BB-Tree: A main-memory index structure for static multidimensional workloads

Stefan Sprenger
Department of Computer Science
Humboldt-Universität zu Berlin
sprengsz@informatik.hu-berlin.de

Patrick Schäfer
Department of Computer Science
Humboldt-Universität zu Berlin
schaefpa@informatik.hu-berlin.de

Ulf Leser
Department of Computer Science
Humboldt-Universität zu Berlin
leser@informatik.hu-berlin.de

***Abstract*—We present the BB-Tree, a fast and space-efficient index structure for processing multidimensional workloads in main memory. It 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, with occasional re-organizations when data changes. To reduce the frequency of re-organizations, the BB-Tree introduces a novel architecture for leaf nodes, called bubble buckets, which automatically morphs between different representations based on their fill degree and are thus able to buffer large numbers of insertions 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 PH-tree using workloads over real and synthetic data. The BB-Tree is the fastest index for range queries up to a selectivity of 20%, and achieves an exact-match query performance similar to that of the best point access method, and is the most space-efficient index structure.**

I. Introduction

Searching multidimensional data can be sped up by using multidimensional index structures (MDIS). In this work we focus on *multidimensional range queries (MDRQ)* over all (complete-match) or a subset of dimensions (partial-match). MDIS are different from one-dimensional index structures as they cannot exploit a natural sort order in the data. Especially partial-match queries require MDIS to treat all dimensions equally, which is typically achieved by building and maintaining some, often hierarchical, structure on top of the data [10]. Navigation of such a structure necessitates inefficient random access patterns, which quickly leads to the scan outperforming an MDIS when queries are less selective, irrespective of whether data is held on disk [14], or in main memory [12]. Research thus aims at creating an MDIS that can efficiently support exact-match and range queries, has low memory overhead, performs gracefully in read/write workloads, is robust against the data dimensionality, and is faster than scans.

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 and buckets overflow, the structure is rebuilt to achieve a beneficial balance regarding the depths of leafs. 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 efficiently handle fluctuating bucket fill degrees and significantly reduce the frequency of index rebuilds. BB 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: Overflowing regular BB let them morph into super BB. This leaves the rest of the BB-Tree unchanged. Since overflows create k new leaf buckets, a BB can cater a large number of inserts. Eventually, the tree is rebuilt when a super BB overflows. In workloads with hammered inserts, i.e., series of insertions into the same small region, BB help to significantly reduce rebuilds and greatly improve the write performance with minimal performance impact, at the cost of a slightly deeper tree. BB keep the inner search tree (IST) stable over long periods of data changes, which enables the adaptation of the inner nodes to cache lines. The number of delimiter values is adapted to the cache line size, to improve utilization. Furthermore, we store the inner nodes in a flat and static array to avoid pointer chasing, decrease random accesses, and reduce cache misses. Typically, such an optimization either makes the index completely static [6], [11] or requires delta stores [7], [9]. In contrast, BB-Trees manage large numbers of changes in-place without rebuilds. Also, eliminating pointers improves space efficiency. We compared the BB-Tree to sequential scans, and four MDIS [2], [4], [14], [15]. The BB-Tree's performance is virtually unaffected by the data dimensionality. It beats all competitors for range queries up to a selectivity of 20%, and has the best space efficiency; for less selective queries it is only beaten by a scan.

II. Related Work

MDIS have been researched for decades [3]. The kd-tree [2] 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 DMBS, e. g., PostgreSQL. The Vector Approximation-

file (VA-file) [14] 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. Many approaches were developed for disk-based data storage, but can be adapted to main-memory settings [12]. The PH-tree [15] is a recent main-memory MDIS, which integrates the concepts of PATRICIA-tries and hypercubes. The R-tree [4] is probably the most prominent method for handling spatially extended objects, and is frequently used for storing point objects [5]. It uses minimum bounding rectangles (MBR) to represent all objects of a certain subtree. MBRs are used for pruning. The R*-tree [1] improves partitioning by aggressively reinserting data objects leading to a more efficient search performance, employed by several DBMS to manage spatial data, e. g., SQLite.

III. 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. 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 can digest a large number of insertions without hurting the tree balance considerably. Eventually, the BB-Tree must be rebuilt.

Data Organization. A BB-Tree consists of two components: A k-ary search tree and a set of BB. Inner nodes of the IST recursively split the data space into k disjoint subsets according to a delimiter dimension and $k-1$ values. Data are kept in regular BB, which hold up to b_{max} m-dimensional data objects, and can dynamically expand to cope with a larger number of objects. When queried, inner nodes are used to reduce the data space. Once, all irrelevant subtrees have been pruned, the remaining BB are scanned to determine the 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 k, the BB-Tree maps inner nodes to the cache line size. (2) A single dense array makes pointers superfluous. Array indexes of child nodes are calculated in constant time using the tree level and the fan out k. The BB-Tree linearizes inner nodes in a breadth-first order, reducing memory pressure while increasing cache efficiency. (3) Binary search efficiently searches a specific delimiter value.

Leaf nodes. All data objects are stored in BB, each has a capacity of b_{max} objects that determines the balance between time spent in pruning the IST, and the time spent in leaf scanning. A large b_{max} results in large leaf nodes, less inner nodes, and low tree depth, preferable for lowly selective queries. In contrast, a lower capacity results in smaller leaf nodes and a deeper tree structure, beneficial for highly selective queries.

Delimiter values. Upon rebuilt, BB-Trees choose their delimiter dimensions in the order of the dimension's cardinality, moving high cardinality dimensions, and thus a presumably higher pruning power, to the top. If a dimension has more than

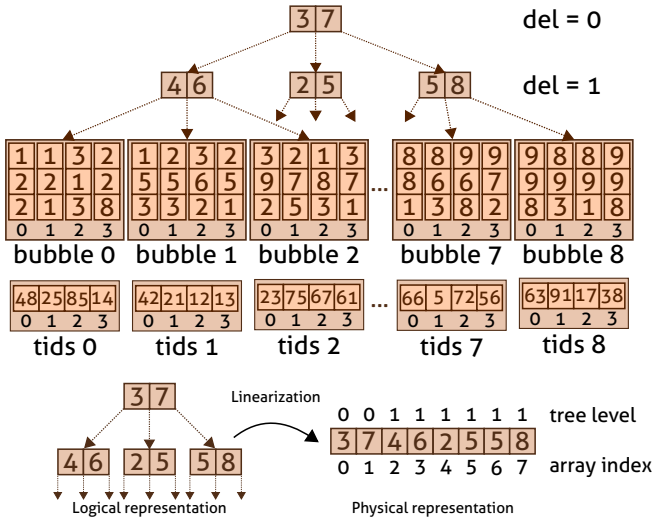


Fig. 1: A BB-Tree ($k = 3$, $b_{max} = 4$) of height $h = 2$ managing $n = 36$ data objects of dimensionality $m = 3$. The IST is linearized as a single immutable array.

k distinct values, delimiters 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 m of the data set, BB-Trees omit the dimensions with the smallest cardinalities in the IST. This scenario is quite frequent due to the high fanout of the tree, resulting in a rather flat BB-Tree for very large data sets. If h is larger than m, we employ dimensions multiple times as delimiters in a round-robin fashion. Figure 1 illustrates a BB-Tree with two levels, and nine BB storing 3d objects.

Bubble Buckets. We describe two techniques to cope with changing data, namely BB and index rebuilds. All leaf nodes are implemented as elastic BB. 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 delimiter values. Super BB employ the delimiter dimension based on the largest number of distinct values, and these $k-1$ delimiter values are chosen to evenly distributed data objects among the k regular child BB. Regular BB morph into super BB upon overflow.

Inserts. The complete procedure for inserting objects is as follows: We first traverse over the IST to determine the bucket that is responsible. If the chosen BB is a regular BB and has free space, we simply insert the object. If not, we morph the regular BB into a super BB, and insert the data object. To insert into a super BB, we check if it contains less than $k * b_{max}$ objects, determine the regular (child) BB, and insert the object; if all fails, we reorganize the index.

Building and Reorganizing a BB-Tree. A BB-Tree initially consists of one regular BB. After b_{max} objects have been inserted, this regular BB morphs into a super BB. Eventually

Data Set	n	m	Distinct per m	Raw Size
UNIFORM	10k-10M	5 to 100	10k-1M	0.19-190.74MB
CLUST	10k-10M	5	10k-1M	0.19-3.8GB
POWER	10k-10M	3	1k-10M	0.11-114.44 MB

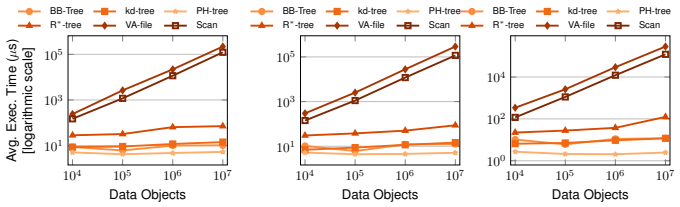
TABLE I: Real and synthetic data sets used.

the super BB overflows, triggering a rebuild. A rebuild consists of four steps. First, we compute how many regular BB are required, while leaving capacity for new inserts, and 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. Second, we randomly sample $R_{samples}\%$ representatives of the whole data set, and 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 of roughly equal size, an estimate of the value distribution of the current level, and obtain $k-1$ delimiter values. Finally, objects are inserted into their respective BB.

Search Algorithms. BB-Trees are designed for partial- and complete-match range, but also support exact-match queries. All search queries first exploit the linearized inner nodes to efficiently find BB that may hold relevant data objects while pruning all others. This is followed by sequential scans over all candidate BBs to determine the matching data objects. Evaluation of search queries potentially follows multiple paths through the tree.

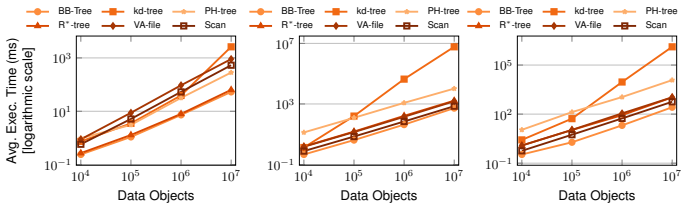
IV. Evaluation

We compare the BB-Tree with the state of the art by executing workloads over synthetic and real-world data sets. We executed all experiments on a server with two Intel Xeon E5-2620 CPUs (each 2 GHz, 64-byte cache lines, six cores, 12 hardware threads) and 32 GB RAM. All experiment are single-threaded. All competitors are completely kept in main memory. All data sets are inserted in random order. Experiments run three times and we present the arithmetic mean. We compare the BB-Tree to general-purpose MDIS: the kd-tree [2], the PH-tree [15], the R*-tree [1], the VA-file [14] and the sequential scan [12]. For the R*-tree, we used an open-source, main memory implementation (https://libspatialindex.github.io/). For the main-memory PH-tree, we used a public implementation (https://github.com/tzaeschke/phtree-1). For the kd-tree, the VA-file and the sequential scan, we used our own main-memory implementations [12]. We evaluated the BB-Tree with $k = 17$, as $k - 1 = 16$ four-byte floats fit into one cache line, and empirically set $b_{max} = 2,500$. The BB-Tree implementation is freely available (https://hu.berlin/bbtree). We evaluate on three data sets Table I. Uniform Data (UNIFORM) is synthetic and facilitates experiments with arbitrary data set sizes (n), dimensionalities (m) and query selectivities. Clustered Data (CLUST) features five-dimensional data set in up to 20 clusters



(a) UNIFORM ($m = 5$) (b) CLUST (m=5) (c) POWER (m=3)

Fig. 2: Performance of exact-match queries.



(a) UNIFORM ($m = 5$) (b) CLUST (m=5) (c) POWER (m=3)

Fig. 3: Performance of synthetic complete-match range queries.

generated using [8] . Real world Sensor Data (POWER) is from the DEBS 2012 challenge (http://debs.org/?p=38).

Exact-Match and Range Queries Figure 2 shows the average execution time of n exact-match MDRQ given n objects. Each exact-match query retrieves a randomly-chosen, existing data object. To store 10^4 objects, the BB-Tree employs just one super BB with $k = 17$ regular nodes ($b_{max} = 2,500$). 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. Although the BB-Tree scans the BB, it is very competitive, as it effectively prunes the data using the IST.

Figure 3 shows the average execution time of complete-match MDRQ, generated by using two objects from the data set as upper and lower bounds. The obtained MDRQ have varying average selectivity; UNIFORM: 0.4% ($\sigma = 0.9\%$), CLUST: 19.8% ($\sigma = 19.7\%$), POWER: 12.6% ($\sigma = 13.1\%$). For CLUST, one range query may span multiple clusters, therefore average selectivities are higher than for UNIFORM, although both data sets have identical generation properties. The BB-Tree has the overall best performance, outperforming the other contestants, by up to three orders of magnitude. For UNIFORM, the R*-tree's performance is similar to that of the BB-Tree. Figure 4 shows the performance of range queries on ten million objects from UNIFORM, depending on query selectivity. We do not plot the kd-tree as its query execution time was orders of magnitude higher when selecting more than 1% of the data. The BB-Tree outperforms all other MDIS regardless of the query selectivity. It beats the scan for queries with a selectivity of up to 20%, and for less selective queries remains close to that of a scan. The BB-Tree has a high cache

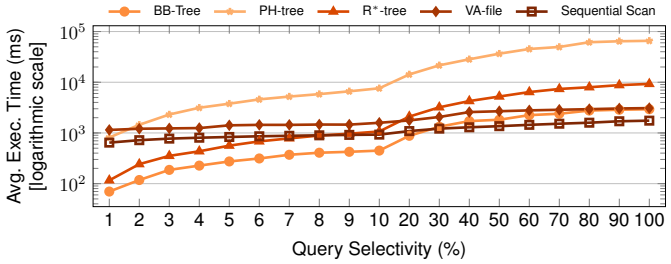
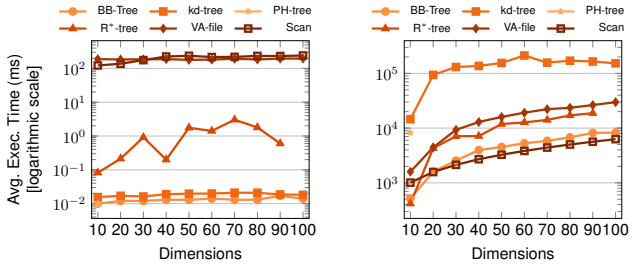


Fig. 4: Performance of range queries depending on query selectivity; kd-tree is omitted as its performance decreases severely for less selective queries (n=10M, m=5, UNIFORM).



(a) Exact-Match Queries (b) Range Queries

Fig. 5: Performance of exact-match and complete-match range queries (average selectivity = 1%, σ = 0.7%) depending on dimensionality (n=10M, UNIFORM).

efficiency, similar to a sequential scan, as it follows most predicted branches, leading to few pipeline flushes.

Impact of Dimensionality. We measured 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 growing dimensionality very low single-dimension selectivities pose serious challenges to MDIS, as pruning becomes less useful. Figure 5 shows the runtimes for m between ten and 100. The PH-tree ran out of memory for m higher than ten. Likewise, the R*-tree ran out of memory for $m \geq 100$ dimensions. For exact-match queries, all methods except the R*-tree are mostly unaffected by dimensionality. For MDRQ, all methods behave similarly and show a degradation roughly proportional to m, whereas slow-down is more pronounced for lower m. Scans become slower with increasing m as more delimiters must be compared. On a CLUST workload (not shown) with $m = 5$ vs $m = 10$ the B-Tree behaves similar as for UNIFORM.

Space Consumption. Figure 6 shows the space consumption of the contestants for ten million data objects. 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.

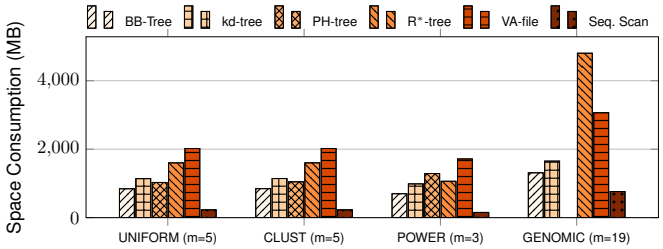


Fig. 6: Space consumption of the competitors (n=10M).

V. 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.

VI. Acknowledgments

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

References

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.
[2] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 1975.
[3] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.
[4] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD, Proc. of Annual Meeting*, pages 47–57, 1984.
[5] K. V. R. Kanth, S. Ravada, and D. Abugov. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. *SIGMOD*, 2002.
[6] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. *SIGMOD*, 2010.
[7] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. *ICDE*, 2013.
[8] E. Müller, S. Günnemann, I. Assent, and T. Seidl. Evaluating Clustering in Subspace Projections of High Dimensional Data. *PVLDB*, 2(1):1270–1281, 2009.
[9] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. *SIGMOD*, 2009.
[10] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.
[11] B. Schlegel, R. Gemulla, and W. Lehner. k-ary search on modern processors. *DaMoN*, 2009.
[12] S. Sprenger, P. Schäfer, and U. Leser. Multidimensional Range Queries on Modern Hardware. *SSDBM*, 2018.
[13] S. Wang, D. Maier, and B. C. Ooi. Fast and Adaptive Indexing of Multi-Dimensional Observational Data. *PVLDB*, 9(14):1683–1694, 2016.
[14] R. Weber, H. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *VLDB*, 1998.
[15] T. Zäschke, C. Zimmerli, and M. C. Norrie. The PH-tree: a space-efficient storage structure and multi-dimensional index. *SIGMOD*, 2014.