Optimizing Bw-tree Indexing Performance

James Hunter, Justin Levandoski, David Lomet, Sudipta Sengupta
Microsoft Research
Redmond, WA, USA
{jahunter, justin.levandoski, lomet, sudipta}@microsoft.com

Jianguo Wang
University of California
San Diego, CA, USA
csjgwang@cs.ucsd.edu

***Abstract*—The Bw-tree is a latch-free, B-tree style index that is part of Deuteronomy key-value stores, which have been deployed both in the cloud and in "boxed" products. It can be layered on a log structured storage manager or used as a main memory index, and it can support transactions when embedded in a system with a Deuteronomy transactional component. Its high performance is a result of its latch-free, log structured properties, coupled with an intelligent but straightforward implementation of index search. In this paper, we present our optimized Bw-tree index that is substantially faster than our initial implementation. We apply an integrated set optimizations, new ones and refinements of prior techniques, to produce Bw-tree indexing with up to 40 percent better performance for single key lookups, while also improving range search performance. And our original implementation performs comparably to the best main memory indexes. Uniquely, however, it is designed to be used with data residing on secondary storage, and hence includes the overheads required to keep data paginated.**

I. Introduction

A. Competitive Environment

There has been continuing interest in key-value stores, both transactional and non-transactional [7]. This is part of a longer trend to take pieces of database technology "out of the box", i.e., out of the surrounding database system, to enable applications to use a lighter weight, and perhaps higher performing data management system [28].

Nowhere is this interest more intense than in the cloud services space, where, for example, Microsoft's DocumentDB exploits the Bw-tree key-value store as its indexing engine for documents [27]. And key-value stores are an increasingly important component in application development in the Open Source cloud world, where, e.g., RocksDB [26] has been increasingly used for cloud services. Performance can be turned into market advantage for the cloud provider, providing faster service at lower cost.

In this paper we show how to substantially improve the performance of the Deuteronomy Bw-tree [13]. The Bw-tree performance already is competitive with the very best main memory indexes with memory resident data. However, we anticipated that improved performance was possible based on earlier work on cache sensitive data structures and improved intra-node search, and this paper confirms this. Nonetheless, new techniques were required to boost performance substantially. This paper describes what we did and the results we achieved.

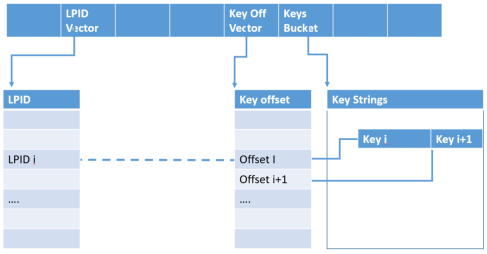


Fig. 1: The Bw-tree's original page layout.

B. Prior Bw-tree Implementation

The Bw-tree, as originally implemented, has a straightforward but nonetheless high speed page search within the "consolidated" part of the Bw-tree node. It uses a good binary search over a vector of offsets to variable length byte string keys. Alternatively, it can use a comparison based on a user comparison function. But the focus here is on byte string keys. In a Bw-tree consolidated page, byte string keys are stored densely in a packed string "bucket" indexed by fixed length offsets from the bucket start. Figure 1 illustrates the page format. Range search uses a cursor that remembers the last key of the already returned range. Records are returned in batches of about a page in size, producing good performance, especially for long ranges. The resulting system provides excellent performance and scales well on multi-core hardware via cache locality and latch-free execution.

C. Optimization Intentions

The unique advantages of the Deuteronomy architecture [17], i.e. transactional functionality, in a transactional component or TC, clearly separated from data access functionality, in a data component or DC, permitted us to optimize page search without concern about transactional implications. This enabled us to focus on search speed inside of a consolidated Bw-tree node. This search is, in fact, similar to that required in a traditional B-tree, but without transactional distractions.

Variable length byte string based keys are very general, and assume even greater generality when the key byte strings are the result of normalization, i.e., the order preserving conversion to byte strings of keys of other types. The big advantage of such normalization is avoiding a function call

per comparison that is sometimes required when supporting a variety of data types, as byte string comparisons can be done inline.

Normalized keys can also be adapted to deal with compound keys, where no special measures are needed when doing an index search using the normalized (compound) byte string [2]. Using normalized keys with prefix key compression (see below) enables a single Bw-tree instance to be used for indexing, for example, separate tables of a SQL database. A table id can be prepended to the keys of each table, such that the table id separates the data of one table from another (or one database from another, should user ids also be prepended). This prepending clusters data by table, with the root (and perhaps a few additional Bw-tree nodes) acting as a catalog. Further, because of prefix compression [4] , the table ids are largely factored out of the lower parts of the tree, meaning that, further down the tree, the augmented keys do not consume additional space, nor do they increase the cost of comparisons.

D. Our Approach and Contributions

With this setting in mind, we focused on improving both search speed and memory footprint. Both are important. Search speed is obvious. Memory footprint becomes critical in the cloud multi-tenancy setting where smaller memory footprint leads to better sustained performance as memory resources are varied up and down based on a tenant's use pattern. Having a small memory footprint for internal nodes (above the data nodes at the leaf) is particularly important as retaining the index in main memory means no record access need bring more than a single Bw-tree node into the cache.

Our approach exploits and extends prefix B-tree [4] techniques in which index terms are shortened and common prefixes factored out of pages. Putting prefix techniques into practice revealed interesting opportunities for additional optimizations. These are described in section II.

Optimizing search performance requires very careful attention to in-page data format, both to produce a very fast page search and to minimize processor cache misses. For this, we exploit some of the techniques from earlier work [18], but extend and modify them to produce even better results. We found that page organization is very sensitive to even small changes. Page layout and factors influencing it are presented in section III.

Using the best possible search algorithm is essential for peak performance. Such an algorithm has to exhibit cache locality, minimize the number of probes, and have very low cost per probe. Interestingly, while we tried many different approaches, our original binary search turned out to provide the best performance. We modified this algorithm to deal with varible length keys. We describe this not widely known algorithm, our modifications to it, and why it has excellent performance in section IV.

Range performance is improved by superior page search performance, but even more so by avoiding the need to re-search a page. We had used a purely logical "cursor" in [15]. And indeed, our cursor continues to be "logical" in that it is effective regardless of changes to the underlying tree structure. However, by using a "cookie" like approach, we can remember where we are in a physical sense and only need to validate that our search point is physically unchanged in order to avoid searching index pages. This is the subject of section V, where we extend the notion of cursor to support additional use cases that we expect to be important.

The combination of the techniques in sections III through V greatly improved upon the prior Bw-tree performance. We did extensive experiments comparing the techniques we introduced in these sections. These experiments guided our research and were decisive in our picking the most effective collection of techniques. We describe the experiments and the resulting high performance in section VI.

Search, including B-tree page search, is a very extensively studied area. We carefully selected what we consider the most relevant of this extensive prior art and describe how it influenced our work in section VII. We end the paper with a discussion section that includes interesting extensions and conclusions in section VIII.

II. Key Compression

A. Prefix B-trees

Bayer and Unterrauer in 1978 introduced the prefix B-tree [4]. This B-tree variant systematically factors out the common prefix of keys on a page (actually, the common prefix of the full range of possible keys on the page). This reduces the memory needed for the keys, hence permitting more keys and associated data per page. Further, by factoring out the common prefix, the key directed search can be done at higher performance as each comparison works with a smaller number of bytes. This is particularly important for index pages, where the data payload is small (a logical page identifier or LPID), and hence the number of entries per page is large, and search speed is essential.

In addition to factoring out common page prefixes, the prefix B-tree includes what it refers to as suffix compression. When splitting a data page, one does not need to post a key present on the page to the parent index page. Instead, one can post a "separator" S, a value (from the space of keys) that simply separates the keys of the two resulting pages, such all keys on the low page are less than S, and all keys on the high page are greater than or equal to S. Now, because any separator that does this will work, one can choose the shortest separator, hence "compressing" the size of the value that would otherwise have been a full key. We choose a shortest "separator" for index pages as well, however, in this case, the separator must be an index entry on the page, not a value that separates two entries.

We exploit both of these prefix B-tree techniques. One focus is on maximizing a common prefix when the key space is only sparsely populated, which is a common case. A second focus is to choose a split point that maximizes both prefix compression and shortness of the index entry posted when data (leaf) pages split.

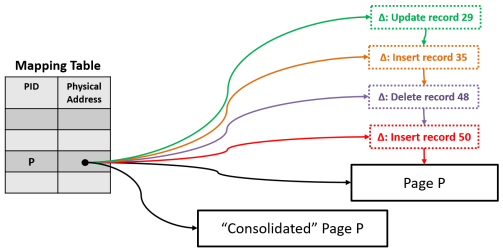


Fig. 2: The Bw-tree page organization showing delta updating.

B. The Bw-tree and Sparse Keys

The Bw-tree is not your usual B-tree. It is designed to be latch-free and log-structured. The result is that the Bw-tree update process differs in important ways from a B-tree. We designed the Bw-tree for great generality. The idea is to be able to deal well with arbitrary keys distributed in arbitrary ways, and these factors led us in a direction that differs from the specifics of prefix B-trees.

Replacement Consolidation: When a Bw-tree page is consolidated, entries on it are organized to optimize search speed. Delta updating, where record updates are prepended to the page, allow records to be changed without the need to reorganize the page. See Figure 2. This organization permits us to inexpensively change state without reorganizing (consolidating) the page. Eventually the page needs to be re-organized to improve search speed. At this point, the existing page is replaced by a new optimized page that includes the effects of the absorbed delta updates, which are applied in a batch.

This updating paradigm means that consolidation does *not* involve updating in-place. Consequently, the cost of consolidation is close to being the same regardless of how data is organized. And this means that we can accept an approach that would be expensive to use with incremental updating, but that is simple and effective with our page replacement form of updating. It is the delta updating that reduces the frequency of consolidations and makes the overall approach cost effective. This is the only place where whether the index is a Bw-tree or a vanilla B-tree really matters.

Sparse Keys: B-trees divide the key space into page-size units of disjoint regions providing complete coverage of the entire space. When keys are sparsely distributed, empty sections of key space will of necessity be attached to regions in which keys are present. These empty regions may be skewed or somewhat evenly interspersed with populated regions. For example, keys consisting of alphanumeric data (which are fairly common) have a number of contiguous sections but are spread over a much larger set of all possible bytes.

The consequence of sparse keys is that the prefix found for prefix B-trees, which is based on finding a prefix for the key space of the page will frequently be shorter than a prefix based solely on the keys present in the page. Thus, instead of a key prefix for the entire key space of a page, we strive to find the longest prefix among keys actually present on a page. Given our replacement consolidation approach, dealing with potential changes in the length of a page's prefix when an update changes the prefix factored out of the page's entries does not degrade our performance. Initially such an update is stored as a delta record. Subsequently, the page is consolidated, and at this time, it is completely replaced with a page based on the new prefix of the actually present keys.

C. Optimizing Key Prefixes and Split Points

As indicated above, the prefix B-tree both used shortest separators and did prefix key compression. We wanted both as well, but how to combine the two was not immediately clear. If we choose the shortest separator, we might not get the longest prefix, and similarly in the other direction. So the problem became: what do we do to simultaneously produce short separators and long prefixes, without degrading B-tree effectiveness via nodes splits that are too skewed for the index to work effectively? We describe our approach below.

1) Short Index Terms: When splitting a data page, we want to find a short separator. When splitting index pages, we need to find an existing index term. We then post the value we have found to the parent page as the new index term for one of the resulting pages. In [4], it is suggested that one needn't simply accept the index term that results from splitting at the midpoint of the page. Rather, a minor deviation from the midpoint may produce a shorter index term while having minimal impact on the resulting B-tree performance. The analysis in [16] confirms that modest split point deviations have modest impact on storage utilization, which we use as a proxy for indexing effectiveness. Indeed, even a page split at the .33 or .67 place (a 2:1 split ratio) only drops the storage utilization by 10%. And for our log structured Bw-tree, this does not result in wasted space as pages are packed into variable length contiguous storage, not fixed size blocks.

2) Long Prefixes and Space Minimization: As with the length of the index term, the length of common prefixes for the pages resulting from a page split will vary based on where we choose to split a page. Our procedure is as follows. We start at the .33 entry for the page, and search until we reach the .67 entry on the page. We designate the pages resulting from a split as L and R, with number of entries $No(L)$ and $No(R)$, and prefix lengths $Len(L)$ and $Len(R)$. To determine the "goodness" of the split point solely for determining common prefix, we compute the space saved in the two resulting pages versus the space consumed in the existing pages. We have then

$Saved(L|R) = No(L) * len(L)) + No(R) * len(R)$ where N is the number of entries on the page. Saving space is important for cache performance and memory footprint. It is also important for search performance. $Saved$ is the total space saving due to prefix compression, not the increase in saving as a result of the page split. There is no guarantee that a page split will result in any additional space saving, but it is a common result.

3) Optimization Function: We need to combine the impact on the pages resulting from a split with the impact of search

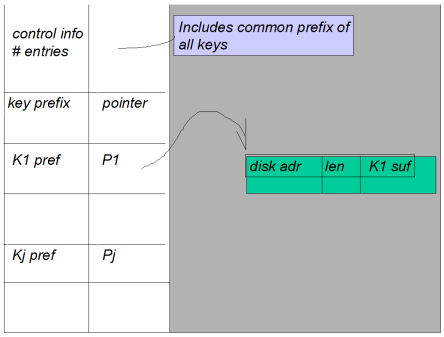


Fig. 3: The B-tree page layout from [18].

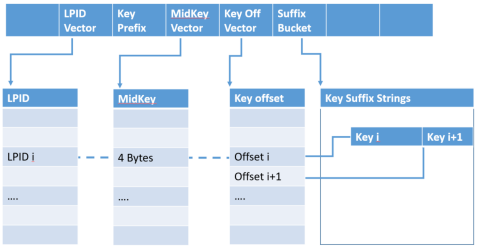


Fig. 4: The Bw-tree's revised page layout.

performance on the parent of the splitting page. We speculated that index term size ($TSize$) should be weighted more than simply by its storage because this index term will be involved in all searches that lead to the split pages, while the keys on a page will only be relevant to search on the specific page. So we tried the heuristic
$max(Score(L|R) : Saved(L|R) - N * TSize(L|R))$
and we tested values of N between 2 and 8, finding that a value of 4 produced the best performance in our experiments. If only SavedSpace is used, then there is a risk of a forcing the page split at a point with a long index term. Thus, we searched between the .33 entry and the .67 entry and selected the split point $L|R$ which had the highest Score.

III. Page Organization

In the "modern era" of multi-core and memory hierarchies, the details of page organization play an important role in how well B-tree style indexes perform. This is a much studied area [24], [25], [6], [18]. We describe in this section how we produce a storage efficient and effective organization for high performance. Our starting point is the page organization suggested in [18], shown in Figure 3.

Our new large scale organization for Bw-tree pages is shown in Figure 4. This organization has a resemblance in some respects to the organization in Figure 3. However there are some new important elements that we have changed. We discuss these elements below.

Header structure layout (104 bytes):
General base page information: 48 bytes
Pointer to key prefix bytes: 8 bytes
Pointer to fixed keys: 8 bytes
Pointer to key offsets: 8 bytes
Pointer to key suffix bytes: 8 bytes
Pointer to page values (LPIDs of child nodes): 8 bytes
Pointer to low key bytes: 8 bytes
Size of high key: 2 bytes
Size of low key: 2 bytes
Size of boundary key prefix: 2 bytes
Size of page key prefix: 2 bytes

Fig. 5: The Bw-tree's revised page header.

A. Page Header

One would not necessarily think that how one organizes header information on a page would be a performance factor. But it turned out that it was. It is important to keep the information that you always need to access clustered together to improve processor cache performance. Our system experienced a 5% performance gain when we changed the clustering in the header information. A schematic of the header is shown in Figure 5. The fields in the figure will appear in the discussion below. Accessing the page header is a cost that, given our high performance search method, is not insignificant, so we need to exercise care at this stage as well.

1) Page Verification: When entering a page, we need to verify that we have reached the correct page because many things can change during a tree traversal when latches are not being held. Simple CRUD operations are done under epoch control, i.e., the operation starts by entering an epoch. This keeps it from seeing reused memory or LPIDs. However, it does not prevent a page from splitting. So we need to check the page high boundary to ensure that the page contains the key(s) requested. If it does not, which is rare, we may need to follow a side pointer to a new page. Including the high boundary in the header instead of at the high end of the page boosted performance.

For index pages, the high key of the page serves as the high boundary, and is itself an index entry (the highest one) on the page, as well as being the parent's index term for locating the page. Keeping this high key in the page header means that the high key need not enter into the determination of the key prefix for the page. This can be important in the case of sparse keys, where index terms must partition the space, but entries only partially fill the space. That is, it may enable us to factor out a longer prefix than would otherwise be possible.

2) Page Key Prefix: We have factored out more than just the prefix for the search space of the page from keys stored on the page, we have factored out the shared prefix of the actual entries. We must check whether the search key shares this

prefix or not. If it does share this longer prefix, we proceed to search the middle key vector (see the next subsection).

If the search key does not match the page's key prefix, then how to proceed depends upon whether we are at a leaf data node or an internal index node.

leaf node: A record for the key does not exist on the leaf page and so the search reports failure as the record is not present in the tree.

index node: If the search key prefix is smaller than the page prefix (which is a prefix for entries on the page, not for the key space), the search proceeds from the lowest index term. If the search key prefix is larger than the extended prefix, the search proceeds from the last pointer on the page, which is the pointer associated with the high key present in the page header.

B. Mid-Key Vector

Our binary search is done over a vector of fixed size entries. Having fixed size entries is essential for fast binary search where one can compute the location of the next search probe. Each entry of the vector consists of the next four bytes of the key after the page prefix has been removed. Removing the prefix thus makes it more likely that the search can be resolved within the mid-key vector because comparisons are being done on deeper parts of search key and page keys. Entries at which a comparison in the mid-key vector yields equality require that the tail of the key be searched, which adds to the cost of the search probe.

For a successful search, we need to be able to handle both short keys, which do not fully consume their mid-key vector entry, and long keys, which need to have their suffix stored and accessible for comparison. We describe how we treat long keys and their suffixes in the next subsection. Here we describe how we represent short keys.

We need to make sense of comparisons when keys are of different lengths. This is not an issue of choosing a padding character as defined, e.g. by the SQL standard. That can be handled via a form of key compression [2]. Rather, after all such massaging, our resultant index terms or records may still have variable length entries in the mid-key vector. When there is equality on an existing comparable key part, we want the shorter key to compare low. So the problem is to represent a variable length mid-key entry such that the shorter entry or key compares low when using a four byte comparison instruction.

For the first three bytes of the mid-key vector, we do the following. (1) If the mid-key vector entry has a corresponding byte, that byte is stored in the appropriate mid-key vector byte of the entry. If the mid-key entry is short, we insert "00000000"b (binary zero) for any unused (empty) byte. It is the task of the last byte in the mid-key entry to make the comparison outcomes correct by distinguishing between bytes in the key that are zero and bytes that are empty because the key is short.

We then need two representations:

- We use the low order two bits of the last byte (of the four byte vector entry) for a field that tells us the length of the entry when the entry is less than four bytes. The rest of this last byte is set to "000000"b. The entry can be anywhere then between zero ("00"b) and three ("11"b).
- For keys with mid-key entries that are four bytes (or longer), we set the fourth byte in the mid-key entry as follows:
  - If its fourth byte is greater than three ("00000011"), we use the mid-key's fourth byte as the fourth byte of its mid-key entry value. In this case, our representation uses no extra storage for representing the key.
  - If its fourth byte is not greater than three, then we set the last byte to "00000011" and store this fourth byte as part of the key suffix.
- To make our comparisons correct, we encode the search key's corresponding mid-key in the same way as we encode it in the mid-key vector, extending it to four bytes as we do the entries in the mid-key vector.

This technique, on rare occasions, increases the key suffix by one byte, but overall, the impact is to add one eighth of a bit to the length of a key's representation as there are only four of 256 values for the fourth byte that require an extra byte.

C. Other Page Elements

1) Tail Suffix: We use two data structures to deal with the remainders of keys after we have removed prefix and mid-key. One part is a suffix location vector SL, addressed in the same way and corresponding to the element in the mid-key vector, that contains the offset to the suffix for the mid-key element. The SL offset points to a contiguous set of bytes that hold the actual suffixes, in the same order as the keys of the page in the suffix bin SB. Thus, $SL(i+1) - SL(i)$ is the length of the ith suffix stored in SB. This representation allows us to avoid explicitly representing both location and length of the suffix. This is how we deal with the full key in our original representation 1.

2) Columnar Data Structures: Accessing only the fields you know you will need is an important way to optimize processor cache utility. In our page search, we always access parts of the page header, including the key prefix. We then search the midkey vector. Sometimes we need to access the key suffix. Once we have found the desired entry, we need to access the entry's payload (LPID or data). Given this access pattern we provide a separate vector for each of midkey vector, key suffix vector, and data. These are shown in Figure 4.

IV. Binary Search

A. Big vs Little Endian

Using a vector of fixed size fields for our binary reduces the cost of a probe to a relatively simple computation (see the next subsection). However, we want also to optimize the cost of the comparison as well. Our target key type is a string of bytes, so it would be tempting to use a byte string compare. However, even the best string comparisons are iterative byte by byte compares.

In [18], we described how to avoid this. We convert the "little endian" sequence of key bytes (1,0,3,2) into the "big endian" 32-bit integer with byte order (0,1,2,3), and then perform comparisons on the latter. The advantage of swapping the key bytes and storing them in a single 32-bit integer is that the result of comparing the latter is given by a single word comparison instruction, rather than by four byte string comparisons. We perform this conversion both for search keys and for mid-key vector entries. This conversion is done once per page search for the search key, and once per page reorganization for the mid-key vector entries.

B. Search and Equality

Our binary search is, of necessity, different from the search of a vector with fixed size entries. We cannot, on some probes, resolve a comparison based only on comparing search key middle with mid-key entry. Rather, to fully determine the outcome requires then that we compare key suffixes. And suffix comparisons are quite expensive, as we must both access them, incurring two probable cache fault, one at the the suffix offest vector, and one at the suffix itself. We must then perform a variable length byte string comparison.

The frequent advice for binary search is to ignore the equality case and fully search the implicit binary tree. The idea is that separating out the equality case at each probe is more costly than dealing with it once at the end of the search. Early search termination, possible by handling the equality case at each probe, does not pay off vs delaying it until the end. In addition, when searching index nodes, it is unlikely that search will terminate early as we use separators as entries, and they are exceedingly unlikely to themselves be keys subject to search.

We tried avoiding separating the equality case when doing comparisons at the mid-key vector. This would avoid the expensive check of the suffix until the binary tree traversal completed, at which point we might have to look at a number of adjacent entries before determining the outcome. Unfortunately, this did not produce good results. The problem is that, in fact, we can have several entries that are equal in the mid-key vector, and they required searching using a linear search at the end of the binary search. So we reverted to doing a full key compare (mid-key vector entry plus tail) when the mid-key comparison produced an equality result. The resulting binary search, including dealing with equality at each probe, produced fairly consistently better performance than the alternative of waiting until the end to deal with equality.

C. Binary Search

1) Borrowed Binary Search: In [18], Shar's method [10] was suggested as an efficient technique. Its advantage is that it separates the vector into two parts, each an integral power of two in size. This means that a simple shift that halves the step size works well, and it is even possible to unroll the search loop for further gain.

We did not use Shar's method. We tried it and it consistently lost to another binary search that we found in [21] (a correction

```
while (auto_t half=n/2) {
   auto middle=lower+half;
   lower=((*middle)<=needle)?middle:lower;
   n-=half;
}
return ((*lower)==needle)?lower:notFound;
```

Fig. 6: Binary Search Routine from [21]

```
while(auto half = n/2) {
  auto middle = lower + half;
  page_key_middle = key_layout.GetKeyMiddle(middle);
  if(key.middle() != page_key_middle) {
    // Fast path.
    lower = (key.middle() < page_key_middle) ? lower : middle;
  } else {
    // Slow path.
    // Need to compare suffixes.
    Slice page_key_suffix = key_layout.GetKeySuffix(middle);
    int result = key.suffix().compare(page_key_suffix);
    if(result != 0) {
      lower = (result < 0) ? lower : middle;
    } else {
      found_equality = true;
      return (exclude_equality) ? middle + 1 : middle;
    }
  }
  n -= half;
}
```

Fig. 7: Variable Length String Binary Search Routine

of a search technique from Khuong [11]). The basic form of this search is illustrated in Figure 6. Like binary search in general, how to compute the position for the next compare is the critical step. Classically, this involves a $ceil$ function that rounds up non-integer division results to the next higher integer. The "slick" part of the binary search of Figure 6 is that it does this very efficiently via discarding the remainder and getting the $ceil$ by subtracting the result from the prior step size($n- = half$). Another thing happens below the level of the C-language description. With the C code as written here, a conditional move replaces the more usual conditional branch. And for searches of the size that we are performing within a node, the conditional move had better performance.

2) Our Variable Length Binary Search: Our binary search needed to deal with variable length keys. We adapted the binary search in Figure 6 to this task. The result is given in Figure 7. The code lines in red are either the same as or perform the same function as the code in 6. The "slow path" is our handling of the equality outcome, where we compare suffixes. Our code is written in C++, which is the reason for some of the syntactic differences. Further, our full code epilogue is not shown.

V. Ranges and Cursors

A. Initial Range Support

In the previous sections, we have focused on optimizing the performance of the search to data from the top of the index tree. This is a critical measure for update performance and especially for the retrieval of a specific key. However,

what makes B-tree style indexes so compelling within database systems and key value stores is that they also support key ordered range scans.

We previously [15] achieved very good performance for range scans by returning batches of records instead of returning a single record at a time. The batches were approximately page size batches, meaning that we only visited a page once before moving on in the search. We tracked progress in the range search by remembering the start key for the remainder of the search. In our tree, this can be the high boundary key for a page we have completed searching. This boundary key will direct us to the next page when the search resumes. The high boundary key was recorded in a cursor data structure returned to the user along with the records encountered in the page just visited.

B. Physiological Cursors

Thus we maintained what might be regarded as a purely logical cursor via which the user then can request additional records in sequence from higher parts of the range. With this purely logical cursor, we would again search the tree from root to leaf to find the page containing the keys begining at the cursor provided key. However, storing physical information in the cursor, i.e., the page associated with the cursor key, permits us to avoid the full tree search, and go directly to the page where the keys we want are stored. This is, of course, another way of optimizing search, i.e., by obviating the need for it, at least on some of the index pages.

This form of cursor would then contain information that would not be understandable nor useful outside of the context of the specific index tree and the specific range search. Storing physical information in a cursor is similar to a web search returning a cookie. Both techniques retain state so that subsequent interactions are either possible or made more efficient. Thus, a range cursor contains both the lower bound key for the remaining range (the logical part) and the next page LPID where the search is to continue. We already store the LPID of the next page in a Bw-tree page because the Bw-tree is a form of B-link tree, complete with side pointers.

C. Ensuring Cursor Safety

Each implementation of an index tree has its own, perhaps unique, characteristics, but common to all is that the index cannot simply accept the physical information that comes from a user. It might be either corrupted or simply out-of-date. So the system must be able to validate that the physical information agrees with the logical information in the cursor, and hence can be used to short-circuit the index search and proceed directly to the data page containing the desired range records. We need to do such validation.

To verify that the remembered LPID is still the identifier for the page where the range search will continue, we consult the mapping table (which maps LPIDs to page addresses) and use the address in the mapping table to access the page. We then check control information in the page to determine that the page contains the keys we need, i.e. that its low key bound is equal to the cursor key. If validation fails, we re-traverse the index from the root of the tree.

Note that our cursor is not fully physical in that we do not store the memory address of the page in the cursor. Thus, our cursor remains valid even if the LPID now points to a page that has had subsequent delta updates or been consolidated to absorb deltas. Since page splits move the high keys to a new page, our cursors also remain valid should the page split. (Of course, when we have descending order cursors, that would not be the case.) Only more extreme and less frequent restructurings would invalidate the cursor, e.g. deletion of the page. For page deletes, our cache manager guarantees that a deleted page will be testably deleted. If it is used for another purpose, our search validation will detect that. If validation fails, we then re-traverse the Bw-tree, starting at the root.

D. Additional Cursor Forms

Physiological cursors can play other roles as well, where the idea is to remember a location in the index so that subsequent searches can bypass parts of an index. One form of index that can be supported in this way is what we call a "prefix cursor". The idea of a prefix cursor is to locate all the records with keys that are all prefixed by the string provided for the prefix cursor. The initial "seek" to position the cursor would then remember the LPID of the page that is the root of the smallest subtree that includes all keys that share the prefix. How small the subtree determines how many nodes on the path to these keys can be bypassed. So such a cursor would contain the key prefix and the LPID of this subtree root.

A prefix cursor is particularly useful when dealing with compound keys, such as keys that have prefixes that include user ids and/or table ids. Such a cursor permits one to remember where to start a search for, say, a particular user's data. This capability of letting a user remember, via a cursor, the location of his data, is similar in functionality to the RocksDB [26] variation that supports hashed access to essentially a forest of trees, one for each user. But here the implementation is even simpler, the access more direct, and the capability of accessing ranges across users, tables, etc. is preserved.

VI. Experimental Results

This section provides experimental evaluation of the optimizations presented in this paper. Our experiments use a mix of real-world production workloads along with synthetic workloads. This paper focuses on in-memory indexing of the Bw-tree. Thus our experiments are all done in this in-memory setting.

There are two aspects that our experiments highlight, which we characterize as memory and performance.

Memory: By memory, we mean two things, (1) the effectiveness of our key compression technology in removing prefixes from the keys, and (2) the overall size of the interior indexing nodes of the Bw-tree, which depends mostly on our choice of short separators for index terms. It is our ability to achieve short separators while being

Processor	Intel® Xeon® E5-4650L 32 cores, 64 hyperthreaded
Memory	192 GB DDR3-1600 SDRAM
OS	Windows® Server 2012 R2

TABLE I: Configuration of the machine used in experiments.

able to factor out long prefixes that is responsible for our good performance results.

Performance: We ran three forms of tests, (1) read only that measure the pure search speed of our optimized indexes, (2) read plus update that measures the cost of maintaining our optimized indexes while sustaining a read load, and (3) read scan that measures the impact of our optimized indexes on scan performance.

A. Implementation and Setup

1) Implementation: We ran experiments against three Bw-tree variants. (1) Our Bw-tree implementation was described in a previous paper [13] and ships in a number of Microsoft production systems including SQL Server Hekaton, Azure DocumentDB, and Bing. We refer to this implementation as **`Base`** in the rest of this section. (2) We wanted to capture the effect of using the prefix B-tree technique to determine the separator chosen when splitting leaf pages is at the midpoint. This variant is called **`Prefix`**. It uses our mid-key vector. And the prefix extracted is the common prefix of entries on the page, not the key space. Finally, variant (3) **`Prefix+Split`** is an implementation that includes our page organization and determines prefix compression using the split point optimization as presented in Section II-C3, using the common prefix of the page entries, and the optimized split point.

2) Machine Configuration: Our experimental machine set up is described in Table 1. Each of its four CPUs reside in separate NUMA nodes, which are organized as a ring.

3) Datasets: We use three datasets in our experiments; two from real production workloads and one synthetic. None of these was selected to show our method to best advantage.

Azure DocumentDB: This dataset contains 9 million records from a Microsoft internal database hosted by Azure DocumentDB. The Bw-tree serves as the inverted index in DocumentDB with JSON document terms (keys) mapped to bitmaps representing the documents containing the term (payloads). See [27] for more information. Keys are from a sparse key space and are variable length, with an average size of 32 bytes, and payloads are 50 bytes.

Windows Server Storage Deduplication: This data comes from a real enterprise deduplication trace in Windows Server used to generate a sequence of chunk hashes for a root file directory and compute the number of deduplicated chunks and storage bytes. This trace contains 27 million total chunks and 12 million unique chunks. Keys are 20-byte SHA-1 hash values that uniquely identify a chunk, while the value payload contains a 44-byte metadata string. The total raw size of the index is 1.6 GB.

Synthetic: Our synthetic dataset consists of 28-byte fixed-size keys with 100-byte payloads. Each key is generated by RockDBs GenerateKeyFromInt() function and consists of an 8-byte prefix and a 20-byte suffix, where each prefix is shared by 1,000 suffixes. Keys are selected randomly from the key space, using a uniform distribution.

B. Memory Impact

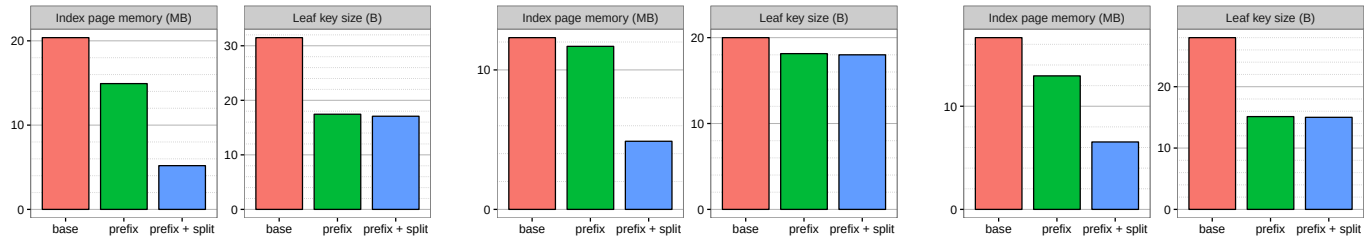
Here we present results that describe the effectiveness of our compression techniques in two respects. (1) How effective is tail compression (choosing a short separator) for the various techniques. Tail compression is particularly important in determining the size of the index levels of the tree, since the size of the common prefix shrinks as one goes up the tree. (2) What is the size of the key prefix that we can extract from keys on a given page. Factoring out this common prefix is the only compression that takes place on the leaf level of the tree.

The numbers reported are for a Bw-tree index filled with all the records in a dataset. The bar charts 8a, 8b, and 8c show the results of our experiments on data sets DocumentDB, De-dup, and synthetic workloads respectively. All figures show, on the left, the impact of the compression technique used on index memory size and, on the right, the size of the key in data pages after the prefix was factored out.

Memory footprint is extremely important for at least two reasons. (1) Processor cache effectiveness is improved, improving execution time performance. (2) In a cloud setting, memory footprint can be reduced further while limiting the performance impact.

DocumentDB is perhaps the most realistic data set, being real data consisting, in part, of document words. What the 8a shows is the dramatic reduction in index size of using the techniques in this paper. Both **`Prefix`** and **`Prefix+Split`** show good space reductions and very effective key size reductions in the data pages. The index size reduction is mostly due to the ability to choose a shorter separator for index terms (i.e. suffix compression). Note that the difference between classic prefix B-tree techniques and ours is modest when factoring out a common prefix for a leaf data page, though the key size reduction is substantial for both. But, it is important to remember that our more aggressive prefix methods enable more flexibility in choosing the separator without compromising on prefix removal. And this shows up dramatically in the memory footprint of the resulting Bw-tree index, where index size is reduced to about 3/4 of the**`Base`** case for **`Prefix`**, but about 1/4 of the **`Base`** case for **`Prefix+Split`**. Our synthetic workload has results generally in line with the DocumentDB workload results.

The De-dup workload results show that varying workloads can have dramatically differing impacts on the resulting effectiveness of techniques. Data leaf key compression is quite modest for both **`Prefix`** and **`Prefix+Split`**. However, the results for **`Prefix+Split`** show a dramatic improvement over **`Prefix`**, enabled once again by greater flexibility in choosing the split point for leaf pages (and without compromising the prefix removal optimization.



(a) DocumentDB (b) Deduplication (c) Synthetic

Fig. 8: Index size (on left) and data page key size (on right)

Level	Key size	Prefix size		
		Key-space	On-page	Net on-page
1	11.43	0	0	0
2	18.05	5.45	6.87	1.42
3	17.65	13.89	15.56	1.67
(leaf) 4	31.49	15.94	16.60	0.66

Fig. 9: DocumentDB page entry key prefixes are longer than its key space prefixes.

We want to highlight that there are times when the prefix for a node's entries can be noticably longer than the prefix for its key space. We extract this information in Figure 9 for the DocumentDB data set. But the difference in prefix sizes was less significant with the other datasets. The important points are that longer prefixes (1) further reduce index size while (2) making index search faster, while not having a downside.

More complete results of our experiments in terms of memory impact are summarized in Figure 13 in the appendix.

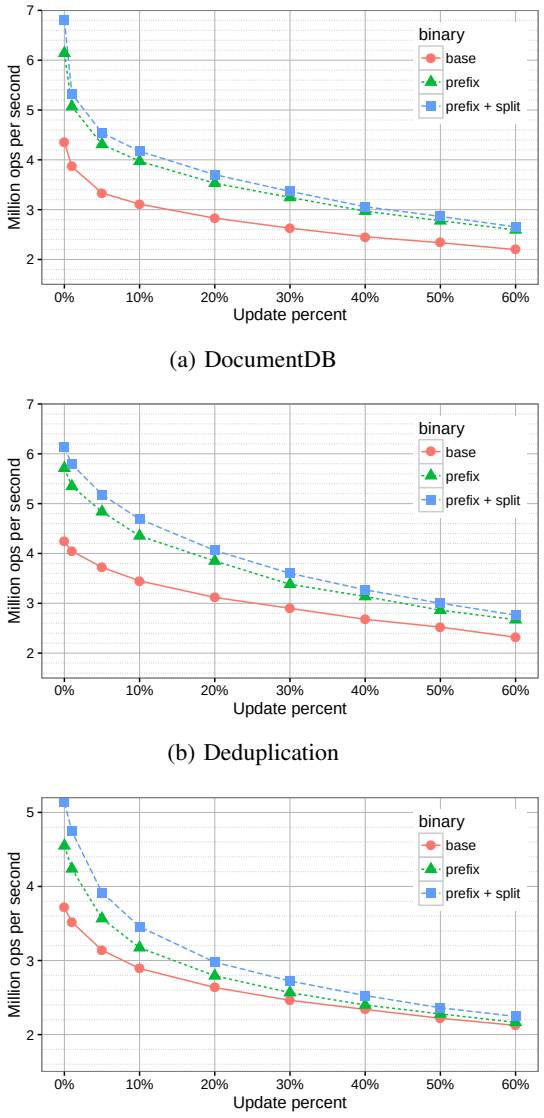
C. Performance

Our primary purpose in exploring page organization is to improve indexing performance. In this section, we report on the performance of the **`Base`**, **`Prefix`** and **`Prefix+Split`** organizations executing operations under a variety of controlled conditions using our three workloads.

1) Search and Update Performance: Updates to the Bw-tree require page maintenance to periodically rebuild pages from scratch with, e.g., when we apply delta record updates during consolidation or create new pages during a split. Our page organization optimizations require more CPU overhead to create an optimized page layout as opposed to the (unoptimized) **`Base`** implementation. Our goal in this experiment is to determine the impact of this extra organizational overhead logic on overall performance.

Different experimental runs fill the index with all records of one of our datasets. Then we start eight worker threads that randomly choose a search key from the full key range. For varying probabilities from 0% to 100% (reported on the x-axis), it performs an update on the Bw-tree for the key, otherwise it performs a read.

Figure 10 reports the results for this experiment run on all datasets. For all datasets and for all implementation variants, we observe an initial drop in performance when going from



(a) DocumentDB

(b) Deduplication

(c) Synthetic

Fig. 10: Throughput for read/update workload on all datasets.

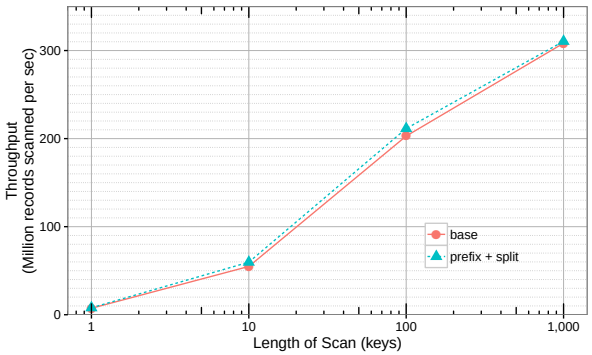


Fig. 11: Range scan performance.

0% to 1% updates. Updating increases execution path due to pointer chasing of delta chains. Further, all page organization strategies require page and tree re-organization. And this is irrespective of the particular form of page organization. The main takeaway is that for all update ratios across all datasets, our optimized pages (**`Prefix+Split`**) consistently give better overall results. Hence, the extra logic necessary to build optimized pages consistently pays of in superior performance.

The improved splitting of **`Prefix+Split`**) over **`Prefix`**) also shows up as a modest performance improvement. While there is not a large improvement in factoring out the prefixes, the shorter separators of **`Prefix+Split`**) result in higher performance via its shorter compares in the index and its smaller index memory footprint improves processor caching.

2) Range Scan Performance: This section studies the effect of our range scan optimizations presented in Section V as well as our page organization improvements. For this experiment, we run the same experiment used in previous work [15]. The index is loaded with the synthetic dataset with 48 worker threads, distributed across 4 NUMA nodes, employed to perform range scans of various sizes. With 95% probability, a worker begins a scan by generating a random key using a uniform distribution. It then opens a scan cursor on the Bw-tree using the generated key as its start position and scans for n records (with n varying between runs and reported on the x-axis). With 5% probability, a worker performs a blind update of a random key chosen using a Zipfian distribution, so that updates are clustered around a small number of keys.

Figure 11 reports the results for this experiment, with the numbers for **`Base`** matching what was previously published in [15]. We see an 8-9% improvement over **`Base`** over small scans, as well as a consistent 1% improvement over **`Base`** over large scans, due to both the scan improvements as well as the the page organization optimizations, which are particularly helpful for short scans that spend a larger fraction of their time traversing the index to position the cursor at the start point. Figure 11 graphs the results for scans up to 1000 records. The table in Figure 12 shows the full set of results.

VII. Related Work

There is a huge amount of work on optimizing search structures- hash based, index tree based, and combinations of the two. Some of the settings have been specific to secondary storage search; some have been restricted to main memory; some have been specialized to specific application requirements. We can only sample this work here. We focus on general purpose index trees in the context of data management systems.

A. Optimized B-trees

Optimizing B-trees started in the 1970's with the work on prefix B-trees [4]. Clearly, we owe much to that work. The idea of extracting a prefix, and producing index terms that need not be keys but separators instead all came from this. The big deficiency in the paper was the lack of a fast binary search.

One line of optimization work focused on reducing inter-node pointers so that keys are denser in index nodes. Cache sensitive search trees [CSS-trees] [24] replaced pointers down the tree with address computation to find descendent pages. The negative, acknowledged by the same authors in [25] is that the CSS-tree does not respond well to updates, though there are cases where updates are rare, or the tree is re-generated instead of being updated in place. Cache conscious B^+-trees [CSB^+-trees] [25] were intended to make B^+-trees cache sensitive like CSS-trees while enabling easy updating. This method too involved reducing node pointers in index pages of the tree by "segmenting" collections of nodes and using address (offset) computation within a segment.

Another line of work focused on cache block alignment. One thinks about the nodes of a B-tree in terms of a number of cache lines, and works to minimize the number of cache lines touched. The fractal pre-fetching B^+-tree [fpB^+-tree] [6] works at two levels. At a large scale, it looks like a B^+-tree. Within its pages, however, there is another tree (the name "fractal" captures this "tree within a tree" organization). The inner tree organization is guided by cache line size and alignment for its nodes.

B. B-tree Alternatives

There have been a number of alternatives suggested as replacements for B-trees, especially in the context of main memory systems, where the ability to paginate the index for secondary storage is perhaps no longer necessary.

The mass-tree [19] is a variant of an index tree that shares a lot in common with tries [10]. Tries permit early parts of keys to be factored out as the indexing proceeds. Indeed, one way to view prefix B-trees is to think about them as an effort to give B-trees at least some of the advantages of tries. Prefix B-trees give up some of the prefix factoring to be compatible with secondary storage. However, tries (including the mass-tree) are very fast for in-memory use.

AVL trees [1] have been suggested as a balanced tree structure for in-memory indexing. And it has been suggested for main memory databases as well [9]. Index nodes are small and connected via pointers, which does not fit with the use of secondary storage. A similar situation arises with t-trees [12], which are also not suitable for secondary storage contexts.

An indexing technology that shares the latch-free characteristics of the Bw-tree is skip lists [23]. This indexing structure constructs "towers" of index terms connected via linked lists. We previously compared the original Bw-tree with linked lists [13] and found the Bw-tree to be almost a factor of four faster due to better processor cache performance.

C. Observations

With the exception of the mass-tree, most of the related work was done earlier than the appearance of processors with a large number of cores. Thus, none of them except skip lists is latch-free. Further, the memory hierarchies of processors have gotten deeper, with the penalty for accessing main memory now in the 100's of cycles. Thus, avoiding latch induced or memory induced stalls has become enormously more important, and will only increase in importance. These trends will necessitate either modifying or replacing indexing technology from an earlier era to cope with this new environment. The Bw-tree is the direct result of focusing on these considerations.

VIII. Discussion

A. Memory

Index tree performance depends crucially on how many of the internal index pages of the index tree are in main memory. We report on fully cached performance here, where leaf pages are also memory resident. However, in a cloud-based multi-tenancy setting, we want to squeeze the memory footprint of our data while maintaining great performance. In this case, index size being small is an enormously important factor. And our index size reduction (memory consumed by the tree above the leaf level) ranges is anywhere from a factor of two to a factor of four smaller than uncompressed index size (our base case). And we see significant gains compared with classical prefix B-tree techniques.

B. Performance

Using the full set of optimizations and compressions that we have implemented, we achieved a record access performance improvement of between 20% and 40%. This is, of course, much smaller than the order of magnitude gains that the Bw-tree demonstrated before compared with conventional B-tree alternatives. Node search is about half the cost incurred in accessing a record. Other costs include crossing interfaces, transit between pages, copying records out of pages, etc. Thus, gaining 40% in performance means that we have made the search about 2.5X faster. Even driving the search cost to zero would have "only" doubled record read performance. Further, the Bw-tree base case is, itself, already very well optimized, so these performance gains are on top of that.

Index optimization has only a modest impact on our range performance, mostly at small ranges. For small ranges (less than a page of data), the improved index search is the critical factor, as index traversal approaches half of the cost for these ranges. Our gain would have been more impressive but for the fact that previously we were already batching records returned by a scan, and, indeed, we were already using a range cursor as well.

C. Future Work

We are determined to explore how to move main memory techniques to a cloud setting where multi-tenancy rules. This paper may be considered a step in that direction. Indexing performance when data is cached is hugely important for data management systems, whether fully main-memory based or simply with cached data. With a cloud focus on partially cached data, shrinking the index space required enables a strategy of retaining all internal (index) nodes in main memory. This serves to limit potential I/O to only the access of data pages at the leaves of the tree. Our optimized Bw-tree makes significant improvements in both indexed access speed and index memory size.

To attack the multi-tenancy problem successfully for a transactional key-value store will require more than this. Inventive ways of tolerating increased latencies will become important if a performance collapse is to be avoided. There is a fundamental difficulty when latency within a transaction increases. Latency in this case increases the "conflict cross-section" between transactions. This is unavoidable regardless of concurrency control method, without imposing limitations on the nature of the workload, e.g. permitting only execution with foreknowledge of transactional accesses so that data can be pre-staged into the cache. But some concurrency control is more tolerant of latencies and conflicts than others. Multi-version methods are probably a step in the right direction, but it is unclear if they are sufficient.

References

[1] G. Adelson-Velsky, E. Landis. "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences (in Russian). 146: 263266. English translation by M. Ricci in Soviet Math. Doklady, 3:12591263, 1962.

[2] G. Antoshenkov, D. Lomet, J. Murray: Order Preserving Compression. ICDE 1996: 655-663

[3] Asynchronous method invocations. $https : //en.wikipedia.org/wiki/Asynchronous_method_invocation$

[4] R. Bayer and K. Unterauer, "Prefix B-Trees," *TODS*, vol. 2, no. 1, March 1977.

[5] Bernstein, P., Lomet, D., Bakerman, T., MacDonald, W., Malek, S., Mitlak, W., Schweiker, R., Tupper, J., Velardocchia, L. B-Tree Access Method DB-Kit Final Report. Wang Institute Project Course Report, (April, 1986)

[6] S. Chen, P. Gibbons, T. Mowry, and G. Valentin, "Fractal Prefetching B$\pm$Trees: Optimizing Both Cache and Disk Performance," in *SIGMOD*, 2002, pp. 157–168.

[7] DB-Engines Ranking of Key-value Stores. http://db-engines.com/en/ranking/key-value+store

[8] T. Bolt, G. Carpenter. Key Compression in the B-tree node manager. Wang Institute DB Course Project, Feb. 1987.

[9] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, David A. Wood: Implementation Techniques for Main Memory Database Systems. SIGMOD Conference 1984: 1-8

[10] D. Knuth, *The Art of Computer Programming: Sorting and Searching*. Menlo Park, CA: Addison-Wesley, 1998, vol. 3.

[11] P. Khuong, "Binary search: doing it less wrong," http://repnop.org/pd/slides/bsearch.pdf, 2014.

[12] Tobin J. Lehman, Michael J. Carey: A Study of Index Structures for Main Memory Database Management Systems. VLDB 1986: 294-303

[13] J. Levandoski, D. Lomet, and S. Sengupta, "The Bw-Tree: A B-tree for New Hardware Platforms," in *ICDE*, 2013, pp. 302–313.

[14] J. Levandoski, D. Lomet, S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. in *PVLDB 6(10)* 877-888 (2013).

[15] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang, "Multi-Version Range Concurrency Control in Deuteronomy," *PVLDB*, vol. 8, no. 13, pp. 2146–2157, 2015.
[16] David B. Lomet: Digital B-Trees. *VLDB*, 1981: pp. 333-344
[17] D. Lomet, A. Fekete, G. Weikum, and M. Zwilling. Unbundling transaction services in the cloud. In *CIDR*, 2009.
[18] D. Lomet, "The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account," *SIGMOD Record*, vol. 30, no. 3, pp. 64–69, 2001.
[19] Y. Mao, E. Kohler, and R. T. Morris, "Cache Craftiness for Fast Multicore Key-value Storage," in *Eurosys*, 2012, pp. 183–196.
[20] "MongoDB. `http://www.mongodb.org/`."
[21] T. Neumann, "Trying to speed up binary search," http://databasearchitects.blogspot.com/2015/09/trying-to-speed-up-binary-search.html, 2015.
[22] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet, "AlphaSort: A Cache-Sensitive Parallel External Sort," *VLDB Journal*, vol. 4, no. 4, pp. 603–627, 1995.
[23] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, 1990.
[24] J. Rao and K. Ross, Cache Conscious Indexing for Decision-Support in Main Memory. *VLDB*, 1999: 78-89
[25] J. Rao and K. A. Ross, "Making b^+-trees cache conscious in main memory," in *SIGMOD*, 2000, pp. 475–486.
[26] RocksDB: A persistent key-value store for fast storage environments. http://rocksdb.org/
[27] D. Shukla et al: Schema-Agnostic Indexing with Azure DocumentDB. in PVLDB 8(12): 1668-1679 (2015)
[28] A. Silberschatz, S. Zdonik, et al., Strategic Directions in Database SystemsBreaking Out of the Box, ACM Computing Surveys, Vol. 28, No. 4 (Dec 1996), 764-778.
[29] StackOverflow, http://stackoverflow.com/questions/34799081/what-are-the-scalability-benefits-of-async-non-blocking-code

APPENDIX

A.1 Table of Range Performance Results

The differences we measured in range performance between our prior implementation **`Base`** and our page optimized implementation **`Prefix`** are modest, but nonetheless interesting at small ranges. These differences are hard to see in the graph at Figure 11. So we present them here in a tabular form in Figure 12

Binary	Scan length	Mill. recs./sec	Vs. base
Base	1	7.32	
Prefix + split	1	7.90	7.9%
Base	10	54.84	
Prefix + split	10	59.69	8.8%
Base	100	202.91	
Prefix + split	100	211.42	4.2%
Base	1,000	308.35	
Prefix + split	1,000	310.47	0.7%
Base	10,000	334.12	
Prefix + split	10,000	337.64	1.1%
Base	100,000	337.39	
Prefix + split	100,000	342.46	1.5%
Base	1,000,000	338.73	
Prefix + split	1,000,000	342.44	1.1%

Fig. 12: Table of Range Performance Results and Performance Improvements.

A.2 Table of Compression Results for Various Data Sets

We present here a table containing a more complete set of the information we generated using our various workloads and page organization techniques. The table documents what is in the bar charts but also includes information about separator size. One "odd" thing stands out in this table. It is the occasional appearance of larger index terms at a higher level of the tree, which may seem strange when considering that at each level, we try to promote a short indexing term to the parent. This is particularly marked in the **`Prefix`** case, which demonstrates that inflexibility in choosing where to split can have a detrimental impact on index memory space reduction, and also, as we shall see next, on search performance.

"Dedup" experiment

Binary	Level	Key size	Prefix size	Remainder size
Base	1	20.00	0.00	20.00
Base	2	20.00	0.00	20.00
Base	3	20.00	0.00	20.00
Base	(leaf) 4	20.00	0.00	20.00
Prefix	1	20.00	0.00	20.00
Prefix	2	20.00	0.00	20.00
Prefix	3	20.00	0.94	19.06
Prefix	(leaf) 4	20.00	1.84	18.14
Prefix + split	1	1.00	0.00	1.00
Prefix + split	2	1.95	0.00	1.95
Prefix + split	3	2.85	0.94	1.92
Prefix + split	(leaf) 4	20.00	2.00	18.00

"DocDB" experiment

Binary	Level	Key size	Prefix size	Remainder size
Base	1	79.33	0.00	79.33
Base	2	71.89	0.00	71.89
Base	3	58.68	0.00	58.68
Base	4	40.44	0.00	40.44
Base	(leaf) 5	31.49	0.00	31.49
Prefix	1	81.56	0.00	81.56
Prefix	2	74.58	4.10	71.42
Prefix	3	60.93	17.18	47.71
Prefix	4	41.73	21.32	26.61
Prefix	(leaf) 5	31.49	18.15	17.45
Prefix + split	1	11.43	0.00	11.43
Prefix + split	2	18.05	6.87	14.23
Prefix + split	3	17.65	15.56	4.54
Prefix + split	(leaf) 4	31.49	16.60	17.08

"Synthetic" experiment

Binary	Level	Key size	Prefix size	Remainder size
Base	1	28.00	0.00	28.00
Base	2	28.00	0.00	28.00
Base	3	28.00	0.00	28.00
Base	4	28.00	0.00	28.00
Base	(leaf) 5	28.00	0.00	28.00
Prefix	1	28.00	6.00	22.00
Prefix	2	28.00	6.00	22.00
Prefix	3	28.00	6.28	21.75
Prefix	4	28.00	6.99	21.01
Prefix	(leaf) 5	28.00	12.76	15.13
Prefix + split	1	7.25	6.00	1.25
Prefix + split	2	8.01	6.00	2.01
Prefix + split	3	13.81	6.99	6.82
Prefix + split	(leaf) 4	28.00	12.99	15.01

Fig. 13: Table of Key Compression Results.