Tree Data Structures and Efficient Indexing Techniques for Big Data Management: A Comprehensive Study

Dimitrios Samoladas
Department of Computer Engineering and Informatics
University of Patras
Patras, Rion, Greece
samoladas@ceid.upatras.gr

Christos Karras
Department of Computer Engineering and Informatics
University of Patras
Patras, Rion, Greece
c.karras@ceid.upatras.gr

Aristeidis Karras
Department of Computer Engineering and Informatics
University of Patras
Patras, Rion, Greece
akarras@ceid.upatras.gr

Leonidas Theodorakopoulos
Department of Management Science and Technology
University of Patras
Patras, Koukouli, Greece
theodleo@upatras.gr

Spyros Sioutas
Department of Computer Engineering and Informatics
University of Patras
Patras, Rion, Greece
sioutas@ceid.upatras.gr

ABSTRACT

In the modern era where data is produced from multivariate sources, there is an urge to handle such data in an efficient yet effective manner. Therefore, applications that necessitate such capabilities shall make use of data structures and indexing mechanisms that can perform fast index operations along with low complexity as per insertion, deletion, and search. In this work, we survey B+ Tree, QuadTree, kD Tree, R Tree, and others along with efficient indexing techniques for big data management in order to provide a generic overview of the field to readers. Ultimately, we provide some indexing experiments as per insert operations and response times.

CCS CONCEPTS

• **Information systems** → **Data structures**.

KEYWORDS

Data Structures, B+ Tree, kD Tree, QuadTree, Indexes, Big Data Management

ACM Reference Format:
Dimitrios Samoladas, Christos Karras, Aristeidis Karras, Leonidas Theodorakopoulos, and Spyros Sioutas. 2022. Tree Data Structures and Efficient Indexing Techniques for Big Data Management: A Comprehensive Study. In *26th Pan-Hellenic Conference on Informatics (PCI 2022), November 25–27, 2022, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3575879.3575977

PCI 2022, November 25–27, 2022, Athens, Greece

ACM ISBN 978-1-4503-9854-1/22/11…$15.00
https://doi.org/10.1145/3575879.3575977

1 INTRODUCTION

In the past decade, there has been a significant increase in the amount of information and data that is being produced. As a result of this matter, there has been an increasing interest in developing high-performance systems that are able to provide responses to queries across terabytes of data in a matter of seconds.

Furthermore, efficient indexing and searching techniques are currently gaining a lot of attention and they should be considered a priority for big data analysis. Such types of solutions have the potential to provide consumers with insightful information about their data. This can also be applied in economics [1, 34] as well as in query expansion schemes [21]. However, in order to successfully retrieve and manage such information in terms of index size and search time, optimization of indexing methods is necessary, which is a process that may be rather challenging to carry out.

In this work, we survey emerging tree data structures and we present some index mechanisms along with the centralized and distributed variants of specific trees. The next section provides an extensive overview of such data structures in detail.

2 OVERVIEW

In this section, we will discuss about indexes and their utility in the database world. The indexes that we will talk about are B+ Tree, QuadTree, kD Tree and we will also mention their distributed implementations according to third-party works and experiments on them.

But first, let us initiate with some basic concepts of indexes. Indexes are structures that arose out of the need to quickly look up information stored in databases. Conventionally, if we wanted to search for all records that belong to a range or generally satisfy a criterion, we would have to access the records one by one and check if they meet the conditions. For a database with N items, this would require $O(N)$ time, which for the huge databases required today, as we have seen due to Big Data, is prohibitive for any undertaking.

Therefore, an index is any data structure that can improve search time. To achieve this improvement it sacrifices space and extra writes to storage to maintain its structure. There is a wide variety

of indexes that satisfy different needs of a wide variety of data, such as spatial, temporal, text, multidimensional data, etc. Choosing the right index for the task we want to perform is a major part of the whole process since it can lead to time complexities for search from $O(logN)$ to $O(1)$.

2.1 B+ Tree

B^+ Tree is a tree version of the B-tree and an extension of the binary tree in essence. The primary distinction between B and B^+ trees is that B^+ trees are leaf-oriented. Thus, rather than containing key-value pairs, internal nodes only hold keys, whereas data is stored on the leaves. The node of a B^+ tree may contain up to n indices, but no less than $[n/2]$ indices, and $n-1$ keys. The root is an exception because, regardless of n, it may have at least two children. The keys of each node are sorted, and pointers are interpolated between them. In other words, if we have keys $K_1, K_2 \ldots, K_{n-1}$ and indices $P_1, P_2, \ldots, P_n$, then the index in position I corresponds to child nodes with keys less than K_i and higher than or equal to K_{i-1}. Figure 1 is an example of the structure that we have explained. To hold N keys, the B^+ tree requires $O(N)$ storage capacity.

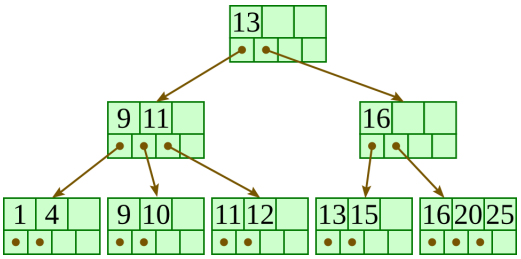

Figure 1: Representation of a $B+$ tree with $n = 4$.

B^+ trees are balanced. That is, the distance of the root from any leaf is the same for all leaves. This property makes the B^+ tree reliable since any change occurring, the tree makes sure it is always balanced which keeps its performance constant in insertions, deletions and searches.

2.1.1 Insert Operations. To insert a new element into the tree, we begin at the root and compare the key to be inserted with the keys already existing in the node. We choose the proper index based on what the key belongs to. We repeat the same procedure in a recursive manner till we reach the sheet. If the leaf has room for extra data (that is, there are less than $n-1$ keys), the new key is added. If the sheet is full, tearing will occur.

When splitting, a new leaf is created, half of the old leaf's keys are transferred to it, and the smallest key of the new leaf is transferred to the parent. Despite the fact that the parent is already full, we have merely separated once more by pushing up the middle key instead of the minor one. This procedure is repeated until a parent node does not need splitting. If we reach the root and further splitting is required, we generate a new root with a key and two pointers. The temporal complexity of inserting a new key is $O(logbN)$, where N is the number of keys in the tree and b is the capacity of the node in children.

2.1.2 Range Search. For range search, B^+ trees are modified such that their leaves are connected serially by pointers, similar to a linked list. To locate the data that falls within the search range, we begin with a straightforward search for the smallest key that falls within the search range. This search is conducted by locating the leaf that the left end of the search range corresponds to. Then, after the keys have been sorted, we identify the first key that is larger than the left end of the search range and proceed serially to the right, comparing each key to the right end of the search range. Since the leaves are connected by pointers, we do not need to retrace to access the next leaf but may instead utilise the pointer it has. Range-based search has a complexity of $O(logbN + m)$, as $logbN$ is required to locate the leftmost key and m is the number of keys retrieved.

2.1.3 Distributed implementation of B+ Tree index. Initially, we have the creation of the tree. The tree as a whole will be stored in an HBase table where each row of it represents a node of the tree. Then, we have an arbitrary ROWKEY and as columns, we have information concerning the node such as the keys that show the following nodes the range of keys covered by the specific node and in the case of the leaves the data values.

Using a MapReduce task, the index was created and stored in HBase. It receives the data of a dataset from the Mapper and turns it into an appropriate key-value format. After the records have been created, they are passed to a custom partitioner in order to divide the data uniformly throughout the reducers. The default MapReduce hash partitioner is thus not utilised. The uniform distribution of data in the reducers is accomplished by constructing intervals or chunks, as they are referred to in the Partitioner, by subtracting the smallest key from the biggest key in the dataset and dividing the resulting value by the number of reducers. The preceding result of the operation returns the interval that determines the endpoints of each interval. When the records reach the partitioner, their key is determined and they are forwarded to the appropriate reducer.

In the reduction step, local $B+$ trees are generated, which are subsequently connected to form the global tree. There are two nodes inside the reducer, the current and the next. Each pair that enters the reducer is added to the leaf that is currently being created. If the leaf is already full, a new leaf is produced and the entered pair is added to it. The leaf is then inserted into a buffer. When the buffer is filled, the data (the newly produced HBase leaves) are written. A final function is responsible for constructing the remainder of the tree from the already-made leaves. The function ends when a node with fewer children than the tree's order is encountered. This function constructs the tree by recursively using the process outlined for the leaves.

2.2 QuadTree

QuadTree [30] is a reasonably basic geographical data indexing approach. Each node in a QuadTree represents a box that covers a portion of the area being indexed, with the root node representing the whole region. Each node is either a leaf, which includes one or more points in the space and no children, or an internal node, which has precisely four children, one for each quadrant resulting from splitting the space of the node by two vertical axes, as seen in Figure 2. The name of the tree is derived from the term quadrants.

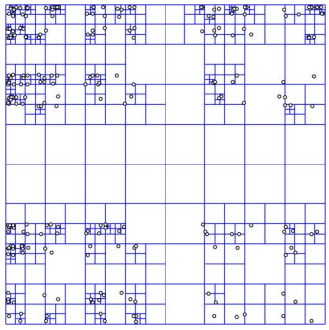

Figure 2: Representation of how space is split by a QuadTree.

2.2.1 Insert Operation. Inserting data into QuadTree is pretty concise. We begin at the root and identify the quadrant to which the desired location belongs. As we travel further into the tree, we repeat the same procedure for each node until we reach a leaf node. Once the leaf is reached, the point is added to the list of remaining points on the leaf. If this list exceeds a predefined maximum number of components, the original single node without quadrants will be divided into quadrants.

These quadrants become new leaf nodes, whereas the node that was split becomes an internal node. The components of the former node are then relocated to their respective quadrants. After inserting many items, the resulting tree structure resembles Figure 3.

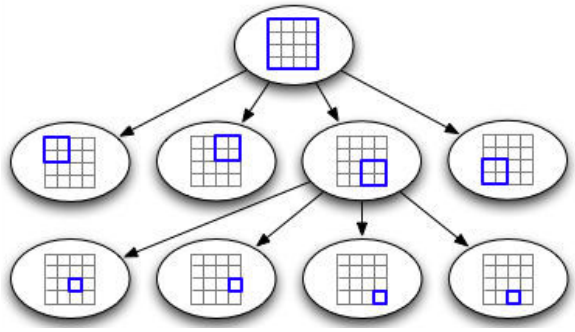

Figure 3: The internal structure of QuadTree.

2.2.2 Range Search. To locate the points inside an interval, we begin with the root. We evaluate each child node to see whether it intersects the desired space region. If so, we go to the next child node and repeat the procedure until we reach a leaf. As soon as we reach the leaf, we examine each of its components and return it if it falls inside the search region.

In a QuadTree, the search complexity (range query) is $(n + h)$, where n is the number of nodes in the tree and h is its height. This intricacy is a consequence of the variety of the search space and the structure of the tree (the order in which the data is entered affects the creation of the tree). If we have a lengthy query that returns essentially all of the tree's nodes, then all should be returned, thus we have $\Theta(n)$. If we have a short query that returns the data of a single subtree, it is sufficient to traverse all of the levels of the subtree; hence, we have $\Theta(h)$. So in total, we have $\Theta(n + h)$.

2.3 kD Tree

The kD Tree is a tree structure used to represent points in k-dimensional space. It shares similarities with QuadTree, but its implementation is different. Each internal node in the structure divides the space into two parts. The right child of the node represents the right part and the left child represents the left part. The split does not necessarily takes place in the middle of the space as in QuadTree, but at some point chosen so that as much as possible there is an equal distribution of the remaining points of the space in the two parts. This of course requires knowing all the points from the start and is not possible in serial point input. The decomposition we have described is shown in Figure 4.

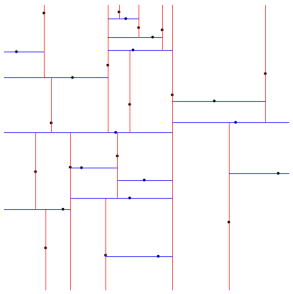

Figure 4: Representation of splits of a two-2D kD Tree.

As shown in Figure 4, each node of the tree corresponds to a point in space. The splitting from level to level is done in terms of axes alternately. The whole process is described in the next section where we analyze the Insert operation.

2.3.1 Insert Operation. Each process of input splits the space. By checking the associated point coordinates from the root, we identify the node to which the point belongs. Since we are at the root, the x coordinate is examined. If the point to be inserted has a smaller x than the point represented by the node, the inserted point will move to the left; otherwise, it will move to the right. At each level of the tree, the division axis alternates clockwise.

By checking where the point will go depending on the axis that divides the specified plane, we proceed recursively to the next nodes. For instance, if it separates it at level x, we check level x, level y, and level z if it is a 3D kD Tree, then level x again, and so on. When we reach the leaf to which the point belongs, we split the space of the leaf according to the axis that is next in line according to the round-robin Algorithm. The topology of a two-dimensional kD tree following the insertion of certain points is seen in Figure 5.

Inserting elements into the tree has time complexity $\Theta(log2N)$ or $O(N)$ in the worst case where N is the number of elements in the tree.

2.3.2 Range Search. Spatial search in kD Tree follows the same method as QuadTree. We start from the root and check if the children intersect with the search space. If so, we continue recursively to the next nodes until we reach the leaf. There we check if the

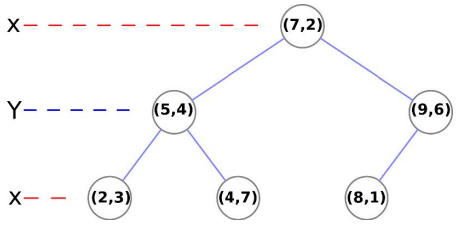

Figure 5: Representation of the structure of a two-dimensional kD Tree. The space complexity of a kD Tree is $O(N)$ where N is the number of inserted points.

points belong to the search space and return them if the answer is yes. An optimization happening here and in QuadTree is the case where the search space encompasses the entire child. In this case, we do not need to continue the search in the subtree spanning the child, but we return all elements belonging to it.

An orthogonal range query takes $O(\sqrt{N} + \mathrm{m})$time in the worst case, where m is the number of points returned. In the more general case where we have a kD Tree in k-dimensions, the time complexity of a k-dimensional range query is $O\left(n^{1-1/k} + m\right)$.

2.3.3 Comparison of kD Tree with QuadTree. QuadTree and kD Tree are quite comparable. Some contend that kD Tree is a k-dimensional extension of QuadTree. Nonetheless, they vary in terms of both their structure and the temporal complexity of certain of their operations.

The manner in which the space between the two trees is divided is different. QuadTree splits the space into 2^k subspaces, where k is the number of dimensions and kD has no predetermined partition independent of the data but divides the space into two halves depending on some measure (such a metric is for example the mean value with respect to the corresponding axis). Due to the exponential growth of the number of dimensions in the QuadTree, this difference makes the kD tree more efficient as the number of dimensions rises.

Another distinction is the simplicity of tree modification. In the event of a single modification, the kD tree may need to be rebuilt, but the QuadTree is more sensitive to alterations that may disrupt its equilibrium or initiate a chain of changes. Lastly, both trees are highly efficient in range searches, with the kD Tree edging out the QuadTree owing to its superior fit to the data and its insignificant production of empty quadrants if the data is not evenly distributed. Accessing empty quadrants increases the cost of the range search, giving kD Trees an advantage over non-uniform data.

2.4 R Tree

In this Section, the R-tree index is introduced, as well as its distributed implementation along with its usefulness in the realm of databases.

This Section is based on the initial paper on R-trees [12] and on the work in [16]. R-Tree is a height-balanced tree with entries stored in its leaves, similar to B-Tree. It specialises in storing spatial data using coordinates as its keys. The tree is entirely dynamic since it permits insertions, removals, and searches without requiring periodic reorganisations for optimal functioning.

The entries in an R-Tree leaf are of the form (mbr, id), where mbr (Minimal Bounding Rectangle) refers to the minimum n-dimensional rectangle that encompasses the item to be inserted. In the case of points, the mbr is the point itself, but for polygons of general shape, the mbr contains coordinates that span the whole polygon without any gaps. The id refers to the data associated with the mbr in the tree and is either the data itself or a key identifying the location of the data in a database. The records in R-Tree internal nodes have the form $(mbr, child - pointer)$, where the child-pointer is a reference to a node at the next level of the R-Tree. In the internal node, the mbr spans all of the rectangles in the node indicated by the child pointer. R-tree and its internal structure are determined by a number of variables which are listed below:

- Each leaf node contains from m to M children (entries) unless it is the root. The same applies to internal nodes.
- The root has at least two children unless it is a leaf. If it is a leaf, it can have a single child (the first record that enters the tree).
- All leaves are at the same level within the tree.

Figure 6 shows the internal structure of the tree. The height of an R-Tree for N data is at most $\lceil \log_m N \rceil - 1$ while the spatial complexity for all nodes except the root is m/M.

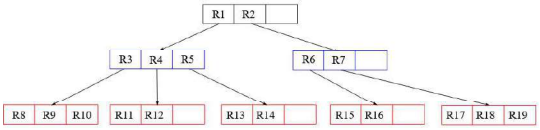

Figure 6: Internal structure of R-Tree.

2.4.1 Insert. Insertion into R-Trees is relatively similar to insertion into B-Trees with the distinction that filled nodes are split and these splits are propagated up the tree. The method for putting new elements into the tree is as follows:

(1) First the root is visited. If the root is a leaf then the record is saved to the root and returned.
(2) If the root is not a leaf we continue recursively until we reach a leaf. To get to the leaf we check at each node whose the mbr of the child needs to be expanded the least to include the new record. If more children claim the registration, differences are resolved by the new registration going to the child whose mbr covers a smaller area.
(3) We continue recursively performing step 2 from node to node until we reach the leaf.
(4) When we get to the leaf, we enter the new record. If the leaf has children less than or equal to M, the algorithm ends. Otherwise, we apply a strategy of fragmentation. If the split caused chained splits that reached the root, then we create a new root with children of the two nodes resulting from the split of the old one.

In terms of splitting, what we are essentially doing is splitting the node into two nodes and splitting the records of the old node between them. Then the pointers of the old node's parent node are updated to replace the old pointer with the two new pointers. If the parent node overflows, the process is repeated. It is important

here to choose the method by which the records of the old node are shared with the new ones in order to reduce as much as possible the overlaps and the empty space inside the *mbr* as well as to minimize the total space covered.

2.4.2 STR Packing (Sort Tile recursive). Having covered the insertion of a single element in the previous section, a method for bulk insertion of data into an R-Tree is presented named STR packing [25].

The difference with the serial input of data in the tree lies in the existence of overlaps and its efficient structure. A dataset in the real world is usually not in an efficient format, meaning that the data is not efficiently ordered so that when serialized it creates an efficient structure. This means that elements that are far apart and ideally should be on different nodes end up on the same node due to the serialization causing the MBR of the corresponding node to swell sharply and creating unnecessary empty space. Also, the splitting strategies, while they do the job fairly well, are by no means perfect and create a large amount of overlap when there is no favourable partitioning of elements, again due to their serialization. All of this, if viewed globally, leads to an unbalanced tree from some splits with the result that some subtrees need to be accessed much more than others, while many times we have unnecessary accesses when searching for elements due to overlaps (looking at the MBR a subtree appears to contain a particular element, but ultimately it is not there but in some adjacent overlapping subtree).

With STR we can solve all of these problems by creating a tree that breaks up the space optimally and without overlaps. However, STR to achieve this result puts quite a burden on main memory compared to serial input since it keeps all the data of a dataset in memory to make the necessary modifications before building the tree. Below we will describe the algorithm assuming that we have a file with r points, where each node can have up to n elements and the R-Tree will be two-dimensional:

(1) We first calculate two values: $P = \lceil r/n \rceil$ and $S = \lceil \sqrt{P} \rceil$. The aim is to divide the space of r points into $\sqrt{r/n}$ vertical pieces (slices) that have enough points to create $\sqrt{r/n}$ nodes.
(2) We first sort the data with respect to the x coordinate and divide it into S vertical slices. Each slice receives $S \times n$ consecutive points from the sorted data. The last slice can have less than $S \times n$ points.
(3) We sort the data of each slice by the y coordinate and group them sequentially by n. These groups at the lowest level are the leaves of the tree.
(4) We gather all the leaves created in each slice into a list and recursively follow the previous steps to create the internal nodes. For internal nodes the elements listed as points above are replaced by rectangular MBRs of each leaf.
(5) We perform step 4 until a list of internal nodes less than or equal to n is generated. This list contains the children of the root we are creating.

By sorting the elements in each layer with respect to x, and then using partitioning we achieve vertical partitioning of the space into non-overlapping pieces. For the same reason, we then sort by y and split so that there are no overlaps in the horizontal split. The partition resulting from the STR algorithm described is shown in Figure 7.

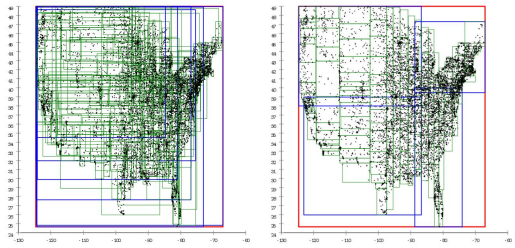

Figure 7: R-Tree structure with serial input of elements (left) and with the STR algorithm (right). The spatial representation shows how much more efficiently split and zero-overlapping the tree is using STR as opposed to the densely overlapping structure resulting from serial input using the quadratic split strategy.

The above procedure is generalized to dimensions greater than 2. We follow the same process by sorting on the first dimension and dividing by $S = \left\lceil P^{\frac{1}{k}} \right\rceil$ vertical slices where each slice has $\left\lceil P^{\frac{k-1}{k}} \right\rceil$ sequential elements. Each slice is then processed recursively as if we had a $k-1$ dimensional dataset.

2.4.3 Range Search. The range search follows the same process in R-Tree as in similar trees. Below the steps of the search algorithm based on a range S and node T are shown:

(1) If T is not a leaf, we check each of its children to see if its MBR intersects the range S. For all children intersecting the search range, we perform the steps of this algorithm.
(2) If T is a leaf, we check all its elements and return those that belong to the range S.

The complexity of the algorithm cannot be clearly defined as it largely depends on the nature of the data, how it was inserted into the tree, and the scope of the search. A lot of overlap due to serial input of scrambled data leads to a lot of extra searches in subtrees and thus the performance decreases. Depending on the scope of the search, the accesses can be limited to a few leaves-children of an internal node or extend to a large number of subtrees and leaves, reducing or increasing the search time respectively. Finally, the data itself may not be uniformly distributed in space and exhibit sparse and dense regions leading to individual accesses in specific subtrees. However, in most cases the tree structure allows the elimination of irrelevant subtrees from the search to a satisfactory level, which leads to efficient searches in general cases.

2.5 Distributed kD Tree index and QuadTree

In [28], the authors present a data management system based on HBase named MD-HBase. This system is intended to address the urge for mass input of new data and real-time spatial query response as required by Location Based Services (LBS). Within this system, they implement two data structures, KD tree, and QuadTree, which are stored according to the key-value data model. But before moving on to the implementation details, we will initiate by explaining the

special way these indexes will partition the space and how this partitioning helps to store them efficiently.

In a key-value database, as we saw in previous sections, objects are stored ordered by a key and partitioned by the range of the key space. As a key, we use the Z-value of the dimensions on which the index is built. But how is this Z-value obtained? If we consider that the multidimensional space is divided into subspaces of equal size and each dimension is numbered in the binary system, then the z-order of the specific subspace is given by joining the names of the spaces that contain it, adding at the end the name of the subspace itself. Such a z-order is shown in Figure 8, where for example for the subspace (01, 01) the z-order is 0011. This follows from the surrounding space named 00 and the subspace itself (01, 01) which has the name 11. If we join the two names we get the z-order 0011 of the subspace (01,01).

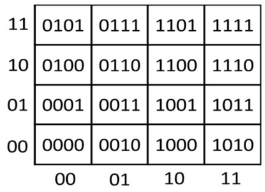

Figure 8: Binary Z-ordering.

Extending this concept of space partitioning to kD Trees and QuadTrees, we may encode the subspaces created by these trees while adding points, as seen in Figure 9. The Longest Common Prefix that occurs in the subspace is used for encoding. In the right subspace generated by the kD Tree in the right portion of Figure 9, for instance, the number 1 is the longest common prefix shared by subspaces in this space. Therefore, this number will serve as its name.

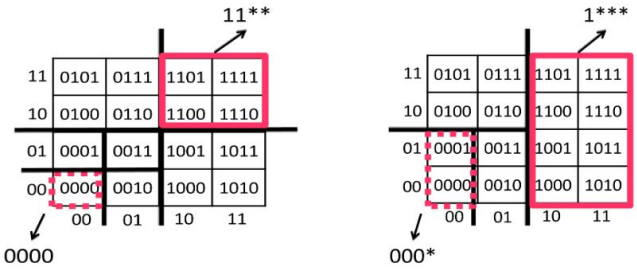

Figure 9: Left: Coding of the spaces generated by QuadTree. Right: Encoding of the spaces generated by the kD Tree.

The design of indexes in the distributed system takes advantage of the Z-ordering and longest common prefix properties. First, if space A surrounds space B, the name A is appended to the name B. Second, the name of the subspace is enough for defining the areas' bounds in all dimensions. The second attribute is a consequence of the fact that the name of a space is derived from the union of the names of all the spaces within a dimension, and so may help eliminate subtrees that do not belong to range queries.

By storing the Z-values of the spaces created by multidimensional structures such as kDTree and QuadTree, Z-values have allowed us to reduce their dimensions to one.

2.6 Index format and storage

Two levels comprise the index: the Index Layer and the Storage Layer. There is an ordered series of subspace names in the Index Layer. This ordered sequence is stored in a BigTable and has a B+ tree structure as a result. The two tiers of this B+ tree are ROOT and META. The ROOT layer is never subdivided and always leads to the META layer, which points to the data. At all levels, each row is quite small since it carries just a few bits of information, such as the name of the subspace, the location of the contents of the subspace, and some query processing metadata. This enables the index as a whole to grow to extremely big sizes without the Index Layer expanding too large proportionately while maintaining a large index size inside the same partition of the BigTable. Figure 10 illustrates the format stated before.

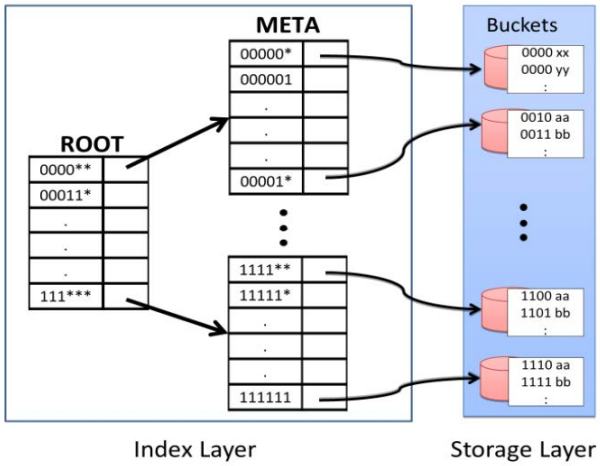

Figure 10: Storage format of the index stored in BigTable while the Storage Layer follows another way of storage.

HBase was used for the Storage Layer. Data in the Storage Layer is organized into buckets. For experimental reasons, different data storage models were implemented. These models are as follows:

- Table share model (TS): In this model, all buckets are placed in the same HBase table with keys of their z-values. This model allows efficient partitioning of the space simply by updating the rows of the table.
- Table per bucket model (TPB): In this model, each bucket has its own table and therefore the Storage Layer has multiple HBase tables. This model allows parallelism to be exploited since multiple tasks can be started simultaneously on different tables.
- Hybrid model: A balance between the previous two models. First, the space is partitioned and we create a table for each subspace created. After the initial split, if there is an overflow and additional splits are required, the newly created subspaces share the same array as their enclosing space.

- Region per bucket model (RPB): As we saw previously, a table in HBase consists of regions. This model uses a table for the data and regions as buckets. This model, however, required changes in the strategy and process by which HBase internally splits the table into regions, so that the split is done at the appropriate point.

Finally, tree and index data structures along with their insert/delete and search operations complexities are summarized in Table 1.

3 EXPERIMENTAL RESULTS

3.1 B+ Tree

The data used in the experiments were created by simulating similar massive datasets using a data generator to assess the distributed index. The simulated dataset was one gigabyte per size and it includes nine million records. This file was initially stored in HDFS before the index is constructed and experiments are set.

Initially, the performance of tree construction is assessed depending on the selected class. Figure 11 illustrates the assessment outcomes for two distinct groups. Clearly, the class of the tree is of utmost significance throughout the construction of the tree. It is crucial to fit the tree order to the amount of data, as a disproportionate choice of parameters (big dataset, tiny tree order) hinders the speed of both construction and search owing to the numerous additional visits necessitated by the high number of internal nodes. Lastly, with a tiny class, a very deep and massive tree is generated, which may need dividing it into numerous RegionServers when the dataset is enormous, resulting in degraded performance.

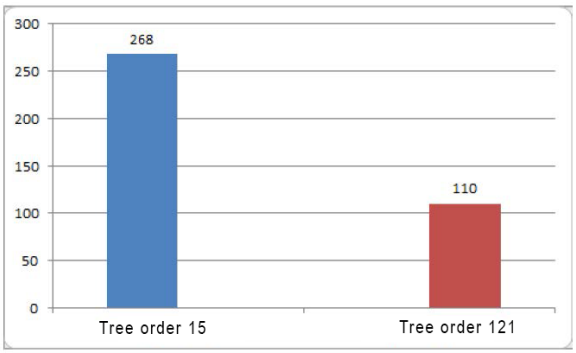

Figure 11: Comparison of performance as a function of the order of the B+ tree.

The construction of the centralised and distributed B+ trees was then compared. The results are shown in Figure 12. Clearly, the building time of a centralised B+ tree is much longer than that of a distributed B+ tree. This is because the distributed tree uses the parallelism offered by MapReduce in order to be created quickly. In fact, the findings are even more astounding when we realise that the dataset used to create the centralised tree was 300MB, but it was 1GB for the distributed tree owing to capacity constraints.

Finally, the range query performance of the centralised and distributed frameworks was evaluated. For each structure, 100 random range queries were executed on the 300MB and 1GB datasets,

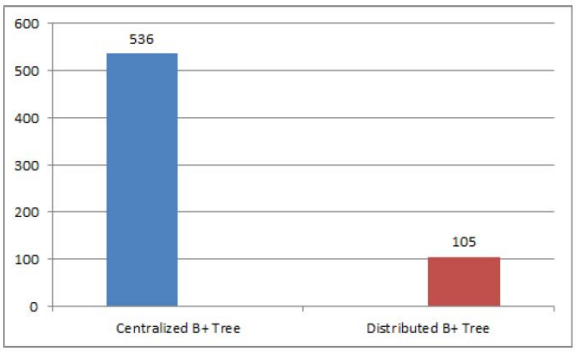

Figure 12: Comparison of centralized and distributed B+ tree construction performance.

respectively. The results are shown in Figure 13. Although the concentrated tree dealt with far less data than the distributed tree, its performance was not superior. These assessments led to the conclusion that HBase and Hadoop are very competent technologies for managing large data sets and executing queries on them quickly.

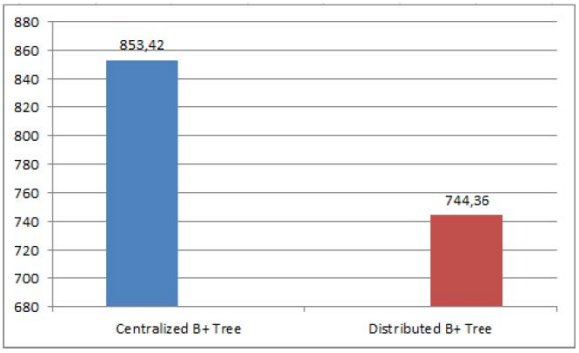

Figure 13: Performance comparison on range queries between centralized and distributed B+ tree implementation.

To further enhance the findings, the prewarm approach mentioned in Section 2.1.2 was used, and tests were conducted to establish the extent to which this strategy reduces the search time. The results are shown in Figure 14. As it can be seen, the prewarm approach greatly improves the time, and maybe to a higher amount than it seems, given that the data is not large enough to be divided into several RegionServers and to benefit even more from the prewarm technique. However, its advantage may be shown in smaller datasets as well.

3.2 Index Experiments

The experiments were conducted on an Amazon EC2 cluster ranging in size from 4 to 16 nodes. Each node had 4 virtual cores, 15.7 GBs memory, 1.6 TB hard disk and 64bit Linux operating system. The experiments were done for all the models we mentioned above.

The first measurements concerned the insert throughput and are shown in Figure 15. The measurements were made in relation to the load the system had to deal with. The load was simulated with

Table 1: Tree and Index Data structures and their Insert/Delete and Search Complexities.

Proposition	Reference	Dataset Type	Insertion/Deletion Complexity	Search Complexity
B-tree	[10]	Temporal	$O(n\log(n))$	$O(n\log(n))$
B+-tree	[13]	Temporal	$O(n\log(n))$	$O(n\log(n))$
B*-tree	[24]	Temporal	$O(n\log(n))$	$O(n\log(n))$
Compact B-tree	[7]	Temporal	$O(n\log(n))$	$O(n\log(n))$
T-tree	[24]	Temporal	$O(2n\log(n))$	$O(n\log(n))$
Masstree	[26]	Temporal	$O(n\log(n))$	$O(n\log(n))$
Fully Persistent B-tree	[6]	Temporal	$O\left(\log_B n + \log_2 B\right)$	$O\left(\log_B n + t/B\right)$
UB-tree	[29]	Spatio-temporal	$O(n\log(n))$	$O(n\log(n))$
PaIndex	[41]	Spatio-temporal	$O(n\log(n))$	$O(n\log(n))$
MLB+-tree	[36]	Seismic Data	$O(n\log(n))$	$O(n\log(n))$
SR-tree	[23]	Image feature vectors	$O(n\log 3(n))$	$O(n\log 3(n))$
E-tree	[40]	Spatial	$< O(n\log(n))$	Not calculated
ER+-tree	[2]	OpinRank Review	Not calculated	Not calculated
SUSHI	[11]	Color histogram and Synthetic data	$O(n^2\log(n))$	Not calculated
R-tree	[12]	Geographical and Multimedia	$O(dn\log(n))$	$O(n\log(n))$
R+-tree	[33]	Geographical and Multimedia	$O(n\log(n))$	$O(n\log(n))$
R*-tree	[3, 4]	Geographical and Multimedia	$O(n\log(n)) + Re-insertion\ complexity$	$O(n\log(n))$
Hilbert R-tree	[15]	Spatial	$O(\log(n) + M\log(n))$	$O(n\log(n))$
SS-tree	[38]	Multi-media data	$O(n\log(n)) + Re-insertion\ complexity$	$O(n\log(n))$
BFM & R-tree	[37]	Multidimensional	Not calculated	Not calculated
ST^{2}B-tree	[8]	Multidimensional	Not calculated	Not calculated
DCC & R-tree	[35]	Medical data	$O(nkt)$	$O(n\log(n))$
X-tree	[5]	Spatial data	Not calculated	Not calculated
aX-tree	[31]	Spatial data	Not calculated	Not calculated
X+-tree	[9]	Spatial data	Not calculated	Not calculated
R*Q-tree	[14]	Spatial data	$O((kndt)(n\log(n)))$	Not calculated
LSM-tree	[27]	Spatial	$O(1)$	$O(n)$

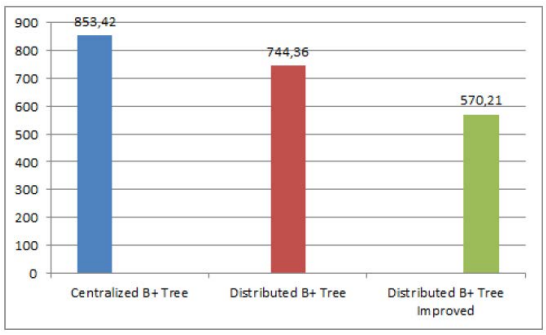

Figure 14: Range query performance for centralized, distributed, and improved B+ Tree.

generators producing 10,000 inputs per second and their number varied from 2 to 64.

Due to the expense of dividing a bucket, the TPB (table per bucket) and TS (table share) models have poor scalability. In contrast, these splits are performed asynchronously utilizing the HBase technique for separating areas in the RPB model. This technique is quite inexpensive per cost. The first two types prevent additional activities from occurring until the bucket split has been completed. Therefore, despite the fact that these two models make more use of parallelism in the import load distribution, their throughput is limited by the lengthy delay caused by splits.

The range query performance of the models for the QuadTree and kDTree indexes was then evaluated. For purposes of comparison, MapReduce was added to the experimental analysis, which verifies for each point whether it falls within the search range and returns it. On almost 400,000,000 points acquired by mimicking moving objects on the streets of San Francisco, experiments were conducted. Figure 16 depicts search times as a function of altering selectivity (i.e. the range of data covered by the specified query).

As shown in Figure 16, all of the previously proposed models are more efficient than ordinary MapReduce or Z-order searches. Particularly for queries with a high degree of selectivity, the provided models demonstrate an improvement of almost twofold or even thrice. In addition, the response time to queries is proportional to the selectivity for the work models, while the MapReduce implementation results in poorer response times regardless of the selectivity. A thorough examination reveals that the TPB approach is the most effective. In queries with a high degree of selectivity, the RPB model has greater execution times than the other models.

Regarding the two structures, QuadTree and kD Tree queries with high selectivity demonstrated superior performance with QuadTree than with kD Tree. However, when the selectivity of

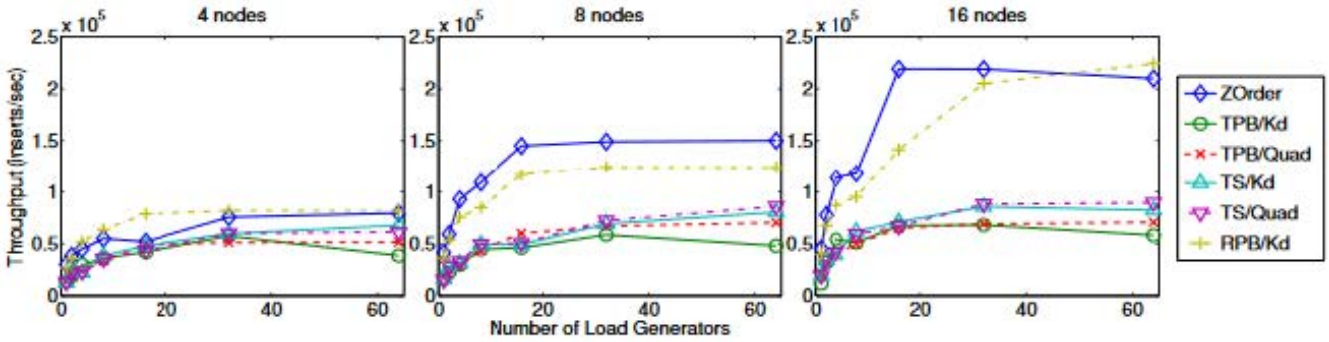

Figure 15: Insert operation performance of the models.

QuadTree drops, so does its reaction time. This is because kD Tree produces fewer buckets than QuadTree, yet QuadTree performs subtree removal more effectively than kD Tree. Due to this, the tree with the fewest buckets, i.e. the kD Tree, will be more efficient in cases of high selectivity when time is mostly used by making subqueries for each bucket. In contrast, with low selectivity, searching inside a bucket consumes more time than making subqueries for each bucket (since we are not searching in many due to low selectivity). Consequently, the tree that removes buckets more effectively throughout the search will be more efficient. We find that the performance of range queries relies directly on the capacity to remove superfluous searches (this property is referred to as pruning).

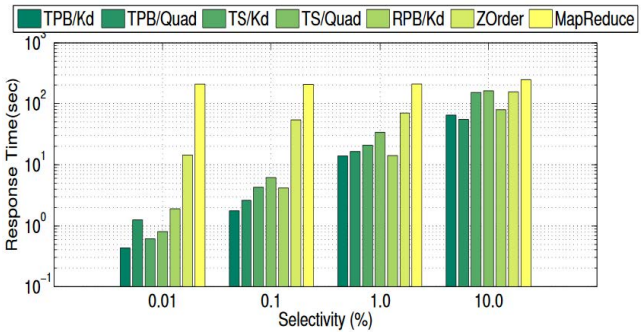

Figure 16: Log-scaled response times as a function of selectivity for the kD Tree and QuadTree indexes.

Finally, measurements were also made for nearest neighbour searches (kNN queries). The measurements were made on the same data as before and for various values of k (number of neighbours). The results of the measurements are shown in Figure 17.

The Figure reveals that the TPB model from QuadTree has the greatest performance, with a response time of around 250ms. Response times grow as k increases, but not exponentially, indicating that we are dealing with an efficient kNN algorithm. When k is less than or equal to 100, TPB/Quad outperformed the other models. The TPB/Quad model outperforms the TPB/Kd model because the lower bucket size decreases the bucket scan time. Both TS model implementations display equal response times for tiny k.

Comparing the two trees, QuadTree has lower reaction times for small k, but longer response times for high k. This is due to the size of the bucket. The search extension is uncommon for small k since the closest neighbours are often included inside the same bucket. Consequently, QuadTree discovers neighbours quicker than kDTree owing to the smaller buckets it produces. For large values of k, the likelihood of expanding the search to more buckets is high. The performance of QuadTree is inferior to that of kD Tree due to the frequent expansions caused by its tiny buckets.

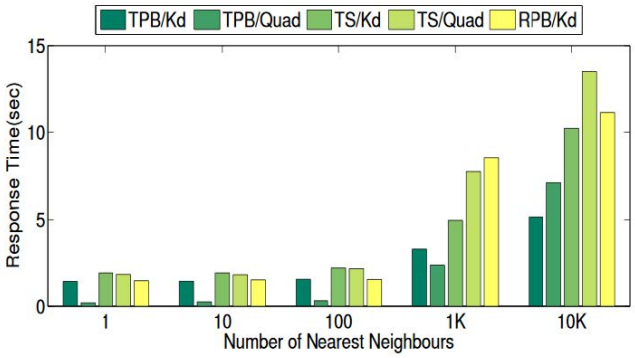

Figure 17: Response times for nearest neighbour queries as a function of the number of neighbours.

4 CONCLUSIONS

In the context of this comprehensive study, well-known tree data structures are surveyed and covered including B+ Tree, QuadTree, kD Tree, R Tree and others which are summarized in the overview table of the preceding section. Moreover, we present the data types they support as well as the insertion/deletion complexity and the search complexity per worst-case scenario. Additionally, we present some efficient indexing techniques with experiments including their centralized and decentralized implementation as per time and insert performance evaluation of fundamental indexing models. Ultimately, this review will serve as an informative summary paper that will familiarize readers with tree data structures and indexing mechanisms for big data management. Future directions of this

work include the integration of sampling schemes [17–20, 22] for approximations along with Tiny Machine Learning (TinyML) methods running on embedded devices [32, 39] which will enable us to run ML models on data structures on top of micro-controllers.

REFERENCES

[1] Hera Antonopoulou, Vicky Mamalougou, and Leonidas Theodorakopoulos. 2022. The Role of Economic Policy Uncertainty in Predicting Stock Return Volatility in the Banking Industry: A Big Data Analysis. (2022).

[2] Balamurugan Balasubramanian, Kamalraj Durai, Jegadeswari Sathyanarayanan, and Sugumaran Muthukumarasamy. 2019. Tree based fast similarity query search indexing onoutsourced cloud data streams. *Int. Arab J. Inf. Technol.* 16, 5 (2019), 871–878.

[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. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[4] Norbert Beckmann and Bernhard Seeger. 2009. A Revised R*-Tree in Comparison with Related Index Structures. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 799–812. https://doi.org/10.1145/1559845.1559929

[5] Stefan Berchtold, Daniel A Keim, and Hans-Peter Kriegel. 1996. The X-tree: An index structure for high-dimensional data. In *Very large data-bases*. 28–39.

[6] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. 2020. Fully persistent B-trees. *Theoretical Computer Science* 841 (2020), 10–26. https://doi.org/10.1016/j.tcs.2020.06.027

[7] Peter Bumbulis and Ivan T. Bowman. 2002. A Compact B-Tree. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 533–541. https://doi.org/10.1145/564691.564753

[8] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A Nascimento. 2008. ST2B-tree: a self-tunable spatio-temporal B+-tree index for moving objects. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 29–42.

[9] MN Doja, Sapna Jain, and M Afshar Alam. 2012. SAS: Implementation of scaled association rules on spatial multidimensional quantitative dataset. *International Journal of Advanced Computer Science and Applications* 3, 9 (2012).

[10] Goetz Graefe. 2006. B-Tree Indexes for High Update Rates. *SIGMOD Rec.* 35, 1 (mar 2006), 39–44. https://doi.org/10.1145/1121995.1122002

[11] Stephan Günnemann, Hardy Kremer, Dominik Lenhard, and Thomas Seidl. 2011. Subspace clustering for indexing high dimensional data: a main memory index based on local reductions and individual multi-representations. In *Proceedings of the 14th International Conference on Extending Database Technology*. 237–248.

[12] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[13] Christian S Jensen, Dan Lin, and Beng Chin Ooi. 2004. Query and update efficient B+-tree based indexing of moving objects. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 768–779.

[14] Pan Jin and Quanyou Song. 2011. A novel index structure r* q-tree based on lazy splitting and clustering. In *2011 IEEE International Conference on Computer Science and Automation Engineering*, Vol. 3. IEEE, 405–407.

[15] Ibrahim Kamel and Christos Faloutsos. 1993. *Hilbert R-tree: An improved R-tree using fractals*. Technical Report.

[16] Aristeidis Karras, Christos Karras, Dimitrios Samoladas, Konstantinos C. Giotopoulos, and Spyros Sioutas. 2022. Query Optimization in NoSQL Databases Using an Enhanced Localized R-tree Index. In *Information Integration and Web Intelligence*, Eric Pardede, Pari Delir Haghighi, Ismail Khalil, and Gabriele Kotsis (Eds.). Springer Nature Switzerland, Cham, 391–398.

[17] Christos Karras and Aristeidis Karras. 2022. DBSOP: An Efficient Heuristic for Speedy MCMC Sampling on Polytopes. https://doi.org/10.48550/ARXIV.2203.10916

[18] Christos Karras, Aristeidis Karras, Markos Avlonitis, Ioanna Giannoukou, and Spyros Sioutas. 2022. Maximum Likelihood Estimators on MCMC Sampling Algorithms for Decision Making. In *Artificial Intelligence Applications and Innovations. AIAI 2022 IFIP WG 12.5 International Workshops*, Ilias Maglogiannis, Lazaros Iliadis, John Macintyre, and Paulo Cortez (Eds.). Springer International Publishing, Cham, 345–356.

[19] Christos Karras, Aristeidis Karras, Markos Avlonitis, and Spyros Sioutas. 2022. An Overview of MCMC Methods: From Theory to Applications. In *Artificial Intelligence Applications and Innovations. AIAI 2022 IFIP WG 12.5 International Workshops*, Ilias Maglogiannis, Lazaros Iliadis, John Macintyre, and Paulo Cortez (Eds.). Springer International Publishing, Cham, 319–332.

[20] Christos Karras, Aristeidis Karras, and Spyros Sioutas. 2022. Pattern Recognition and Event Detection on IoT Data-streams. https://doi.org/10.48550/ARXIV.2203.01114

[21] Christos Karras, Aristeidis Karras, Leonidas Theodorakopoulos, Ioanna Giannoukou, and Spyros Sioutas. 2022. Expanding Queries with Maximum Likelihood Estimators and Language Models. In *Proceedings of the ICR'22 International Conference on Innovations in Computing Research*, Kevin Daimi and Abeer Al Sadoon (Eds.). Springer International Publishing, Cham, 201–213.

[22] Christos Karras, Aristeidis Karras, Dimitrios Tsolis, Konstantinos C. Giotopoulos, and Spyros Sioutas. 2022. Distributed Gibbs Sampling and LDA Modelling for Large Scale Big Data Management on PySpark. In *2022 7th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*. 1–8. https://doi.org/10.1109/SEEDA-CECNSM57760.2022.9932990

[23] Norio Katayama and Shin'ichi Satoh. 1997. The SR-tree: An index structure for high-dimensional nearest neighbor queries. *ACM Sigmod Record* 26, 2 (1997), 369–380.

[24] Tobin J Lehman and Michael J Carey. 1985. *A study of index structures for main memory database management systems*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[25] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings 13th international conference on data engineering*. IEEE, 497–506.

[26] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.

[27] Fei Mei, Qiang Cao, Hong Jiang, and Lei Tian Tintri. 2017. LSM-Tree Managed Storage for Large-Scale Key-Value Store. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 142–156. https://doi.org/10.1145/3127479.3127486

[28] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *2011 IEEE 12th International Conference on Mobile Data Management*, Vol. 1. 7–16. https://doi.org/10.1109/MDM.2011.41

[29] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. 2000. Integrating the UB-tree into a database system kernel.. In *VLDB*, Vol. 2000. Citeseer, 263–272.

[30] Hanan Samet. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)* 16, 2 (1984), 187–260.

[31] Grace Samson, Lu Joan, Mistura M Usman, Aminat A Showole, and Hadeel Jazzaa Hadeel. 2018. Large spatial database indexing with aX-tree. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 3, 3 (2018), 759–773.

[32] Nikolaos Schizas, Aristeidis Karras, Christos Karras, and Spyros Sioutas. 2022. TinyML for Ultra-Low Power AI and Large Scale IoT Deployments: A Systematic Review. *Future Internet* 14, 12 (2022). https://doi.org/10.3390/fi14120363

[33] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*. Technical Report.

[34] Leonidas Theodorakopoulos, Hera Antonopoulou, Vicky Mamalougou, and Konstantinos Giotopoulos. 2022. The drivers of volume volatility: A big data analysis based on economic uncertainty measures for the Greek banking system. *Banks and Bank Systems* 17, 3 (08 2022), 49–57. https://doi.org/10.21511/bbs.17(3).2022.05

[35] Xinlu Wang, Weiming Meng, and Mingchuan Zhang. 2019. A novel information retrieval method based on R-tree index for smart hospital information system. *International Journal of Advanced Computer Research* 9, 42 (2019), 133–145.

[36] Yida Wang, Changhai Zhao, Zengbo Wang, Jiguo Du, Chao Liu, Haihua Yan, Jiamin Wen, Hongjun Hou, and Kun Zhou. 2018. MLB+-tree: A Multi-level B+-tree Index for Multidimensional Range Query on Seismic Data. In *2018 5th International Conference on Systems and Informatics (ICSAI)*. IEEE, 1176–1181.

[37] Zhu Wang, Tiejian Luo, Guandong Xu, and Xiang Wang. 2013. A new indexing technique for supporting by-attribute membership query of multidimensional data. In *International Conference on Web-Age Information Management*. Springer, 266–277.

[38] David A White and Ramesh Jain. 1996. Similarity indexing with the SS-tree. In *Proceedings of the Twelfth International Conference on Data Engineering*. IEEE, 516–523.

[39] Angelos Zacharia, Dimitris Zacharia, Aristeidis Karras, Christos Karras, Ioanna Giannoukou, Konstantinos C. Giotopoulos, and Spyros Sioutas. 2022. An Intelligent Microprocessor Integrating TinyML in Smart Hotels for Rapid Accident Prevention. In *2022 7th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*. 1–7. https://doi.org/10.1109/SEEDA-CECNSM57760.2022.9932982

[40] Peng Zhang, Chuan Zhou, Peng Wang, Byron J Gao, Xingquan Zhu, and Li Guo. 2014. E-tree: An efficient indexing structure for ensemble models on data streams. *IEEE Transactions on Knowledge and Data engineering* 27, 2 (2014), 461–474.

[41] Shaoming Zhang, Xudong Liu, Mingming Zhang, and Tianyu Wo. 2017. PaIndex: An online index system for vehicle trajectory data exploiting parallelism. In *2017 4th International Conference on Systems and Informatics (ICSAI)*. IEEE, 696–703.