UpBit: Scalable In-Memory Updatable Bitmap Indexing

Manos Athanassoulis
Harvard University
manos@seas.harvard.edu

Zheng Yan*
University of Maryland
zhengyan@cs.umd.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

ABSTRACT

Bitmap indexes are widely used in both scientific and commercial databases. They bring fast read performance for specific types of queries, such as equality and selective range queries. A major drawback of bitmap indexes, however, is that supporting updates is particularly costly. Bitmap indexes are kept compressed to minimize storage footprint; as a result, updating a bitmap index requires the expensive step of decoding and then encoding a bitvector. Today, more and more applications need support for both reads and writes, blurring the boundaries between analytical processing and transaction processing. This requires new system designs and access methods that support general updates and, at the same time, offer competitive read performance.

In this paper, we propose scalable in-memory **Up**datable **Bit**map indexing (UpBit), which offers efficient updates, without hurting read performance. UpBit relies on two design points. First, in addition to the main bitvector for each domain value, UpBit maintains an update bitvector, to keep track of updated values. Effectively, every update can now be directed to a highly-compressible, easy-to-update bitvector. While update bitvectors double the amount of uncompressed data, they are sparse, and as a result their compressed size is small. Second, we introduce *fence pointers* in all update bitvectors which allow for efficient retrieval of a value at an arbitrary position. Using both synthetic and real-life data, we demonstrate that UpBit significantly outperforms state-of-the-art bitmap indexes for workloads that contain both reads and writes. In particular, compared to update-optimized bitmap index designs UpBit is $15-29\times$ faster in terms of update time and $2.7\times$ faster in terms of read performance. In addition, compared to read-optimized bitmap index designs UpBit achieves efficient and scalable updates ($51-115\times$ lower update latency), while allowing for comparable read performance, having up to 8% overhead.

1. INTRODUCTION

Bitmap indexing is a popular indexing technique for large data sets. It leverages fast bitwise operations [9, 10, 20, 21, 32], and is commonly used for a number of applications ranging from scientific data management [33] to analytics and data warehousing [11, 19, 24, 26, 27, 36]. Bitmap indexes are used by several popular database systems, including open-source systems like PostgreSQL and commercial systems like Oracle [25], SybaseIQ [19, 21], and DB2 [7].

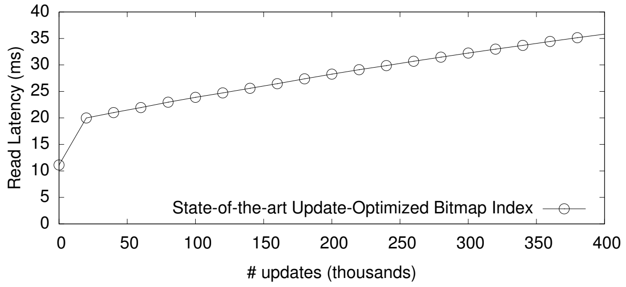


Figure 1: The problem: Read latency of the state-of-the-art update-optimized bitmap index UCB [8] does not scale with # of updates.

Bitmap indexes demonstrate significant benefits especially for equality or low selectivity queries as we can quickly get all qualifying rowIDs. In addition, they efficiently support analytical queries that include logical operations in the `where` clause, as well as ad-hoc queries with multiple selections. Bitmap indexes are also useful for counting values, as well as counting NULLs, something typically not supported by other indexes.

A basic bitmap index consists of multiple bitvectors[1]; one for each distinct value of the domain of the indexed attribute A. The k_{th} position of a bitvector corresponding to value v is set to one if the k_{th} position of the attribute A is equal to v. While without compression the storage requirements for a bitmap index are very high (the space required is proportional to the product of the relation size multiplied by the domain cardinality of the indexed column), several compression techniques have been proposed to greatly reduce bitmap index size [1, 13, 14, 15, 16, 18, 34]. Typically these techniques apply variations of run-length encoding in order to reduce the size down to the same order of magnitude as the indexed column. Hence, every read operation requires a bitvector decode while every update operation requires a decode followed by an encode.

The Problem: Scalability for Updates. The major drawback of bitmaps indexes is offering updates. This is exacerbated by the increasing need of applications to require support for both efficient reads and updates, to deliver both good read performance and data freshness [2, 3, 5, 17, 22, 30].

Read-optimized bitmap index designs are not suitable for updates. The reason is that updating a bitmap index requires a series

*Work done while the author was a year-long intern at Harvard University.

[1]We use the term *bitmap index* for the access method and *bitvector* for its building block throughout this paper.

SIGMOD '16, June 26–July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915964

of decoding and encoding actions on the individual bitvectors that correspond to the old and new values. In order to efficiently support updates, a bitmap index ideally should avoid repetitive decodings and re-encodings.

Update Conscious Bitmaps [8] (UCB) is the state-of-the-art design for update-optimized bitmap indexes. The core idea of UCB design is the auxiliary *existence bitvector*[2] (EB) [21] that stores information about whether specific bits in the actual bitvectors are valid or not. A significant drawback of UCB is that read performance does not scale with updates. Figure 1 shows that the read latency of UCB increases by more than $2\times$ as updates are applied to the bitmap index. In other words, as more updates arrive, read queries become increasingly more expensive (the graph and the experimental setup is further discussed in Section 4).

UpBit: Scalable Updates in Bitmap Indexing. We present UpBit, a scalable in-memory **Up**datable **Bit**map index design. UpBit achieves both (i) efficient reads, similar to read-optimized indexes, and (ii) lower update time than both read-optimized and update-optimized bitmap indexes. UpBit read performance remains stable as more updates, deletes, or inserts are applied.

Similar to read-optimized bitmap indexes, for every value v of the domain of the indexed attribute, UpBit maintains a *value bitvector* (VB). Each value bitvector stores a bit set to one at the positions having value v for the indexed column, and zero otherwise. UpBit introduces two new design elements to achieve efficient updates and to maintain good read performance as updates accumulate.

First, for every value bitvector, UpBit introduces a corresponding *update bitvector* (UB). Every incoming update results in two changes, one in the update bitvector corresponding to the old value, and another in the update bitvector of the new value. The aggregate size of the uncompressed bitvectors for UpBit is twice as much as for a read-optimized bitmap index, however, at any point in time each update bitvector has only a small number of bits set to one. This allows for negligible storage and CPU decoding overhead. In order to maintain high compressibility the update bitvectors are periodically merged with value bitvectors and re-initialized. When queried, UpBit combines – through a bitwise XOR – the value bitvector and the corresponding update bitvector. When updated, UpBit only needs to flip two bits, one in the UB of the new value and on in the UB of the old value.

Second, UpBit introduces *fence pointers* to allow efficient retrieval of arbitrary values. Fence pointers allow direct access to any position of a compressed bitvector. As a result, fence pointers help to avoid unnecessary decodings by allowing a read query to jump directly to the relevant area and decode only a small part of the bitvector that contains the target information. Using fence pointers as pivot elements also enables efficient multi-threaded decoding of a bitvector during querying with minimal synchronization needed during reconstruction.

While the use of multiple update bitvectors seems like more work than updating in-place or simply having a single existence bitvector, it allows UpBit to achieve drastically better read and update performance, following the intuition of the RUM Conjecture [4], because: (i) there is no single bitvector that receives all updates, hence compressibility is higher and updating does not create a single bottleneck on the auxiliary bitvectors; the cost is distributed to multiple update bitvectors, (ii) fence pointers minimize decoding and encoding action when updating and reading from the update bitvectors, and (iii) UpBit maintains a threshold of compressibility after which value bitvectors and update bitvectors are merged, and update bitvectors are reset. When the number of updates stored in an update bitvector exceeds this threshold, UpBit performs the merge operation during the following read query on the corresponding domain value, which involves this particular bitvector. Each read query decodes and on-the-fly merges the corresponding value and update bitvectors, hence, in the merge-back operation the read query initiates a write-back as well. Since UpBit maintains one update bitvector per value of the domain, each pair of VB and UB can be independently merged in a query-driven fashion, without having to alter the structure of the entire bitmap index.

UpBit for Modern Data Stores. A bitmap index can be beneficial for a modern data system, regardless of whether the physical data organization is a single column, or groups of columns. UpBit's design is orthogonal to the underlying data layout, and it benefits the read latency in either case. The benefit of using UpBit is pronounced when data is organized in groups of columns, as using a secondary index avoids reading more data.

Contributions. In summary, we make the following contributions.

- We introduce UpBit, an updatable bitmap index that allows fast updates without sacrificing read performance. UpBit is based on two design points:
  - We introduce *update bitvectors*, a set of additional auxiliary bitvectors (one per value of the domain), that distribute the update burden.
  - We introduce *fence pointers* for compressed bitvectors as a way to avoid decoding entire bitvectors and efficiently access only their relevant parts. Fence pointers also enable efficient multi-threaded bitvector decoding when querying.
- We show that we can achieve scalability in terms of the number of updates by periodically merging the changes to make sure the compressibility of the auxiliary bitvectors is high.
- Through experimentation with synthetic and real data we demonstrate the efficiency of UpBit in a variety of scenarios varying update rate, data size, and cardinality of the indexed column. We show that UpBit offers consistently negligible or small read latency overhead when compared with a read-optimized bitmap index. In addition, when compared with a state-of-the-art update-optimized bitmap index, UpBit achieves significantly lower update cost and much faster reads, scaling with the number of updates.

2. BITMAP INDEXES & UPDATES

In this section, we provide background on bitmap index literature and we motivate our work by showing that existing bitmap indexes are not suitable for workloads with frequent updates.

Bitmap Indexes. Initially designed for static analytic workloads, bitmap indexes have been used in a variety of applications, targeting both equality and range queries. A general binary representation of the content of a column was first introduced using a variable number of bits per value [32], however, typically today bitmap indexes use a single bit per row [20, 21], and one bitvector per value containing ones for rows that are equal to this value and zeros otherwise. An unencoded bitmap index is depicted in Figure 2. The figure shows a column with three different values, and three bitvectors, each containing a bit set to one when the corresponding position is equal to the value each stores.

Keeping Bitvectors Small. Bitvectors contain a lot of redundant data (a bit for every value of the domain for every position of the

[2]Called *existence bitmap* in prior work; here we use *existence bitvector* to maintain consistent terminology throughout the paper.

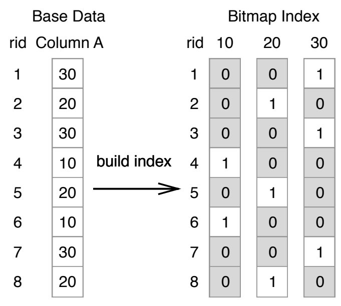


Figure 2: Read-Optimized Bitmap Index.

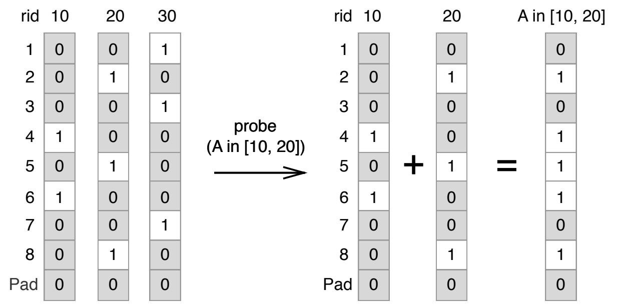
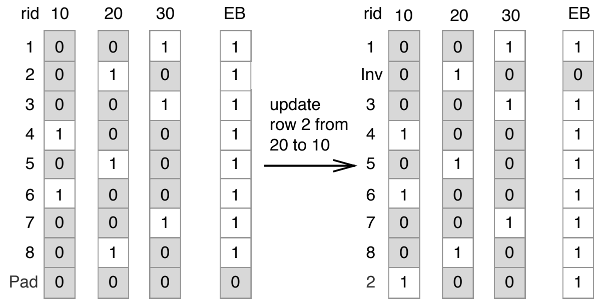
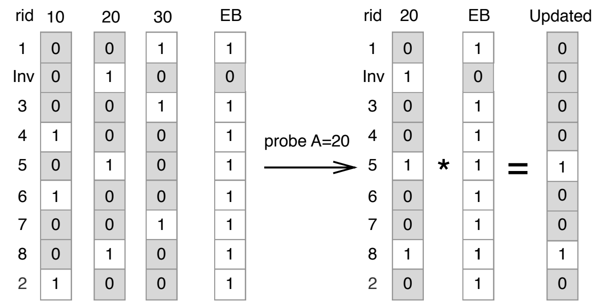


Figure 3: Searching a Bitmap Index for $A \in [10, 20]$.



(a) Update value of second row from 20 to 10 using UCB.



(b) Probe for value B using UCB.

Figure 4: Illustration of Update Conscious Bitmaps [8].

column). This redundancy can be easily reduced by employing appropriate encoding schemes [9, 10]. In the case of range queries, bitmap indexes either perform bitwise OR operations between the corresponding bitvectors, as shown in Figure 3, or employ a different encoding scheme (range encoding), which however, has much greater space requirements [10]. Read performance is further improved by using compression in conjunction with encoding. The first such design was Byte-aligned Bitmap Compression (BBC) [1]. Word-Aligned Hybrid (WAH) [34] encoding applies run-length encoding and has a few follow-up variations. To ensure memory-alignment on modern architectures, WAH splits bits into words (32 bits each), which become the unit of compression. Using this encoding, a maximum run of 2^{32} identical bits can be represented by a single WAH fill word. However, in practice runs with identical bits are shorter and require less bits to be encoded. This was addressed by follow-up proposals which use a few bits per word to signify a long run of identical bits, or a specific pattern interleaved with long runs. CONCISE [13] and Position List Word Aligned Hybrid (PLWAH) [14] improve compression ratio when a single bit or a short pattern interrupts a long run. In this work we use the bitvector implementation of FastBit [33] which relies on WAH compression [34].

Updates. Traditionally, bitmap indexes have been built for read-intensive workloads, and as a result they are not well-suited for updates [9, 32]. This happens because in-place updates cause a costly decoding of the whole bitvector and a re-encoding to store its updated version. For example to update rowID k from value v_1 to value v_2 we would need to go to position k in the value bitvector for value v_1 and set the bit to zero, and then set the k_{th} bit of the value bitvector for v_2 equal to one. Both bitvectors need to be decoded in order to find the bits to be updated. The decoding begins at the very beginning of the bitvector, and after the update the bitvector needs to be encoded as a whole anew. On the contrary, handling new rows is easier as new bitvectors can be appended at the end of the current ones without the costly decode-then-encode cycle.

Update-Conscious Bitmaps. To handle the update problem, a number of approaches have been proposed. The state of the art for update-optimized bitmap index is the Update Conscious Bitmaps (UCB) [8], which supports efficient deletes. Updates are treated as a delete-then-insert using the new delete mechanism, while the new value is appended in the end of the bitvector and a mapping between the invalidated position and the appended value is kept.

Update Conscious Bitmaps use – in addition to a bitvector for each value of the domain – a bitvector to store whether a given row of the indexed column has been updated, called the existence bitvector (EB) [21]. When a position x in EB is 0, the corresponding rows of all value bitvectors are invalidated. When a bit of EB is 1, the bitmap index can be accessed as usual. That way, every delete is performed by setting to 0 the bit of the EB corresponding to the deleted row. In-place updates are transformed to delete-then-append operations (Figure 4a). By avoiding decoding and encoding the value bitvectors at the time of an update, UCB offers faster updates than other approaches, and particularly efficient deletes.

Reading from UCB, however, requires an additional AND operation between the value and the existence bitvector (Figure 4b). When all value bitvector positions are valid, and hence, every bit of the existence bitvector is set to one, this bitwise AND is very efficient. As more updates or deletes are applied, however, the existence bitvector becomes less compressible because it contains more bits that have been set to zero. Figure 1 plots the average read latency of UCB as updates are applied over a relation with 100M values of synthetic data in an in-memory setup (more details about the experimental setup are in Section 4.1). The y-axis plots the read latency as a function of the updates cumulatively applied on UCB, shown on the x-axis. UCB read performance does not scale with the number of updates. The reason is three-fold. First, the scattered updates deteriorate the compressibility of the bitvectors. Second, every update initiates both a decode and re-encode operation. Third, in order to support updates, UCB uses an additional level of indirection maintaining the mapping of rowIDs for which the values stored in the bitvector has been invalidated.

Figure 5 shows the average query latency in the same experiment presented in Figure 1, when compared with a read-optimized bitmap index that supports in-place updates. The black bars corre-

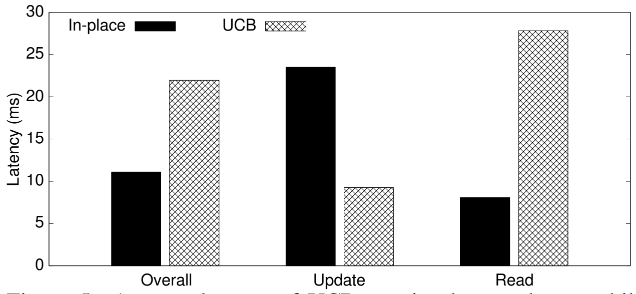


Figure 5: Average latency of UCB vs. in-place updates: while UCB offers 2.88x faster execution for updates, searching with a read-optimized bitmap index that employs in-place updates is 3.57x faster, leading to a 2x overall performance degradation for a mixed workload, with 10% updates.

spond to a bitmap index with naïve in-place updates and the pattern bars correspond to UCB. The first two bars show the average query latency for the full workload (90% read queries and 10% updates). The second pair of bars shows the average *update* latency for the in-place updates and UCB, and the third pair shows the average *read* latency for the two approaches. While UCB offers 2.57× faster update performance when compared to in-place updates, both UCB's read latency (3.45×), and overall average query latency (1.98×) is significantly higher than in-place updates. When answering a read query, UCB needs to perform an additional AND operation with the existence bitvector which, in turn, needs to decode the entire value bitvector. In addition, UCB needs to consult a translation table for every invalidated row, and consequently do another AND operation. The fast update operation – just update a single bit in a highly compressible bitvector to begin with – is not enough to address the slow query time. This behavior is exacerbated as more updates accumulate, as shown in Figure 1.

In order to better understand why UCB degrades as more updates are accumulated, Figure 6 shows the breakdown of the response time of equality queries and updates for UCB. In particular, Figure 6 presents the breakdown of the response time of equality queries and updates when we use the UCB design, (i) after the first update, (ii) after 1K updates, (iii) after 10K updates, (iv) after 200K updates, and, (v) after 1M updates. The cost to update in-place does not depend on the number of past updates, while both costs to update and to read using UCB depends heavily on the past updates. Further, UCB initially minimizes the updating cost because it only changes one bit in a highly compressible bitvector and completely avoids the update cost by updating the existence bit. This approach leads to very fast updates as long as there are only a few bits in the existence bitvector that are set to zero, and as a result, the existence bitvector is highly compressible. As we have more updates, however, the existence bitvector becomes less compressible, hence, updating it becomes more expensive, leading to higher overall update cost. During a read query, UCB always has to pay the price of decoding the bitvector corresponding to the search predicate, as would any design. Decoding the value bitvectors is a fixed cost that is not affected by the number of updates. UCB, however, requires the additional step of the bitwise AND operation between the bitvectors and the EB, creating a cost factor which increases as more updates accumulate (and EB becomes less compressible).

Updating Bitmap Indexes With Low Impact on Read Performance. Figure 6 helps us pinpoint what should be optimized in order to offer efficient updates. The update cost of UCB contains the cost of updating the existence bitvector. While this cost is negligible for up to 10k updates, as more updates are applied it increases exponentially. In addition, the read time due to the bitwise AND between the value and the existence bitvector is the

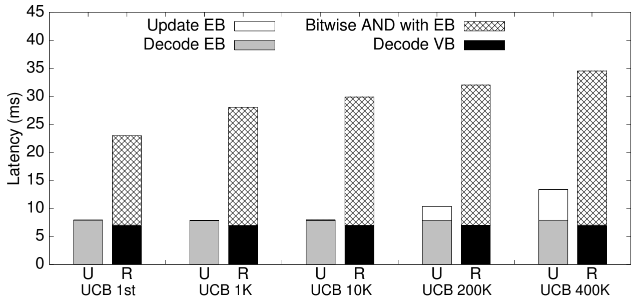


Figure 6: Breakdown of update and read time: as more updates accumulate UCB spends more time in updating the EB when updating and ANDing VB with EB when querying.

most expensive operation. The approach proposed in this paper, UpBit, aims at addressing these two factors. First, by using multiple update bitvectors and periodically merging them during query time with the corresponding value bitvector, we guarantee that the cost to update the auxiliary bitvector is always kept low. Second, by distributing the update overhead to multiple UB (as opposed to one EB) we have higher compressibility and we bound the query cost as well. In addition, having multiple UBs allows the adaptive reconstruction of VB by merging them on a query-driven basis.

3. UpBit: EFFICIENT READS AND WRITES FOR BITMAP INDEXES

In this section, we present UpBit in detail, and describe how it can offer efficient updates without compromising read query performance. UpBit shifts the update burden from a single bitvector (the existence bitvector of UCB) to multiple update bitvectors, and builds an efficient way to search the value of a specific position of an encoded bitvector by introducing fence pointers.

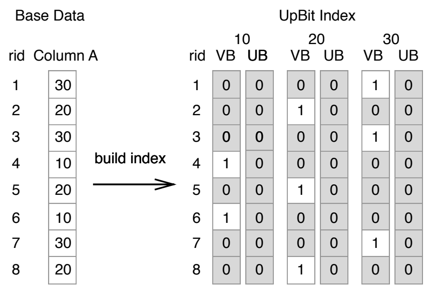
We begin by introducing the notation and the UpBit structure. Then we describe each operation: probing the index, deleting a row, updating a value, inserting new data, and any auxiliary operation needed to support these actions.

3.1 UpBit Data Structure

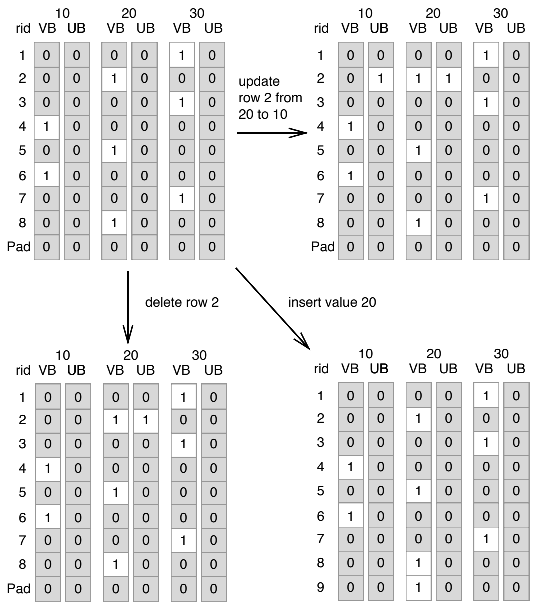
Notation. UpBit is a bitmap index over an attribute A of a table T. The domain of A has d unique values, hence UpBit contains d value bitvectors $VB = \{V_1, V_2, ..., V_d\}$ and d update bitvectors, $UB = \{U_1, U_2, ..., U_d\}$.

The Internals of UpBit. For every value of the domain of the indexed attribute, UpBit stores a value bitvector showing which rows contain the corresponding value. For example, Figure 7a shows the three bitvectors corresponding to the attribute A values 10, 20, and 30, respectively. Rows 4 and 6 are equal to 10, rows 2, 5 and 8 are equal to 20, and rows 1, 3, and 7 are equal to 30, and as a result, the value bitvectors have the corresponding bits set to one. In addition, UpBit uses one additional bitvector per value, termed the update bitvector (UB). Each update bitvector is initialized with zeros with size equal to the one of the value bitvectors of the index. The update bitvectors are used to mark which bits of the value bitvector are changed. In particular, the current value of a row of the UpBit is given by the XOR between the corresponding position of the value and the update bitvectors. The update bitvectors distribute the burden of each update to multiple bitvectors, keeping the update cost low, and, at the same time, the compressibility of these bitvectors high in order to have minimal impact on read performance.

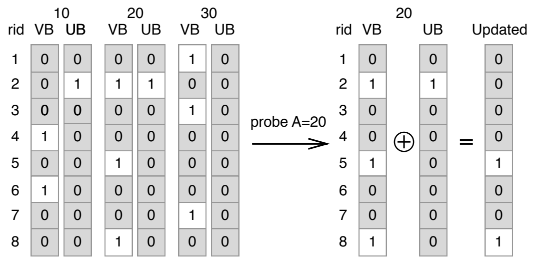
Value-Bitvector Mapping (VBM). A common operation in any bitmap index is to locate the bitvector, V_i, corresponding to a specific value, v, of the domain. Any query that needs to search the



(a) The internals of UpBit.



(b) Updating, deleting, and inserting using UpBit.



(c) Search for a single value with UpBit.
Figure 7: Illustration of UpBit.

bitmap index performs this action. This is typically implemented using a dictionary which returns the bitvector id i, given v.

No Space Overhead. The update bitvectors are initially empty, thus, they can be very efficiently compressed as 0-Fill words. In the beginning of UpBit's operation, the update bitvectors are practically single words, hence, they have negligible impact on the overall size of the index.

Interpreting Update Bitvectors. In order to return the value of the position of any value bitvector, UpBit performs a bitwise XOR between this value bitvector and the corresponding update bitvector, as shown in Figure 7c. When a column is indexed with UpBit the update bitvectors are initialized to zero (shown in Figure 7a). Effectively, the XOR between the VB and UB at the initial state of the index will always result to the VB itself. As an optimization, we always maintain a counter of the number of bits set to one in each update bitvector and if it is equal to zero, we can avoid performing the XOR altogether.

3.2 UpBit Operations

Let us now consider, in detail, how each of the probe, delete, update, and insert operations are designed using UpBit.

Searching. We first discuss how UpBit is probed for exact match queries; that is, how we find whether a value, *val*, exists in the indexed column, and in which position.

The first step is to find the bitvector i that corresponds to *val*, using the VBM which links values to bitvectors. The next step is to check whether any updates have occurred that are marked on U_i. If all bits in U_i are unset then no updates exist; more specifically, there have been no updates on *val*. Otherwise, if even a single bit is set, then updates on *val* exist and UpBit has to account for them. In order to return the right bitvector UpBit performs a bitwise XOR between V_i and U_i.

search (index: *UpBit*, **value:** *val*)

```
1: Find the i bitvector that val corresponds to
2: if $U_i$ contains only zero then
3:     return $V_i$
4: else
5:     return $V_i \oplus U_i$
6: end if
```

Algorithm 1: Searching UpBit for value *val*.

For example, in Figure 7c we probe for value equal to 20. In this case we first check the update bitvector for value 20 and if not all bits are set to zero, UpBit performs a XOR between U_i and V_i in order to get the positions that contain 20 (Algorithm 1).

Deleting a Row. We continue with how UpBit handles the deletion of a row. In this case we assume that the workload consists of the delete of a specific row, k.

delete_row (index: *UpBit*, **row:** k)

```
1: Find the val of row k
2: Find the i bitvector that val corresponds to
3: $U_i[k] = \neg U_i[k]$
```

Algorithm 2: Deleting row k with UpBit.

In order to make sure that the index will never return row k as an answer to any query, we first need to retrieve the value of this row (line 1 of Algorithm 2), and then find the update bitvector corresponding to this value B_i (line 2 of Algorithm 2). The first operation constitutes one of the main challenges when supporting general updates in bitmap indexes: *retrieve the value of an arbitrary row*. The last step of deleting a row is to negate the contents of the selected update bitvector for row k: $U_i[k] = \neg U_i[k]$. The whole process is shown by Algorithm 2, and is depicted in Figure 7b in the bottom left part. In prior work, deletes are supported through the existence bitvector which invalidates the whole row [8, 21], in order to avoid the costly retrieval of the value of an arbitrary position. Here, we discuss how to efficiently do so.

get_value (index: *UpBit*, **row:** k)

```
1: for each i ∈ $\{1,2,...,d\}$ do
2:     $temp\_bit = V_i.get\_bit(k) \oplus U_i.get\_bit(k)$
3:     if temp_bit then
4:         Return $val_i$
5:     end if
6: end for
```

Algorithm 3: Get value of row k using UpBit.

Retrieving the Value of a Row. Algorithm 3 shows how to retrieve the value of row k. Instead of going to base data which may be on disk, we search all bitvectors. This operation can be aggressively parallelized as it is embarrassingly parallel: each bitvector search can run independently. In fact, for every bitvector we run Algorithm 4 using a small amount of additional metadata to avoid decoding the whole bitvector, effectively reducing the decoding cost.

get_bit (bitvector: *B*, row: *k*)

```
$pos = fence\_pointer.nearest(k)$
while $pos < k$ do
    if $isFill(B[pos])$ then
        $value, length = decode(B[pos])$
        if $(pos + length) * 31 < k$ then
            $pos += length$
        else
            Return $value$
        end if
    else
        if $pos * 31 - k < 31$ then
            Return $B[pos] \& (1 << (k \% 31))$
        else
            $pos++$
        end if
    end if
end while
```

Algorithm 4: Get k_{th} bit of a bitvector using UpBit.

Reducing Decoding Cost. A large fraction of the time to get the value of the k_{th} row goes to decoding the bitvectors before the XOR operation, and reading the k_{th} bit. The reason why this cost is high, is that in order to access the k_{th} bit we need to decode all previous bits; there is no way to decode a subset of the bitvector knowing in which position the decoding starts with the existing encoding schemes. We address this performance limitation by adding *fence pointers* which enable efficient partial decoding close to any arbitrary position[3]. Both encoded and not encoded bitvectors are organized in words. The words of a not encoded bitvector contain its raw bits, while the words of an encoded bitvector may represent multiple unencoded words. The metadata of fence pointers form a small separate index of the encoded bitvector which allows direct access to the encoded words corresponding at known positions of the not encoded bitvector. Fence pointers have a predefined granularity on the not encoded bitvector, g. For a bitvector with n not encoded words, fence pointer i stores the offset of the encoded word that contains the unencoded word $i \cdot g: \; i = 1, 2, ..., \frac{n}{g}$. Every encoded word is aligned to an unencoded word (the reverse is not always true, i.e., a single encoded word encodes multiple unencoded words). As a result the granularity g is only approximate. Figure 8 details such an example. The array on the top contains the unencoded word id (fence) and the offset of the encoded word that starts with the contents of the said word id. An encoded word always starts at the beginning of a not encoded word, which in turn always contains 31 bits. In this example we are looking for the position of bit 62073, this will be in the $62073 \div 31 = 2002$ unencoded word, in position $62073 \bmod 31 = 11$. Using the fence pointer index we find that we need to decode $w97$ to retrieve unencoded word 2002, and then look for the 11th bit. If we were looking for the position of bit 62150 we would probably need to decode more than one word (e.g., continue to word $w98$) depending on the exact distribution of the bits. In general, fence pointers help to avoid both unnecessary CPU processing (decoding) and memory consumption (storing the unnecessary parts of the decoded bitvector). However, if the granularity becomes too fine (g becomes too small) the overhead of maintaining fence pointers increases; we effectively use more space to index the same information. In Sections 4.4 and 4.6 we show what granularity is beneficial through extensive experimentation.

Updating a Row. The most challenging operation is updating the value of a row. We assume that the operation requests to change the value of row k to val. The top part of Figure 7b depicts the update operation of row 2 from 20 to 10.

[3]The existence bitvector of UCB can also benefit from fence pointers; UCB needs to checks whether a position is valid.

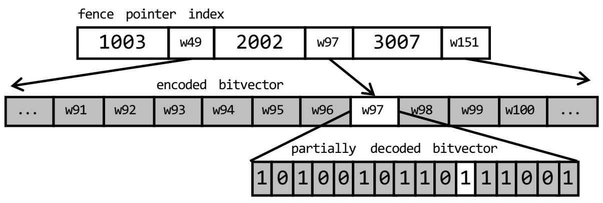


Figure 8: Fence pointers allow efficient partial decoding by storing a mapping between encoded and unencoded words of the bitvector.

In order to indicate the new value, UpBit flips the bit of position k of the bitvector corresponding to val (U_i). However, this is not enough, because we need to negate the bit of position k of the bitvector corresponding to the old value as well (U_j). Since we do not necessarily know the old value of position k, we need to retrieve this value similarly to what we described before. The last two steps are the negation of the corresponding bits: $U_i[k] = \neg U_i[k]$, and $U_j[k] = \neg U_j[k]$. Algorithm 5 describes this process.

update_row (index: *UpBit*, row: *k*, value: *val*)

```
Find the $i$ bitvector that $val$ corresponds to
Find the old value $old\_val$ of row $k$
Find the $j$ bitvector that $old\_val$ corresponds to
$U_i[k] = \neg U_i[k]$
$U_j[k] = \neg U_j[k]$
```

Algorithm 5: Update row k to val with UpBit.

Inserting a Row. The last operation with a bitmap index is inserting a new row. This operation does not require to update existing values, hence, we do not need to decode any bitvectors; we extend the current bitvectors so there is some necessary bookkeeping.

We first need to find the bitvector corresponding to val (U_i). Following Fastbit implementation of WAH, we use active words [34, 35] to append at the end of the bitvectors. The last word stores two variables: the literal value and the number of bits. Hence, we do not need to fully decode the bitvector to get the last word. Typically, the new bit can be appended, and the size of the bitvector is increased without a physical expansion of the encoded bitvector. In rare cases, the last word is not enough and the bitvector needs to be extended. In this case, the bitvector is extended by a word and additional padding space is added in the end of the bitvector for future inserts. Once the padding space is available, we increase the U_i size by one element and we set the new bit equal to one on the B_i bitvector. For example, in the bottom right part of Figure 7b we see how inserting value 20 is handled. The overall process is also described in Algorithm 6.

insert_row (index: *UpBit*, value: *val*)

```
Find the $i$ bitvector that $val$ corresponds
if $U_i$ does not have enough empty padding space then
    Extend $U_i$ padding space
end if
$U_i.\#elements++$
$U_i[\#elements] = 1$
```

Algorithm 6: Insert new value, val.

While the insertion can happen directly at V_i as well, we select to use U_i because it is typically smaller and more compressible, hence every operation is more efficient. In addition, we follow a merging policy between value and update bitvectors during read queries to exploit existing processing as we discuss in Section 3.3. A major difference of our approach when compared against prior work [8] is that because it does not invalidate rows, there is no need for keeping a mapping when the value of a row is updated. However, as updates or deletes accumulate update bitvectors become less compressible and need to be merged back with the main value bitvectors.

3.3 Scaling With the Number of Updates

As UpBit accumulates updates, deletes and inserts, its update bitvectors become unavoidably less compressible, leading to expensive bitwise operations and decodings. We address this problem by merging each update bitvector with the corresponding value bitvector, when their size is larger than a threshold.

The decision whether the two bitvectors are going to be merged is based on the number of accumulated updates. We describe in Section 4.4 how to tune the merging threshold. Once the threshold is determined to T updates, UpBit can start merging the update and the value bitvector as follows: it monitors the number of updates applied to each bitvector independently and when the accumulated updates are more than T, the corresponding update bitvector is marked as "to be merged". In the next search operation involving this bitvector the merging between the value and the update bitvector will happen in order to produce the result expected by the user ($V_i \oplus U_i$). This result will also be saved as the new value bitvector while the update bitvector will be re-initialized. This process is described in Algorithm 7.

merge (index: *UpBit*, bitvector: *i*)

```
1:  $V_i = V_i \oplus U_i$
2:  $comp\_pos = 0$
3:  $uncomp\_pos = 0$
4:  $last\_uncomp\_pos = 0$
5:  for each i $\in \{1, 2, ..., length(V_i)\}$ do
6:      if $isFill(V_i[pos])$ then
7:          $value, length += decode(V_i[pos])$
8:          $uncomp\_pos += length$
9:      else
10:         $uncomp\_pos++$
11:     end if
12:     if $uncomp\_pos - last\_uncomp\_pos > THRESHOLD$ then
13:         $FP.append(comp\_pos, uncomp\_pos)$
14:         $last\_uncomp\_pos = uncomp\_pos$
15:     end if
16:     $comp\_pos++$
17: end for
18: $U_i \leftarrow 0s$
```

Algorithm 7: Merge UB of bitvector *i*.

After the update threshold is met, we perform the merge during a subsequent query evaluation. The reason is that at this point we calculate anyway the result of the XOR between the value and the update bitvector to return it as a result of the read query. Hence, by writing back after a read query we avoid the unnecessary overhead of performing bitwise XOR during updates. When the bitvectors are merged, U_i is reset to zeros, while V_i is updated and its fence pointers are recalculated during encoding. Last but not least, UpBit's multiple update bitvectors allow the merging mechanism to happen in a bitvector-by-bitvector case. On the contrary, prior work UCB uses the existence bitvector, and as a result it needs to perform a merge operation of the existence bitvector with all value bitvectors at once, increasing the merging overhead drastically.

4. EXPERIMENTAL ANALYSIS

In this section we provide a detailed experimental analysis of UpBit against state-of-the-art updatable bitmap indexes and state-of-the-art read-optimized bitmap indexes. We show that UpBit brings drastically faster overall workload execution. When compared to the update-optimized bitmap index UpBit achieves both faster updates and faster reads. On the other hand, when compared to read-optimized bitmap indexes it achieves similar read performance and much faster update performance.

4.1 Experimental Methodology

Infrastructure. Our experimentation platform is a server running Debian "Wheezy" with kernel 3.18.11. The server is equipped with four sockets each with an Intel Xeon CPU E7-4820 v2, running at 2GHz. Each processor has 16MB L3 cache, and supports 8 hardware threads, and 8 hyper-threading threads, for a total of 64 hardware contexts with HT enabled. The main memory is 1TB, corresponding to 256GB per socket, and the storage is based on a RAID-5 configuration of 300GB 15KRPM SAS hard disks.

Implementation. The implementation of UpBit is a standalone implementation in C++. A key component in UpBit is the design of bitvectors. For this, we built on top of the FastBit bitvector [33] which we modified to support update bitvectors, fence pointers, and multi-threading. In order to parallelize the search of different bitvectors we use C++11 threads.

Tested Approaches. We compare UpBit with both state-of-the-art update-aware bitmap indexes and read-optimized indexes. Specifically we compare *UpBit* against (i) read-optimized bitmap indexes supporting *in-place updates*, for which we modify FastBit, and (ii) *UCB*, which we reimplemented using FastBit in order to provide a more efficient implementation and to use the same underlying bitvector logic for a fair comparison.

Select Operator API. We design and we test UpBit as a drop-in replacement for a select operator in a modern system. Its input is a column where we want to perform a selection and its output is a set of qualifying positions in the column. This is equivalent to a standard scan or index scan operator. We also compare against a modern hardware optimized scan (with multi-core, SIMD and tight for-loops) as a sanity check.

Workloads. We experiment with both synthetic and real-life data sets. We generate synthetic integer data varying two key parameters for the performance of a bitmap index: the size of the data set, n, and the cardinality of the domain, d. The distribution of the values follow either the uniform or the zipfian distribution. We further test with the Berkeley earth data [6] and with TPC-H [28].

Performance Metrics. Experiments showing response time always show the average of ten runs with standard deviation below 3% while throughput experiments show real-time behavior using a moving average of a small fraction of the experiment duration. We show both complete workload performance – that is, a mix of reads and writes – and also individual read and write performance.

4.2 Scalable Updates With UpBit

Read Performance. The first experiment demonstrates the scalability of UpBit in terms of the number of updates. The experiment has the same setup as the one shown in Figure 1. The initial dataset consists of $n = 100$M tuples, with domain cardinality $d = 100$. The workload contains equal number of updates and equality queries, both uniformly distributed across the domain (we discuss range queries later on). Figure 9 shows the results for UCB and UpBit. While UCB read latency increases unsustainably, UpBit scales, offering stable latency as more updates are applied.

Impact of Updates. Next, we show that the average read latency of a workload with updates is minimally impacted, and that the update latency is much faster than any previous approach. We show these observations for a variable percentage of updates in the workload.

We experiment with a synthetic data set consisting of $n = 100$M values, taken from a domain with $d = 100$ unique values. We present average read and update time for three different workloads: one that contains 1% updates in the query mix, one that contains 5% of updates, and finally one that contains 10% updates in the query mix of 100k operations.

Figures 10 and 11 show the results. The white bars correspond to a read-optimized bitmap index that employs in-place updates,

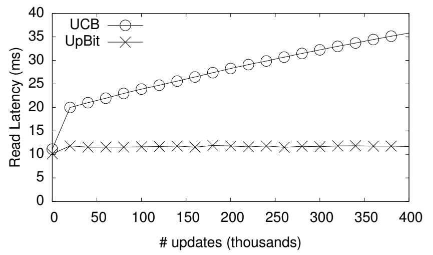


Figure 9: When stressing UpBit with updates, it delivers scalable read performance, addressing the most important limitation observed for UCB.

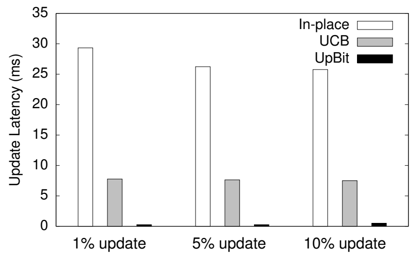


Figure 10: UpBit delivers $51-115\times$ faster updates than in-place updates and $15-29\times$ faster updates than state-of-the-art update-optimized bitmap index UCB.

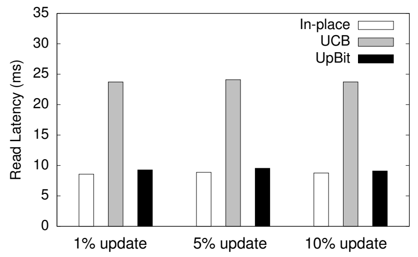
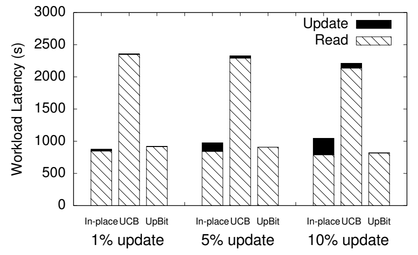
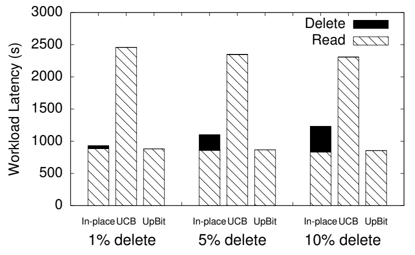


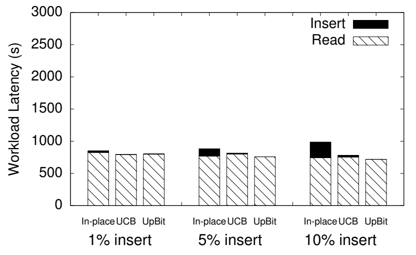
Figure 11: UpBit outperforms update-optimized indexes by nearly $3\times$ in terms of read performance while it loses only 8% compared to read-optimized indexes.



(a) UpBit vs. UCB vs. in-place for updates.



(b) UpBit vs. UCB vs. in-place for deletes.



(c) UpBit vs. UCB vs. in-place for inserts.

Figure 12: As we vary the percentage of updates, deletes or inserts from 1% to 10%, UpBit has the lowest overall workload latency when compared with any other setup. UpBit achieves similar read performance to a read-optimized bitmap index and drastically better updates (a) and deletes (b) than both read-optimized and update-optimized indexes. When inserting new values (c) all approaches have a similar low overhead on read performance. In-place updates cannot gradually absorb the new values, hence, inserting cost does not scale.

the grey bars to UCB and the black bars to UpBit. UCB delivers $3.43-3.77\times$ faster update latency than in-place updates. Our approach is significantly faster delivering $51-115\times$ faster update time than in-place updates ($15-29\times$ faster than UCB). The biggest factor of in-place updates is to actually decode and then encode the corresponding value bitvectors, while for UCB it is decoding and updating the existence bitvector. On the other hand, UpBit updates two highly compressible update bitvectors leading to much lower update latency. Figure 11 shows the impact in read latency. The read latency of the approach with in-place updates is effectively the ideal read latency of a bitmap index because there is no auxiliary bitvectors or other data to read from. UCB on the other hand merges the existence bitvector with the value bitvector when answering each query leading to a $2.71-2.77\times$ performance degradation in read latency. UpBit, however, delivers read latency with at most 8% overhead which accounts for combining with XOR the relevant parts of the value and the update bitvectors.

Figure 12a shows the overall workload latency of the previous experiment as we vary the percentage of updates in the workload. We see that the update overhead for in-place updates is very low for a very small number of updates while it drastically increases as we have more updates in the workload. On other hand, UCB has small update cost in all cases but the read cost is prohibitively high, leaving UpBit as the approach that combines low update cost scaling with updates and very low read performance overhead, leading to faster workload latency against both in-place updates and UCB.

Impact of Deletes. Next, we repeat the above experiment replacing updates with deletes. Figure 12b shows the overall workload latency of 100k operations where 1%, 5%, or 10% of them are deletes and the rest lookups. Similarly to the previous experiments, in-place updates have a drastically increasing overhead as the percentage of deletes increases and UCB has very high read overhead. On the other hand, UpBit has $1.06-1.44\times$ lower workload latency than in-place deletes by combining low read overhead and very efficient deletes.

Impact of Inserts. Figure 12c compares in-place updates, UCB, and UpBit when only inserting new values, using the same setup as the previous experiments. Since inserts are treated differently than updates and deletes (using the available padded space of the value and the update bitvectors to append new values) we expect different performance as well. Except for in-place updates that suffer as the percentage of inserts increases, UCB and UpBit perform almost the same because appending new values in the bitvectors is treated in a similar way using active words.

Mixed Workload. Figure 13 puts everything together, using a workload that write requests consist equally of Updates, Deletes and Inserts (UDI). We vary the aggregate percentage of UDI operations between 1% and 10% as in our previous experiments. The impact of updates dominates, and UpBit is the approach with the fastest workload latency. As the percentage of UDI increases the performance of in-place degrades, while UCB is slower but maintains stable performance. UpBit outperforms both approaches.

We repeat similar experiments varying a number of parameters, one at a time, to shed light on UpBit's behavior. We vary the cardinality of the indexed column, the data size, and the data distribution. Lastly, we experiment with a real-life data set which typically uses bitmap indexing.

Scaling With Cardinality. Domain cardinality is a critical parameter for bitmap indexes as it defines the number of maintained bitvectors. In this experiment we study the effect of cardinality. Figure 14 shows the overall workload latency of 100K queries over a relation with $n = 100$M values, which has a domain with $d = 1000$ unique values. On average each bitvector has now $10\times$ fewer bits set to one, and as a result, read time is about $10\times$ faster than the corresponding experiment with $d = 100$ (Figure 13). The update cost, however, remains similar to the previous experiment, and the gain when compared with the in-place updates is much bigger.

Scaling With Data Size. The behavior of all three approaches remains the same when we increase data size. Figure 15 shows the same experiment with 10K queries, repeated with $n = 1$B values

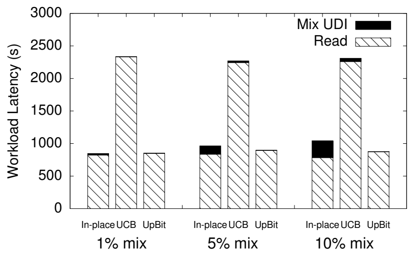


Figure 13: For general UDI workload, the overhead of maintaining a gradually less compressible EB overwhelms UCB, while UpBit offers faster workload execution than both approaches.

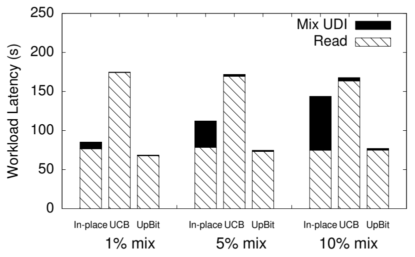


Figure 14: For a data set with larger domain cardinality ($d = 1000$) the update cost is relatively higher, and UpBit has a bigger benefit over in-place updates for the same number of updates.

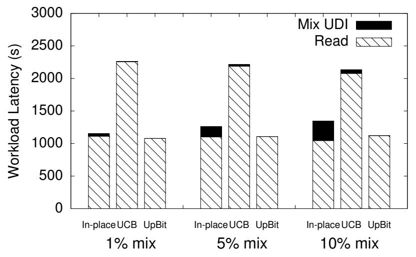


Figure 15: When increasing the data set size ($n = 1B$, $d = 100$), the qualitative behavior of all approaches remain the same. The average latency increases linearly with the data set size.

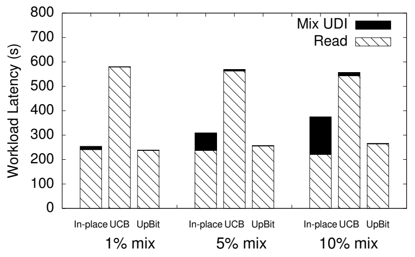


Figure 16: For skewed data (zipfian with $S = 1.5$), the latency decreases as most bitvectors are nearly empty. UpBit faces a small overhead because it has the same distribution of FPs in all VBs.

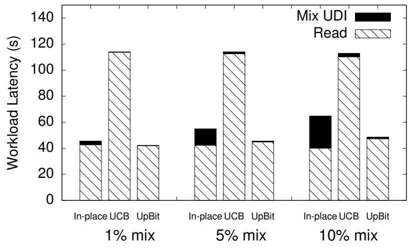


Figure 17: UpBit outperforms all other approaches with real data as well (Berkeley Earth data set with $n = 31$M values, and domain cardinality $d = 114$) for a workload with 1%, 5% or 10% updates.

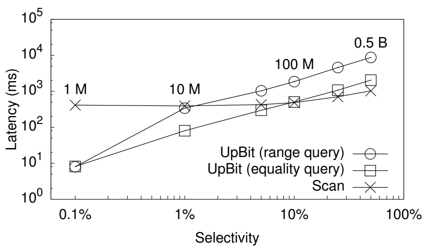


Figure 18: Compared with a fast scan, UpBit is faster for range queries with up to 1% selectivity. Equality queries with similar selectivity are much more efficient because we avoid the bitwise OR between VBs.

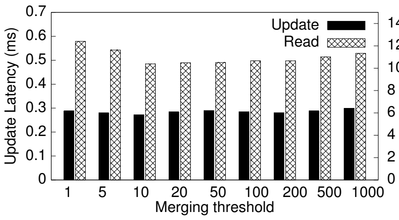
and the domain cardinality as in the initial experiment ($d = 100$). Figure 15 looks almost identical to Figure 13. The data size does not affect the performance trends and the relative behavior of different bitmap index designs. As we index more data over the same attribute we have more bits per bitvector, maintaining, however, the same number of value and update bitvectors. As a result, the relative performance of the different approaches is the same, and the absolute performance increases linearly with the data set size.

Zipfian Distribution. The distribution of the data affects the behavior of all approaches in different ways. In a uniform distribution all bitvectors get the same number of bits set to one, and as a result, they have similar compressibility. In distributions with skew, some bitvectors will contain more ones and others will be almost empty, leading to very fast querying. Figure 16 shows the overall workload latency of an experiment with $n = 100$M and $d = 100$, when the data follow the zipfian distribution ($S = 1.5$; about 40% of the items have the two most popular values, and the remaining are uniform in the domain). Read and update requests are uniformly distributed over the domain. We observe that all approaches are faster than the case where the distribution is uniform, because most bitvectors contain very few bits set to one and are highly compressible. The update cost, however, remains approximately the same as in the previous case because we still have to decode and then encode the update values. As a result, the contribution of the update cost is now increasing faster as the number of updates increases. UpBit is slightly slower than the read-optimized bitmap index, because having bitvectors with drastically different number of bits set equal to one would require a different granularity of fence pointers in each value bitvector. This would incur more translation overheads since UpBit would have to keep multiple mappings of fence pointers and positions; one for each value bitvector.

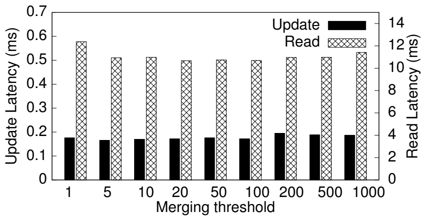
Berkeley Earth Data Set. Finally, we use UpBit to index the Berkeley earth data set showing significant performance benefits over both in-place updates and UCB. The data set contains measurements from 1.6 billion temperature reports, and is used for several climate studies. It contains information about temperature, date of measurement, and location (longitude and latitude). We use binning to query based on rounded temperature. Figure 17 shows that UpBit offers up to 1.34× smaller workload latency than in-place updates and 2.33× than UCB.

4.3 UpBit vs Modern Fast Scans

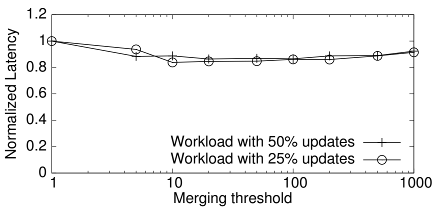
Here we compare UpBit with an in-house modern fast scan which uses tight for-loops, and exploits the available processing in terms of multi-core and SIMD implementing standard practices from the literature [23, 31, 37, 38]. Figure 18 shows the latency of a select operator over a synthetic data set of $n = 1$B values and $d = 1000$ when executed with UpBit and the modern scan, while varying the selectivity on the x-axis between 0.1% and 50%. We observe that UpBit is 65× faster than the modern scan for selecting 0.1% of the column. Since we are selecting from a column with one billion values, 0.1% values still correspond to 10^6 elements. The speedup drops sharply to 1.09× for selectivity 1%, and the break-even point is at 1.52%. When selecting 5% or more of the data UpBit is anywhere between 2.4× and 8.7× slower than the scan. We observe very similar behavior for different data sizes. Figure 18 also plots a line corresponding to the UpBit when each read query is an equality query. In this case, even though the queries have different selectivity, they all address only one bitvector; for every query the domain cardinality is different. Similar to what we observed before, UpBit is 65× faster than the scan for selectivity 0.1%. UpBit's benefits remains for 1% selectivity (4.6×) and 5% (1.4×), while for 10% selectivity UpBit and the scan have virtually the same performance. For higher selectivity the scan is more efficient: 1.5× for 25% selectivity and 1.99× for 50% selectivity. The difference between a range and an equality query with the same selectivity is the additional cost to perform the bitwise OR between the value bitvectors.



(a) Read and update latency as a function of merging threshold for a workload with 20% updates.



(b) Read and update latency as a function of merging threshold for a workload with 50% updates.



(c) Merging threshold for the overall workload combining reads and updates.

Figure 19: In workloads that have both read and update queries the optimal threshold of bits set to one before merging is 10. For lower threshold we merge too often and for higher threshold we disturb the compressibility of the update bitvectors.

4.4 Tuning UpBit

Merging UBs to VBs. As updates are being stored in the update bitvector they become less compressible, hurting the overall performance. Hence, UpBit merges value and update bitvectors, as described in Section 3.3. Figures 19a and 19b show the average update and read latency as we vary the merging threshold on the x-axis, for two different workloads; Figure 19a corresponds to a workload with 20% updates and Figure 19b to a workload with 50% updates. Note that the update latency uses the y-axis on the left hand-side and the read latency the y-axis on the right hand-side. For both workloads, we observe a small increase in update time as the merging threshold increases, since the updated bitvectors are becoming less compressible. This trend, however, is almost hidden because the update latency is dominated by the cost to get the old value of the updated row (about 93% of the execution time of an update goes to the *get_value* operation for our experiment with domain cardinality $d = 100$). Using merging threshold equal to one result in the most expensive read latency, since every read has to merge all pending updates (a cost more pronounced in Figure 19b where indeed every read is merging the previous update), but it drops sharply for threshold between 10 and 20. For higher threshold values the read cost gradually increases as we operate on less compressible UBs. In order to make the correct decision about what threshold value we should choose we look at the overall workload latency. Figure 19c shows the workload latency for both experiments, normalized by the slowest setup; which is for threshold equal to one. In this case, the expensive read latency dominates. On the other hand, the normalized workload latency is 15% faster for threshold equal to 10 or 20 for both workloads. For bigger thresholds the overall workload latency gradually increases, and eventually reaches 0.92 for merging every 1000 updates. A merging threshold of 10 updates leads experimentally to the fastest workload execution, hence, it is used in rest of our experimentation.

Fence Pointers. Fence pointers help in reading only the useful part of the bitvectors, however, as their granularity becomes finer they have a substantial space overhead. Here we vary the granularity of the fence pointers and we show the achieved read latency and the corresponding size overhead. Figure 20 shows the overall system performance as we vary the granularity of fence pointers, in particular, the number of bits between two consecutive fence pointers. In this experiment we have 100 million rows with 100 distinct values and we show the average latency sustained, with a workload having 10% updates and 90% read queries. The x-axis shows the granularity of the fence pointers, while the last bar shows the performance of UpBit when no fence pointers are employed. The solid line shows the space overhead as a ratio of the size of the overall bitmap index. On the left hand-side we have a fence pointer after every value effectively paying the price of having the data uncompressed, leading to very expensive reads, however, as we make the fence pointers granularity coarser there is a sweet-spot that has minimum latency and small size overhead. This optimization fits the test data set ($n = 100$M, $d = 100$). For each data set a similar optimization experiment is executed.

Parallel Bitvector Reading. A key operation to support efficient updates on bitmap indexes is to be able to efficiently read an arbitrary value of the column from the bitmap index in order to decide which bitvector will need to be updated. Algorithm 3 can be parallelized in a way that different bitvectors are accessed simultaneously. In addition, with fence pointers we avoid reading unnecessary parts of the bitvectors. Figure 21 shows how getting the value at an arbitrary position scales with number of threads.

UpBit Scaling With Hardware Contexts. In addition to parallel bitvector reading, UpBit employs multi-threaded execution of read queries. All approaches decode bitvectors using multiple threads. Figure 22 shows how the average read and update latency of in-place updates, UCB, and UpBit scales with the number of threads for a workload with $n = 1$B values and domain cardinality $d = 100$. The solid lines show the average update time and the dotted lines show the average read time. Both in-place and UpBit read performance scales for up to 8 threads, at which point, the overhead of on-the-fly merging the value bitvector with the update bitvector for UpBit dominates the read latency, while UCB cannot take advantage of multiple threads. The update latency for in-place updates scales for up to 32 threads (which is the number of physical cores of the machine used). Updates with UpBit scale up to 8 threads. The update bitvectors are sparse and have up to 10 bits set to one at any point of time because of the periodic merge with the value bitvectors. Hence, breaking an update bitvector to more than 8 partitions does not help read performance. On the contrary, it requires more bookkeeping and increased overhead of result distributing and collecting, leading to an overall increase in update latency.

Tuning Summary. UpBit has three major parameters that are exposed for tuning: (i) the UB-VB merging threshold, (ii) the fence pointer granularity, and (iii) the level of parallelism used. Statically defined values deliver robust performance for various workloads. UpBit, however, allows fine tuning of these knobs in the case of a workload that may require different tuning.

4.5 The Impact of UpBit Design Elements

In order to more clearly understand the performance benefit of the two main design elements of UpBit, we evaluate UpBit performance when only fence pointers are enabled (UpBit-FP). We measure the impact of fence pointers with different granularity, when there is only one update bitvector, similar to the existence bitvector employed by UCB. When updating this bitvector, instead of decoding the whole bitvector we can use the fence pointers to decode only the literal word containing the bit we are about to update. That way we avoid the initial unnecessary decoding, but we may still have to decode and then encode until the end of the existence bitvector if the update changes the overall size of the encoding.

Figure 23 shows how UpBit-FP performs in an experiment with

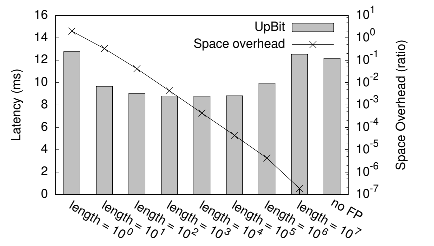


Figure 20: UpBit's optimal behavior needs fence pointers every 10^3-10^5 values having less than 0.5% space overhead.

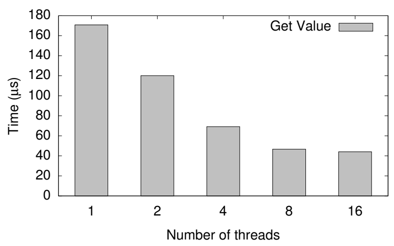


Figure 21: Bitvectors parallel scans scale with number of threads, leading to 3.9× improvement in *get_value*.

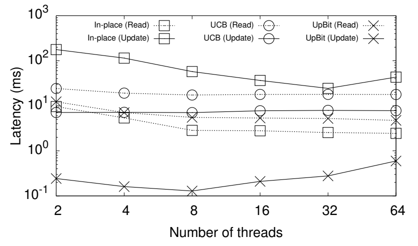


Figure 22: Updates with UpBit are two orders of magnitude faster than other approaches and scale for up to 8 threads.

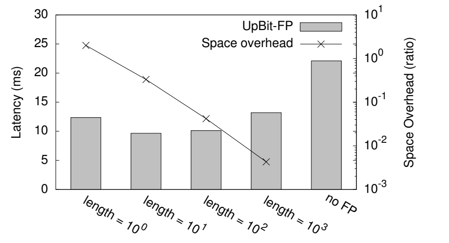


Figure 23: Fence pointers alone offer more than 2× better performance, having less than 10% space overhead.

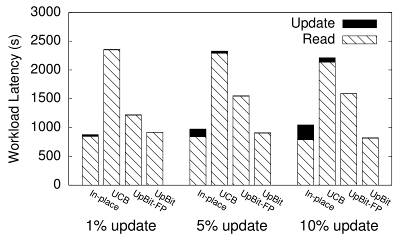


Figure 24: Both fence pointers, and update bitvectors contribute towards the overall performance gains of UpBit.

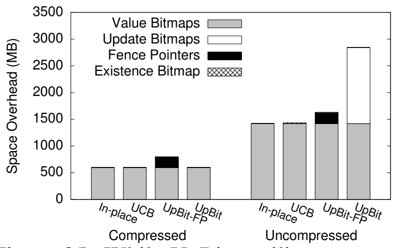


Figure 25: While UpBit auxiliary structures add a lot of raw space, when compressed the space overhead of UpBit is negligible.

data size $n = 100$M rows, cardinality $d = 100$, and a workload with 10% updates and 90% read queries. We observe that fence pointers can indeed help to decrease average latency by 2.29×, requiring, however, significant space overhead of about 15%. Almost the same benefit (2.18×), can be achieved for only 4% space overhead.

Full UpBit Design. We now put the whole design together. We repeat the same experiment as in Figure 12a but now we also include UpBit-FP. Figure 24 shows that the performance gains of UpBit is due to both design decisions of introducing (i) update bitvectors, and (ii) fence pointers. The efficiency of UpBit is the result of the combination of the two design elements. Fence pointers on the existence bitvector alone would be less effective because as we accumulate updates the auxiliary bitvector does not maintain compressibility. On the other hand, a system with update bitvectors like UpBit needs fence pointers to amortize their cost.

4.6 Read-Update-Memory Tradeoff

UpBit is an updatable bitmap index which makes use of additional metadata to support fast updates with low overhead on read queries. Naturally, it is interesting to find out at what cost we achieve this performance tradeoff. Figures 20 and 23 highlight the tradeoff between read performance and memory size. In addition, Figure 25 shows the breakdown of the size of the four update-aware approaches used in our experimentation. Update bitvectors result in doubling the uncompressed size of the bitmap index. However, they can be efficiently compressed because they are sparse. Similarly, the fence pointers add about 0.5% for UpBit, but they result in more space overhead to have the maximum performance gain if they are not used with update bitvectors (UpBit-FP). Finally, the auxiliary bitvector employed in UCB and UpBit-FP approaches has negligible size. UpBit maintains low space overhead by periodically merging update bitvectors with value bitvectors. The high compressibility of the sparse update bitvectors due to efficient encoding, allows for low update and merging overhead.

We further analyze the average latency of UpBit as we vary the granularity of fence pointers separately for read and update queries, for an experiment with $n = 100$M values and domain cardinality $d = 100$. In particular, Figure 26 shows the average read latency, update latency, and memory overhead when we vary the fence pointers granularity between one for every single value, and one every 10^7 values, which is virtually identical with what we observe when we have no fence pointers. Figure 26 shows a triangular tradeoff between read latency, update latency, and memory overhead. For very dense fence pointers both read and update latency are high and, of course, memory overhead is very high, resulting in no benefits. As we make the fence pointers sparser we observe that both read and update latency drop. The minimum read latency is achieved when we store a fence pointer every 10^5 values, and the minimum update latency when we store a fence pointer every 10^3 values. As we further decrease the granularity of fence pointers updates become very expensive as we have to decode larger parts of each bitvector. In summary, we want to minimize all three values: read latency, update latency, and memory overhead. If we have hard requirements (e.g., in read latency or memory size) we will have a specific range of values that we can select. In any case, we can select a "sweet spot" corresponding to the requirements of the application at hand. For example, depending on how many update and read queries comprise the workload, we select the appropriate fence pointer granularity to minimize the overall workload latency.

4.7 UpBit on TPC-H

We now continue to demonstrate that UpBit is effective on full queries with the standard TPC-H benchmark (scale factor 100). We integrate UpBit in an in-house column-store prototype system that supports full select-project-join queries. We experiment using as select operator, either UpBit or an optimized scan. The workload consists of a variation of TPC-H Q6; the selectivity of the `l_quantity` clause is varied from 2% to 100%, and we index on the `l_quantity` column. Q6 has two more select predicates, hence the overall query selectivity is between 0.08% to 3.9%, accordingly (more details about query generation can be found in Appendix B).

Figure 27 shows that UpBit achieves significant benefits compared to the scan-based execution. For query selectivity between 0.08% and 2.4%, (indexed column selectivity $2\% - 60\%$), UpBit is up to 6× faster, while for higher query selectivity ($2.4\% - 3.9\%$, indexed column selectivity $> 60\%$), the scan approach is up to 1.4× faster. Using UpBit for full SQL queries on larger data sets results in similar behavior as observed earlier (e.g., Figure 18).

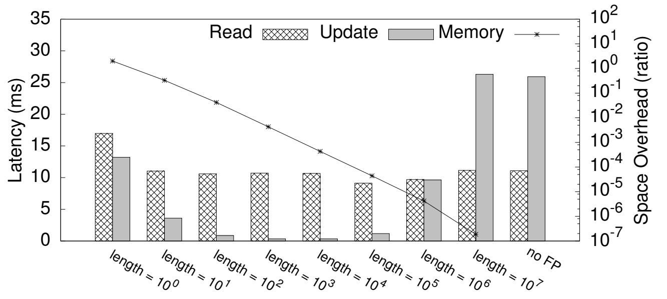


Figure 26: In order to decide fence pointer granularity we analyze the expected workload to get the best combination of performance and memory overhead.

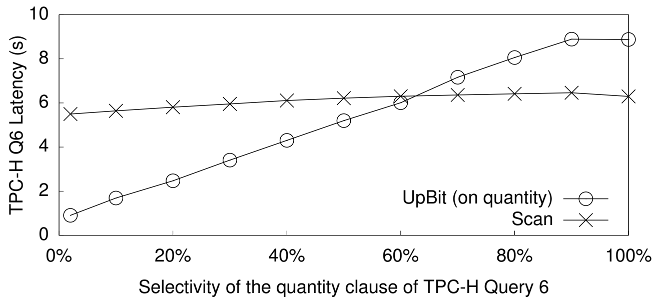


Figure 27: UpBit achieves significant benefits up to 6× compared to the scan-based execution of TPC-H Q6, when varying the selectivity of the `l_quantity` clause.

4.8 Discussion

Designing Bitmap Indexes. The analysis of UpBit leads to three core observations about designing bitmap indexes. First, introducing fence pointers mitigates one of the most important costs of bitmap indexes when updating: decoding and re-encoding bitvectors. In addition, distributing the cost of updating to several highly compressible update bitvectors allows for (i) very efficient updates, (ii) efficient on-the-fly merges during reads, and (iii) query-driven absorption of the updates. Finally, both fence pointers and update bitvectors increase the memory requirement of the index by a small fraction, however, we show that we can keep the memory overhead low and that we can fine-tune it based on the expected workload.

Protecting Updates. While in this paper we focus on serial execution, UpBit is ideally situated to absorb concurrent updates compared to past work. The introduction of one update bitvector per value of the domain means that UpBit can operate in parallel for queries that update or read different values. Locking can happen in a fine grained way at the granularity of each individual value/update bitvector, providing maximum flexibility for concurrency compared to state-of-the-art techniques that essentially have a single point of access for all queries (e.g., either in-place updates or a single existence bitvector).

5. RELATED WORK

Updating Bitmap Indexes. The state of the art in update-friendly bitmap indexes is Update Conscious Bitmaps (UCB) [8]. USB supports updates by using an existence bitvector (EB) [21] which is updated when a row is deleted. Updates in this context are handled by a delete-then-insert operation. In order to efficiently support inserts UCB bitvectors are expanded by a single synthetic 0-fill-pad fill word which can represent very large bitvectors comprised only of zeros. In an event of an insert, or of a delete-then-insert, the new values are stored in the additional words. UCB, however, needs to store the mapping information when physical rows get invalidated. In addition to that, EB becomes less compressible as updates arrive because more bits are set to one, and there are fewer opportunities for space efficient encoding. UCB offers very low update times for a small number of updates, however, both update time and read time increases as updates accumulate.

In this paper we address this limitation by introducing UpBit with two key design elements. First, UpBit employs update bitvectors (one per value of the domain) to distribute the update burden from the single bitvector that UCB employs, the existence bitvector. The difference now is that the auxiliary bitvectors can maintain their compressibility which is the main cost factor when querying using a bitmap index. The way to maintain auxiliary bitvector compressibility is by merging the update bitvectors and the value bitvectors based on a threshold. In particular, when a performance-driven threshold of bits set in an update bitvector is reached, it is marked for merging in the next read query. Then, in the next read query for the given value the update bitvector is merged back to the value bitvector in a query-driven way. This design was not practical with UCB because, contrary to UpBit, a query-driven merge would result into updating all bitvectors for some read queries. Second, UpBit introduces fence pointers in the update bitvectors to allow for efficient reads of arbitrary positions of each bitvector without decoding them entirely. This design reduces unnecessary decodings, and offers both better update time and query time when compared with the state of the art.

HICAMP Bitmap Index. HICAMP bitmap index [29] uses the DAG structure of HICAMP memory prototype [12]. HICAMP (Hierarchical Immutable Content-Addressable Memory Processor) offers architectural support for efficient concurrency safe shared structured data accesses. The HICAMP bitmap index uses HICAMP memory in order to efficiently deduplicate bitvectors. When any part of a bitvector is updated the memory management unit of HICAMP memory delivers efficient deduplication, in order to offer both space and update optimizations. HICAMP bitmap can be updated with $O(log(n))$ cost, and the read cost is $O(m \cdot log(n))$ where m is related to the number of non-zero words of the bitvector.

HICAMP bitmap index, orthogonally to UCB and UpBit, offers efficient concurrency-safe bitmap indexing with aggressive deduplication building directly on top of the HICAMP prototype memory which is not yet available outside hardware simulation. UpBit can take advantage of the deduplication support of HICAMP memory to store for free most of the update bitvectors and potentially absorb more updates before merging update and value bitvectors.

6. CONCLUSIONS

In this paper, we show that state-of-the-art bitmap indexes suffer in terms of update performance, and they do not scale with the number of updates. We introduce UpBit, a new bitmap index which allows for both efficient reads and writes and scales with the number of updates without hurting read performance.

Instead of updating bitvectors directly, when an update query arrives, UpBit buffers the update allowing multiple updates at the same time while supporting fast response times for read queries, having access to the latest updates with negligible overhead. UpBit uses dedicated *update bitvectors* for every value of the domain to buffer inserts, deletes and updates. At the same time it uses *fence pointers* across these bitvectors to allow for fast navigation by minimizing decoding. We demonstrate that UpBit achieves faster workload latency compared with both state-of-the-art read and update-optimized bitmap indexes, combining the best of both worlds.

Acknowledgments. We thank the reviewers and the DASlab group for their valuable feedback. This work is supported by SNSF Grant No. P2ELP2_158936 and by NSF Grant No. IIS-1452595.

7. REFERENCES

[1] G. Antoshenkov. Byte-aligned Bitmap Compression. In *Proceedings of the Conference on Data Compression (DCC)*, pages 476–476, 1995.

[2] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: Efficient Online Updates in Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 865–876, 2011.

[3] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. Online Updates on Data Warehouses via Judicious Use of Solid-State Storage. *ACM Transactions on Database Systems (TODS)*, 40(1), 2015.

[4] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 461–466, 2016.

[5] J. Becla and K.-T. Lim. Report from the first workshop on extremely large databases (XLDB 2007). *Data Science Journal*, 7, feb 2008.

[6] Berkeley. Berkeley Earth Data. *http://berkeleyearth.org/data/*.

[7] M. Cain and K. Milligan. IBM DB2 for i indexing methods and strategies. *IBM White Paper*, 2011.

[8] G. Canahuate, M. Gibas, and H. Ferhatosmanoglu. Update Conscious Bitmap Indices. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 15–25, 2007.

[9] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *ACM SIGMOD Record*, 27(2):355–366, 1998.

[10] C.-Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. *ACM SIGMOD Record*, 28(2):215–226, 1999.

[11] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[12] D. R. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi. HICAMP: Architectural Support for Efficient Concurrency-safe Shared Structured Data Access. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 287–300, 2012.

[13] A. Colantonio and R. Di Pietro. Concise: Compressed 'N' Composable Integer Set. *Information Processing Letters*, 110(16):644–650, 2010.

[14] F. Deliège and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 228–239, 2010.

[15] F. Fusco, M. Vlachos, X. Dimitropoulos, and L. Deri. Indexing million of packets per second using GPUs. In *Proceedings of the Conference on Internet Measurement Conference (IMC)*, pages 327–332, 2013.

[16] F. Fusco, M. Vlachos, and M. P. Stoecklin. Real-time creation of bitmap indexes on streaming network data. *The VLDB Journal*, 21(3):287–307, 2011.

[17] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, Near Real-time, Scalable Data Warehousing. *Proc. VLDB Endow.*, 7(12):1259–1270, 2014.

[18] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A Tunable Compression Framework for Bitmap Indices. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 484–495, 2014.

[19] R. MacNicol and B. French. Sybase IQ Multiplex - Designed For Analytics. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1227–1230, 2004.

[20] P. E. O'Neil. Model 204 Architecture and Performance. In *Proceedings of the International Workshop on High Performance Transaction Systems (HPTS)*, pages 40–59, 1987.

[21] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. *ACM SIGMOD Record*, 26(2):38–49, 1997.

[22] Oracle. Oracle Database 12c for Data Warehousing and Big Data. *Oracle White Paper*, 2013.

[23] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core CPUs. *Proceedings of the VLDB Endowment*, 1(1):610–621, 2008.

[24] P. Russom. High-Performance Data Warehousing. *TDWI Best Practices Report*, 2012.

[25] V. Sharma. Bitmap Index vs. B-tree Index: Which and When? *Oracle White Paper*, 2005.

[26] K. Stockinger. Bitmap Indices for Speeding Up High-Dimensional Data Analysis. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, pages 881–890, 2002.

[27] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 553–564, 2005.

[28] TPC. Specification of TPC-H benchmark. *http://www.tpc.org/tpch/*.

[29] B. Wang, H. Litz, and D. R. Cheriton. HICAMP Bitmap: Space-Efficient Updatable Bitmap Index for In-Memory Databases. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 1–7, 2014.

[30] C. White. Intelligent business strategies: Real-time data warehousing heats up. *DM Review*, 2002.

[31] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[32] H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit Transposed Files. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 448–457, 1985.

[33] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Lauret, J. Meredith, P. Messmer, E. J. Otoo, V. Perevoztchikov, A. Poskanzer, O. Rübel, A. Shoshani, A. Sim, K. Stockinger, G. Weber, and W.-M. Zhang. FastBit: interactively searching massive data. *Journal of Physics: Conference Series*, 180(1):012053, 2009.

[34] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing Bitmap Indices with Efficient Compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38, 2006.

[35] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on Design and Implementation of Compressed Bit Vectors. Technical report, Lawrence Berkeley National Laboratory, 2001.

[36] M.-C. Wu and A. P. Buchmann. Encoded Bitmap Indexing for Data Warehouses. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 220–230, 1998.

[37] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 145–156, 2002.

[38] M. Zukowski, P. A. Boncz, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, 2005.

APPENDIX

A. IMPLEMENTATION

UpBit is implemented as a standalone prototype access method in C++. A key component for efficient bitmap indexing is the design of bitvectors. For this, we built on top of the state-of-the-art bitmap representation when it comes to read performance using the bitvectors implementation of FastBit [33]. We have modified FastBit to support existence bitvectors (to implement UCB), update bitvectors, fence pointers, and multi-threading.

A key challenge when implementing UpBit is how to find the old value of a given row (`get_value`) in order to set the corresponding bit in the update bitvector. We parallelize this process using C++11 threads. We take advantage of the available hardware parallelism (number of cores) in order to concurrently search all value bitvectors for the given row. Fence pointers are particularly helpful because we do not need to decode the bitvectors in their entirety.

A.1 Integration

UpBit can be easily integrated to a full database system. UpBit exposes a traditional secondary index interface, allowing for initial build, point query, range query, delete, update, and insert operations. The space management is currently independent from a database system since it is more efficient to store the encoded bitvectors contiguously. In our prototype column-store implementation integrating UpBit has minima overhead, since it can directly use a column as input and can also separately manage the storage used for storing the index durably. Finally, we augment the implementation of UpBit with a method to convert the final bitvector after querying a column using the index to a list of rowIDs.

B. TPC-H QUERY 6

Here we discuss in detail how we generated the TPC-H queries used in the experiment presented in Section 4.7. In this experiment we use a modified TPC-H Query 6, where we vary the selectivity. The SQL code of Q6 is shown in Listing 1. The value of the parameter `DATE` is the first of January of a randomly selected year within $[1993, 1997]$. The dates of the TPCH data span the year 1992 to 1998 for a total of 7 years. The parameter `DISCOUNT` is randomly selected within $[0.02, 0.09]$, where the possible values are distributed in the range $[0, 0.1]$, in increments of 0.01, having, as a result, 11 evenly distributed distinct values. Finally, the value of the parameter `QUANTITY` is either 24 or 25, while all possible values of the domain are the integers in the range $[1, 50]$, distributed uniformly. As a result, an average Query 6 selects the rows corresponding to 1 of the 7 years, 3 of 11 possible values of discount, and 24 or 25 of 50 possible values of quantity, leading to selectivity $\frac{1}{7} \cdot \frac{3}{11} \cdot \frac{24.5}{50} \approx 1.91\%$. In our experimentation (Figure 27) we index the `l_quantity` column and we vary the selectivity of on this column from 2% (`l_quantity < 1`) to 100% (`l_quantity < 50`). As a result, the overall Q6 selectivity varies between $\frac{1}{7} \cdot \frac{3}{11} \cdot \frac{1}{50} \approx 0.08\%$ and $\frac{1}{7} \cdot \frac{3}{11} \cdot \frac{50}{50} \approx 3.9\%$.

Listing 1: TPCH Q6

```
SELECT
    sum(l_extendedprice * l_discount)
        as revenue
FROM
    lineitem
WHERE
    l_shipdate >= date '[DATE]'
    AND l_shipdate <
        date '[DATE]' + interval '1' year
    AND l_discount between
        [DISCOUNT] - 0.01 AND [DISCOUNT] + 0.01
    AND l_quantity < [QUANTITY];
```