Cache-Sensitive Skip List: Efficient Range Queries on modern CPUs

Stefan Sprenger, Steffen Zeuch, and Ulf Leser

Humboldt-Universität zu Berlin, Institute for Computer Science,
Unter den Linden 6, D-10099 Berlin, Germany
{sprengsz,zeuchste,leser}@informatik.hu-berlin.de

Abstract. Due to ever falling prices and advancements in chip technologies, many of today's databases can be entirely kept in main memory. However, reusing existing disk-based index structures for managing data in memory leads to suboptimal performance due to inefficient cache usage and negligence of the capabilities of modern CPUs. Accordingly, a number of main-memory optimized index structures have been proposed, yet most of them focus entirely on single-key lookups, neglecting the equally important range queries. We present Cache-Sensitive Skip Lists (CSSL) as a novel index structure that is optimized for range queries and exploits modern CPUs. CSSL is based on a cache-friendly data layout and traversal algorithm that minimizes cache misses, branch mispredictions, and allows to exploit SIMD instructions for search. In our experiments, CSSL's range query performance surpasses all competitors significantly. Even for lookups, it is only surpassed by the recently presented ART index structure. We therefore see CSSL as a serious alternative for mixed key/range workloads on main-memory databases.

Keywords: Index Structures; Main-Memory Databases; Scientific Databases

1 Introduction

Over the last years, various index structures were designed for fast and space-efficient execution of search operations in main memory, like the adaptive radix tree (ART) [13] or Cache-Sensitive B^+-tree (CSB^+) [18]. By reducing cache misses, improving cache line utilization, and exploiting vectorized instructions, they outperform conventional database index structures, like B-trees [5], which were mostly designed to reduce disk accesses. Most of these novel index methods focus on single-key lookups and show suboptimal performance for range queries, despite their importance in many applications. Use cases for range queries are numerous, such as: queries in a data warehouse that ask for sales in a certain price range, analysis of meteorological data that considers certain yearly time periods in long time series, and Bioinformaticians who build databases of hundreds of millions of mutations in the human genome that are analyzed by ranges defined by genes [9].

In this paper, we introduce the Cache-Sensitive Skip List (CSSL), a novel main-memory index structure specifically developed for efficient range queries on modern CPUs. CSSL is based on skip lists as described in [16], yet uses a very different memory layout to take maximal advantage of modern CPU features like CPU-near cache lines, SIMD instructions, and pipelined execution. In this work, we focus on read performance but provide a technique for handling updates, too. Besides many other use cases, we see CSSL as perfectly suited for scientific databases that prefer fast reads over fast writes and need range queries in many cases. Especially the bioinformatics community, which is confronted with an exponentially growing amount of genomic data that is mostly analyzed with range queries to investigate certain genomic regions [20], may benefit from our approach.

We evaluated CSSL on data sets of various sizes and properties and compared its performance to CSB$^+$-tree [18], ART [13], B$^+$-tree [7], and binary search on a static array. We also include experiments with real-world data from the bioinformatics domain to investigate performance on non-synthetic key distributions. For range queries and mixed workloads, CSSL is consistently faster than all state-of-the-art approaches, often by an order of magnitude; also its lookup performance is way ahead of all competitors except ART.

The remaining paper is structured as follows. The next section introduces skip lists, the index structure that CSSL is based on. Section 3 presents the Cache-Sensitive Skip List as our main contribution. Section 4 describes algorithms for executing lookups and range queries on CSSL. In Section 5, we compare CSSL against other state-of-the-art index structures using synthetic as well as non-synthetic data. Section 6 discusses related work, and Section 7 concludes this paper.

2 Preliminaries

Skip lists were originally presented as a probabilistic data structure similar to B-trees [16]. Skip lists consist of multiple lanes of keys organized in a hierarchical fashion (see Figure 1). At the highest level of granularity, a skip list contains a linked list of all keys in sorted order. In addition to this so-called *data list*, skip lists maintain *fast lanes* at different levels. A fast lane at level i contains $n * p^i$ elements on average, where n is the number of keys to be stored and $0 < p < 1$ is a parameter. Skip lists were originally proposed as probabilistic data structures, as the elements to be stored in higher lanes are randomly chosen from those at lower lanes: Every element of fast lane i appears in fast lane $i + 1$ with probability p. This scheme allows for efficient updates and inserts, yet makes the data structure less predictable.

In our work, we use a deterministic variant of skip lists, so-called *perfectly balanced skip lists* [15]. In balanced skip lists, the fast lane at level $i+1$ contains every $1/p$'th element of the fast lane at level i. Accordingly, for $p = 0.5$ a lane at level $i + 1$ contains every second element of level i, in which case a skip list resembles a balanced binary search tree. Figure 1 shows a balanced skip list over nine integer keys with two fast lanes for $p = 0.5$.

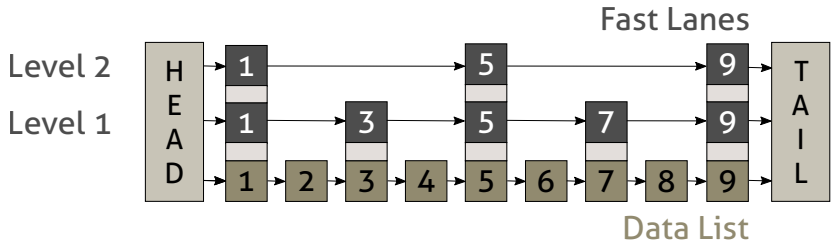

Fig. 1. A balanced skip list that manages nine keys and two fast lanes; each fast lane skips over two elements ($p = 1/2$).

In case of a low p value, fast lanes skip over many elements, therefore, fast lanes can be considered sparse. In case of a high p value, fast lanes skip over few elements, therefore, fast lanes can be considered dense. Fast lanes are used to narrow down the data list segment that may contain the searched element to avoid a full scan. For instance, a search for key 6 would traverse the skip list of Figure 1 as follows. First, search determines the first element of the highest fast lane at level 2 by using the head element. Second, the fast lane will be traversed until the subsequent element is either equal to the searched element, in which case search terminates, or greater than the searched element. In this example, search stops at element 5. Third, search moves down to the next fast lane. In this example, traversal jumps to element 5 of the fast lane at level 1. Fourth, steps two and three are repeated until the data list is reached. Fifth, the data list is scanned until the searched element is found or proven to be non-existing. In a fully built balanced skip list for $p = 0.5$, search requires $O(log(n))$ key comparisons in the worst case. Parameter p directly influences the structure of the fast lane hierarchy and should be chosen depending on the expected number of keys. If p is too high, only few keys need to be compared per fast lane when searching, but a lot of fast lane levels are required to fully build a balanced skip list. If p is too low, a lot of keys need to be compared per fast lane when searching, but only few fast lane levels are required to fully build a balanced skip list.

Besides single-key lookups, skip lists also offer very efficient range queries. Since the data list is kept in sorted order, implementing a range query requires two steps: 1) Search the first element that satisfies the queried range, and 2) traverse the data list to collect all elements that match the range boundaries.

In the original paper [16], skip lists are implemented using so-called *fat keys*. A fat key is a record that contains a key and an array, which holds pointers to subsequent elements for every fast lane and for the data list. The advantage of this approach is that all nodes are uniform, which simplifies the implementation. Furthermore, if a key is found in an upper lane, search immediately stops as all instances of a key are kept in the same record. On the other hand, such an implementation is space inefficient, because it requires space for $O(m * n)$ pointers (if m is the number of fast lane levels), although most values in higher levels are padded with NULL.

Searching in skip lists using fat keys requires to follow many pointers. This layout is suboptimal on modern CPUs, as it incurs many cache misses due to

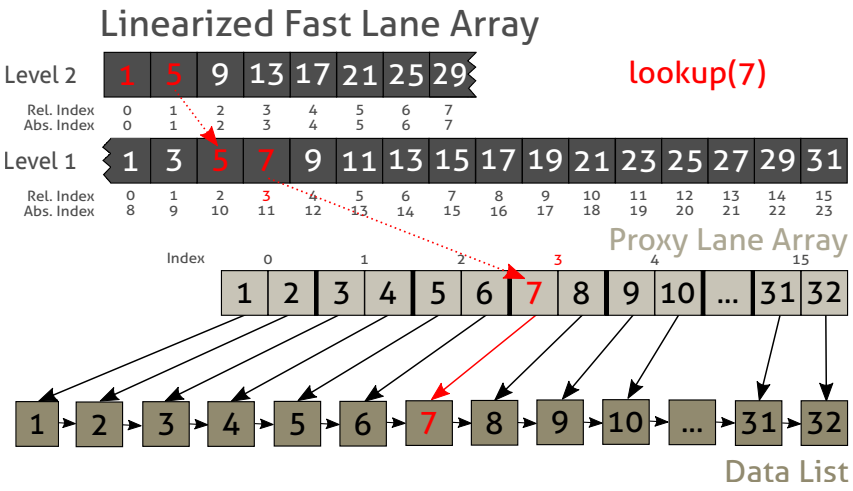

Fig. 2. A Cache-Sensitive Skip List that manages 32 keys with two fast lanes ($p = 1/2$).

jumps between non-contiguous parts of allocated memory. Even when searching the data list, cache utilization is suboptimal due to the *fatness* of keys. For instance, in a skip list that stores 32-bit integer keys and maintains five fast lanes in addition to the data list, each node takes $4\ bytes + 6 * 8\ bytes = 52\ bytes$ of memory on a 64-bit architecture. Given that a cache line is typically 64 *bytes*, each traversal step fills almost an entire cache line although only a small part of it is used. Typically, traversal steps just need the key and one pointer to find the subsequent element on a certain fast lane, i.e., $4\ bytes + 8\ bytes = 12\ bytes$.

3 Cache-Sensitive Skip List

In this paper, we present Cache-Sensitive Skip List as alternative implementation for balanced skip lists, which uses a radically different memory layout that leads to much higher efficiency in today's CPU architectures. The first and most obvious idea is to keep fast lanes as separate entities in dense arrays. This leads to less cache misses, improves the utilization of cache lines, and allows to use SIMD instructions. Figure 2 shows a Cache-Sensitive Skip List that manages 32 integer keys with two fast lanes for $p = 0.5$. The traversal path, which search would take to find key 7, is highlighted in red.

CSSL's main contributions are threefold: First, fast lanes are linearized and managed in one dense array, which is called *Linearized Fast Lane Array*, instead of being kept in data list nodes. This improves utilization of cache lines when executing a lookup or range query. Second, by linearizing fast lanes we eliminate the need to store and follow pointers. For a given n, the number of fast lane elements is known a-priori since we build on balanced skip lists. Thus, we can simply compute the position of follow-up elements within the array, making pointers completely superfluous. In Figure 2, pointerless traversal over fast lanes is indicated by dotted arrows. In our current implementation, we always preallocate a certain amount of memory per fast lane based on a hypothetical maximum

t of keys. As long as $n < t$, all inserts can be managed inside the data structure; as soon as n exceeds t, we rebuild fast lanes and increase t by a fixed fraction (see Section 3.2 for details on an update strategy). Third, CSSL uses SIMD instructions to iterate over matching keys when executing range queries, which is especially useful in the case of large ranges. We exploit the lowest fast lane, i.e., the fast lane at level 1, to search for the last key that satisfies the queried range. To the best of our knowledge, CSSL is the first index structure that can make significant use of SIMD instructions when executing range queries.

Our approach to linearization of fast lanes has the following benefits compared to conventional skip lists: First, CSSL need less memory. Let k be the size of a key and r be the size of a pointer. Ignoring space requirements for data objects, which is equal in both layouts, conventional skip lists require $n * (m * r + r + k)$ space, whereas CSSL only require $n * (r + k) + \sum_{i=1}^{m} p^i * n * k$. Second, traversing linearized fast lanes has a better cache line utilization because we always use the whole cache line content until we abort search and jump to a lower layer. In the case of 32-bit keys, 16 fast lane elements fit into one 64-byte cache line while only one fat key of a conventional skip list fits into it. Third, since traversal of linearized fast lanes accesses successive array positions, we can make use of prefetched cache lines. Fourth, array-based storage of fast lane elements allows the usage of SIMD instructions and enables data-level parallelism. Given that s is the size of a SIMD register and k is the key size, $\frac{s}{k}$ fast lane elements can be compared in parallel. Modern CPUs usually feature SIMD registers having a size of 128 or 256 bits, thus four or eight 32-bit integers can be processed per instruction. For the implementation of CSSL, we use Intel's AVX instructions [2] that support 256-bit SIMD registers.

3.1 Optimizations

Besides these main concepts, we apply a number of further optimizations to fully exploit modern CPUs. First, we always tailor the size of fast lanes as multiples of the CPU cache line size (see Figure 3). This especially affects the highest fast lane level. Second, we introduce an additional lane, called *proxy lane*, between the lowest fast lane and the data list (see Figure 2). For each key, the proxy lane maintains a pointer to its corresponding data object. Connections between the proxy lane, which is implemented as an array of structs, and the fast lane at level 1 are implicit: The i'th fast lane element is part of the struct that can be found at index $i - 1$ of the proxy lane. We use the proxy lane to connect the lowest fast lane with the data list. Third, in practice we observed that searching the highest fast lane is very expensive in terms of CPU cycles if it contains lots of elements. This is especially the case if the number of fast lanes is kept small and the highest fast lane contains a lot more than $1/p$ elements. In the worst case, we have to scan the whole lane, while searching the remaining fast lanes can never require more than $1/p$ comparisons per lane. We accelerate searching the highest fast lane by using a binary search instead of sticking to a sequential scan.

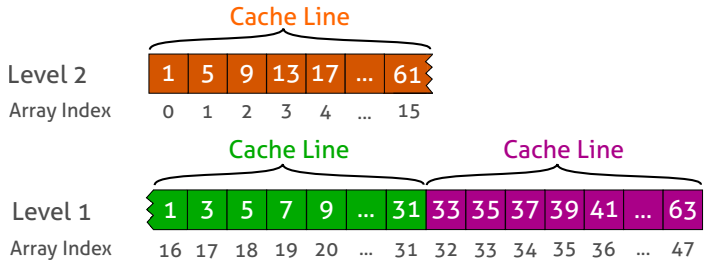

Fig. 3. Linearized fast lane array of a CSSL that indexes all 32-bit integers in $\{1, .., 64\}$ with two levels ($p = 1/2$).

3.2 Updates

In our implementation, a CSSL is initialized with a sorted set of keys. Nonetheless, we still want to support *online updates*. In the following, we describe techniques for inserting new keys, updating existing keys, and removing keys.

Inserting keys: Since CSSL employs dense arrays for managing fast lanes, directly inserting keys into fast lanes would require a lot of shift operations to preserve the order of fast lane elements. For this reason, new keys are only inserted into the data list, which is implemented as a common linked list. We create a new node and add it at the proper position. As soon as the fast lane array gets rebuilt to allocate more space, new keys are also reflected in the fast lane hierarchy. Nonetheless, we can find new keys in the meantime. If search does not find a key in the fast lanes, it moves down to the data list and scans it until the key is found or proven to be non-existing. The insert algorithm can be implemented latch-free by using an atomic compare-and-swap instruction for changing pointers in the data list.

Deleting keys: In contrast to insertions, we cannot delete keys from the data list but leave fast lanes untouched, because this would lead to invalid search results. In the first step of deleting a key from CSSL, we need to eliminate it from the fast lane array. Just changing the corresponding entry to $NULL$ would require reshift operations to close gaps in the array. Therefore, we replace to-be-deleted entries with a copy of the successive fast lane element. This allows fast deletes but leaves the fast lane structure intact. We end up with duplicates in the fast lane array that are removed as soon as the array gets rebuilt. As last step, the *next* pointer of the preceding node in the data list is changed to point to the successor of the to-be-removed-node and the node is deleted.

Updating keys: Updates are basically implemented as an insert operation followed by a deletion.

Though being based on balanced skip lists, which leads to less flexibility compared to common skip lists, CSSL is able to handle online updates. By limiting in-place updates on the fast lane array, we can keep the number of cache invalidations small.

4 Algorithms

In this section, we describe in detail algorithms for executing lookups and range queries using CSSL. We start by presenting the lookup algorithm, because the execution of range queries is based on it.

Lookups: Pseudocode for lookups is shown in Algorithm 1. If search is successful the element's key will be returned, if not INT_MAX will be returned. The algorithm can be split into multiple parts. First, the highest fast lane is processed with a binary search (see Line 1). Second, the remaining fast lanes are searched hierarchically to narrow down the data list segment that may hold the search key (see Lines 2-8). We scan each fast lane sequentially instead of employing a binary search, because we need to compare only $1/p$ elements per fast lane level. Third, if the last fast lane contains the searched element, it is immediately returned (see Line 9); otherwise the associated proxy node is loaded and all keys of the data list are compared with the searched element (see Lines 10-12). INT_MAX is returned if no matching element is found (see Line 13).

Algorithm 1: lookup(key)

```
 1: pos = binary_search_top_lane(flanes, key);
 2: for (level = MAX_LEVEL - 1; level > 0; level--) {
 3:   rPos = pos - level_start_pos[level];
 4:   while (key >= flanes[++pos])
 5:     rPos++;
 6:   if (level == 1) break;
 7:   pos = level_start_pos[level-1] + 1/p * rPos;
 8: }
 9: if (key == flanes[--pos]) return key;
10: proxy = proxy_nodes[pos - level_start_pos[1]];
11: for (i = 1; i < 1/p; i++)
12:   if (key == proxy->keys[i]) return key;
13: return INT_MAX;
```

Range Queries: Pseudocode for range queries is shown in Algorithm 2. Search returns pointers to the first and last data list element that match the given range defined by *start* and *end*, i.e., it returns a linked list that can be used for further processing. Execution of range queries is implemented as follows.

First, the first matching element is searched similar to executing a lookup (see Lines 1-16 of Algorithm 2). Second, the algorithm jumps back to the lowest fast lane and scans it using vectorized instructions to find the last element that satisfies the queried range. Using AVX, CSSL can process eight 32-bit integer keys in parallel (see Lines 17-25). Third, the proxy node, which is associated with the matching fast lane entry, is loaded and compared with the range end to determine the last matching element (see Lines 29-35). Fourth, range search returns a struct that provides pointers to the first and last matching element in the data list (see Line 36).

Algorithm 2: searchRange(start, end)

```
1: RangeSearchResult res;
2: pos = binary_search_top_lane(flanes, start);
3: for (level = MAX_LEVEL - 1; level > 0; level--) {
4:   rPos = pos - level_start_pos[level];
5:   while (start >= flanes[++pos])
```

```
 6:     rPos++;
 7:   if (level == 1) break;
 8:   pos = level_start_pos[level-1] + 1/p * rPos;
 9: }
10: proxy = proxy_nodes[rPos];
11: res.start = proxy->pointers[1/p - 1]->next;
12: for (i=0; i < 1/p; i++) {
13:   if (start <= proxy->keys[i]) {
14:     res.start = proxy->pointers[i]; break;
15:   }
16: }
17: sreg = _mm256_castsi256_ps(_mm256_set1_epi32(end));
18: while (rPos < level_items[1] - 8) {
19:   creg = _mm256_castsi256_ps(
20:     _mm256_loadu_si256((__m256i const *) &flanes[pos]));
21:   res = _mm256_cmp_ps(sreg, creg, 30);
22:   bitmask = _mm256_movemask_ps(res);
23:   if (bitmask < 0xff) break;
24:   pos += 8; rPos += 8;
25: }
26: pos--; rPos--;
27: while (end >= flanes[++pos] && rPos < level_items[1])
28:   rPos++;
29: proxy = proxy_nodes[rPos];
30: res.end = proxy->pointers[1/p - 1];
31: for (i=1; i < 1/p; i++) {
32:   if (end < proxy->keys[i]) {
33:     res.end = proxy->pointers[i - 1]; break;
34:   }
35: }
36: return res;
```

5 Evaluation

We compare CSSL to other index structures optimized for in-memory storage. We also include B^+-tree [7] as baseline approach, though we note that it is designed to be stored on disk. We compare competitors w.r.t. performance of range queries (see Section 5.1), performance of lookups (see Section 5.2), performance on a mixed workload (see Section 5.3), and space consumption (see Section 5.5). An evaluation with real-world data from the bioinformatics domain can be found in Section 5.4; results are very similar to those for synthetic data sets. Search performance is measured in throughput, i.e., how many queries are processed per second. For our main evaluation, we use n 32-bit integer keys with dense and sparse distribution. For dense distribution, every key in $[1, n]$ is indexed; for sparse distribution n random keys from $[1, 2^{31})$ are indexed. We evaluate

CSSL with two configurations, $CSSL_2$ with $p = 1/2$ and $CSSL_5$ with $p = 1/5$, to investigate effects on dense and sparse fast lanes. The theoretical optimum for the number of fast lanes would use so many fast lanes that the uppermost fits exactly into the L1 Cache of the CPU. In our current CSSL implementation, we give the number of desired fast lanes as parameter, which we set to a value close to the optimum. In the experiments, it was set to nine.

We compare to the following approaches:

- the adaptive radix tree (ART) [13], a recent radix tree variant designed for main memory,
- the CSB^+-tree [18], a cache-sensitive variant of the B^+-tree,
- a binary search (BS) on a static array,
- and a B^+-tree [1] as baseline approach.

For ART and CSB^+, we used implementations provided by the authors. For CSB^+, we had to implement range queries. We consider BS as the only index structure that is read-only by design.

Our test system consists of the following hardware: a Intel Xeon E5-2620 CPU with 6 cores, 12 threads, 15 MB Level 3 Cache, 256-bit SIMD registers (AVX) and a clock speed of 2 GHz. The evaluation system runs Linux and has 32 GB RAM. All experiments are single-threaded. All competitors including CSSL were compiled with GCC 4.8.4 using optimization *-O3*. We use PAPI [3] to collect performance counters.

5.1 Range Queries

The goal of CSSL is to achieve high range query performance by employing a data layout tailored to the cache hierarchy of modern CPUs, which also can be traversed using SIMD instructions. In this section, we evaluate all approaches for range queries on 16M and 256M 32-bit integer keys w.r.t. different range sizes (0.1%, 1%, and 10% of n). We determine to-be-evaluated ranges by selecting a random key from the set of indexed elements as lower bound and adding the range size to define the upper bound. For dense distribution, this creates a range covering $|upper\_bound - lower\_bound|$ elements. For sparse distribution, ranges are created in the same way, yet contain less elements, which usually leads to higher throughput.

Figure 4 shows results for executing range queries on 16M keys. Both CSSL configurations outperform all contestants for both key distributions and all evaluated range sizes. In contrast to all competitors, CSSL does not need to follow pointers when iterating over matching keys but can use SIMD instructions to traverse the fast lane array, which results in an outstanding performance. The usage of SIMD instructions accelerates the performance of CSSL by a factor between 2 to 3, depending on the concrete configuration (data not shown). $CSSL_5$ is faster than $CSSL_2$, which is due to the fact that fast lanes skip over five instead of only two elements, thus less keys have to be compared when searching for the range end (see Lines 17-28 of Algorithm 2). The sequential access pattern of CSSL has several benefits as revealed by analyzing performance counters (see

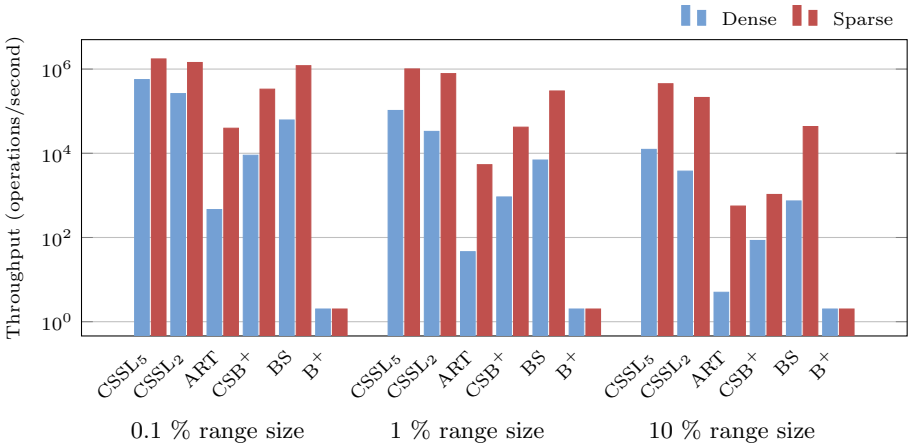

Fig. 4. Range query throughput for 16M 32-bit integer keys w.r.t. different range sizes (logarithmic scale).

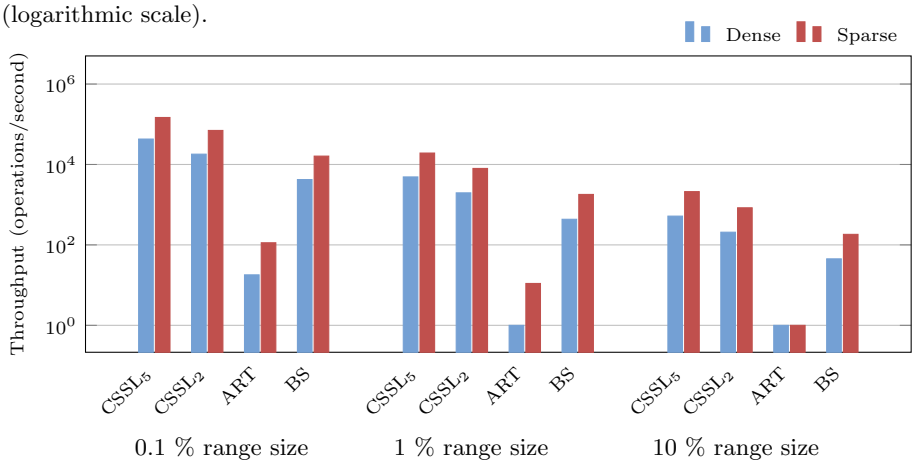

Fig. 5. Range query throughput for 256M 32-bit integer keys w.r.t. different range sizes (logarithmic scale).

Figure 6). CSSL utilizes most prefetched cache lines, which leads to only few cache misses. Furthermore, CSSL generates less branch mispredictions than the contestants, because it processes mostly consecutive positions of the fast lane array. This benefits the number of CPU cycles needed to execute a range query.

For this experiment, BS is the second best competitor followed by CSB$^+$, ART and B$^+$. By eliminating pointer accesses and taking cache line sizes into account, CSB$^+$ is able to reduce cache misses significantly compared to B$^+$-tree as shown in Figure 6.

For 16M dense keys, CSSL$_5$ is up to 16.8X faster (10.4X for sparse data) than the second best competitor BS. Compared to all competitors, CSSL achieves the best relative performance for large range sizes, i.e., the speedup factor is the highest for large ranges, because it can traverse matching keys without chasing pointers. Figure 5 shows results for executing range queries on 256M keys. Both

Performance Counter	$CSSL_5$	$CSSL_2$	ART	CSB^+	BS	B^+
Dense						
CPU Cycles	**202k**	661k	501M	27M	3.4M	1,070M
Branch Mispredictions	**12**	15	813k	46	13	1.4k
Level 3 Cache Hits	**8k**	24k	1.3M	49k	21k	1.6k
Level 3 Cache Misses	**21**	7.3k	2.7M	243k	7.4k	7.8M
TLB Misses	**5**	13	1.6M	99	24	381k
Sparse						
CPU Cycles	**5k**	13k	4.5M	620k	59k	1,095M
Branch Mispredictions	**13**	16	16k	4.6k	**13**	832
Level 3 Cache Hits	**139**	373	14k	364	325	1.8k
Level 3 Cache Misses	**23**	165	28k	5.7k	278	7.4M
TLB Misses	**3**	5	19k	958	10	369k

Fig. 6. Performance counters per range query on 16M 32-bit integer keys (10 % range size).

CSB^+ and B^+ were not able to index this amount of data, because they ran out of memory. Again, CSSL outperforms BS and ART significantly.

5.2 Lookups

We evaluate the execution of single-key lookups. Lookups are a common operation in database management systems and needed for various use cases. Figure 7 shows our evaluation results concerning lookup performance on 16M 32-bit integer keys for all contestants. ART achieves the best performance for both distributions. Furthermore, ART is the only competitor that can boost performance on dense keys, for instance by using lazy expansion; the remaining competitors show identical results on both distributions. CSSL achieves the second best performance, closely followed by BS and CSB^+. B^+ shows the worst performance. The density of fast lanes has almost no influence on executing lookups as $CSSL_2$ and $CSSL_5$ show an identical performance. ART is 4.4X faster than CSSL for dense keys, and 2.4X faster than CSSL for sparse keys.

In Figure 8, we present performance counters per lookup on 16M 32-bit integer keys for all competitors. ART produces no branch mispredictions and only few level 3 cache misses, while B^+-tree shows the worst performance parameters. As in the case of range queries, CSSL produces only few cache and TLB misses. Though being optimized for range queries, CSSL is able to achieve a lookup throughput that outperforms BS, CSB^+ and B^+ and is almost as fast as ART in the case of sparse keys.

5.3 Mixed Workload

Many real-world applications do neither use lookups nor range queries exclusively, but employ a mix of both. We investigate the throughput when executing a mixed workload consisting of an equal number of lookups and range queries. In this experiment, we run a benchmark of 1M randomly generated queries, i.e., 500k lookups and 500k range queries, on 16M dense and sparse 32-bit integer

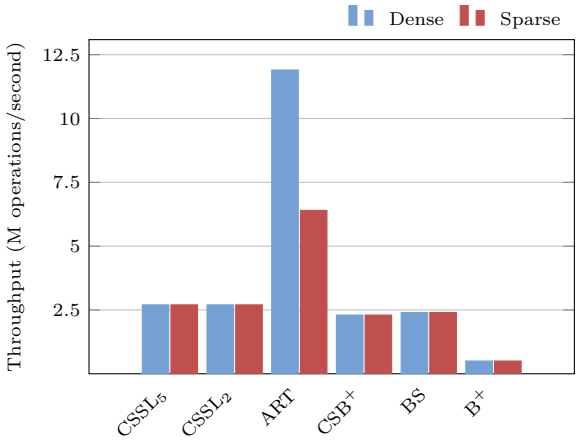

Fig. 7. Lookup throughput for 16M 32-bit integer keys.

Performance Counter	$CSSL_5$	$CSSL_2$	ART	CSB^+	BS	B^+
Dense						
CPU Cycles	927	956	**209**	1,068	1,036	5,889
Branch Mispredictions	9	13	**0**	1	12	12
Level 3 Cache Hits	11	8	**2**	3	21	28
Level 3 Cache Misses	5	8	**2**	5	9	39
TLB Misses	**1**	3	2	3	4	20
Sparse						
CPU Cycles	926	951	**383**	1,054	1,029	5,789
Branch Mispredictions	9	13	**0**	3	12	12
Level 3 Cache Hits	11	8	5	**3**	20	29
Level 3 Cache Misses	5	8	**3**	4	10	38
TLB Misses	**1**	3	4	5	4	20

Fig. 8. Performance counters per lookup on 16M 32-bit integer keys.

keys. For range queries, we always use a range size of 500k. Figure 9 shows the results of this experiment.

CSSL shows the best performance across all competitors when confronted with a mixed workload. As in the case of the range query benchmark, it is followed by BS, CSB^+, ART and B^+. Although ART shows the best single-key lookup performance, CSSL is magnitude faster when running a workload that also includes range queries besides lookups. This emphasizes the need for a fast range query implementation in index structures.

5.4 Evaluation with Genomic Data

We evaluate all competitors on real-world data from the bioinformatics domain to investigate their performance when managing data that features a non-synthetic key distribution. As data source, we used the 1000 Genomes

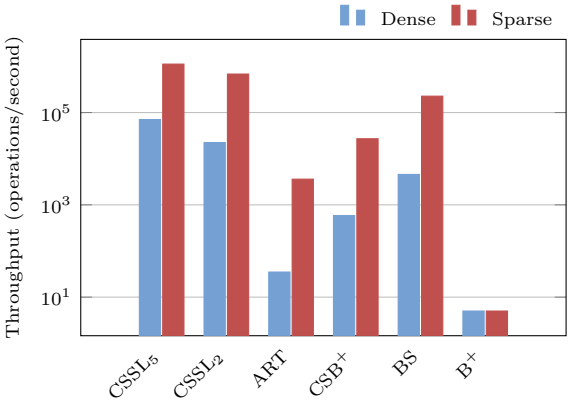

Fig. 9. Throughput for a mixed lookup/range query workload on 16M 32-bit integer keys (logarithmic scale).

Project [19] that sequenced the whole genomes of 2,504 people from across the world. Data is provided in text files and can be downloaded from the project website for free. We indexed the genomic locations of all mutations that were found on chromosomes 1 and 2, i.e., 13,571,394 mutations in total, and queried them using randomly generated ranges of different sizes (0.1%, 1%, and 10% of the featured genomic interval). Figure 10 shows results of this benchmark.

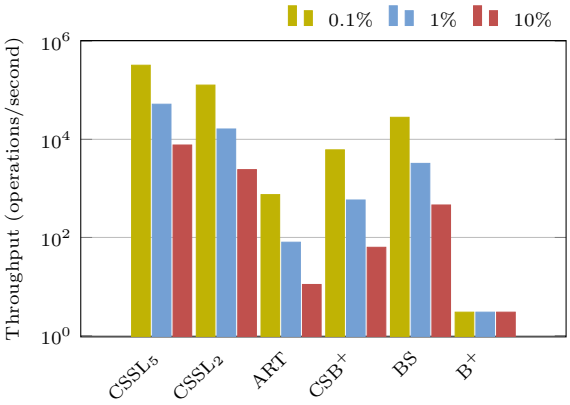

Fig. 10. Range query throughput for genomic data (13,571,394 mutations) w.r.t. different range sizes (logarithmic scale).

As for synthetic data, CSSL dominates all competitors in executing range queries. Again, BS achieves the second best throughput, followed by CSB^+, ART and B^+. All competitors, except B^+, show better performance for smaller range sizes, which is due to the fact that less mutations are covered, i.e., less keys need to be compared. For a range size of 10 %, $CSSL_5$ is 16.7X faster than BS, 121.6X faster than CSB^+, and 696X faster than ART.

5.5 Space Consumption

We compare the space consumption of all competitors for storing 16M 32-bit integer keys, i.e., 64 MB of raw data (see Figure 11). As already seen in the evaluation of search performance, ART is better suited for managing dense data than sparse data. For a dense key distribution, ART requires the least space followed by BS and CSSL. The tree-based approaches B^+ and CSB^+ show the worst memory consumption. For a sparse key distribution, BS achieves the best result followed by $CSSL_5$ and ART. Again, B^+ and CSB^+ achieve the worst results. For 16M keys, $CSSL_2$ requires 1.8X more memory than $CSSL_5$, because fast lanes hold more entries.

ART's space efficiency would probably grow for larger keys. Then, ART is able to employ further optimization techniques, e.g., path compression, that are not beneficial for small keys [13].

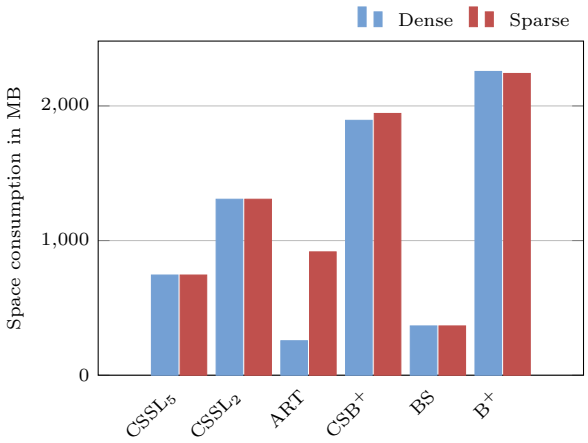

Fig. 11. Space consumption for 16M 32-bit integer keys (lower is better).

6 Related Work

Although concepts like tailored data layouts, index traversal with SIMD instructions, and pointer elimination have been investigated before [11, 17, 18], to the best of our knowledge, we are the first to combine these to accelerate range queries. Skip lists [16] were proposed as a probabilistic alternative to B-trees [5]. In the last years, they have been applied in multiple areas and have been adapted to different purposes, e.g., lock-free skip list [8], deterministic skip list [15], or concurrent skip list [10]. In [21], Xie et al. present a parallel skip list-based main-memory index, *PI*, that processes query batches using multiple threads. CSSL is based on [15], but employs a cache-friendly data layout that is tailored to modern CPUs.

There are several others approaches addressing in-memory indexing [6, 11–13, 17, 18], yet few specifically target range queries. CSS-trees [17] build a tree-based dictionary on top of a sorted array that is tailored to cache hierarchy and can be used to search in logarithmic time. CSS-trees are static by design and need to be completely rebuilt when running updates. Rao and Ross [18]

introduce the CSB$^+$-tree, a cache-conscious B$^+$-tree [7] variant, which minimizes pointer accesses and reduces space consumption. As shown in Section 5, CSSL outperforms CSB$^+$-tree significantly for all workloads. Masstree [14] is an in-memory database that employs a trie of B$^+$-trees as index structure. It supports arbitrary-length keys, which may be useful when indexing strings. We did not include Masstree in our evaluation, because its implementation is multi-threaded, which prevents a fair comparison. Instead, we considered its base index structure, the B$^+$-tree, as competitor. In [22], Zhang et al. introduce a hybrid two-stage index that can be built on top of existing index structures like B-trees or skip lists. They also propose a paged-based skip list implementation that is tailored to main memory. In contrast to CSSL, it is completely static by design and does not exploit SIMD instructions.

The adaptive radix tree [13] is a main-memory index structure based on radix trees. ART employs adaptive node sizes and makes use of CPU features like SIMD instructions to boost search performance. While it achieves high lookup performance currently only superseded by hash tables [4], its support for range queries is much less efficient since these require traversing over the tree by chasing pointers. As shown in Section 5, CSSL outperforms ART significantly for range queries. We assume that the results of our comparison between CSSL and ART would carry over to other index structures based on prefix trees, such as generalized prefix trees [6], or KISS-Tree [12]. Another recent data structure is FAST [11], a binary search tree tuned to the underlying hardware by taking architecture parameters like page or cache line size into account. It achieves both thread-level and data-level parallelism, the latter by using SIMD instructions. Similar to CSSL, FAST does not need to access pointers when traversing the tree. However, FAST is optimized for lookup queries only, where it is clearly outperformed by ART [13]. Therefore, we did not include it in our evaluation.

7 Conclusions

We presented the Cache-Sensitive Skip List (CSSL), a main-memory index structure for efficiently executing range queries on modern processors. CSSL linearizes fast lanes to achieve a CPU-friendly data layout, to reduce cache misses, and to enable the usage of SIMD instructions. We compared CSSL with three main-memory index structures, the adaptive radix tree, a CSB$^+$-tree, and binary search, and one baseline, a B$^+$-tree. CSSL outperforms all competitors when executing range queries on synthetic and real data sets. Even when confronted with a mixed key/range workload, CSSL achieves the best results in our evaluation. CSSL's search performance and memory consumption is influenced by the number of elements each fast lane skips over ($1/p$). Sparse fast lanes show better results regarding memory consumption and range query execution.

In future work, we will add multithreaded query execution to further accelerate read performance. We plan to work on both inter- and intra-query parallelism.

8 Acknowledgments

Stefan Sprenger and Steffen Zeuch are funded by the Deutsche Forschungsgemeinschaft through graduate school SOAMED (GRK 1651).

References

1. B+ tree source code (C '99). http://www.amittai.com/prose/bpt.c
2. Introduction to Intel®Advanced Vector Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions
3. PAPI. http://icl.cs.utk.edu/papi/
4. Alvarez, V., Richter, S., Chen, X., Dittrich, J.: A comparison of adaptive radix trees and hash tables. In: 31st IEEE Int. Conf. on Data Engineering (2015)
5. Bayer, R., McCreight, E.: Organization and Maintenance of Large Ordered Indices. In: SIGFIDET (1970)
6. Boehm, M., Schlegel, B., Volk, P.B., Fischer, U., Habich, D., Lehner, W.: Efficient In-Memory Indexing with Generalized Prefix Trees. In: BTW (2011)
7. Comer, D.: Ubiquitous B-tree. ACM Computing Surveys 11(2), 121–137 (1979)
8. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: Proc. 23rd annual ACM symposium on Principles of distributed computing. pp. 50–59 (2004)
9. Hakenberg, J., Cheng, W.Y., Thomas, P., Wang, Y.C., Uzilov, A.V., Chen, R.: Integrating 400 million variants from 80,000 human samples with extensive annotations: towards a knowledge base to analyze disease cohorts. BMC bioinformatics 17(1), 1 (2016)
10. Herlihy, M., Lev, Y., Luchangco, V., Shavit, N.: A provably correct scalable concurrent skip list. In: Conf. on Principles of Distributed Systems (2006)
11. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In: Proc. of the Int. Conf. on Management of Data. pp. 339–350 (2010)
12. Kissinger, T., Schlegel, B., Habich, D., Lehner, W.: KISS-Tree: Smart latch-free in-memory indexing on modern architectures. In: Proc. of the Eighth Int. Workshop on Data Management on New Hardware. pp. 16–23 (2012)
13. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: 29th IEEE Int. Conf. on Data Engineering (2013)
14. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proc. of the Seventh EuroSys Conference. pp. 183–196 (2012)
15. Munro, J.I., Papadakis, T., Sedgewick, R.: Deterministic skip lists. In: Proc. of the third annual ACM-SIAM symposium on Discrete algorithms. pp. 367–375 (1992)
16. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM 33(6), 668–676 (1990)
17. Rao, J., Ross, K.A.: Cache Conscious Indexing for Decision-Support in Main Memory. In: Proc. of 25th Int. Conf. on Very Large Data Bases. pp. 78–89 (1999)
18. Rao, J., Ross, K.A.: Making B^+-Trees Cache Conscious in Main Memory. In: Proc. of the 2000 ACM SIGMOD Int. Conf. on Management of Data. pp. 475–486 (2000)
19. The 1000 Genomes Project Consortium: A global reference for human genetic variation. Nature 526(7571), 68–74 (2015)
20. Xie, X., Lu, J., Kulbokas, E., Golub, T.R., Mootha, V., Lindblad-Toh, K., Lander, E.S., Kellis, M.: Systematic discovery of regulatory motifs in human promoters and 3' UTRs by comparison of several mammals. Nature 434(7031), 338–345 (2005)
21. Xie, Z., Cai, Q., Jagadish, H., Ooi, B.C., Wong, W.F.: PI: a Parallel in-memory skip list based Index. arXiv preprint arXiv:1601.00159 (2016)
22. Zhang, H., Andersen, D.G., Pavlo, A., Kaminsky, M., Ma, L., Shen, R.: Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In: Proc. of the Int. Conf. on Management of Data. pp. 1567–1581 (2016)