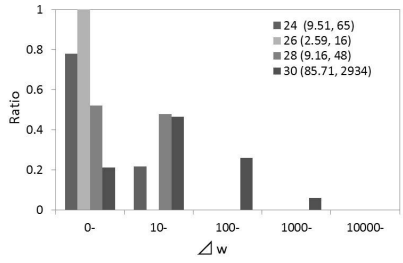

Figure 11: Distributions of the errors for the various numbers of keys in leaf nodes (synthetic data). The error amounts get bigger as the number of keys increases, because the maximum error amounts widen due to the increase of H_8. The values in parentheses are similar to Figure 10.

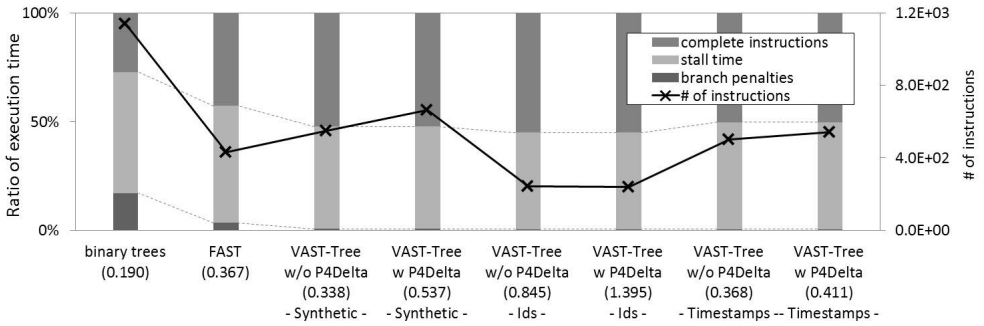

Figure 8: Ratios of execution time and counts of instructions with the exact-match scans of three techniques: binary trees, FAST, and VAST-Tree. The total number of keys is 2^{25}, and the values in parenthesis indicate IPC in each technique.

there are 2^{28} keys.

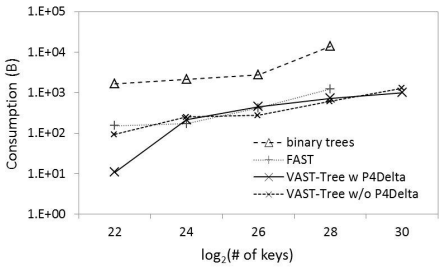

Figure 12: Comparison of averaged memory bandwidth (B) for a single tree search consumed by all the techniques. VAST-Tree achieves the saving of memory bandwidth consumption as compared to binary trees and FAST because of the conditional branch-free and compressed structure.

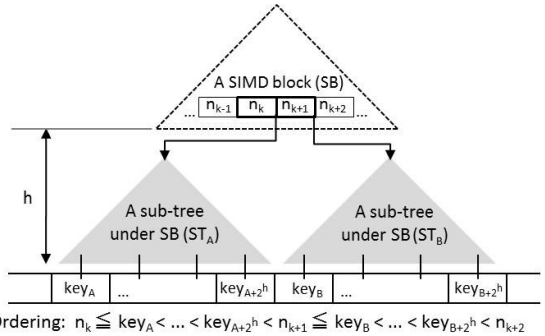

Figure 13: An analysis of the errors on the ordering of keys under a certain SIMD block (SB). The number of keys under these sub-trees (ST_A and ST_B) is assumed to be 2^h.

5. DISCUSSION

VAST-Tree is obviously influenced by the distribution of keys as shown by Figure 10. Section 5.1 details an analysis of the errors caused by the key distribution, and Section 5.2 then presents a optimized technique to reduce the penalty of the worst errors in tree traversal by using a error model detailed in Section 5.1.

5.1 Error Model of Tree Traversal

First, we analyze how many errors occur during tree traversal in VAST-Tree. Figure 13 shows key ordering under a certain SIMD block (SB) which includes comparison keys: n_k and n_{k+1} (n_k<n_{k+1}). There are two sub-trees, ST_A and ST_B; ST_A includes keys from key_A to key_{A+2^h} and ST_B includes keys from key_B to key_{B+2^h}, respectively.We assume that search key $skey$ follows n_k<$skey$<n_{k+1}.If an error happens in SB, that is, $skey$ is erroneous as n_{k+1}<$skey$<n_{k+2}, $skey$ is compared with keys in ST_B, not ST_A. Finally, $skey$ must be located in key_B because all the keys under ST_B are bigger than $skey$ ($skey$<key_B)[4]. That is why at most a single error happens during tree traversal.

Next, based on the analysis above, we use a geometric distribution to approximate the amount of the errors (Δw).

[4]Because the lossy compression is not applied to the leftmost keys (minimum values) in the headers of compression blocks, these keys are compared correctly.

Specifically, because the only one error happens through tree traversal from the root to the bottom, the first error after consecutive and successful comparisons from the root is modeled based on the well-known coin-toss model. Let the error probability (the random variable determined by the distribution of keys) of level h be p_h, the amount of errors (Δw) be dw_h and the height of trees be H. The random variable of the total error, W, can be written as:

$$W = \frac{1}{H}\left(\sum_{h=1}^{H} dw_h \times p_h \sum_{k=1}^{h}\left(1-p_{k-1}\right)^{k-1}\right) \quad (1)$$

In practice, the error happens in P_{16} and P_8 of VAST-Tree. Figure 14 shows each assigned variable of P_{16} and P_8 used in the following equations. According to these variables, Equation (1) is transformed into the equation below:

$$W = \frac{1}{H_{16}+H_8} \sum_{k=1}^{\frac{H_{16}}{CH_{16}}+\frac{H_8}{CH_8}} w_k \quad (2)$$

w_k means the amounts of the errors in each compression block and is defined as:

$$w_k = \sum_{l=1}^{\frac{CH_{16}}{SH_{16}}+\frac{CH_8}{SH_8}} dw_l \times p'_k \sum_{n=1}^{l}\left(1-p'_{n-1}\right)^{n-1} \quad (3)$$

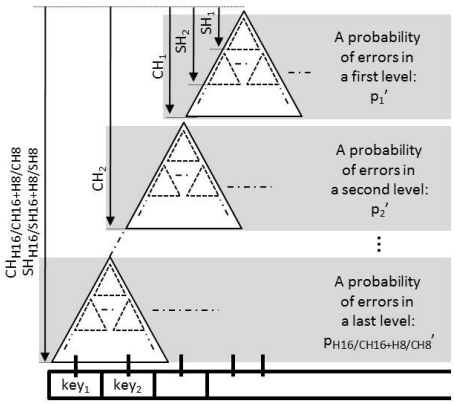

Figure 14: Defining each variable in P_{16} and P_8.

dw_l represents the amounts of the errors caused by comparison using SIMD operations. It is quantified as follows:

$$dw_l = \begin{cases} 2^{H_{16}+H_8-l\times SH_{16}} & (l \le \frac{H_{16}}{CH_{16}}) \\ 2^{H_8-(l-\frac{H_{16}}{CH_{16}})\times SH_8} & (otherwise) \end{cases} \quad (4)$$

p'_k represents the probability of the errors in the k-th compression block from the top as shown in Figure 14. The probability in a compression block is assumed to be same because the bits of the comparison keys in the block are cut by the same number of bits. Therefore, p'_k is defined as follows:

$$p'_k = \frac{c_k \times r_k}{qr}, \quad (5)$$

where qr is the input range of scans, c_k is the total number of comparison keys at the same level of the k-th SIMD block from the top, and r_k is the error range caused by the lossy compression of comparison keys. For example, if seven bits are cut from each key, the error range is 128 ($=2^7$) for each comparison key. Thus c_k is represented as:

$$c_k = \begin{cases} 2^{H_{32}+(k-1)\times CH_{16}} & (k \le \frac{H_{16}}{CH_{16}}) \\ 2^{H_{32}+H_{16}+(k-\frac{H_{16}}{CH_{16}}-1)\times CH_8} & (otherwise) \end{cases} \quad (6)$$

The number of cut bits obviously depends on the key distribution in leaf nodes, so r_k is also determined by this distribution. Let the random variable of d-gaps on keys be X. Assuming that the compression block of CH_{16} is located at the height of h (similar to SB in Figure 13), $2^{h+CH_{16}}$ keys must be packed in leaf nodes under this block. If the total of these d-gaps exceeds the range of 2^{16}, the error range starts to widen. If Y_k is the random variable of the summation under the k-th compression block, r_k is defined as:

$$r_k = \begin{cases} 2^{log_2 Y_k - 16} & (k \le \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k > 16) \\ 0 & (k \le \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k \le 16) \\ 2^{log_2 Y_k - 8} & (k > \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k > 8) \\ 0 & (k > \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k \le 8), \end{cases} \quad (7)$$

where Y_k can be calculated by X as shown below:

$$Y_k = \begin{cases} \sum_{n=1}^{H_{16}+H_8+(1-k)\times CH_{16}} X_n & (k \le \frac{H_{16}}{CH_{16}}) \\ \sum_{n=1}^{H_8+(1-k-\frac{H_{16}}{CH_{16}})\times CH_8} X_n & (otherwise) \end{cases} \quad (8)$$

Figure 15 compares these actual and estimated errors output by our error model explained above. Although our estimation model deviates slightly at the small and large $1/\lambda$ values, it roughly replicates the actual amount of the errors incurred by VAST-Tree due to the distribution of keys. Moreover, these errors are inclined to increase with the increase of d-gaps in keys. Therefore, the parameters (H_{32}, H_{16}, and H_8) of VAST-Tree must be determined carefully from the distribution of d-gaps.

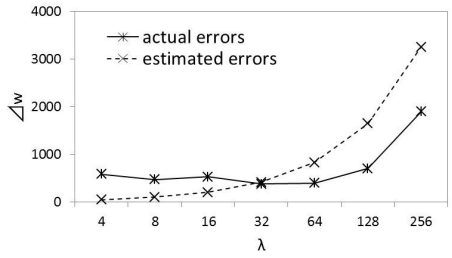

Figure 15: Estimation of the errors incurred by VAST-Tree. This evaluation was conducted by using 2^{30} keys, and synthetic data sets with varying $1/\lambda$.

5.2 Minor Error Optimization

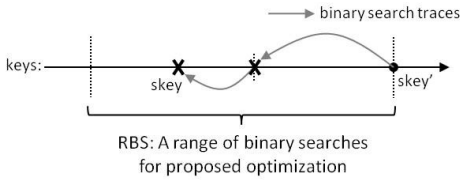

Figure 16: An overview of our proposed optimization technique for binary searches so as to minimize the penalty of the worst errors. $skey'$ is initial position after tree traversal on the branch nodes of VAST-Tree, and $skey$ is the search key. $skey$ needs to be in the range of RBS.

In this section, we present a technique so as to minimize the penalty of the worst errors as described in Section 4.1.2. Specifically, we apply binary searches to tree traversal with the worst errors instead of scanning sequentially for the error correction. Let the search key be $skey$, and the result of tree traversal with the error be $skey'$. Figure 16 overviews the optimization based on binary search. Note that $skey$ needs to be in the range of RBS with high frequency so as to strengthen the use of binary searches. Based on our analysis of the errors, RBS is defined as follows:

$$RBS = t \times \sqrt{V(W)}, \quad (9)$$

where t and $V(W)$ represent a certain constant value and the variance of W, respectively. The relationship between $V(W)$ and the rate follows Chebyshev's inequality regarding to t, and is represented by the following inequality:

$$P\left(|W - E(W)| \geq t \times \sqrt{V(W)}\right) \leq \frac{1}{t^2} \quad (10)$$

Therefore, t is decided so as to insure that $skey$ is in the range of RBS. The remaining question is when to exploit the above optimized technique. In the optimized VAST-Tree, if the inequality below is satisfied, the binary search is applied so as to minimize the number of cache lines that must be read for the error correction.

$$\frac{|k - k'|}{CK \times E(X)} > log_2(t \times \sqrt{V(W)}) \quad (11)$$

The left term represents the estimated amount of read cache lines for the original error correction, and the right term represents the amount of read cache lines by using the optimized error correction with binary searches.

Table 6 shows the timestamp counts of a single search with the worst error. These counts are obtained by a $rdtsc$ instruction, which has ability to extract the value of timestamp counters inside $Intel$ processors. The original VAST-Tree algorithm is noted as SEQ, and the optimized one is noted as BS, respectively. As a result, the optimized one saves the worst processing time with the data sets of $Twitter - Ids$ and $Twitter - Timestamps$ by 7.1% and 18.5%, respectively.

Type of Correction	SEQ	BS	%
Ids	557285	523514	93.9
Timestamps	551297	449340	81.5

Table 6: Hardware-based timestamp counts of a single tree search with the worst error. The right-most column represents the reduction ratios of the worst cases ($\frac{BS}{SEQ}$).

6. RELATED WORK

Recently, a lot of techniques have been proposed for modern computing infrastructures to improve the performance of database kernel operations such as sorts, joins, scans, compression, and indices [22][5][21][7][12][8]. In particular, since the early study [3] reported that the bottlenecks between processors and memory degraded the performance of database kernels as explained in the Introduction, many techniques using hardware-conscious approaches have been raised. A recent report provides a comprehensive survey of these kinds of advances [17]. This section provides a brief overview about related works in the fields of indexing and index compression techniques.

6.1 Indexing on Modern Hardware

T-trees was initially proposed as a replacement of disk-based indices [16]. Later, a lot of cache-conscious B+trees [19][20][10][11][4] were proposed to decrease cache misses in tree traversal. In implementing memory-based B+trees, one technical consideration is how large the branch nodes are. Two previous studies [11][4] concluded that the performance of B+trees is improved by using node sizes larger than cache line size, because small node size is inclined to increase tree height, which yields more TLB misses than larger node size does. Moreover, there are other cache-conscious techniques that use query buffering so as to save memory bandwidth [27]. This approach holds a bunch of queries that access the same branch node, and processes these queries simultaneously so as to minimize the total number of node accesses. As a result, it decreases the number of cache lines, and the consumption of memory bandwidth. The technique is useful for processors that have many cores, though the situation of multi-threads is not considered.

Although these cache-conscious techniques are successful, they ignore the branch-free and alignment techniques used by recent studies [28][25]. These days, index techniques have been accelerated by using SIMD instructions [23][14]. In particular, FAST successfully incorporates cache-conscious, aligned, and branch-free techniques, and is now the fastest of the known technique for in-memory usage and read-only workloads. However, the alignment technique used to decrease the total cache lines and TLB creates a lot of padding areas, which greatly increases the total index size of FAST as the number of keys increases. As FAST does not support incremental updates, the same authors proposed a updatable and multi-core efficient index structure, called PALM [24]. To the best of our knowledge, this is the first study to address the trade-off between optimization techniques of modern processors and the total index size; we apply SIMD instructions and processor-friendly compression to indices of large data sets, and show that it runs well on modern processors.

Recently, the application of many-core architectures such as GPUs and other devices has become a popular topic, and there is an index technique for GPU implementation [13]. FAST was prototyped and evaluated on GPUs [14], too. However, the data on host memory must be transferred to GPU memory, and this transfer time is reported to make up 15% to 90% of the total [6]. This problem is expected to be solved by subsequent processors that are integrated at the die level with GPUs such the next generation Intel processors called "Ivy Bridge", or future graphic devices. On the other hand, Intel developed an advanced x86-based many-core architecture, called $Intel^{®}MIC$ (Many Integrated Core), and some index techniques [24][15] were evaluated on this platform. Therefore, the techniques to harness these hardware are of considerable importance in the database community.

6.2 Compression Techniques for Indices

Many compression techniques for indices such as "prefix/suffix truncation" and "NULL suppression" [2][9] were proposed by early database researchers, but they targeted compaction of indices rather than processing efficiency. Our approach applies truncation to branch nodes so as to harness SIMD instructions, and then realizes compact and efficient data structure for branch nodes.

Leaf nodes can be regarded as a sequence of keys in ascending order, and there are some kind of processor-friendly compression techniques that can be applied to such data. We show here two recent techniques: P4Delta [28] and VSEncoding [25]. P4Delta, explained in the previous section, is a heuristic and parametric approach. P4Delta is popular as the compression technique applied to the inverted lists of search engines [26]. On the other hand, VSEncoding is a parametric approach, and finds the optimal parameters by using dynamic programming. Our compression of keys on leaf nodes is based on P4Delta rather than VSEncoding

because VSEncoding has no ability to access compressed elements randomly. That is, it is difficult to reference the offset of integer arrays as explained in Section 3.3.1. As the amount of information dynamically grows in the future, the application of compression is of significance for the analysis of massive data sets.

7. CONCLUSIONS

We have introduced VAST-Tree, a state-of-the-art technique that can efficiently and compactly process the indices of massive data sets. There are two key advances in VAST-Tree; the lossy compression of middle and bottom branch nodes, and the lossless compression of the keys in leaf nodes. The lossy compression achieves higher compression ratios, and although the compaction yielded by the lossless compression is not so high, its technique fits VAST-Tree well since it reduces the number of cache lines that must be read for error correction. It is the first study to address the trade-off between the optimization techniques of modern processors and total index size. Finally, we modeled the errors of VAST-Tree, and presented an optimized technique to minimize the penalty of worst errors in tree traversal.

8. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? Proceedings of the VLDB Endowment, pages 266–277, 1999.

[2] R. Bayer and K. Unterauer. Prefix B-trees. ACM Transactions on Database Systems, 2:11–26, 1977.

[3] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In Proceedings of VLDB'99, pages 54–65, 1999.

[4] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. SIGMOD Record, 30(2):235–246, 2001.

[5] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. Proceedings of VLDB'08, 1:1313–1324, 2008.

[6] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. Proceedings of VLDB'10, 3:670–680, 2010.

[7] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In Proceedings of SIGMOD'06, pages 325–336, 2006.

[8] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In Proceedings of SIGMOD'04, pages 215–226, 2004.

[9] G. Graefe. Modern B-tree Techniques. Foundations and Trends in Databases, 3(4):203–402, 2011.

[10] G. Graefe and P.-A. Larson. B-Tree Indexes and CPU Caches. In Proceedings of ICDE'01, pages 349–361, 2001.

[11] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B+-trees. Proceedings of SIGMETRICS'03, 31, 2003.

[12] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. Proceedings of the VLDB Endowment, 4:314–325, 2011.

[13] T. Kaldewey, J. Hagen, A. Di Blas, and E. Sedlar. Parallel search on video cards. In Proceedings of HotPar'09, pages 9–9, 2009.

[14] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In Proceedings of SIGMOD'10, pages 339–350, 2010.

[15] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Designing Fast Architecture Sensitive Tree Search on Modern Multi-Core/Many-Core Processors. ACM Transactions on Database Systems, 9(4), 2011.

[16] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In Proceedings of VLDB'86, pages 294–303, 1986.

[17] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. Proceedings of the VLDB Endowment, 2:1648–1653, 2009.

[18] R. Matthew. When Multicore Isn't Enough: Trends and the Future for Multi-Multicore Systems. In In HPEC, 2008.

[19] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In Proceedings of VLDB'99, pages 78–89, 1999.

[20] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In Proceedings of SIGMOD'00, pages 475–486, 2000.

[21] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In Proceedings of IPDPS'09, pages 1–10, 2009.

[22] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In Proceedings of SIGMOD'10, pages 351–362, 2010.

[23] B. Schlegel, R. Gemulla, and W. Lehner. k-ary search on modern processors. In Proceedings of DaMoN'09, pages 52–60, 2009.

[24] J. Sewall, J. Chhugani, C. Kim, and P. Satish, Nadthur Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modification to B+Trees on Many-Core Processors. In Proceedings of VLDB'11, 2011.

[25] F. Silvestri and R. Venturini. VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming. In Proceedings of CIKM'10, pages 1219–1228, 2010.

[26] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In Proceedings of SIGIR'09, pages 147–154, 2009.

[27] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In Proceedings of VLDB'03, VLDB '2003, pages 405–416, 2003.

[28] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In Proceedings of ICDE'06, pages 59–71, 2006.