BB-Tree: A practical and efficient main-memory index structure for multidimensional workloads

Stefan Sprenger
Humboldt-Universität zu Berlin
Berlin, Germany
sprengsz@informatik.hu-berlin.de

Patrick Schäfer
Humboldt-Universität zu Berlin
Berlin, Germany
schaefpa@informatik.hu-berlin.de

Ulf Leser
Humboldt-Universität zu Berlin
Berlin, Germany
leser@informatik.hu-berlin.de

ABSTRACT

We present the BB-Tree, a fast and space-efficient index structure for processing multidimensional read/write workloads in main memory. The BB-Tree uses a k-ary search tree for pruning and searching while keeping all data in leaf nodes. It linearizes the inner search tree and manages it in a cache-optimized array, creating the need for occasional re-organizations when data changes. To reduce the frequency of such re-organizations, the BB-Tree introduces a novel architecture for leaf nodes, called bubble buckets, which can automatically morph between different representations depending on their fill degree and are thus able to buffer a large number of insertions or deletions in-place. We compare the BB-Tree to scanning, main-memory variants of the R*-tree, the kd-tree, and the VA-file, and the recent PH-tree using different multidimensional workloads over real and synthetic data sets. The BB-Tree is the fastest access method for range queries up to a selectivity of around 20% (after which it is only beaten by scanning), the fastest method in read/write workloads, and achieves an exact-match query performance similar to that of the best point access method. In addition, it is the most space-efficient method of all considered index structures. We also describe a parallel range query operator and show that it scales with the number of physical cores.

1 INTRODUCTION

Many data sets are inherently multidimensional, with typical dimensionalities ranging from two to a few dozen. We give three examples: (1) Precision Medicine is based on the comparison of a patient's mutational landscape to that of background populations and disease cohorts. Each mutation is a multidimensional object [13], with dimensions like genomic location, type of the mutation, prevalence in a population, functional impact, etc. Oncologists query such data seeking for mutations with certain properties to discover commonalities across diseases or treatment results [21]. (2) In modern machine surveillance, a battery of sensors measure multiple properties of parts of engines, such as temperature, vibration, electric currents, humidity, accelerations in all three spatial dimensions, etc. Analyzing such data for specific events or situations often induces executing series of multidimensional range queries (MDRQ) [20]. (3) In data warehousing, commercially relevant events are described by multiple, often hierarchically organized dimensions, leading to the famous OLAP cube [11]. Slicing such a cube, i.e., selecting (aggregated) events based on values of certain dimensions, often boils down to MDRQ, for instance selecting all sales in a certain date and price range.

Searching multidimensional data can be sped up by using multidimensional index structures (MDIS). MDIS can support different types of queries; in this work we focus on range queries over all (complete-match MDRQ) or a subset of the dimensions of a data space (partial-match MDRQ). MDIS are different from one-dimensional index structures because they cannot exploit a natural sort order in the data. Especially partial-match queries require MDIS to treat all dimensions equally, which typically is achieved by building and maintaining some, often hierarchical, structure on top of the data [26]. Navigation of such a structure necessitates inefficient random access patterns, which quickly leads to the situation that MDIS are outperformed by scans when queries are less selective, irrespective of whether data are held on disk [34] or in main memory [28]. The aim of this research is thus to create an MDIS that can efficiently support exact-match and range queries, that has low memory overhead, that performs gracefully in mixed read/write workloads, that is robust against the dimensionality of the data (up to a certain point) and that is faster than scans even for less selective queries.

In this paper, we present the BB-Tree, a novel main-memory MDIS which fulfills these requirements. Conceptually, a BB-Tree is an *almost-balanced* k-ary search tree, where inner nodes recursively split the data space into k partitions according to a delimiter dimension and $k-1$ delimiter values. Data objects are stored in leaf nodes (buckets). When too many data points are inserted (or deleted) and buckets overflow (or underflow), the structure is rebuilt to achieve a balance that is beneficial regarding the depths of leaves. Within this general and well-known layout, the BB-Tree combines a number of advanced techniques that yield its superior performance.

As the main contribution, BB-Trees introduce elastic buckets, called *bubble buckets* (BB), that can efficiently handle strongly fluctuating bucket fill degrees and that significantly reduce the frequency of index rebuilds. BB automatically morph between different representations, depending on their number of stored data objects. We distinguish between *regular* and *super* BB. Regular BB can hold up to b_{max} data objects and are implemented using arrays. Super BB are composites and consist of a routing node and a set of up to k regular BB, hence, they locally add a further level to the tree. BB can dynamically grow and shrink: Overflowing regular BB let them morph into super BB, and underflowing super BB let them morph back into regular BB. Both operations leave the rest of the BB-Tree unchanged. Since overflows create k new leaf buckets, a BB can cater for a rather large number of inserts. Eventually, the tree must be rebuilt when a super BB overflows. In workloads with *hammered* inserts, i.e., series of insertions into the same small region of the space, BB help to significantly reduce the number of rebuilds and thus greatly improve the performance of writes with only minimal influence on query performance though a locally slightly deeper tree.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.16

BB help to keep the inner search tree (IST) of the BB-Tree stable over long periods of data changes, which enables an adaptation of the inner nodes to cache lines, the basic unit of data transfers between main memory and on-die CPU caches. We always choose k depending on how many delimiter values fit into one cache line to improve cache line utilization. For instance, when implementing delimiter values with four-byte floats and running on a machine with 64-byte cache lines, k is set to 17. Furthermore, we store the inner nodes of the BB-Tree in a flat and static array to avoid pointer chasing during search, to decrease random access patterns, and thus to reduce cache misses, especially at the last cache level. Typically, such an optimization either makes the index structure completely static [15, 27] or creates the need to manage delta stores [19, 23]. In contrast, BB-Trees manage all changes in-place and also can, due to the separation between the IST and the leaves and due to the concept of BB, manage a large number of updates without index rebuilds. Additionally, eliminating pointers improves space efficiency.

We furthermore describe and evaluate a special technique for the parallelization of MDIS queries, which effectively avoids complicated data partitioning. In the parallel range query operator, search queries are evaluated by first navigating the IST to determine all buckets that may hold matching data objects. This step is performed by a single thread as the tree, due to its high fan out, is quite low even for very large data sets. In the next step, which strongly dominates the runtime of queries, all qualifying BB are scanned in parallel. As a result, the performance of the parallel range query operator scales with the number of physical cores.

In a comprehensive evaluation, we compared the BB-Tree to sequential and parallel scans and to four other MDIS, namely the recent PH-tree [35], and main-memory adapted variants of the R*-tree [12], the kd-tree [4], and the VA-file [34]. We used different real and synthetic data sets of different sizes with dimensionalities between three and 100. We evaluated complete-match and partial-match range and exact-match queries, and also considered mixed read/write workloads. The BB-Tree is the fastest method for range queries up to a selectivity of around 20% after which it is only outperformed by a scan. For exact-match queries, the BB-Tree is almost as fast as the best point access method, the PH-tree; for more than ten dimensions it even shows a superior performance. It is the fastest MDIS regarding insertions and the overall fastest method regarding deletions. Its performance is virtually unaffected by the dimensionality of the data. The BB-Tree has the best space efficiency among all MDIS, an important property for in-memory data structures.

A preliminary version of this work will appear in [29]. Here, we extend [29] by dynamic updates, parallel execution of range queries and provide extended experiments.

2 RELATED WORK

We structure our discussion of related work into two parts: (1) Main-memory indexing and (2) multidimensional indexing.

Main-Memory Indexing. The recent focus on main-memory database systems led to the development of several main-memory index structures. A popular example is the adaptive radix tree (ART) [18], designed for efficient execution of exact-match queries especially over longer keys. However, ART does not support multidimensional data and executing range queries requires a costly traversal of its radix tree. The cache-sensitive skip list [30] is a main-memory index focussing on range queries. It uses a CPU-friendly data layout aligned with the sizes of cache lines, a technique that was previously suggested by other one-dimensional index structures, such as FAST [15] or the KISS-Tree [17]. Schlegel et al. [27] showed how to linearize k-ary search trees in a read-only in-memory setting, a technique we also use for the BB-Tree. We introduced bubble buckets to handle updates efficiently. None of the aforementioned index structures is able to index multidimensional data for partial-match range queries.

Multidimensional Indexing. MDIS have been researched for decades, leading to a multitude of different methods [10].

One of the most popular MDIS is the kd-tree [4], which organizes multidimensional point objects in a binary search tree by splitting the data space at each node using one of the dimensions as delimiter. It is integrated into several mature database systems, e. g., PostgreSQL[1]. K-D-B-trees [25] combine the concepts of B-trees [2] and kd-trees to optimize I/O behaviour. Quadtrees [9] are similar to kd-trees, but split the space in all dimensions at each node, which is less efficient for high dimensionalities. The Vector Approximation-file (VA-file) [34] is a mixture between an MDIS and a sequential scan that divides the space into cells of equal size using hash functions to allow for efficient pruning. All of these approaches were originally developed for disk-based data storage but can be adapted to main-memory settings [28]. Only the K-D-B-tree keeps its structure balanced when data changes, whereas the VA-file is an essentially immutable index. A recent main-memory based MDIS is the PH-tree [35], which integrates the concepts of PATRICIA-tries and hypercubes.

The R-tree [12] is probably the most prominent access method for handling spatially extended objects, but is also frequently used for storing point objects [14]. It uses minimum bounding rectangles (MBR) to represent all objects belonging to a certain subtree. These MBR are used for pruning. The R*-tree [3] is an R-tree variant that improves partitioning by aggressively reinserting data objects leading to a more efficient search performance. It is employed by several database systems to manage spatial data, e. g., SQLite[2]. PR-trees [1] optimize I/O in disk-based systems, and X-trees [5] target data of very high dimensionality. The CR-tree [16] is an R-tree variant that compresses inner nodes to pack more entries into MBR, which increases space and cache efficiency. Recently, Qi et al. [24] proposed a novel R-tree packing technique that provides asymptotically optimal I/O search complexity. However, their technique is restricted to static data and requires a complete reconstruction of the index with every update. Accordingly, the latter three approaches are not directly relevant for our work.

There also exist a number of interesting works, which are further away from our own research. Wang et al. [32, 33] exploit the characteristics of observational data, e. g., immutability, continuous dimension values, and append-only insertions, to achieve high query efficiency. In contrast, the BB-Tree is a general-purpose MDIS that supports updates in any order. ELF [6] executes multi-column selection predicates on in-memory data, but requires delta stores to handle updates.

3 THE BB-TREE INDEX STRUCTURE

In a nutshell, a BB-Tree is a main-memory optimized MDIS for point data. It combines the pruning power of an *almost-balanced* k-ary search tree with the efficiency of scans in main memory.

[1] https://www.postgresql.org/docs/9.6/static/spgist.html
[2] https://sqlite.org/rtree.html

Notation	Description
n	Size of the data set.
m	Dimensionality of the data set.
h	Tree height.
k	Inner nodes split the space into k subparts according to a delimiter dimension and $k-1$ delimiter values.
t	Number of available hardware threads.
B_{match}	Number of bubble buckets that need to be scanned to evaluate a certain range query.

Parameter	Description
b_{max}	Capacity of a regular bubble bucket.
$R_{samples}$	When reorganizing the inner search tree, we use $R_{samples}\%$ of all data as samples.

Table 1: Frequently used notations and input parameters.

The inner search tree (IST) is linearized and stored in a cache-optimized yet immutable array. Data objects are stored in special leaf nodes, the bubble buckets (BB), which are able to digest a large number of insertions or deletions without affecting the IST and without hurting the tree balance considerably. Nevertheless, in case of long series of hammered inserts or deletions, the BB-Tree must be rebuilt to keep its balance. In this case, the structure of the novel tree is determined using random sampling. In the following, we describe the different components and techniques of the BB-Tree in detail; Table 1 summarizes our notation.

3.1 Data Organization

A BB-Tree consists of two components: A k-ary search tree and a set of bubble buckets. Inner nodes of the search tree recursively split the data space into k disjoint subsets according to a delimiter dimension and $k-1$ delimiter values. All data are kept in BB, which initially hold up to b_{max} m-dimensional data objects, but can dynamically expand and shrink to cope with varying number of objects in the region they represent. When searching in a BB-Tree, the inner nodes are navigated to reduce the data space. Once all certainly irrelevant regions (or subtrees) have been pruned, the remaining BB are scanned to determine the true query results.

Inner search tree. The entire IST is implemented as a single, immutable array. This has several advantages: (1) Cache lines are the basic unit for transferring data between main memory and CPU caches. By choosing an appropriate value for k, the BB-Tree tailors its inner nodes to the individual cache line size of the CPU, which increases cache line utilization and reduces the amount of data transferred through the cache hierarchy. (2) Using a single dense array for representing a balanced IST makes pointers superfluous. Array indexes of child nodes are calculated in constant time based on the current tree level and the fan out k. The BB-Tree linearizes inner nodes in a breadth-first order, which reduces memory pressure and increases cache efficiency during traversals. (3) By using an array representation, searching a specific delimiter value within an inner node (as necessary during searching) can be efficiently implemented using a binary search, which requires only $log_2(k-1)$ instead of $(k-1)$ comparisons. Note that all these accesses occur within a single cache line, which means that they do not produce any cache miss.

Leaf nodes. All data objects are stored in bubble buckets. The elasticity property of BB is described in Section 3.2. For now, we assume that every BB has a maximum capacity of b_{max} objects. b_{max} is an important parameter of the BB-Tree as it determines, at query time, the balance between time spent in tree searching, resulting in pruning, and time spent in scanning, producing the query results. A high value leads to large leaf nodes storing more objects, which in turn requires less inner nodes and thus a less deep tree. Such a structure is preferable for less selective query workloads: More work is put on scans where the comparison of the data objects with the query lead to many matches, whereas less time is spent in pruning which, for less selective queries, is not effective anyway. In contrast, a lower BB capacity results in smaller leaf nodes and a deeper tree structure, which is beneficial for highly selective queries as more time is invested in successful pruning and less in scans producing almost no matches.

Delimiter values. For a good search performance, it is crucial that delimiters allow to prune subtrees and non-relevant BB as early as possible. When being rebuilt, BB-Trees choose their delimiter dimensions in the order of the number of distinct values of a dimension, moving dimensions with a high cardinality and thus a presumably higher pruning power to the top. If a dimension has more than k different values, delimiter values are determined such that each subtree features a roughly equal number of objects. If the number of inner node levels, h, is smaller than the dimensionality of the data set, m, BB-Trees thus omit the dimensions with the smallest cardinalities in the IST. Note that this scenario is quite frequent due to the high fan out of the tree which makes the BB-Tree rather flat even for very large data sets. As an example, assume a BB-Tree over one Billion objects, a fan out of $k = 17$, a b_{max} value of 1,000, and a fill degree of 50%. Adressing the resulting two million BB requires only six IST levels. Thus, low-cardinality dimensions, which are anyway problematic in terms of pruning power, do not clutter the tree. On the other hand, if h is larger than m, we employ dimensions multiple times as delimiters in a round-robin fashion. This scenario occurs especially for data sets with a dimensionality between two and four (depending on the number of objects). A special case occurs when low-cardinality dimensions are used as delimiters (see Section 3.5).

Example. Figure 1 illustrates a BB-Tree with $k = 3$, $h = 2$ (two tree levels), and nine BB managing three-dimensional data objects (buckets three to six are not displayed). Each (regular) BB can hold up to $b_{max} = 4$ data objects. Individual data objects are identified by tids. At the first level, the shown BB-Tree splits the data space into $k = 3$ partitions according to the first dimension. All data objects having a value of three or less in this dimension are held in the left subtree. All data objects having a value of seven or less, but larger than three, in this dimension are held in the middle subtree; all other data objects can be found in the right subtree. At the next level, the data space is recursively split according to the second dimension. Note that this example uses two dimensions as delimiter, although $m = 3$. Given a fan out of $k = 3$, two tree levels are sufficient for distinguishing between nine BB. Figure 2 illustrates the linearization of the IST. We link the linearized IST with the corresponding BB by maintaining an array of pointers, where entry i references the i-th BB.

SIMD. Although processing inner nodes with Single Instruction Multiple Data (SIMD) instructions sounds promising at first glance, especially because the tree is linearized and packed into a dense array, we were not able to obtain any performance benefits

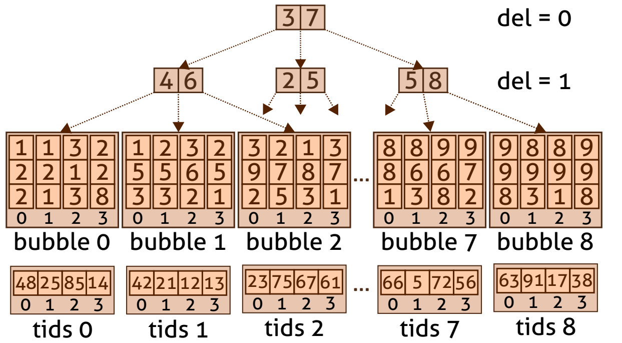


Figure 1: A BB-Tree (k = 3, b_{max} = 4) of height h = 2 managing n = 36 data objects of dimensionality m = 3; buckets three to six are omitted.

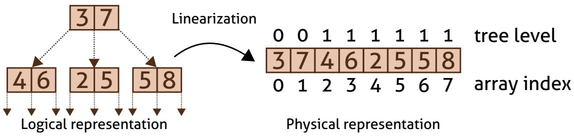


Figure 2: Linearized storage of the inner search tree.

through SIMD parallelism. Compared to a binary search, scanning inner nodes with SIMD instructions does not save many comparisons yet incurs overhead. For instance, when employing 16 delimiter values, which perfectly fit into one 64-byte cache line, a binary search requires $log_2(16) = 4$ comparisons, while a SIMD search on 256-bit registers needs two comparisons (or four comparisons if only 128-bit SIMD registers are available). These small savings are outweighed by the overhead induced by SIMD scans, especially data transfers between regular and vector registers [7], and the necessary scalar evaluation of the results of SIMD instructions.

3.2 Bubble Buckets

Until now, we described the BB-Tree as a static index structure and omitted the treatment of overflowing or underflowing leaf buckets. We lift this restriction and describe two techniques to cope with changing data, namely bubble buckets (this section) and index rebuilds (next section).

All leaf nodes are implemented as elastic bubble buckets. There exist two types of BB: A regular BB is implemented as a C++ *std::vector*, which is a dynamically growing and shrinking array, and takes inserts up to its maximum capacity b_{max}. In contrast, a super BB locally adds a further level to the tree. It consists of an inner node and a set of k regular nodes. The inner node holds a delimiter dimension and a set of delimiter values. As usual, super BB employ the dimension that has the largest number of distinct values as delimiter, and the $k-1$ delimiter values are chosen such that the data objects are as evenly distributed as possible among the k regular child BB. Super BB morph into regular BB upon underflow, and regular BB morph into super BB upon overflow.

Inserts. The complete procedure for inserting objects is as follows: We first traverse over the inner nodes to determine the bucket that is responsible for the new object. If the chosen BB is a regular BB and has free space, we insert the object and are done. If there is no free space, we morph the BB into a super BB, and insert the data object; this also happens when the chosen BB already is a super BB. To insert into a super BB, we first check whether the super BB currently contains less than $k * b_{max}$ objects. If this is the case, we determine the appropriate child BB, which must be a regular BB, and insert the object; otherwise we reorganize the index.

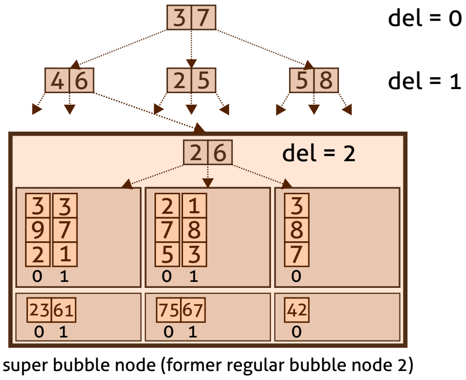


Figure 3: Assuming that a new data object (3 8 7) with tid 42 gets inserted into the BB-Tree from Figure 1, the second BB morphs into a super BB that consists of k regular nodes; dimension two is employed as delimiter.

BB can thus accommodate up to $k * b_{max}$ inserts into the same region before the index needs a rebuild. If objects are deleted during insert-heavy workloads, this period gets even longer. Within this time, the IST of the BB-Tree is stable, and the local depth is increased by at most one. However, to keep the algorithms simple we currently do not balance the size of the child nodes of a super BB, which, in theory, could lead to cases where all inserts accumulate in one child node. This would for instance happen when objects of an one-dimensional data set are inserted in a certain sort order; for such situations, other index structures are more appropriate, such as [18].

Deletes. When deleting an object, we first search the IST to determine the responsible BB and delete the object there. In the case of a regular BB, no further processing is performed. This implies that a BB-Tree may have empty leaf buckets; however, due to the dynamic size of their implementation, the memory overhead is minimal. We nevertheless rebuild the index when more than 10% of all BB are empty to get rid of superfluous inner nodes. If we delete from a child of a super BB, this bucket checks the total number of objects it contains and morphs into a regular BB in case the number is smaller than $p * b_{max}$. In the default setting, we set $p = 0.5$ to prevent pathological cases of constantly morphing BB when the b_{max}-th object is inserted and deleted iteratively.

Example. Consider again the BB-Tree from Figure 1. When we insert a new data object (3 8 7), the second bucket overflows and morphs into the super BB shown in Figure 3. Here, the super BB uses the third dimension as delimiter.

3.3 Building and Reorganizing a BB-Tree

A BB-Tree is initialized with one regular BB. After b_{max} objects have been inserted, this regular BB morphs into a super BB. With more inserts, this super BB eventually overflows, triggering a rebuild of the index. All operations, except for the very first, operate on a BB-Tree that was the result of an index rebuild.

Such a rebuild consists of four steps. First, we determine how many regular BB are needed to hold the current data, while leaving capacity for new inserts. From this number, we also derive the necessary number of levels of the IST. By default, we set the number of BB to $n/(10\% * b_{max})$ allowing each node to ingest further $90\% * b_{max}$ data objects until it overflows. This parameter may be changed if the expected workload consists of many inserts (lower value, less rebuilds, deeper tree) or few inserts (higher value, less deep tree). Second, we randomly sample $R_{samples} * n$ data objects as representatives of the whole data set. By scanning the sample data, we estimate the cardinality of each dimension. Dimensions are sorted by cardinality and assigned to the h IST levels in descending order. Third, we recursively determine the delimiter values for the inner nodes. Using the sample data, we compute an equi-depth histogram with k buckets, reflecting the distribution of the dimension values of the current level. Using that histogram, we obtain $k-1$ delimiter values such that the data are divided into k partitions of rougly equal size. In the case of a delimiter value, which occurs with a much higher frequency than the others, the derived partitions will be of unequal size. Clearly, this procedure fails for low-cardinality dimensions containing less than k distinct values. In the last step, all objects are inserted into their respective BB.

Obviously, index reorganization is an expensive operation. A random sample must be determined which is scanned multiple times, a new IST is constructed, and data objects must be moved to new locations. We chose pragmatic and fast methods for these steps, which come at certain drawbacks. First, equally splitting a subtree by one dimension is not always possible, which, in the worst case, may lead to an unbalanced BB-Tree (see Section 3.4), where subtrees at the same depth contain an unequal number of data objects. Second, we globally assign dimensions to tree levels, which again can lead to imbalances when dimensions are strongly correlated. Third, we compute the IST structure only on a sample. If the sample is small, the tree is found quickly yet might not optimally represent the data. Contrary, if the sample is large, building the tree needs more time yet probably leads to a better tree structure. We make two notes regarding these issues. First, they are shared by most other updateable MDIS. For instance, the structure of kd-trees strongly depends on the order of the insertions. The K-D-B-tree turns kd-trees into balanced search trees, but at the price of complicated and slow update operations. Second, though we cannot give formal guarantees, for the data sets we used in our evaluation, we never observed any notable imbalance. We are thus confident that unbalanced BB-Trees, which are possible in theory, remain very rare in practice.

3.4 Search Algorithms

BB-Trees focus on partial- and complete-match range queries, but also support exact-match queries.

All search queries have in common that they first exploit the linearized inner nodes to efficiently find those BB that may hold data objects relevant for the search query while pruning all others. This step is followed by sequential scans over all candidate BB to determine the data objects matching the query. Evaluation of search queries may have to follow multiple paths through the tree: Partial-match range queries must consider multiple paths whenever a node splits on a dimension which is not part of the query. Complete-match range queries have to consider multiple paths when the range covers more than one subtree. Even exact-match queries need to consider multiple paths when a low-cardinality dimension (with less than k distinct values) is used as delimiter.

Assuming the usual case, where the navigation of the IST results in only one candidate BB, exact-match queries have a complexity of $O(h * log(k) + b_{max} * m)$: They first perform h times a binary search within inner nodes and eventually scan one BB holding up to b_{max} objects, where a comparison between the query and a data object requires m value comparisons. The tree height (h) depends on the number of stored data objects, the BB capacity and the fan out: $h = log_k(n/b_{max})$. Note that the tree height is increased by one in case the search leads to a super BB. The complexity of exact-match queries is dominated by the scans of leaf nodes, calling for small values of b_{max}. On the other hand, every change of a level during IST traversal may result in a cache miss, making these operations more costly in practice.

Note that the worst-case complexity of the BB-Tree is linear in n. First, this is trivially the case when queries select all indexed objects. Second, this case occurs when less than B_{max} objects are indexed, as these are all stored in one bucket preventing pruning. Third, the worst case also applies, when BB-Trees consist of only one super BB and most data objects are inserted into the same child BB due to a inaptly-chosen delimiter dimension. However, once the super BB overflows, the data objects are distributed to multiple regular BB and the search complexity (for queries requiring only one BB) converges to the formulas given above.

For less selective range queries, scans are more attractive as more of their comparisons actually lead to matches, without any tree navigation in-between. However, determining an optimal b_{max} value would only be possible if all queries had the same, a-priori known selectivity across the entire data space, an assumption that is rather impractical. In practice, every setting of b_{max} implements an expectation on the average selectivities of queries in the future workload. In our evaluation, we will show that our default value leads to a performance that is almost on-par with MDIS specialized in exact-match queries while clearly outperforming all competitors for MDRQ.

3.5 Low-Cardinality Dimensions

We consider a dimension of a data set to have a low cardinality when its number of distinct values is smaller than k. Low-cardinality dimensions are common in real-world data sets and challenge MDIS because they make partitioning generally hard and equal partitions impossible. The problem is less severe for the BB-Tree, as it sorts delimiter dimensions by the number of distinct values, which usually keeps low-cardinality dimensions completely out of the IST. However, if a data set contains less than h dimensions and these dimensions have low cardinalities, a low-cardinality dimension will be chosen as delimiter dimension of an inner node, making it impossible to find distinct delimiter values such that the data is split into k subparts of equal size. In the worst case, where a delimiter dimension features only one distinct value, the IST loses all its pruning power and searching the BB-Tree degenerates to a sequential scan over all leaf nodes.

4 PARALLEL EVALUATION OF RANGE QUERIES

The parallel range query operator uses multiple threads to execute search queries and consists of two phases.

The first phase navigates the inner nodes with a single thread to determine the candidate BB. Traversing over the linearized inner nodes in parallel is complicated, as an optimal scheme would require solving a non-trivial load balancing problem due to different pruning effects in different subtrees. At the same

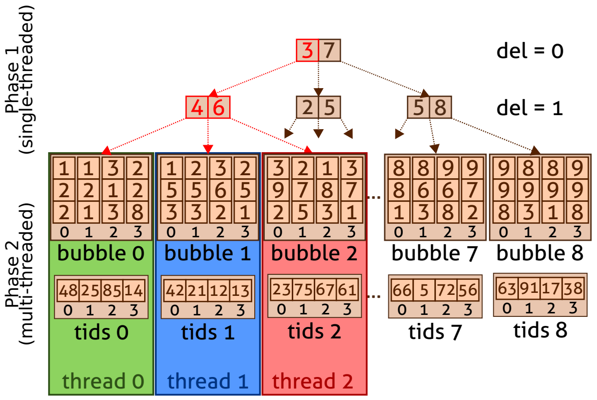


Figure 4: Parallel evaluation of an exemplary range query defined by lower boundary $[1, 0, 3]$ **and upper boundary** $[3, 7, 6]$**.**

time, there is only little to gain as the tree usually is rather flat due to its high fan out.

In the second phase, all candidate BB are scanned in parallel. Let B_{match} denote the number of candidate BB that have been determined by the first phase (super BB are considered as multiple BB). If $B_{match} \geq t$, we divide the candidate BB into t partitions of size B_{match}/t. Each partition is processed by a distinct thread using the same algorithm as in the regular, single-threaded BB-Tree. Hence, if $B_{match} = t$, we obtain the perfect degree of parallelism. If $B_{match} < t$, some threads would remain idle when sticking to the one-thread-per-bucket assignment. In this case, we assign multiple threads to single BB allowing to divide the data objects of one BB into partitions and scan these partitions in parallel.

Example. Figure 4 illustrates the parallel evaluation of a range query in a BB-Tree. The shown search query retrieves all data objects that match the lower boundary $[1, 0, 3]$ and the upper boundary $[3, 7, 6]$. The first, single-threaded phase of the query execution determines that the first three BB may hold data objects matching the range boundaries (the search path is marked in red). In the second, multi-threaded phase, these BB are searched concurrently with one CPU thread per bucket assuming that our imaginary machine features three threads ($B_{match} = t$).

5 EVALUATION

In a comprehensive evaluation, we compare the BB-Tree with state-of-the-art approaches to general-purpose indexing of multidimensional data by executing synthetic and real-world query workloads over synthetic and real-world data sets. Specifically, we aim to answer the following questions: (1) Does the performance of the BB-Tree depend on data set- or workload-specific characteristics, e. g., data dimensionality, data skew, or query selectivity (see Sections 5.4 to 5.7)? (2) How does the BB-Tree perform on mixed workloads that contain both reads and writes (see Section 5.9)? (3) What is the effect of parallelization (see Section 5.10)? (4) How efficient does the BB-Tree utilize memory space (see Section 5.11)?

5.1 Experimental Setup

Hardware. We executed all experiments on a server equipped with two Intel Xeon E5-2620 CPUs (2 GHz clock rate, 64-byte cache lines, six cores, 12 hardware threads) and 32 GB of RAM. In total, the machine features 12 cores and 24 hardware threads. Most experiments are single-threaded, some experiments investigate the parallel range query operator and therefore make use of multiple threads.

Methodology. In our evaluation, the competitors are completely kept in main memory. All data sets are inserted in random order. All experiments measure the average execution time of an operation, e. g., range query. We run experiments three times and present the arithmetic mean.

Competitors. We compare the BB-Tree with multiple approaches to general-purpose multidimensional indexing: the kd-tree [4], the PH-tree [35], the R*-tree [3], the VA-file [34] and the sequential scan [28]. Section 2 provides a brief description of the competitors; for more details we refer the interested reader to the original papers or surveys, like [10], or benchmarks, like [28]. For the R*-tree, we used an open-source, main memory implementation (https://libspatialindex.github.io/) and relied on the default configuration, but slightly adjusted the node capacities such that nodes are aligned to cache lines. For the PH-tree, which is a main-memory MDIS by design, we used a publicly available implementation shared by the authors (https://github.com/tzaeschke/phtree-1). For the kd-tree, the VA-file and the sequential scan, we used our own implementations based on the original publications, but adapted them to main-memory storage following techniques described in [28]. Most contestants, including the BB-Tree, use 32-bit floating-point values to manage dimension data. The R*-tree implementation uses 64-bit floating-point values; the PH-tree implementation uses 64-bit integer values. We evaluated the BB-Tree with $k = 17$, because $k - 1 = 16$ four-byte floating-point values fit into one cache line of the evaluation machine. Based on the observations described in Section 5.3, we empirically set $b_{max} = 2{,}500$. For reorganization, we use $R_{samples} = 10\%$ of all objects as samples to estimate the current data distribution.

Software. All software was implemented in C++ and was compiled with GCC using optimization flag *-O3*. We measured hardware performance counters with PAPI (http://icl.cs.utk.edu/papi/) and space consumption with valgrind (http://valgrind.org/). For the parallel range query operator, we used an open-source thread pool library (https://github.com/vit-vit/CTPL) to enable the reuse of POSIX threads. Our implementation of the BB-Tree is freely available (https://www2.informatik.hu-berlin.de/~sprengsz/bb-tree/).

5.2 Data Sets and Workloads

We evaluate the competitors on four data sets. Table 2 provides the number of data objects (n), the dimensionality (m), the number of distinct values per dimension (for UNIFORM and CLUST, we provide averages over all dimensions), and the raw size of each data set. We primarily use synthetic workloads. Unless noted otherwise, we generate synthetic range queries by randomly choosing two objects from the data set and, for each dimension, we use the smaller (larger) value of both objects as lower (upper) boundary. For GENOMIC, we execute a realistic workload, the Genomic Multidimensional Range Query Benchmark (GMRQB) [28], consisting of eight partial- and complete-match MDRQ templates of varying selectivity.

Data Set	n	m	Distinct Values per Dimension	Raw Size
UNIFORM	10k	5	10k (avg)	0.19MB
	100k	5	95k (avg)	1.91MB
	1M	5	632k (avg)	19.07MB
	10M	5-100	1M (avg)	190.74MB-3,814.7MB
	10M	5	4-64	190.74MB
CLUST	10k	5	10k (avg)	0.19MB
	100k	5	95k (avg)	1.91MB
	1M	5	632k (avg)	19.07MB
	10M	5	1M (avg)	190.74MB
POWER	10k	3	10k; 1k; 1k	0.11MB
	100k	3	100k; 2k; 2k	1.14MB
	1M	3	1M; 4k; 5k	11.44MB
	10M	3	10M; 6k; 8k	114.44MB
GENOMIC	10k-10M	19	1-63,883	0.72MB-724.79MB

Table 2: Data sets used in our experiments.

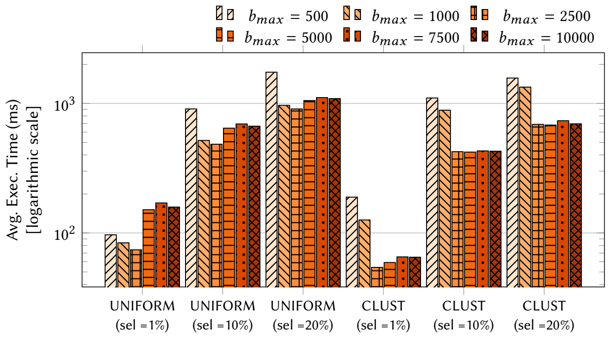


Figure 5: Performance of BB-Trees with different BB capacities (B_{max}) when executing range queries with varying selectivities (1%, 10%, and 20%) (n=10M, m=5, UNIFORM/CLUST).

Uniform Data (UNIFORM). Synthetic data facilitates experiments with arbitrary data set sizes, dimensionalities and query selectivities. We generate uniformly distributed data objects within the domain $[0, 1]$.

Clustered Data (CLUST). The five-dimensional data set CLUST features up to 20 clusters. We used a generator provided by Müller et al. [22] to generate CLUST within the domain $[0, 1]$. Within clusters, data are uniformly distributed.

Sensor Data (POWER). POWER is obtained from the DEBS 2012 challenge (http://debs.org/?p=38) and resembles real-world sensor data of hi-tech manufacturing equipment. As in previous studies [28, 33], we index three dimensions.

Genomic Data (GENOMIC). GENOMIC consists of real-world genomic variant data obtained from the 1000 Genomes Project [31]. We transformed the raw data, originally provided as text files, into 19-dimensional data objects, which can be indexed by the competitors (some attributes were provided as text and had to be converted into floating-point values). As shown in Table 2, the dimensions of GENOMIC have a highly varying cardinality, ranging from one to 63,883 distinct values. We previously defined the Genomic Multidimensional Range Query Benchmark (GMRQB) in [28], which consists of eight realistic partial- and complete-match MDRQ templates restricting between two and 19 dimensions of the data space. The templates are instantiated with concrete values obtained from the 1000 Genomes Project data and have an average selectivity between 10.76% and 0.00001%.

5.3 Impact of Bubble Bucket Capacities

The capacity of BB, as defined by b_{max}, controls the ratio between index probing (navigation of IST) and scanning (evaluation of leaf nodes) when searching in BB-Trees. While small BB put more work on index probing, large BB increase the relative time spent on scanning. As shown in previous work [28, 34], index probing is beneficial for highly selective queries and scanning is superior for less selective workloads.

We study the impact of b_{max} on the performance of range queries with varying selectivities (1%, 10%, 20%) when applied to ten million five-dimensional objects from UNIFORM and CLUST. Our goal is to find a pragmatic configuration providing a robust performance for a wide range of query selectivities and data distributions. Figure 5 shows the results.

For uniform distributions, this experiment confirms that small (large) capacities are beneficial for highly (less) selective queries. While small BB capacities are more efficient than large BB capacities for queries with an average selectivity of 1%, they become less efficient with increasing query selectivity (10% and 20%). For the selectivies considered here, BB holding up to 2,500 objects provide the best performance.

Clustered data lead to a less optimal partitioning, which lessens the pruning power of the IST and puts more work on scanning. As a result, small BB capacities become less efficient, even for small selectivities, because less BB can be pruned. Also for clustered data, BB with a maximum capacity of 2,500 objects provide either the best performance or are on a par with other configurations.

Taking the results of this experiment into account, we set $b_{max} = 2,500$ for all following experiments.

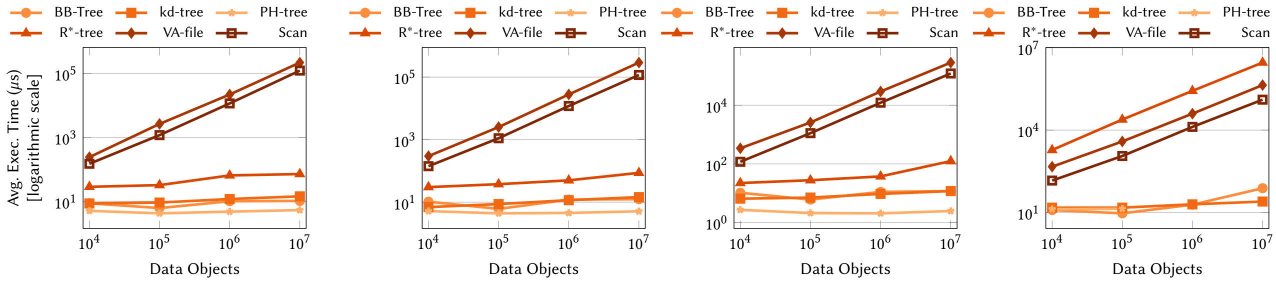
5.4 Exact-Match Queries

Figure 6 shows the average execution time of exact-match queries for the four data sets depending on the number of data objects. We execute n exact-match queries on the contestants given that n denotes the data set size. Each exact-match query retrieves a randomly-chosen, existing data object. To manage 10^4 objects, the BB-Tree does not need an IST, but employs one super BB consisting of $k = 17$ regular nodes ($b_{max} = 2,500$). In general, for exact-match queries, the performance of the BB-Tree is very similar to that of the kd-tree and the PH-tree. It clearly outperforms the R*-tree, the VA-file and the sequential scan for all data sets, often by multiple orders of magnitude. For the largest instance of GENOMIC (10^7 objects), inner nodes at the lowest tree level feature duplicate delimiter values, which require scanning multiple candidate BB and result in minor performance drops.

Typically, MDIS achieve a better exact-match query performance than sequential scans, because they can prune large parts of the data while scans need to consider all data. Although the BB-Tree needs to scan over data objects stored in BB it achieves a very competitive performance. The BB-Tree exploits the linearized inner nodes to effectively reduce the amount of data to consider for query evaluation.

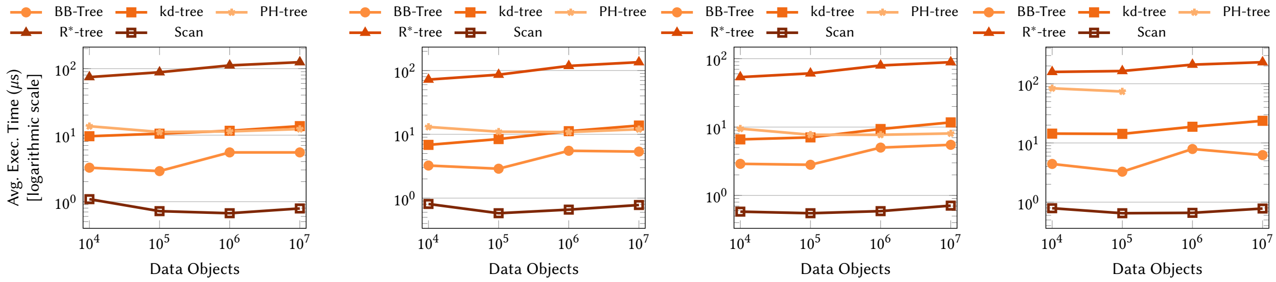
5.5 Insertions and Deletions

Figure 7 presents the average time a contestant needs to ingest a data object. The shown results include the reorganizations of the BB-Tree; in a real system, we would advise to handle rebuilds in background jobs, which strongly increases insert performance. We do not insert entire data sets at once, but load object by



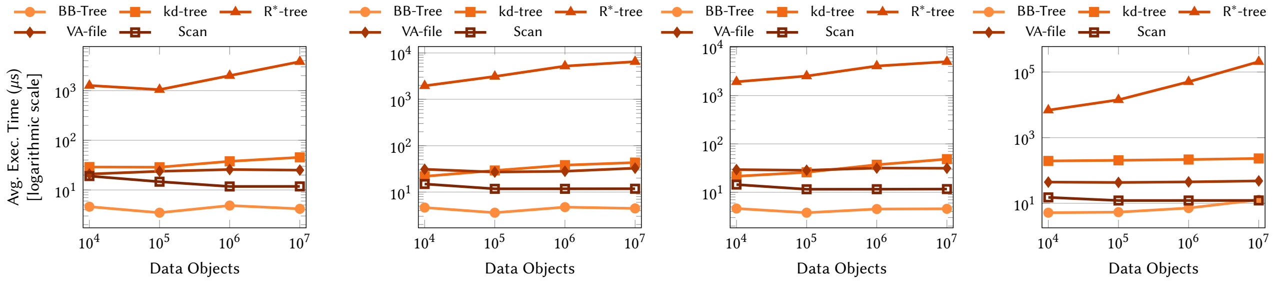
(a) UNIFORM (m=5) **(b) CLUST (m=5)** **(c) POWER (m=3)** **(d) GENOMIC (m=19)**

Figure 6: Performance of exact-match queries on synthetic and real-world data depending on the number of data objects.



(a) UNIFORM (m=5) **(b) CLUST (m=5)** **(c) POWER (m=3)** **(d) GENOMIC (m=19)**

Figure 7: Performance of insert operations on synthetic and real-world data depending on the number of data objects.



(a) UNIFORM (m=5) **(b) CLUST (m=5)** **(c) POWER (m=3)** **(d) GENOMIC (m=19)**

Figure 8: Performance of delete operations on synthetic and real-world data depending on the number of data objects.

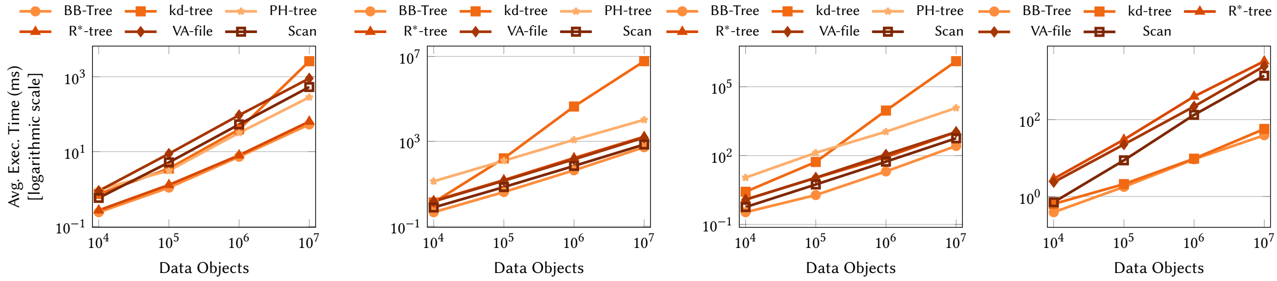
object. Thus, this experiment does not include the VA-file, which supports only bulk inserts, because it requires to know the data distribution beforehand. For instances of GENOMIC with more than 10^5 objects, the space needs of the PH-tree exceeded the available 32 GB of memory.

The scan achieves the highest insert performance, because it implements inserts by appending new data objects to a dynamic array and does not require to deal with node overflows, like the R*-tree. Notably, the BB-Tree shows a better insert performance than the kd-tree and the PH-tree and clearly outperforms the R*-tree. The concept of elastic BB effectively reduces the frequency of rebalancing operations. When dynamically inserting ten million data objects, regardless of the data set, the BB-Tree needed only three reorganizations, which took 6.55s on average (standard deviation σ = 8.75s). Smaller data sets require even less reorganizations.

Figure 8 shows the average time needed for deleting an object from the four data sets depending on data set size. The used implementation of the PH-tree did not provide a delete operator. The delete performance of the BB-Tree correlates with its exact-match query performance, because it first locates the to-be-deleted data object and then removes it from the corresponding BB. The BB-Tree outperforms all other competitors in deleting data objects, even scans, except for the largest instance of GENOMIC.

5.6 Range Queries

Figure 9 shows the average execution time of complete-match MDRQ that we generated by randomly choosing two objects from the data set. Depending on the data distribution, the obtained MDRQ objects have a varying average selectivity; UNIFORM: 0.4% (σ = 0.9%), CLUST: 19.8% (σ = 19.7%), POWER: 12.6% (σ = 13.1%), GENOMIC: 0.2% (σ = 0.2%). For CLUST, one range query may span multiple clusters, therefore average selectivities



(a) UNIFORM (m=5)

(b) CLUST (m=5)

(c) POWER (m=3)

(d) GENOMIC (m=19)

Figure 9: Performance of synthetic complete-match range queries on synthetic and real-world data depending on the number of data objects. Due to the technique used to generate MDRQ (see Section 5.2), average selectivities are as follows: UNIFORM: 0.4% (σ = 0.9%), CLUST: 19.8% (σ = 19.7%), POWER: 12.6% (σ = 13.1%), GENOMIC: 0.2% (σ = 0.2%).

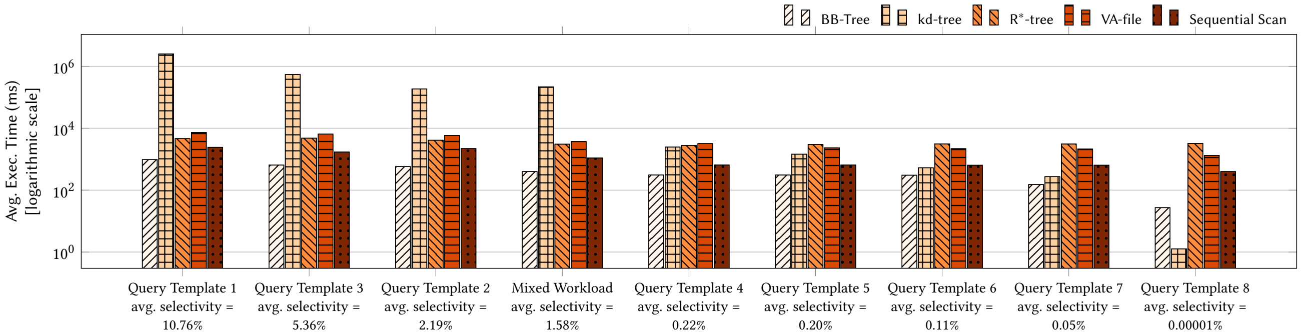


Figure 10: Performance of eight realistic MDRQ templates, including a mixed workload, from GMRQB; query templates are ordered by selectivity (n=10M, m=19, GENOMIC).

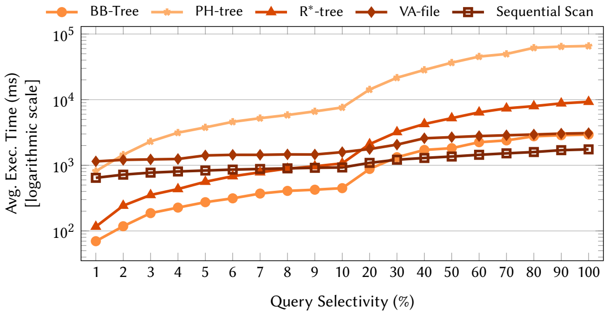


Figure 11: Performance of range queries depending on query selectivity; kd-tree is omitted as its performance decreases severely for less selective queries, strongly impairing the readability of the figure (n=10M, m=5, UNIFORM).

are higher than for UNIFORM although both data sets have an identical size and both are generated within $[0, 1]$. The BB-Tree achieves the best overall performance and outperforms the other contestants, sometimes by up to three orders of magnitude. For UNIFORM, the R*-tree shows a performance similar to that of the BB-Tree. For GENOMIC, the kd-tree performs similar to the BB-Tree. We omit the PH-tree for all range query experiments on GENOMIC, because the implementation given by the authors crashed with failing C++ assertions.

Figure 10 presents the average execution time of the GMRQB on ten million data objects from GENOMIC. Note that most query templates, except query templates 7 and 8, have their first selection predicate in the second level of the BB-tree, which means that the GMRQB workload is rather unfavorable for our index. Nonetheless, BB-Trees consistently achieve the best performance for query templates 1-7, which are partial-match MDRQ querying 5.81 dimensions on average (σ = 4.11), and a mixed workload consisting of all query templates randomly mixed together. Only for query template 8, which resembles an exact-match query as it selects a single data object on average, they are beaten by kd-trees. For data of high(er) dimensionality, such as GENOMIC, R*-trees lose their pruning power and show a worse performance than scans.

Figure 11 shows the performance of range queries on ten million objects from UNIFORM depending on query selectivity. We omit the kd-tree because, compared to the other competitors, its execution time was orders of magnitude higher for queries selecting more than 1% of the data. The BB-Tree outperforms all other MDIS regardless of the query selectivity. It also beats the scan for queries with a selectivity of up to 20%. For less selective queries, the performance of the BB-Tree remains close to that of a scan. Furthermore, the BB-Tree achieves a very high cache efficiency, almost as good as that of a sequential scan, and follows most predicted branches leading to few pipeline flushes (see Table 3).

5.7 Impact of Dimensionality

We measure the performance of exact-match and complete-match range queries on ten million data objects from UNIFORM depending on data set dimensionality. We generate complete-match range queries with an average selectivity of 1% (σ = 0.7%). With a

	BB-Tree	kd-tree	PH-tree	R*-tree	VA-file	Scan
CPU Cycles	**164M**	8,306M	1,908M	252M	2,934M	1,582M
LLC Accesses	1.0M	824M	1.2M	2.5M	1.8M	**0.5M**
LLC Misses	0.7M	0.9M	0.8M	0.5M	1.6M	**0.3M**
TLB Misses	0.3M	1.0M	0.3M	0.3M	0.2M	**0.1M**
Branch Mispr.	**0.1M**	0.7M	3M	0.2M	10M	7M

Table 3: Performance counters per range query (1% selectivity;n=10M, m=5, UNIFORM).

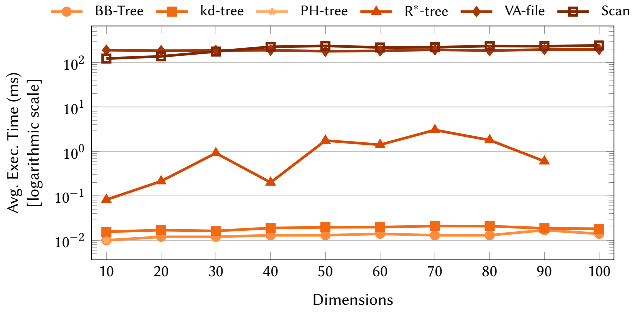


Figure 12: Performance of exact-match range queries (average selectivity = 1%, σ = 0.7%) depending on dimensionality (n=10M, UNIFORM).

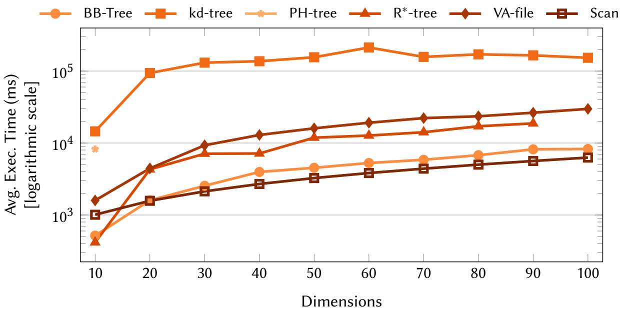


Figure 13: Performance of complete-match range queries (average selectivity = 1%, σ = 0.7%) depending on dimensionality (n=10M, UNIFORM).

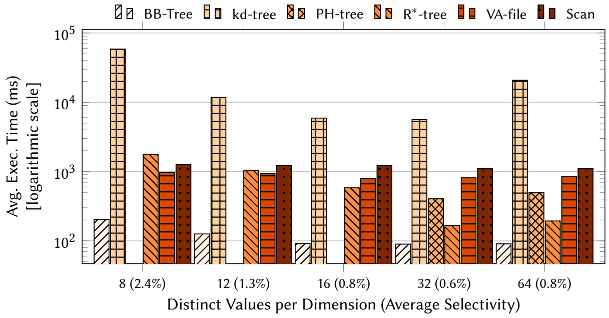


Figure 14: Performance of MDRQ with a varying selectivity depending on number of distinct values per dimension (n=10M, m=5, UNIFORM).

growing dimensionality, this results in very low single-dimension selectivities posing serious challenges to MDIS because pruning becomes less useful. For instance, when running complete-match MDRQ with an overall selectivity of 1% on 100-dimensional uniformly distributed data, where dimensions are not correlated, single-dimension selectivities are approximately 95.50%, as $0.955^{100} \approx 0.01$.

Figure 12 shows the runtimes of exact-match queries for dimensionalities between ten and 100[3]. For such workloads, all methods except the R*-tree are mostly unaffected by the dimensionality of the data space. Similarly, Figure 13 shows the runtimes of complete-match MDRQ depending on the dimensionality. All methods show a performance degradation roughly proportional to the dimensionality of the data space, starting at a dimensionality of 20, because an increasing number of dimensions has to be compared when evaluating queries. The slow-down is more pronounced for lower dimensionalities.

We also executed workloads on instances of CLUST featuring five and ten dimensions (data not shown). All competitors behave very similar as for UNIFORM: Exact-match queries are almost unaffected by the dimensionality of the data space, whereas range queries degrade noteably.

5.8 Low-Cardinality Dimensions

Low-cardinality dimensions are challenging for BB-Trees because they make it impossible to find *k* different delimiter values, which limits the pruning power of the IST. We first study this effect using range queries applied to ten million five-dimensional data objects from UNIFORM with different moderately low cardinalties for all dimensions. Results are shown in Figure 14. At these cardinalities, none of the competitors is affected severely as the differences only correspond to the different query selectivities. Note that in the cases of eight and 16 distinct values per dimension, the data space includes duplicate data objects which is not supported by the PH-tree; therefore, we omit this method in this experiment.

Next, we performed an experiment with extremely low cardinalities (between two and 12) yet used data of higher dimensionality. Figure 15 shows the performance of range queries with a selectivity of 0.00002% (σ = 0.0%), when applied to ten million 50-dimensional objects from UNIFORM. The PH-tree had to be omitted because it produced incorrect results. This experiment shows that the performance of most MDIS drops considerably with lower cardinalities, whereas scans and VA-files are much less effected. However, for such low cardinalities other index structures, like bitmaps [8], are probably a better choice anyway.

5.9 Mixed Workload

In most applications, MDIS are loaded in bulk before running large batches of search queries. Once built, inserts and deletes rarely happen. This experiment studies the contestants when running such workloads on data from GENOMIC. We use ten million data objects, of which we first insert 9,999,900[4]. Next, we run 100 inserts, 100 deletes, 2,800 exact-match queries and 7,000 range queries in random order. For inserts, we use objects, which were not bulk loaded. For exact-match queries and deletes, we randomly choose objects from the data set. This may result in queries asking for previously deleted data objects. For range queries, we

[3] Note that the space requirements of the PH-tree exceeded the available 32 GB of main memory for dimensionalities higher than ten. Similarly, the R*-tree ran out of space for 100 dimensions.

[4] For the VA-file, we insert all data objects at the beginning of the workload, because it only supports bulk inserts.

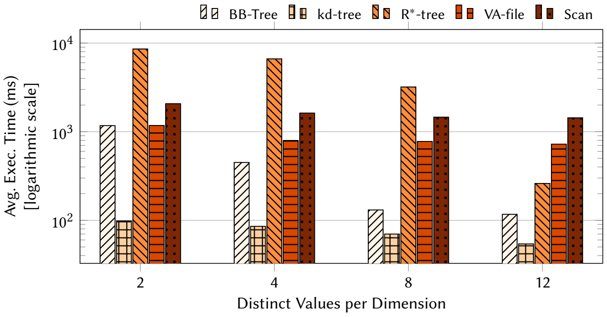


Figure 15: Performance of MDRQ with a selectivity of 0.00002% (σ = 0.0%) depending on number of distinct values per dimension; PH-tree is omitted (n=10M, m=50, UNIFORM).

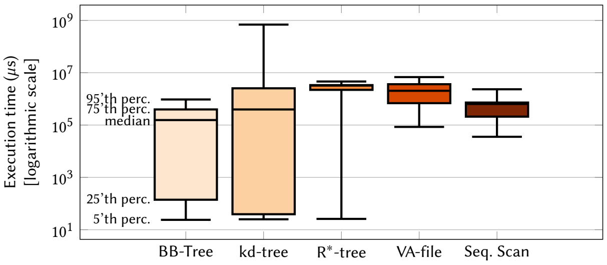


Figure 16: Execution times of single queries (inserts, deletes, exact-match and range queries) from a mixed workload in random order; bulk insert is not included; PH-tree ran out of memory (n=10M, m=19, GENOMIC).

	Bulk Insert (s)	Average/Minimum/Maximum exec. time (ms)
BB-Tree	54.7s	**262.66ms** / 0.005ms / **1,866.73ms**
kd-tree	236.7s	128,735.5ms / 0.011ms / 4,842,752ms
PH-tree		Ran out of memory.
R*-tree	2,316s	2,735.16ms / 0.008ms / 7,735.76ms
VA-file	38.7s	2,704.8ms / 0.004ms / 8,148.82ms
Seq. Scan	**7.8s**	809.83ms / **0.002ms** / 3,117.46ms

Table 4: (1) Total execution time of the bulk insert and (2) average, minimum and maximum execution time of the remaining queries of the mixed workload.

use the mixed workload from GMRQB (avg. sel.=1.58%, σ = 3.58%), which consists of all query templates randomly mixed together. Once again the PH-tree ran out of memory.

Figure 16 summarizes the runtimes. It does not include the bulk insert, because this would focus too much on inserts (ten million inserts vs. 9,900 search queries and deletions). For most contestants, insertions are the fastest operation, which would move all other operations out of the 95'th percentile. Table 4 shows the runtime of the bulk insert and summarizes the execution times of the remaining 10,000 queries. The BB-Tree achieves the highest performance in most cases. Only for the bulk insert, it is outperformed by the scan and the VA-file. The results show that the BB-Tree combines high search performance with fast inserts and deletes.

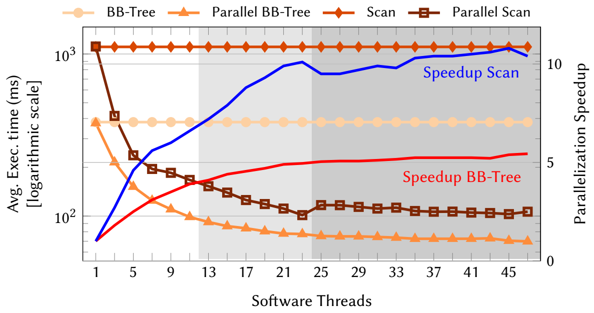


Figure 17: Performance of the mixed workload from GMRQB (avg. selectivity=1.58%) depending on the number of used software threads (n=10M, m=19, GENOMIC).

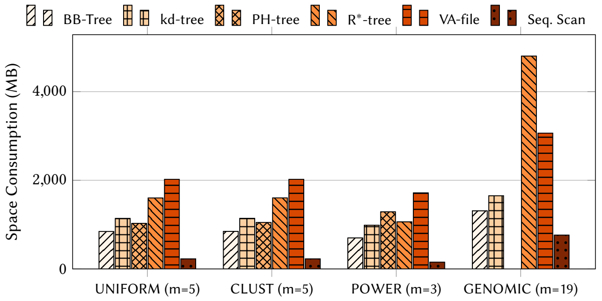


Figure 18: Space consumption of the competitors (n=10M).

5.10 Parallel Evaluation of Range Queries

Figure 17 shows the performance of the parallel range query operator when executing the mixed workload from GMRQB over ten million objects from GENOMIC depending on the number of used software threads. We compare results to a single-threaded BB-Tree, a single-threaded scan, and a parallel scan, which (1) divides the data objects into t partitions, (2) concurrently scans each partition with one thread, and (3) concatenates the results of the individual partitions. The performance of the parallel range query operator of the BB-Tree improves with the number of used threads up to a barrier established by the number of available physical cores (12 on our evaluation machine). Hyper-threading provides only few benefits for the mostly compute-bound BB-Tree, but is useful for memory-bound applications, like scans. Using moderately more threads than supported by the hardware (> 24), does neither provide benefits nor disadvantages. Scanning (up to 10.9X speed-up) benefits more from multi-threading than the BB-Tree (up to 5.5X speedup), because (a) the parallel scan leverages hyper-threading and (b) scan-based MDRQ can be completely parallelized while BB-Trees navigate the IST with a single thread. Nonetheless, BB-Trees outperform scans regardless of the number of used threads.

5.11 Space Consumption

Figure 18 shows the space consumption of the contestants when storing ten million data objects of the data sets used in the evaluation. For GENOMIC, the PH-tree required more than the available 32 GB of main memory. The BB-Tree achieves a high space efficiency, which is mainly enabled by the linearization of its inner nodes. Compared to the other MDIS, it requires the smallest index overhead over the scan.

6 CONCLUSIONS

We presented the BB-Tree as a fast and space-efficient means for storing and querying multidimensional data in main memory. It supports complete- and partial-match range queries, exact-match queries, and dynamic updates. We compared the BB-Tree with state-of-the-art MDIS using different synthetic and real-world workloads over different synthetic and real-world data sets with three to 100 dimensions. The BB-Tree beats all competitors in executing range queries up to a selectivity of 20%; for less selective queries it is only outperformed by a scan. It executes exact-match queries almost as fast as the best competitor, the PH-tree; for higher dimensionalities it even provides the best performance. The BB-Tree achieves the best insert and delete performance. We also presented a parallel variant that accelerates range queries almost linearly with the number of available CPU cores. Of course, BB-Trees are pure main-memory data structures; if data does not fit in memory, disk-based MDIS should be used like the original R*-Tree [12] or the original VA-File [34]. In future work, we intend to support nearest neighbor search and concurrent execution of search queries.

7 ACKNOWLEDGMENTS

Stefan Sprenger is funded by the Deutsche Forschungsgemeinschaft through graduate school SOAMED (GRK 1651).

REFERENCES

[1] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. 2008. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Trans. Algorithms* 4, 1 (2008), 9:1–9:30.

[2] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indices. *Acta Inf.* 1 (1972), 173–189.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *Proc. of the ACM SIGMOD International Conference on Management of Data* (1990).

[4] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* (1975).

[5] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree : An Index Structure for High-Dimensional Data. *Proc. of the 22th International Conference on Very Large Data Bases* (1996).

[6] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. 2017. Accelerating Multi-Column Selection Predicates in Main-Memory - The Elf Approach. In *33rd IEEE International Conference on Data Engineering*. 647–658.

[7] David Broneske and Martin Schäler. 2017. Single Instruction Multiple Data - Not Everything is a Nail for this Hammer. In *FADS@VLDB*.

[8] Chee Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *Proc. of the ACM SIGMOD International Conference on Management of Data*. 355–366.

[9] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4 (1974), 1–9.

[10] Volker Gaede and Oliver Günther. 1998. Multidimensional Access Methods. *ACM Comput. Surv.* 30, 2 (1998), 170–231.

[11] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Min. Knowl. Discov.* 1, 1 (1997), 29–53.

[12] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD, Proc. of Annual Meeting*. 47–57.

[13] Jörg Hakenberg, Wei-Yi Cheng, Philippe E. Thomas, Ying-Chih Wang, Andrew V. Uzilov, and Rong Chen. 2016. Integrating 400 million variants from 80,000 human samples with extensive annotations: towards a knowledge base to analyze disease cohorts. *BMC Bioinformatics* 17 (2016), 24.

[14] Kothuri Venkata Ravi Kanth, Siva Ravada, and Daniel Abugov. 2002. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. *Proc. of the ACM SIGMOD International Conference on Management of Data* (2002).

[15] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. *Proc. of the ACM SIGMOD International Conference on Management of Data* (2010).

[16] Kihong Kim, Sang Kyun Cha, and Keunjoo Kwon. 2001. Optimizing Multidimensional Index Trees for Main Memory Access. In *Proc. of the ACM SIGMOD International Conference on Management of Data*. 139–150.

[17] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. 2012. KISS-Tree: smart latch-free in-memory indexing on modern architectures. *Proc. of the Eighth International Workshop on Data Management on New Hardware* (2012).

[18] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. *29th IEEE International Conference on Data Engineering* (2013).

[19] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. *29th IEEE International Conference on Data Engineering* (2013).

[20] Xin Li, Young-Jin Kim, Ramesh Govindan, and Wei Hong. 2003. Multidimensional range queries in sensor networks. In *Proc. of the 1st International Conference on Embedded Networked Sensor Systems*. 63–75.

[21] Astrid Lievre, Jean-Baptiste Bachet, Delphine Le Corre, Valerie Boige, Bruno Landi, Jean-François Emile, Jean-François Côté, Gorana Tomasic, Christophe Penna, Michel Ducreux, et al. 2006. KRAS mutation status is predictive of response to cetuximab therapy in colorectal cancer. *Cancer research* 66, 8 (2006), 3992–3995.

[22] Emmanuel Müller, Stephan Günnemann, Ira Assent, and Thomas Seidl. 2009. Evaluating Clustering in Subspace Projections of High Dimensional Data. *PVLDB* 2, 1 (2009), 1270–1281.

[23] Hasso Plattner. 2009. A common database approach for OLTP and OLAP using an in-memory column database. *Proc. of the ACM SIGMOD International Conference on Management of Data* (2009).

[24] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2018. Theoretically Optimal and Empirically Efficient R-trees with Strong Parallelizability. *PVLDB* 11, 5 (2018), 621–634.

[25] John T. Robinson. 1981. The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes. *Proc. of the ACM SIGMOD International Conference on Management of Data* (1981).

[26] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures.* Morgan Kaufmann.

[27] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2009. k-ary search on modern processors. *Proc. of the Fifth International Workshop on Data Management on New Hardware* (2009).

[28] Stefan Sprenger, Patrick Schäfer, and Ulf Leser. 2018. Multidimensional range queries on modern hardware. *Proc. of the 30th International Conference on Scientific and Statistical Database Management* (2018).

[29] Stefan Sprenger, Patrick Schäfer, and Ulf Leser. 2019. BB-Tree: A practical and efficient main-memory index structure for multidimensional workloads. *35th IEEE International Conference on Data Engineering* (2019).

[30] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. 2016. Cache-Sensitive Skip List: Efficient Range Queries on Modern CPUs. *4th International Workshop on In-Memory Data Management and Analytics* (2016).

[31] The 1000 Genomes Project Consortium. 2015. A global reference for human genetic variation. *Nature* 526, 7571 (2015), 68–74.

[32] Sheng Wang, David Maier, and Beng Chin Ooi. 2014. Lightweight Indexing of Observational Data in Log-Structured Storage. *PVLDB* 7, 7 (2014), 529–540.

[33] Sheng Wang, David Maier, and Beng Chin Ooi. 2016. Fast and Adaptive Indexing of Multi-Dimensional Observational Data. *PVLDB* 9, 14 (2016), 1683–1694.

[34] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *Proc. of the 24rd International Conference on Very Large Data Bases* (1998).

[35] Tilmann Zäschke, Christoph Zimmerli, and Moira C. Norrie. 2014. The PH-tree: a space-efficient storage structure and multi-dimensional index. *Proc. of the ACM SIGMOD International Conference on Management of Data* (2014).