HICAMP Bitmap: Space-Efficient Updatable Bitmap Index for In-Memory Databases

Bo Wang
Stanford University
bowang@stanford.edu

Heiner Litz
Stanford University
heiner.litz@stanford.edu

David R. Cheriton
Stanford University
cheriton@cs.stanford.edu

ABSTRACT

Bitmap represents an efficient indexing structure for querying large amounts of data and is widely deployed in data-warehouse applications. While the size of a bitmap scales linearly with the number of rows in a table, due to its sparseness, it can be greatly reduced via compression based on run-length encoding. However, updating a compressed bitmap is expensive due to the encoding and decoding overheads, in particular, as re-compression can change the compressed sequence length and data layout. Due to this problem, bitmap indices only perform well for read-only workloads.

In this paper, we propose a bitmap index structure which is both space-efficient and allows fast updates, by building on top of a smart memory model called HICAMP. As a consequence, our approach enables bitmap indices for workloads that exhibit high update ratios as in OLTP workloads. We also present a new multi-bit bitmap design which addresses the candidate checking problem. In our experiments, the HICAMP bitmap index demonstrates 3∼12x reduction in size over B-tree and 8∼30x over other commonly used indexing structures such as Red-Black tree, while supporting efficient updates simultaneously.

1. INTRODUCTION

Database systems use indices to quickly locate data without having to search every row in a table. B-tree is the most popular data structure in operational databases, especially for OLTP workloads which feature many concurrent, short, but latency-sensitive transactions. The popularity of tree-based indexing methods in OLTP stems from the fact that its search and update complexities are both logarithmic, and that the search and update ratios are similar in OLTP workloads[21]. However, the B-tree index is large and hence consumes precious memory capacity that could otherwise be used for data buffering. In today's increasingly popular in-memory databases, memory consumption represents a primary concern when trying to fit in the full data set. Some recent in-memory databases like SAP HANA greatly reduce the use of indices to decrease memory consumption and eliminate effort in index maintenance, which in turn offers higher overall performance [6].

DaMoN'14, June 22 - 27 2014, Snowbird, UT, USA
Copyright 2014 ACM 978-1-4503-2971-2/14/06 ...$15.00.
http://dx.doi.org/10.1145/2619228.2619235

The bitmap index represents an alternative space-efficient indexing structure well-suited for read-heavy environments, such as OLAP databases, that require high performance searching and set operations on read-only or read-mostly data. Bitmap indices provide better performance for searching than for updating. This property fits the search-to-update ratio in OLAP workloads [21]. A basic bitmap index utilizes one bitmap per distinct value of an attribute. Inside each bitmap, the n-th bit is set to non-zero if and only if the n-th record equals to the attribute value this bitmap corresponds to. Without compression, a basic bitmap index requires $N \times V$ bits to index an attribute with N records and V distinct values, i.e. its cardinality. This size is prohibitively large, especially if there exist many distinct values. To reduce the large space used by raw bitmap indices and to make it practical, various bitmap compression algorithms have been proposed [24, 1, 4]. These algorithms leverage the sparsity of bitmaps and apply run-length encoding to suppress long sequences of consecutive zero (or non-zero) bits.

As the size of a compressed bitmap index is much smaller than that of a B-tree, it is desirable to adopt bitmap indices, especially for in-memory databases. However, bitmap indices are not update-friendly for the following reasons. Firstly, because of the non-uniform length of compressed bit sequences, it requires a sequential scan from the beginning to locate the exact position of the bit to be updated. Secondly, changing the bit requires decoding the compressed sequence, updating the bit, and then re-encoding the new sequence. This process is time-consuming, especially when the new sequence is of a different length. Therefore, compressed bitmap indices are ill-suited for OLTP workloads.

Another issue faced by the bitmap index is high cardinality of some attributes because the number of bitmaps inside a basic bitmap index equals to the cardinality of the attribute. To avoid a large number of sparse bitmaps in the case of high cardinality, *binning* is applied to partition the attribute value space into bins. The basic idea of binning is to build one bitmap for a sequence of consecutive values named a *bin* rather than for each distinct value. This method disassociates the number of bitmaps from the attribute cardinality and enables to control the index width. Besides, attributes in a continuous value space can be indexed with bitmap indices in the same way as discrete-valued attributes. However, indexing multiple values with a single bitmap introduces false positives on lookups. In particular, a bit in the bitmap is set when the corresponding record matches any one of the attribute values in that bin. To verify a match in this case, a *candidate check* needs to be performed by reading the record value from the corresponding column. The match is a true positive only if the record value equals to the lookup value. As bitmap indices are stored separately, each candidate check needs to look into the data area of the column. In disk-based databases, candidate checking is expensive as scanning a bitmap may trigger paging in and out multiple data pages. In in-memory databases, candidate checking becomes cheaper, however, it can still incur cache misses and cache pollution.

In this paper, we propose a new bitmap index which simultaneously achieves small memory consumption, fast lookups, and efficient updates by leveraging the Hierarchical Immutable Content-Addressable Memory Processor (HICAMP) [3]. HICAMP introduces a new memory system that stores data in the form of directed acyclic graphs (DAGs). It also provides fine-grained memory deduplication. Designing a bitmap index utilizing these underlying hardware features enables us to minimize the size of a bitmap without applying run-length encoding and to maintain a regular data layout for efficient updates. HICAMP bitmaps can be scanned fast by exploiting the knowledge of the DAG structure. To reduce candidate checks introduced by bitmap binning, we devise a multi-bit bitmap format that mitigates the issue of high cardinality without sacrificing lookup performance.

The remainder of this paper is structured as follows. Section 2 presents the HICAMP architecture and introduces its hardware features. Section 3 presents our new bitmap index structures. Compaction evaluations are presented in Section 4. Section 5 reviews related work before Section 6 concludes the paper and discusses future work.

2. HICAMP MEMORY SYSTEM

The advantages of our bitmap index is achieved by taking advantage of the underlying hardware. In this section, we briefly describe the data organization and deduplication as a primer to understand how HICAMP bitmap indices work. Details of HICAMP architecture can be found in [3].

Data stored in HICAMP memory are divided into *content-unique lines* of a fixed size, e.g. 64 bytes. Each line is immutable and only stored once in the whole memory system. Multiple occurrences of the same line reference the same physical memory location by a *physical line id* (PLID). Memory lines are reference counted. Whenever the reference count of a line reaches zero, the line is reclaimed. An object in HICAMP memory is structured as a DAG, called a *segment*, storing data in its leaf nodes. The internal nodes store the PLIDs pointing to leaf nodes and other internal nodes. A node in the DAG is a memory line. We will use these two terms interchangeably. Figure 1 shows an example of a bit array stored in HICAMP memory as a DAG. For simplicity, the figure shows a line as containing only 8 bits or 2 PLIDs. In a real system, a 64-byte memory line is capable of storing 512 bits or 16 PLIDs. The three unique non-zero lines in this bitmap are stored as the three leaves in the figure. Repetitive occurrences of the leaf 0001 0000 are deduplicated with the same PLID P_2. All-zero lines are denoted with the zero PLID. Zero-valued PLIDs imply zero-valued subtrees.

It is worthwhile to note that the deduplication is performed hierarchically. When one or more objects share a

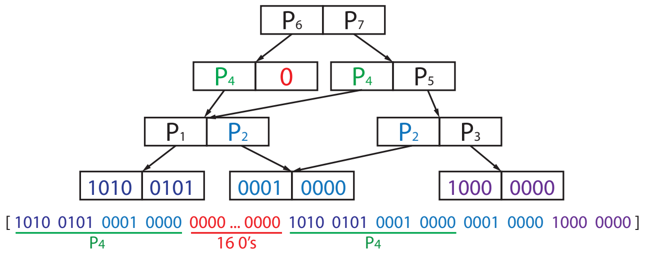

Figure 1: Storing a bitmap as a DAG on HICAMP

sub-DAG of the same data, these sub-DAGs are referred by the same PLID of the sub-DAG root. Only one copy of leaves and internal nodes are stored in the memory. For example, in Figure 1, the sequence 1010 0101 0001 0000 occurs twice and spans two leaves. Rather than using two PLIDs (P_1, P_2) referencing the two leaves respectively, using a PLID P_4 referencing the parent node of these two leaves is sufficient.

Furthermore, HICAMP compacts the path over internal nodes with only a single child reducing the number of internal nodes. Figure 2 shows an example for the case that an internal node contains only two PLIDs.

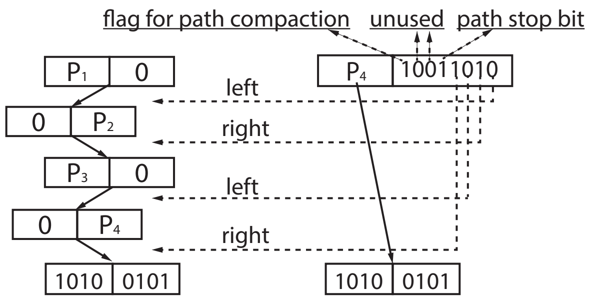

Figure 2: Path Compaction

HICAMP memory seamlessly integrates with conventional memory. The virtual address space of a process consists of a conventional partition and a HICAMP partition. Accesses to virtual addresses in the HICAMP partition are directed to the HICAMP memory controller, which looks up the segment IDs and calculates the leaf offsets based on the virtual addresses. This lookup operation is handled by a hardware component called *iterator register*. The iterator register translates a store operation into a lookup to locate the corresponding leaf to be changed and a lookup-by-content to the content-unique line, then recursively replaces the PLIDs until the root of the DAG.

As the name suggests, iterator registers provide an efficient means to iterate through a data collection sequentially avoiding unnecessary DAG traversals. The path information from the root node to the currently referenced leaf node is cached in the iterator register for efficient access to adjacent leaves. In addition to path caching, iterator registers improve performance by prefetching the next elements to avoid memory stalls. This is particularly helpful in HICAMP because the logically contiguous leaf elements may be physically scattered in memory.

3. HICAMP BITMAP INDEX

The memory structure described in Section 2 enables efficient mapping from database indices to a compacted format while maintaining good manipulability. In this section, we first discuss the standard bitmap on HICAMP and then present our multi-bit bitmap to handle attributes with high cardinality without sacrificing the lookup performance.

3.1 Standard Bitmap Index

The standard bitmap index on HICAMP has a straightforward format. Each bitmap is stored as a HICAMP DAG. On the top-level, a hash table maps the lookup value to the corresponding bitmap. This hash table itself is also implemented as a HICAMP DAG as illustrated in Figure 3. The lookup value is used as the offset into the DAG. The value stored at the corresponding leaf is a reference to the specific bitmap.

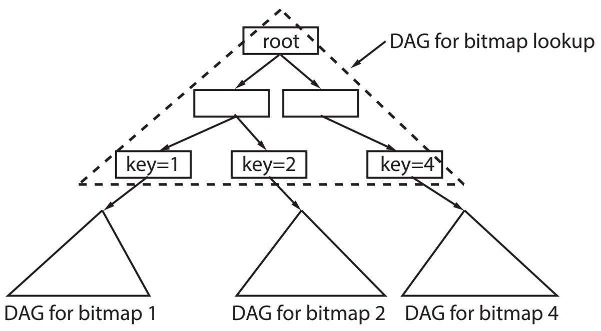

Figure 3: Top-level Structure of HICAMP Bitmap Index

Standard bitmaps are compacted on HICAMP in two ways. Firstly, taking advantage of the fact that most bitmaps are sparse, i.e. many data lines of bitmaps are zero lines, HICAMP memory references them with the zero PLID hierarchically. Thus, long sequences of consecutive zero bits take little memory space. Secondly, non-zero lines with identical contents are stored only once and referenced by the same PLID. It only takes 4 bytes to store a PLID in an internal line, which translates to 1/16 overhead comparing to the 64-byte line deduplicated. Lines in very sparse bitmaps often contain only a single non-zero bit. Because a 64-byte line has only 512 bits, only 512 content-unique lines, i.e. 32KB in space, are needed to cover all possible patterns. For lines with two non-zero bits, there are a few more permutations, but similarly it only requires less than 8MB of memory. With these two mechanisms, sparse bitmaps can be stored at high compaction ratio.

One of the most common operations on index is scanning. For example, to determine all records of a lookup value, the corresponding bitmap is scanned. Procedure `next_set_bit()` shows a fast scanning algorithm on HICAMP bitmaps. Each time `next_set_bit()` is called, the index to the next non-zero bit is returned in logarithmic time. `next_set_bit()` takes three parameters - the current word, the current word offset in the DAG, and the bit offset in the current word. To find the next set bit, bits before the current bit offset in the current word are cleared with bit masks in constant time (i.e. `clear_least_sig_bits()`). If the current word is then zero, the next non-zero word in the DAG is looked up (i.e. `next_non_zero_word()`). `next_non_zero_word()` walks the DAG and skips consecutive zeros. This walk takes O(log n) steps, where n is the size of the bitmap. Using bit masks and shift, the first non-zero bit in the word can be found in constant time [22] (i.e. `first_set_bit()`). Combining the word-level offset and bit-level offset, we obtain the position of the next non-zero bit. Predicates are supported by passing the value in the column for checking in case of a binned bitmap index (Section 3.2).

```
proc next_set_bit($cur\_word, word\_offset, bit\_offset$) $\equiv$
  while $word\_offset \leq num\_words$ do
        $cur\_word :=$ clear_least_sig_bits($cur\_word, bit\_offset$)
        if $cur\_word = 0$
           $word\_offset :=$ next_non_zero_word($word\_offset$)
           $cur\_word :=$ word_value($word\_offset$)
        fi
        $bit\_offset :=$ first_set_bit($cur\_word$)
        $index := word\_offset \times 64 + bit\_offset$
        $bit\_offset := bit\_offset + 1$
        $pred\_test :=$ evaluate_predicate($index$)
        if $pred\_test = true$
           return $index$
        fi
  end
```

HICAMP DAG allows the scan operation to only access non-zero data lines in a bitmap. Even though data lines of a bitmap may scatter non-continuously in the physical memory, iterator register hardware can prefetch logically adjacent lines without CPU intervention thanks to its awareness of the DAG structure.

In contrast to software compressed bitmaps, HICAMP bitmaps support efficient updates. Software compression algorithms employ run-length encoding, which makes updates prohibitive costly. Typically an auxiliary delta structure is maintained to record all changes and the bitmap is rebuilt in batch periodically [19]. HICAMP bitmaps adopt fine-grain deduplication rather than variable-length compression. Thus, in HICAMP bitmaps, the bit to update can be directly located by translating the position to a particular leaf in the DAG and the offset within that leaf. Due to the similarity, the update complexity of HICAMP bitmaps is comparable to that of B-tree, in particular in O(log n) complexity.

3.2 Multi-bit Bitmap Index

Standard bitmap indices are efficient when the cardinality is relatively small. However, if the cardinality is large, a large number of bitmaps need to be stored in the bitmap index. The overhead of metadata may surpass the space savings from deduplication. Additionally, when performing range queries, many bitmaps need to be opened and each one just contains a small number of non-zeros. Handling too many DAGs concurrently can lead to iterator register thrashing and degrade the iteration performance. Moreover, when the attribute value space is continuous rather than discrete, standard bitmap indices cannot be directly applied. Binning can solve the high cardinality problem and enable indexing on continuous attributes. However, it introduces the need for candidate checking which affects the lookup performance.

To reduce the overhead of candidate checking, we introduce a multi-bit bitmap format. In a multi-bit bitmap, a fixed number of bits, e.g. 4 bits, are used to encode a single record. In comparison, a standard bitmap uses only one bit per record. Rather than only denoting the existence, every n bits can encode a signature of $2^n - 1$ distinct values, while n clear bits represent the absence of a record. The signature is used to reduce the need for candidate checking. To encode a record of value x in a n-bit bitmap index, we firstly determine the bin of this record by dividing x by the bin width b, assuming $b \leq 2^n - 1$. Then, the signature of the record, e.g. modulo of x by the bin width b, is stored in the n bits at the position corresponding to this record inside that bin. To lookup records of value x, we can directly scan bin x/b for the signature $x\%b$.

Figure 4 shows an example in a 4-bit bitmap index, i.e. $n = 4$. In this case, every 4 bits encode a record. A signature can have up to $2^4 - 1 = 15$ distinct values. Let bin width b also be 15, then value 50 will be hashed to bin #3 with a signature of $50\%15 = 5$, i.e. $(0101)_2$ in binary. To look up records of value 50, we scan bin #3 for signature $(0101)_2$. There is no need to check the original column when a non-zero element is encountered in a bin.

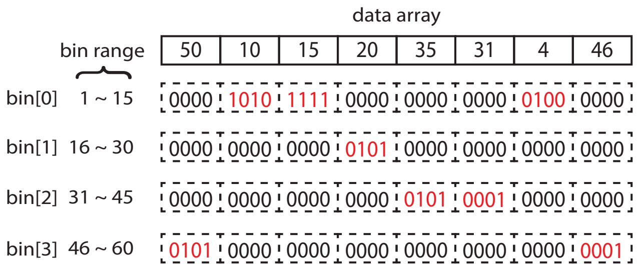

Figure 4: Example of a 4-bit bitmap index

Generalizing the above format, we can set the bin width larger than $2^n - 1$ in a n-bit bitmap index. In this case, more than one distinct attribute values are mapped to the same signature within a bin. For example, if the bin width is $m \times (2^n - 1)$, m distinct attribute values are mapped to one signature. We call these m attribute values a *value group*. More formally, a value group is a fixed number of consecutive values that share the same signature within a bin. For instance, in Figure 5, $n = 4$ and bin width equals to 120, every 4-bit signature represents $120/(2^4 - 1) = 8$ distinct values, i.e. $m = 8$. In this case, candidate checking is still needed because we are not sure which one of the two values this signature stands for. However, the need for candidate checking is limited to non-zero elements of the target signature, i.e. elements in the same value group as the lookup value, rather than every non-zero element in the bitmap as in a standard bitmap index with binning. Therefore, the number of candidate checks reduces to $1/2^n$ on average.

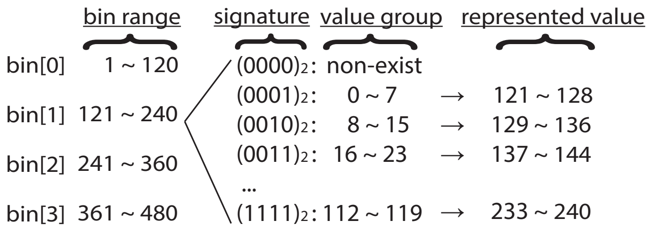

Figure 5: 4-bit bitmap with value group (m = 8)

It is worth noting that multi-bit bitmaps can be large without deduplication, as a flat n-bit bitmap takes $n - 1$ times more memory than a standard bitmap. Most of the extra space is consumed by storing zeros due to the abundant sparsity in a bitmap index. However, on HICAMP most of these zero sequences are deduplicated. Moreover, saving a record with multiple bits reduces the number of records on the same memory line, which in some circumstances may increase the probability of finding duplicate lines.

Multi-bit bitmaps also make binning favorable to both equality queries and range queries. Conventionally, appropriate binning can improve the performance of range queries since fewer bitmaps need to be scanned, which can amortize the cost of candidate checks on boundaries of the range. However, candidate checks slows down equality queries because every hit in the bitmap requires a candidate check. Multi-bit bitmaps solve the candidate checking bottleneck and thus make binning a good fit for both types of queries.

4. EVALUATION

4.1 Methodology

To evaluate the memory consumption of our HICAMP bitmap index, we build a HICAMP simulator on top of ZSim [17]. ZSim is a fast, multithreaded, cycle-accurate simulator using dynamic binary instrumentation [15]. Unlike conventional cycle-driven or event-driven models, ZSim adopts an instruction-driven timing model for faster simulations. While the test binary is executed, ZSim instruments every load and store and redirects accesses to HICAMP memory addresses to our HICAMP memory model. In the HICAMP model, we simulate the DAG structure and the deduplication mechanism. The number of memory lines allocated for data and DSG structure are recorded. To enable efficient scan on bitmaps, we added new instructions to lookup the next (or previous) non-zero word and half-word. These instructions utilize the DAG to efficiently skip all zero words.

4.2 Compaction Results

4.2.1 TPC-H

We evaluate the size of the HICAMP bitmap index using data generated by TPC-H DBGen [5] which mimics real-world datasets. We pick four numeric columns with different cardinalities for testing. Each column consists of 50 millions of records. We compare the HICAMP bitmap index against the following baselines: B+tree (STX B+tree [2]), AVL tree (STL AVL Map [16]), red-black tree (STL library), skip list (CS::SkipList [18]). Additionally, we use FastBit [23], the official implementation of word-aligned hybrid code (WAH), as the software compression algorithm.

Table 1 reports the memory consumption for each of the indexing structures on the TPC-H datasets. We report the average number of bytes required to index one record. The HICAMP bitmap index is over an order of magnitude smaller than B+tree in the cases where the cardinality is small. For larger cardinalities, the number of bytes per record in all tree-based indices and skip list stay roughly the same. The reason therefore is, that these data structures store their records in associative structures indexed by a key. Thus the total space is determined by the number of records, the number of keys and the internal indexing cost of internal nodes and pointers. Both AVL tree and red-black tree are binary

Cardinality	Column name	B+Tree (d=128)	B+Tree (d=1024)	AVL Tree	Red-Black Tree	Skip List	WAH	HICAMP Bitmap
7	line number	25	24	64	64	53	0.9	1.7
50	quantity	25	24	64	64	53	4.4	1.2
2526	ship date	25	24	64	64	53	1.7e-3	0.09
100000	supplier key	23	19	64	64	53	6.8	12.7[‡]

[†] unit: bytes/record   [‡] indexed with 8-bit bitmaps

Table 1: Memory consumption on TPC-H

search trees, i.e. each node can have at most two children nodes. Therefore, many internal nodes are needed to index a large dataset. However, nodes in B+tree have higher fanouts, leading to smaller number of bytes per records compared to the AVL tree and red-black tree.

Bitmap indices perform poorly when cardinality is large. In the case of 100,000 unique values, 70 bytes are required to index a single record. This is because 100,000 bitmaps need to be constructed to build the index, even though each bitmap is very sparse. On average, only 500 out of the 50 millions bits are non-zeros. However, there is also a per-bitmap memory overhead for metadata storage. The space cost on internal nodes is not negligible neither. Additionally, the hashmap used to look up a bitmap, is much denser and contains more non-zero lines, thus requiring more space. To mitigate the high cardinality issue, binning is applied. Due to the sparsity of the data, we use a bin width of 100, respectively every 100 consecutive unique values are mapped to a bitmap. With binning, the number of bytes per record reduces to 4.3 bytes, just 1/5 of that of B+tree.

Although binning can greatly reduce the memory overhead of bitmap indices, it requires costly candidate checking. When scanning a bitmap with a bin width of 100, 99% of the hits are false positives assuming uniform data distribution. For every hit, we have to check the data record to filter out false positives. Candidate checks hurt performance and incur additional memory references. A solution to this issue is to use multi-bit bitmaps. When indexing the column with 100,000 unique values on 8-bit bitmaps, it takes 12.7 bytes per record. Though the 8-bit bitmaps require 3x the memory capacity compared to binning with bin width of 100, no candidate checking is required when iterating on the index. Moreover, the multi-bit bitmap is still up to 50% more space-efficient than B+tree.

HICAMP bitmaps have a similar memory consumption comparing with WAH in most cases. In the high cardinality case, HICAMP bitmaps require slightly more memory, up to 1.87x space. However, HICAMP bitmaps also support efficient updates. The small amount of space overhead is worthwhile.

In the third test case (ship date), the memory consumption of both WAH and HICAMP bitmaps are very small. This is because the ship date is sorted in ascending order. Thus, in bitmap representation, a large number of consecutive bits are either zero or non-zero. This spatial property is greatly beneficial for bitmap compaction. In WAH, this is represented by a bit for value and a number for the run-length. In HICAMP, all copies of these all-zero and all-non-zero lines are saved only once. Because of hierarchical deduplication, the internal nodes can also be deduplicated, thus significantly reducing the size of the index.

Cardinality	B+Tree	AVL/RB	Skiplist	WAH	Multibit
unif 10	25	64	53	1.2	2.0
unif 100	25	64	53	5.7	7.0
unif 1000	25	64	53	7.4	8.0
zipf 10	25	64	53	0.9	1.9
zipf 100	25	64	53	1.2	3.0
zipf 1000	25	64	53	1.3	2.4

Table 2: Memory consumption on uniform/zipf dist.

4.2.2 *Uniform and Zipf Distribution*

In this section, we evaluate the size of multi-bit bitmap on data generated under uniform and zipf distribution. This provides more general comparison beyond TPC-H and the asymptotic size of the multi-bit bitmap index. In most enterprise databases, the number of distinct values per column is small. According to a study on real enterperise databases [10], 80∼90% of columns have fewer than 1024 distinct values. Thus, test data was generated with cardinalities from 10 to 1,000. The number of bits per record is chosen according to the cardinality: 2-bit bitmap is used for the cardinality of 10, 4-bit for 100, and 8-bit for 1,000.

Table 2 shows the memory consumption of various indexing formats on uniform and zipf distributions. The zipf distribution represents many datasets following power law in the real world. Zipf distributed data exhibits a large number of records with a small number of unique values. Therefore, a small number of bitmaps are dense while the rest are very sparse. This situation is a good fit for HICAMP deduplication due to the abundance of zero lines. Therefore, the memory consumption in the case of zipf is smaller than that of a uniform distribution. The HICAMP multi-bit bitmap is 3∼12x smaller than B+tree, 8∼30x smaller than AVL tree, red-black tree and skip list. The size of bitmap index changes with the cardinality, while the size of the tree-based indexing structures remain almost unchanged. WAH encoding can achieve a slightly smaller size, close to that of the HICAMP multi-bit bitmap.

5. RELATED WORK

One of the early bitmap compression algorithms is the byte-aligned bitmap compression (BBC) [1]. BBC is based on the idea of run-length encoding that represents consecutive identical bits by the bit values and the run-length. The bitmap is encoded or decoded one byte at a time. Bitwise logical operations can be performed on compressed bitmaps directly. Word-aligned hybrid code (WAH) [24] compresses bitmaps in a word-aligned fashion, which reduces the decoding frequency and exhibits better performance. Despite high compression ratios and fast bit-wise operations, these coding schemes are not update-friendly. Thus, bitmap indices are mostly used in read-only applications like data warehousing.

Another bitmap-based indexing method is column imprints [19], which partitions the value domain into a small number of bins and uses only one bitmap to record whether a value may appear in a cache line. A bit is set if at least one value in the cache-line falls into the corresponding bin. This structure also adopts run-length encoding, thus requiring a delta structure to record changes.

B+trees are commonly used for update-intensive workloads. The B-link tree [11] represents the most common structure for concurrent updates. The B-link tree avoids all read latches by linking tree blocks at each level together with a next pointer, which greatly improves concurrency. However, write latches are still required as a tree block is modified. Foster B-tree [7] provides optimized latching and improves the update rates without compromising read performance. The recently proposed Bw-tree [14] achieves a latch-free approach with the compare and swap (CAS) instruction for all state changes, including structure modification operations. However, the implementation of a serializable latch-free data structure is non-trivial and requires tens of thousands of lines of C++ code. In comparison, concurrency control on HICAMP bitmap index is much simpler because it operates on a snapshot with a flat data representation. Much of the conflict resolution is handled by the memory controller. Because of no structure modifications in bitmap index, the performance is more predictable.

The advent of in-memory databases inspires new tree-based indexing structures [8, 9, 12]. Fast Architecture Sensitive Tree (FAST) [8] is an architecture sensitive binary search tree which minimizes cache miss and adopts SIMD instructions for data-level parallelism. However, FAST does not support online update. KISS-tree [9] and Adaptive Radix Tree (ART) [12] are in-memory radix trees supporting efficient searching and updating. ART adapts the fanout parameter locally for better trade-off between tree height and space efficiency. ART has a software path compression sharing the similar idea as HICAMP's hardware path compaction. Cache awareness is the new main optimization target. An advantage of the HICAMP bitmap index is the DAG-structure-aware hardware prefetching, which solves the cache miss problem in most pointer-chasing structures.

Novel memory systems create new opportunities in database design. Pelley et.al. [20] reconsidered OLTP durability management to optimize recovery performance and forward-processing throughput with non-volatile RAMs. Litz et. al. [13] proposed new memory features to support multiversion concurrency control in hardware and accelerate transaction processing.

6. CONCLUSION AND FUTURE WORK

In this paper, we have shown that fine-grained data deduplication and the DAG structure of HICAMP memory enable efficient updates on a bitmap index while maintaining a high compaction ratio. The fine-grained deduplication reduces the size of the data structure as well as memory footprints. The experiments show that the size of a bitmap index can be an order of magnitude smaller than B+tree. Utilizing the DAG structure, our HICAMP bitmap index exploits the sparsity of indices enabling efficient scan operations. The HICAMP bitmap exhibits a similar size as software compressed bitmaps but supports efficient updates. Additionally, the HICAMP bitmap supports random lookup to locate the bit to be updated in complexity O(log n). These properties enable bitmap indices for OLTP workloads.

To address the problem of candidate checking, we designed a multi-bit bitmap which encodes a record with multiple bits and decreases the need for candidate checks. In the experiment, multi-bit bitmap demonstrates 3∼30x smaller in size than tree-based indices under both uniform and zipf distributions.

Lastly, this work demonstrates how hardware innovation enables break the conflict between space cost and data manipulation plagued by compression. In-memory databases need online data structures that have small memory consumption and support efficient updates simultaneously.

In the next step, we plan to evaluate concurrent transactional bitmap index update on HICAMP. Besides deduplication and DAG structure, HICAMP also provides a copy-on-write mechanism which efficiently enables light-weight snapshot isolation. Writes to a leaf are not updated in place but on a new local version of the leaf, which merges into the global version at commit time. We would like to implement transactional update on bitmap index and compare it against concurrent tree-based indexing structures. Besides, we are building a columnar in-memory database on HICAMP memory to handle both OLTP and OLAP workloads. We are interested in integrating the bitmap index into the database to evaluate its performance on both types of workloads.

7. ACKNOWLEDGEMENT

The authors would like to thank Michael Chan for his careful review and valuable feedback on the draft. The authors are grateful to Amin Firoozshahian and Alex Solomatnikov of HICAMP Systems, Inc. for providing the HICAMP memory model simulator.

8. REFERENCES

[1] G. Antoshenkov. Byte-aligned bitmap compression. *Technical report, Oracle Corp. U.S. Patent number 5,363,098.*

[2] T. Bingmann. STX B+Tree. https://panthema.net/2007/stx-btree/, 2013.

[3] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi. Hicamp: Architectural support for efficient concurrency-safe shared structured data access. *SIGARCH Comput. Archit. News*, 40(1):287–300, Mar. 2012.

[4] A. Colantonio and R. D. Pietro. Concise: Compressed 'n' composable integer set. *CoRR*, 2010.

[5] T. P. P. Council. TPC-H Specification. http://www.tpc.org/tpch/spec/tpch2.16.0.pdf, 2013.

[6] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[7] G. Graefe, H. Kimura, and H. Kuno. Foster b-trees. *ACM Trans. Database Syst.*, 37(3):17:1–17:29, 2012.

[8] 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 *International Conference on Management of Data*. ACM, 2010.

[9] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. KISS-Tree: Smart latch-free in-memory indexing on modern architectures. In *International Workshop on Data Management on New Hardware*. ACM, 2012.

[10] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core cpus. *Proc. VLDB Endow.*, 5(1):61–72, Sept. 2011.

[11] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, Dec. 1981.

[12] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *International Conference on Data Engineering*. IEEE, 2013.

[13] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson. SI-TM: Reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 383–398, New York, NY, USA, 2014. ACM.

[14] D. B. Lomet, S. Sengupta, and J. J. Levandoski. The Bw-Tree: A b-tree for new hardware platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 302–313, Washington, DC, USA, 2013. IEEE Computer Society.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[16] D. Osmari. STL AVL Map. http://stlavlmap.sourceforge.net/, 2008.

[17] D. Sanchez and C. Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *SIGARCH Comput. Archit. News*, 41(3):475–486, June 2013.

[18] C. Saulnier. CS::SkipList C++ Generic Template Container Library. http://csskiplist.sourceforge.net/.

[19] L. Sidirourgos and M. Kersten. Column imprints: a secondary index structure. In *Proceedings of the 2013 international conference on Management of Data*, pages 893–904. ACM, 2013.

[20] B. T. G. B. B. Steven Pelley, Thomas F. Wenisch. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment*, 7:121–132, Oct. 2013.

[21] K. Stockinger and K. Wu. Bitmap indices for data warehouses. *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, 5:157–178, 2007.

[22] H. S. Warren. *Hacker's Delight*. Addison-Wesley, Boston, MA, USA, 2002.

[23] K. Wu. Fastbit: an efficient indexing technology for accelerating data-intensive science. In *Journal of Physics: Conference Series*, volume 16, page 556. IOP Publishing, 2005.

[24] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, Mar. 2006.