Multidimensional Adaptive & Progressive Indexes

Matheus Agio Nerone
CWI, Amsterdam
matheus@cwi.nl

Pedro Holanda
CWI, Amsterdam
holanda@cwi.nl

Eduardo C. de Almeida
UFPR, Brazil
eduardo@inf.ufpr.br

Stefan Manegold
CWI, Amsterdam
manegold@cwi.nl

***Abstract*—Exploratory data analysis is the primary technique used by data scientists to extract knowledge from new data sets. This type of workload is composed of trial-and-error hypothesis-driven queries with a human in the loop. To keep up with the data scientist's productivity, the system must be capable of answering queries in interactive times. Given that these queries are highly selective multidimensional queries, multidimensional indexes are necessary to ensure low latency. However, creating the appropriate indexes is not a given due to the highly exploratory and interactive nature of such human-in-the-loop scenarios.**

In this paper, we identify four main objectives that are desirable for exploratory data analysis workloads: (1) *low overhead* over the initial queries, (2) *low query variance* (i.e., *high robustness*), (3) *predictable index convergence*, and (4) *low total workload time*. Given that not all of them can be achieved at the same time, we present three novel incremental multidimensional indexing techniques that represent three sample points on a Pareto front for this multi-objective optimization problem. (a) The *Adaptive KD-Tree* is designed to achieve the lowest total workload time at the expense of a higher indexing penalty for the initial queries, lack of robustness, and unpredictable convergence. (b) The *Progressive KD-Tree* has predictable convergence and a user-defined indexing cost for the initial queries. However, total workload time can be higher than with Adaptive KD-Trees, and per-query time still varies. (c) The *Greedy Progressive KD-Tree* aims at full robustness at the expense of only improving the per-query cost after full index convergence.

Our extensive experimental evaluation using both synthetic and real-life data sets and workloads shows that (a) the Adaptive KD-Tree reduces total workload time by up to a factor 2 compared to the state-of-the-art, (b) the Progressive KD-Tree achieves predictable convergence with up to one order of magnitude lower initial query cost, and (c) the Greedy Progressive KD-Tree exhibits the lowest query variance up to three orders of magnitude lower than the state-of-the-art.

I. Introduction

When analyzing a new data set, data scientists impose exploratory queries to extract knowledge from the underlying data. Their workflow has three main steps. (1) They generate a hypothesis about the data; (2) they validate it on a trial-and-error basis by issuing highly selective queries to check small portions of the data; (3) they adjust their hypothesis based on step (2) and repeat until satisfied [1].

This process can require many time-consuming trial-and-error iterations until the data scientist can extract the desired information. There is a direct impact on the query performance of each iteration and the rate on which the data scientists can refine their findings. A study by Liu et al. [2] proposed the maximum data-to-insight time for each hypothesis checking iteration, which should not surpass an interactivity threshold of 500 ms. For small data sets, full scans are fully capable of executing queries within this time limit, but for larger multidimensional data sets, only a multidimensional index that covers the query attributes is capable of pushing query response time below the 500 ms interactivity threshold.

The selection of beneficial indexes is one of the most difficult physical design decision that a database administrator (DBA) has to face, due to the combinatorial explosion of possible indexes and the trade-off between workload speedup vs. index size, creation, and maintenance costs [3]. This task is usually alleviated through the use of self-tuning tools [4]. These tools capture the executing workload, select the most relevant queries, and weigh in index creation/maintenance costs versus the benefits they would bring to most queries. They are then capable of suggesting a collection of indexes that the DBA must evaluate and create or drop.

Although self-tuning tools [4] have been widely successful in data warehouses, they are not capable of facilitating this process for data scientists. There are three key differences when comparing exploratory workloads and classic OLAP. (1) With exploratory workloads, there is no previous workload knowledge. (2) There is no available idle time for a priori full index creation. (3) Data scientists do not have the same skill set as DBAs to weigh in the index suggestions.

Techniques like Adaptive Indexing [5], [6] and Progressive Indexing [7] aim to alleviate the index construction issue on exploratory workloads by creating partial uni-dimensional indexes as a result of query processing. In this way, indexes are automatically created without any human intervention and incrementally refined towards a full index, the more the data is accessed. However, these techniques have very limited use on a broad group of data sets since they only target uni-dimensional workloads. For instance, the 1000 genomes project [8] has human genetic information, the Power data set[1] that contains sensor information from a manufacturing installation, and the SkyServer data set [9] which maps the universe, are some of many examples that perform multidimensional filtering.

Recently, Pavlovic et al. [10] published a study on multi-dimensional adaptive indexes, initially testing a Space-Filling Curve strategy, where multiple dimensions are mapped to one dimension. They used uni-dimensional adaptive indexing techniques on top of the created map. However, the indexing burden in the first queries was too high, making this approach unfeasible for interactive times. They later propose the QUASII index, a d-level hierarchical structure that similarly partitions

[1] https://debs.org/grand-challenges/2012/

the data as the coarse granular index strategy [6]. When accessing one piece, the data is continuously refined until all pieces within that piece are smaller than a given size threshold. This strategy is much more efficient in smearing out the cost of index creation than the Space-Filling Curve Adaptive Indexing. However, it results in two highly undesirable characteristics for exploratory workloads. (1) Due to the continuous piece refinement, it heavily penalizes queries when they first access one piece; (2) since the index prioritizes an aggressive refinement only on areas targeted by the executing query, it is not robust against changes in the access pattern, resulting in performance spikes if the workload suddenly accesses a previously unrefined piece.

This paper introduces three novel algorithms to tackle the problem of multidimensional adaptive indexing under exploratory data analysis. (a) The *Adaptive KD-Tree*, inspired by adaptive indexing, performs indexing using the query predicates as pivots, hence aiming for minimum total response time. (b) The *Progressive KD-Tree*, inspired by fixed-delta progressive indexing, introduces a per-query indexing budget that remains constant during query execution. Hence, a user-controlled amount of indexing is done per query. (c) The *Greedy Progressive KD-Tree* uses a cost model to automatically adapt the indexing budget to keep the per-query cost constant until full index convergence, achieving a low variance per-query.

The main contributions of this paper are:

- We propose a new multidimensional adaptive index called *Adaptive KD-Tree* inspired by standard adaptive indexing techniques.
- We introduce a new progressive indexing approach for multidimensional workloads named *Progressive KD-Tree*.
- We present a cost model for our Progressive KD-Tree to enable an adaptive indexing budget.
- We experimentally verify that our techniques improve total execution time, initial query cost, robustness, and convergence compared with the state-of-the-art.
- We provide an Open-Source implementation[2] of all techniques discussed in this paper including the state-of-the-art we compare with.

We consider relational data sets with d dimension attributes and possibly additional "payload" attributes, and we assume that all queries have a conjunctive selection predicate consisting of d terms, i.e., exactly one term for each dimension attribute.

Outline. This paper is organized as follows. In Section II, we investigate related research that has been performed on uni- and multidimensional automatic/adaptive index creation. In Section III, we describe our novel Multidimensional Adaptive and Progressive Indexes. In Section IV, we perform a quantitative assessment of each of the novel methods we introduce, and we compare them with the state-of-the-art on multidimensional adaptive indexing under three real workloads and eight synthetic workloads. Finally, in Section V, we draw our conclusions and discuss future work.

[2]Our implementations and benchmarks are available at https://github.com/pdet/MultidimensionalAdaptiveIndexing and https://zenodo.org/record/3835562

II. Related Work

The creation of indexes has been a long-standing problem in database automatical physical design. The combinatorial possibilities of indexes when designing a database make the selection of indexes an NP-Hard problem [3]. For most OLAP scenarios, it is possible to capture a relevant workload, select pertinent indexes to the workload, analyze their creation and maintenance overhead in contrast to their query execution benefit, and invest in a priori full index creation [4]. In data exploration, we do not have idle time to invest in full index creation as we do not have any previous information or opportunity to gather workload knowledge. Hence, Adaptive and Progressive Indexing techniques are more promising solutions. They create indexes during query execution, taking the current running workload as priority-refinement hints.

In this section, we briefly discuss the state-of-the-art multi-dimensional index structures and adaptive/progressive indexing techniques.

A. Multidimensional Data Structures

R-Tree [11] is an N-ary multidimensional tree that generalizes the B-Tree. Nodes represent rectangles that bound the insertion points of data (i.e., coverage), and different rectangles may overlap data. Like B-Trees, the insertions and deletions require splitting and merging nodes to preserve height-balance with all leaves at the same depth. The internal nodes keep a way of identifying a child node and representing the boundaries of the entries in the child nodes, while the external nodes store the data. The R-Tree has a variant, the R*-Tree [12], for read-mostly workloads that balances the rectangle coverage and reduces overlapping.

VA File [13] is a flat structure that divides an m-dimensional space in 2^b rectangular cells. Users assign b bits to be distributed over the m dimensions. A unique bit-string of length b is set for each cell, and data objects use a hash method to find the spacial position to each value (i.e., approximation by the bit-string).

KD-Tree [14] is a multidimensional binary search tree, where k is the number of dimensions of the search space that are switched between tree levels. The performance of KD-Trees is of great advantage as searches, insertions, and deletions of random nodes present logarithmic complexity and search of t tuples present sub-linear complexity. The nodes of the tree are insertion points. Therefore, the order of insertion shapes the tree structure but increases the complexity of maintenance when tree re-balancing is needed after deletion.

PH-Tree [15] implements a bit-string prefix sharing tree to reduce the space requirements compared to single key storage. The bit-string representation is used to navigate the dimensions in a Quadtree, where the first bit of the index entry indicates the position in the search space.

Flood [16] is a multidimensional learned index. The learning algorithm's goal is to help to tweak performance parameters of the index, like the layout of the index by choosing between a grid of cells or columns (in a 2-D representation), the size of each cell, and the sort order of each cell or column.

Discussion. To compare these index structures, we must put them in the context of the data exploration scenario. Although Flood has a significant advantage of finding an efficient setup by searching the parameters' space, it is not a good fit for our types of workloads since it requires a considerable amount of time to be invested in model training (i.e., index creation). PH-Trees present efficient lookups, but they are catered to data sets where data points are not evenly spread and share many prefixes. Finally, KD-Trees, VA Files, and R*-Trees have been thoroughly examined, in the main memory context, by Sprenger et al. [17]. The work concludes that the KD-Trees outperform R*-Trees and VA Files due to its point storage design. VA-Files have even a more significant disadvantage for shifting access patterns, common in exploratory data analysis, since it is a non-adaptive structure with a static number of bits assigned per dimension. Considering each technique's main drawbacks and advantages, we decided to use a KD-Tree as our multidimensional index of choice for exploratory data analysis, as a full index baseline and the index structure that holds the data for both our adaptive and progressive solutions. In summary, the reasons are its robust performance against shifting workloads, different from VA Files and PH Trees, the higher performance when compared to R*-Trees, and low index creation cost compared to Flood.

B. Adaptive/Progressive Index

Adaptive Indexing [5], [6] enables efficient incremental index creation, as a side effect of query execution. It only indexes columns that are truly queried and incrementally refines the indexes the more they are queried, eventually converging to a similar performance as a full index. Progressive Indexing [7] differs from Adaptive Indexing techniques in having a limited amount of indexing budget that it can perform per query. This budget allows for increased robustness, predictable performance, and full-index convergence while being highly competitive in total response time. However, most of these techniques are catered to produce uni-dimensional indexes. In the next paragraphs, we discuss the state-of-the-art adaptive indexing techniques that produce multidimensional indexes.

Space Filling Curve Cracking [10] uses a space-filling curve technique that preserves proximity (e.g., Z-Order, Hilbert Curve) to map multiple dimensions into one dimension. This additional step enables the use of uni-dimensional adaptive/progressive indexing techniques. Later on, queries also must be translated to this uni-dimensional mapping.

QUASII [10] Following the adaptive indexing philosophy, QUASII incrementally builds a multidimensional index prioritizing refinement on queried pieces. One significant difference compared to standard adaptive indexing techniques is that QUASII has a more aggressive refinement behavior. When accessing a piece, it recursively refines it until its size drops below a $size\_threshold$. QUASII pays higher refinement costs when a piece is accessed the first time to yield fast query response time when frequently accessing refined pieces.

Discussion. Space-Filling Curve Cracking is the first attempt to perform adaptive indexing of multiple columns. However, as demonstrated by Pavlovic et al. [10], mapping is prohibitively expensive on the first query, excluding this strategy from truly adaptive indexes. QUASII is a more promising solution since it features characteristics that are similar to standard adaptive indexing techniques. However, QUASII's aggressive refinement strategy is undesirable in an adaptive indexing strategy hurting query robustness. Besides, QUASII forces initial queries to pay an unnecessarily high cost. Finally, other techniques are self-proclaimed multidimensional adaptive indexes, like AQWA [18] and SICC [19]. However, they do not focus on exploratory data analysis but rather on adaptive indexing for data ingestion. The main goal of AQWA is to adjust for changes in the data in a Hadoop distributed scenario. Simultaneously, SICC mainly focuses on reducing "over-coverage" in entries of frequent data ingestion in streaming systems. Hence, they do not focus on a low penalty for the initial queries, on robustness or index convergence.

III. Multidimensional Adaptive/Progressive Index

In exploratory data analysis, multidimensional indexes must be created concerning four main objectives. (1) They must not inflict a high penalty over the initial queries since we do not know in advance if a group of columns will be queried only once or multiple times. (2) They should be robust, avoiding performance variations for similar queries, which is undesirable from the user perspective but can happen on incremental indexes due to access to less refined pieces. (3) They should guarantee predictable convergence to an index that yields the same performance as a full index. (4) They should minimize the total response time for a given workload to maximize the number of hypotheses that can be tested in the same amount of time. This section presents three adaptive/progressive indexing techniques for multidimensional data that represent three sample points on a Pareto front for this multi-optimization problem. (a) The *Adaptive KD-Tree* is optimized to achieve the lowest total response time. (b) The *Progressive KD-Tree* is designed to incur a low indexing cost over the initial queries and to have a predictable convergence. (c) The *Greedy Progressive KD-Tree* is designed to have a low performance variance until full index convergence.

The indexing techniques presented in this paper focus on fixed-width numerical data types and assume an uncompressed in-memory columnar data storage using a dense array per column/attribute. To the best of our knowledge, this also holds for all previously proposed adaptive indexing techniques, both uni- and multidimensional. An extension to also support variable-length strings, e.g., using a dictionary encoding and reorganizing only the fixed-width array of indices representing the actual columns, is mainly an engineering exercise that is beyond the scope of this paper and left for future work.

A. Adaptive KD-Tree

The *Adaptive KD-Tree* is a multidimensional adaptive indexing technique that follows the same principles as Adaptive Indexing [5]: (1) It uses the query predicate as hints as to what pieces of the data should be indexed, and (2) only indexes

the necessary pieces to answer the current query. Our index has two main canonical phases. The *initialization* phase only happens when the first query selecting a group of non-indexed columns is executed. In this phase, we create a copy of the original table into our index table. In the *adaptation* phase, we swap rows in the index table to partition it according to the query predicates.

In this section, we describe the general structure of our index, how the per-query adaptation works, and how we perform index lookups and piece scans.

Data Structures. The Adaptive KD-Tree uses a KD-Tree to store information regarding the offset of the pieces: Each node contains a key, a discriminator attribute, two pointers for the left and right children, and the position offset that keeps track of the created pieces. Since it is a secondary index, this position offset points to our index table stored in the decomposition storage model (DSM) format. The index table is initially created as a copy of the original table when the first query is executed.

Adaptation phase. The adaptation phase is triggered in all queries until the index fully converges. The adaptation performs two main steps:

1) In a KD-Tree, each node splits the data set along only one dimension, all nodes on the same level use the same dimension, and the levels from the root to the leaves of the tree iterate through the dimensions in a round-robin fashion. To achieve this, when processing a query, we first iterate through the lower bounds of all column predicates, and then through the upper bounds of all column predicates, inserting these boundaries into the KD-Tree, and pivoting the respective pieces accordingly. For example, given the predicates $6 < A \leq 13$ AND $5 < B \leq 8$, the adaptation order would use this sequence attribute-value pairs: $(A, 6), (B, 5), (A, 13), (B, 8)$.

Using query predicates as pivots rather than median values, as an "optimal" KD-Tree would do, is a conscious choice, favoring workload adaptivity over theoretical guarantees.

2) For each attribute-value pair, we physically adapt the data on the pieces that are relevant to the predicate. This is only done if the piece size is bigger than a previously defined $size\_threshold$. $size\_threshold$ is chosen such that the extra effort of indexing would not outperform a simple scan.

Figure 1 depicts an example of the adaptation phase when executing the first query with predicates $6 < A \leq 13$ AND $5 < B \leq 8$ with $size\_threshold = 4$. In the first part of our example, we have our initialized index table equal to the original table. In the second step, we start the adaptation phase generating the attribute-value pairs $(A, 6), (B, 5), (A, 13), (B, 8)$, and partitioning the index table for each of those pairs. In the example, the second step demonstrates the partition of pair $(A, 6)$. We swap the rows of our table, taking 6 as a pivot for the first column A, and insert in the KD-Tree the pivot 6 with the position offset 6. In the third step, we partition the pair $(B, 5)$, where the table is pivoted in the second column B with pivot 5, later on adding it to the KD-Tree. Note that we could perform this partitioning in both the top ($A \leq 6$) and bottom ($A > 6$) pieces of our

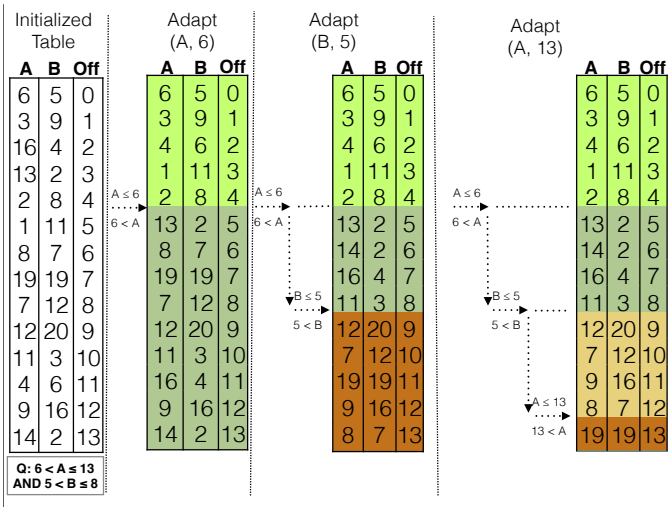


Fig. 1: Adaptive KD-Tree: The adaptation phase with query: $6 < A \leq 13$ AND $5 < B \leq 8$, and $size\_threshold = 4$.

table. However, since the Adaptive KD-Tree only indexes the minimum to answer the query, we only refine the piece that potentially contains query answers (here, $A > 6$), leaving the piece that surely contains no query answers ($A \leq 6$) unchanged. A similar process is done for the next pair $(A, 13)$ depicted as the fourth step. At this step, the resulting piece size reaches the $size\_threshold$, and no further partitioning happens for the last pair $(B, 8)$.

Index Lookup. After performing the necessary adaptations for the query, we perform an index lookup followed by the scan of all pieces that fit our query predicates. The index lookup starts from the root of the KD-Tree and recursively traverses the tree depending on how the query relates to the current node key. In Figure 2 we depict an example of the entire search process for predicates $6 < A \leq 15$ AND $0 < B \leq 5$. The

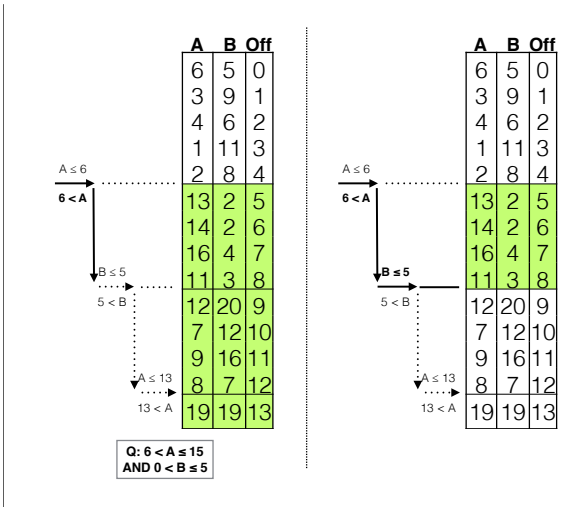


Fig. 2: A search with predicates $6 < A \leq 15$ AND $0 < B \leq 5$ in the Adaptive KD-Tree.

search method starts by comparing the root of the tree, that indexes column A on key 6, with the range $6 < A \leq 15$. We need to check the right child of the root since both predicate boundaries are greater than the node (i.e., where all elements on $A > 6$ are stored.). We now compare the range $0 < B \leq 5$ to the node that indexes column B on key 5. Note that this time, the predicate boundaries are lower or equal to the node's key. Hence, we only need to check its left child. Finally, since the left child is null, we scan the piece starting on offset 5 until offset 9.

Piece Scan. The index lookup returns a list of pieces that we scan to answer the query. For each piece, we have a pair of offsets indicating where they begin and end, and information of which predicates still need to be checked. For example, in Figure 2, on the rightmost column, the index would have returned one piece, with offsets 5 and 9. For this piece, we know that all elements in there are $6 < A$ and $B \leq 5$. Hence, we do not need to apply the lower and higher query predicates of attributes A and B, respectively. However, whenever the index does not match our query predicates exactly, we need to perform a multi-dimensional conjunctive selection on one or more pieces. There are, in general, two ways to perform multi-dimensional conjunctive selections in column stores. (1) We perform the selection on each column individually, creating an intermediate result per column as (candidate) list of IDs (or as bit-vector). Later, intersecting all lists (or and-ing all bit-vectors) to yield the final result. (2) We perform the selection over one column, creating an intermediate as (candidate) list of IDs (or as bit-vector). Then we use this candidate list (or bit-vector) to test the selection predicate on the next column only for those tuples that qualified with the first column and create a revised candidate list (or bit-vector) as an intermediate result reflecting both selections. We continue accordingly for all remaining columns. Option (1) is advantageous for low selectivities, since they focus on sequential scans over the whole data set, while option (2) presents the best performance over high selectivities since, except the first column, we only check elements that qualify. Hence, in all our scans we use option (2) with a candidate list to achieve the best performance.

Interactivity Threshold. The user must provide the Adaptive KD-Tree with an interactivity time threshold τ. In case a simple full scan of the data already exceeds τ, the Adaptive KD-Tree will perform a pre-processing step with the first query. This pre-processing step constructs a partial KD-Tree, using arithmetic means as pivots, until pieces are small enough (i.e., their scan cost is below τ). All subsequent queries will proceed with refining this KD-Tree as previously described. In this way, only the first query exceeds the interactivity threshold τ.

B. Progressive KD-Tree

The *Progressive KD-Tree* is a multidimensional progressive indexing technique inspired by Progressive Quick-Sort [7]. Like one-dimensional progressive indexing techniques, the main goals of Progressive KD-Tree are to limit the indexing penalty imposed on the first query, achieve robust performance, and ensure deterministic convergence towards a full index — irrespective of the actual query workload or data distribution. We accomplish all three goals by indexing a fixed-size portion of the data with each query, independent of the query predicates. The per-query indexing budget (and hence overhead over a scan) and the convergence speed can be controlled by a parameter δ that determines the fraction of the entire data set indexed with each query. Opting for workload-independence, we need to choose the partitioning pivots independent of the query predicates. We use the average value (arithmetic mean) to yield a reasonably balanced KD-Tree also with skewed data. Our experiments in Section IV show that determining the median to guarantee a perfectly balanced KD-Tree is prohibitively expensive and does not pay off. The Progressive KD-Tree follows two phases. In the initial *creation phase*, each query copies a δ fraction of the data out-of-place to our index, while pivoting on the average value of the first dimension. After all data has been copied, in the subsequent *refinement phase*, queries further split the existing pieces until their size drops below a $size\_threshold$. When all pieces reach the qualifying size, we consider that the index has converged to a full index. A fully-converged Progressive KD-Tree will have the same structure as a pre-built full index KD-Tree using arithmetic means as partitioning pivots.

Creation Phase. The creation phase copies the data from the original column into our index while filtering and pivoting it on that column's average value. The filtering process is similar to the Adaptive KD-Tree piece scan when copying and pivoting a dimension of the data. We create a candidate list to keep track of elements that qualify its filters. This candidate list is subsequently refined when copying and pivoting the remaining dimensions.

Figure 3 depicts an example of the creation phase in the iterations *Create 1* and *Create 2*. In the *Create 1* iteration, we allocate an uninitialized table in DSM format, with columns A and B, having the same size as the columns of the original table. We start partitioning in the first dimension A. Unlike the Adaptive KD-Tree, the pivot selection is not impacted by the query predicates. We use the average of that piece's dimension, which is calculated during data loading. In the example, the average value of the whole column A is 9. We then scan the original table and copy the first $N * \delta$ rows to either the top or the bottom of the index depending on how they compare to the pivot. In our example, we index half of our table, since $\delta = 0.5$. In this step, we also search for any elements that fulfill the query predicate. Afterward, we scan the not-yet-indexed fraction of the original table to completely answer the query. In subsequent iterations, as depicted in *Create 2*, we scan either the top, bottom, or both pieces of the index based on how the query predicate relates to the chosen pivot. In our example, the running query has a filter $3 < A \leq 8$, and we only need to scan the upper piece of our index. Finally, we copy and pivot the other half of our table to our index.

Refinement Phase. With the original table no longer required to compute queries, we now perform index lookups. While doing these lookups, we further refine the index pieces until they all have become smaller than a given size threshold,

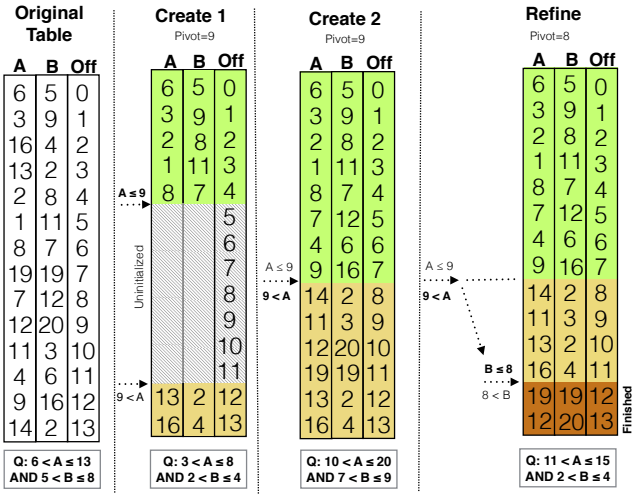


Fig. 3: Progressive KD-Tree with index budget $\delta = 0.5$ and $size\_threshold = 2$. Four queries submitted in the workload.

progressively converging towards a full KD-Tree. We focus on refining pieces of the index required for query processing (i.e., pieces containing query pivots). If these pieces are fully refined (i.e., the pieces containing query pivots children reach a size below $size\_threshold$) and the indexing budget is not over, refinement is continued on a size priority, refining the largest piece first. The refinement is done by recursively performing quicksort operations to swap rows inside the index. Like the creation phase, we also keep track of the sum left and right children of the indexed piece, which is later used as pivots for the children. After all the refinement for that query is completed, we perform a similar index lookup and piece scan as the Adaptive KD-Tree. The only difference is that we need to also take into account pieces where pivoting is not finished.

Figure 3 depicts an example of the refinement phase. In our example, the running query has the filters $10 < A \leq 20$ and $7 < B \leq 9$. A lookup in the index indicates a scan of the bottom piece, and hence that is the piece to be refined on dimension B. We use $\frac{root.right\_sum}{root.end - root.current\_end}$ value as the pivot. In the example, the pivot is the value 8. With $\delta = 0.5$, we are capable of fully refining that piece around 8. Due to our $size\_threshold = 2$, we mark the bottom piece as finished, and no further refinement occurs.

Interactivity Threshold. The user must provide the Progressive KD-Tree with an interactivity time threshold τ and a δ. We distinguish two situations depending on the full scan costs. (1) If a simple scan of the entire table does *not* exceed τ, we use the cost model, presented in the next section, to calculate a δ' such that the first query (incl. indexing a δ' fraction of the data) does not exceed τ. We then use $\delta = min(\delta, \delta')$ for all queries, ensuring that none exceeds τ. (2) If a simple scan of the entire table *does* exceed τ, we use the user-provided δ until the KD-Tree is sufficiently built such that the scan cost per query drop below τ. Then, we calculate a δ' as in situation (1) and proceed as described above.

C. Greedy Progressive KD-Tree

While the δ parameter of Progressive KD-Tree allows us to control both the per-query indexing effort (and hence overhead) and the speed of convergence towards a full index, there is a mutual trade-off. The smaller δ, the lower the overhead, but the slower the convergence; the larger δ, the faster the convergence, but the higher the overhead.

Let t_{scan} denote the time to scan the entire data set, t_{budget} denote the time it takes to pivot/refine a δ fraction of the data set, t'_i denote the net query execution time (i.e., without refining the index) of the ith query Q_i given the current state of the index, and $t_i = t'_i + t_{budget}$ denote the gross execution time (i.e., incl. refining the index) of the ith query Q_i given the current state of the index. The gross execution time t_i of each query with Progressive KD-Tree is bounded by $t_{total} = t_{scan} + t_{budget}$, i.e., $t_i \leq t_{total}$. While this is a tight bound for the first query ($t'_0 = t_{scan} \Rightarrow t_0 = t_{total}$), it gets looser the more queries are being processed and the more of the index is partly constructed, as then the partial index is likely to let queries become faster than a scan ($t'_i < t_{scan} \Rightarrow t_i < t_{total}$). While generally decreasing, t'_i, and hence t_i, can still vary significantly until the index is fully built.

We proposed *Greedy Progressive KD-Tree* as a refinement of Progressive KD-Tree to ensure that, until the index is fully created, each query Q_i has the same gross execution time $t_i = t_0 = t_{total}$, i.e., exploits the full difference between t_{total} and t'_i for indexing. In this way, we speed-up convergence without increasing total query execution time. To do so, we introduce a *cost model* that estimates the net execution time t'_i for each query Q_i and calculates its maximum indexing budget as $t'_{budget,i} = t_{total} - t'_i$, from which we derive δ'_i for each Q_i. The first query uses the user-provided δ, i.e., $\delta'_0 = \delta \Rightarrow t'_{budget,0} = t_{budget}$.

Cost Model. The cost model considers the query and the state of the index in a way that is not affected by different data distributions, workload patterns, or query selectivities. In a nutshell, our cost model can tell how much data will be scanned, hence yielding a conservative δ'_i that guarantees that our query cost will never exceed t_{total}. A conservative δ'_i means the highest possible δ'_i in the worst-case, where any of the construction/refinement boosts the current query execution. However, if the query execution finishes below the t_{total} limit, we perform one extra step called the *reactive* phase to perform an extra indexing until fully consuming the t_{total} limit. The parameters of the Greedy Progressive KD-Tree cost model are summarized in Table I.

Creation Phase. The total time taken in the creation phase is the sum of (1) the index lookup time (i.e., time to access the root node and decide if we scan the top/bottom of our table), (2) the indexing time, and (3) the original table scan.

(1) To calculate the index lookup time, we need to account for the node access and the top/bottom access of each column of our table, where we perform two random accesses $2 * \phi$,one for the root and one to access the indexed table's first column, and $\frac{\alpha * N}{\gamma}$ for the total data we must scan. Since our data has

System	ω	cost of sequential page read (s)
	κ	cost of sequential page write (s)
	ϕ	cost of random page access (s)
	σ	cost of random write (s)
	γ	elements per page
Data set & Query	N	number of elements in the data set
	α	% of data scanned in partial index
	d	number of dimensions
Index	δ	% of data to-be-indexed
	ρ	% of data already indexed
	h	height of the KD-Tree

TABLE I: Parameters for Progressive Indexing Cost Model.

d dimensions, we must account one random access for the additional columns and multiply the sequential scan by $d-1$. The index lookup time is $t_{lookup} = 2 * \phi + \frac{\alpha * N}{\gamma} + (d-1) * \phi$. Simplifying to $t_{lookup} = \frac{\alpha * N}{\gamma} + (d+1) * \phi$.

(2) The indexing time (i.e., index construction time) consists of scanning the base table pages and writing the pivoted elements to the result array. The indexing time is calculated by multiplying the time it takes to scan and write a page sequentially $(\kappa + \omega)$ by the number of pages we need to write summed with the access of each dimension, resulting in $t_{indexing} = (\kappa + \omega) * \frac{N * \delta}{\gamma} + (d-1) * \phi$.

(3) The original table scan, is given by sequentially reading all not-yet-indexed data. The total fraction of the data that remains unindexed is $1 - \rho - \delta$, hence the scan time of the original table is given by $t_{scan} = \frac{(1-\rho-\delta)}{\gamma} * \omega$.

The total time taken for the creation phase is the sum of all three steps, hence $t_{total} = t_{lookup} + t_{indexing} + t_{scan}$ and we set $\delta = \frac{t_{budget}}{(\kappa+\omega) * \frac{N}{\gamma} + (d-1) * \phi}$.

Refinement Phase. In the refinement phase, we no longer need to scan the base table. Instead, we only need to scan the fraction α of the data in the index. However, we now need to (1) traverse the KD-Tree to figure out the bounds of α, and (2) swap elements in-place inside the index instead of sequentially writing them to refine the index. The cost for traversing the KD-Tree is given by the height h of the KD-Tree times the cost of random page access ϕ, resulting in $t_{lookup} = h * \phi$. For the swapping of elements, we perform predicated (i.e., branch-free) swapping [20] to allow for a constant cost regardless of how many elements we need to swap. The total swap cost is the number of elements we can swap times the cost of swapping them, which is two random writes multiplied by the d dimensions, i.e., $t_{swap} = N * \delta * 2 * d * \sigma$. The total cost in this phase is therefore equivalent to $t_{total} = t_{lookup} + \alpha * t_{scan} + t_{swap}$. Finally, we set $\delta = \frac{t_{budget}}{N * 2 * d * \sigma}$ for the adaptive indexing budget.

Interactivity Threshold. With Greedy Progressive KD-Tree, in addition to the mandatory interactivity time threshold τ, the user can additionally provide a "penalty" budget δ or a limit x of queries. We distinguish two situations, depending on the full scan cost. (1) If $t_{scan} < \tau$, we set $t_{total} = \tau$, i.e., ensure that no query exceeds τ, and use our cost model to calculate all $t_{budget,i}$ and δ'_i (incl. the first query's $t_{budget,0}$ and δ'_0) as described above. In this case, we ignore the also provided δ or x. (2) If $t_{scan} \geq \tau$, we distinguish two cases. (2a) In case the user provided a "penalty" budget δ, we start with $t_{total} = t_{scan} + t_{budget}$ with δ, and use our cost model to calculate all $t_{budget,i}$ and δ'_i until the KD-Tree is sufficiently built such that the scan cost per query drop below τ. (2b) In case the user provided a limit x of queries, we use our cost model to calculate the amount of indexing that is required to build a partial KD-Tree such that the remaining scan cost per query are less than τ, distribute this indexing work over x queries, and calculate how much indexing budget $t_{budget++}$ is need for each query. With this, we proceed as in (2a) for the first x queries. In both cases, (2a) & (2b), we then proceed with the user-provided τ as in situation (1).

IV. Quantitative Assessment

In this section, we provide a quantitative assessment of our proposed adaptive/progressive indexes. This section is divided into four parts. First, we define all real and synthetic data sets and workloads used in our assessment. Second, we analyze the impact of δ on the Progressive KD-Tree in terms of first query cost, pay-off, time until full convergence, and total time. Third, we provide an in-depth performance comparison of our proposed adaptive/progressive indexes and analyze their behavior under three real and eight synthetic workloads. We also provide comparisons with the state-of-the-art on multidimensional adaptive indexing QUASII (Q) and use two variations of a full KD-Tree index as a baseline. The first one using the average value of a piece as the pivot (AvgKD), and the second one using medians (MedKD). Finally, we study the behavior of our algorithms when the full scan cost is higher than the interactivity threshold.

Setup. All indexes were implemented in a stand-alone C++ program. All the data is 4-byte floating-point numbers stored in a columnar format (i.e., DSM). The code was compiled using GNU g++ version 9.2.1 with optimization level -O3. All experiments were conducted on a machine with 256 GB of main memory, an Intel Xeon E5-2650 with 2.0 GHz clock, and 20 MB of L3 cache size.

A. Data Sets & Workloads

We use four different data sets in our assessment.

Power. The power benchmark consists of sensor data collected from a manufacturing installation, obtained from the DEBS 2012 challenge[3]. The data set has three dimensions and 10 million tuples. The workload consists of random close-range queries on each dimension, a total of 3000 queries.

Skyserver. The Sloan Digital Sky Survey is a project to map the universe. Their data and queries are publicly available at their website[4]. The data set we use here consists of two columns, *ra* and *dec*, from the *photoobjall* table with approximately 69 million tuples. The workload consists of 100,000 real range queries executed on those two attributes.

Genomics. The 1000 Genomes Project collects data regarding human genomes. It consists of 10 million genomes,

[3]https://debs.org/grand-challenges/2012/

[4]http://skyserver.sdss.org

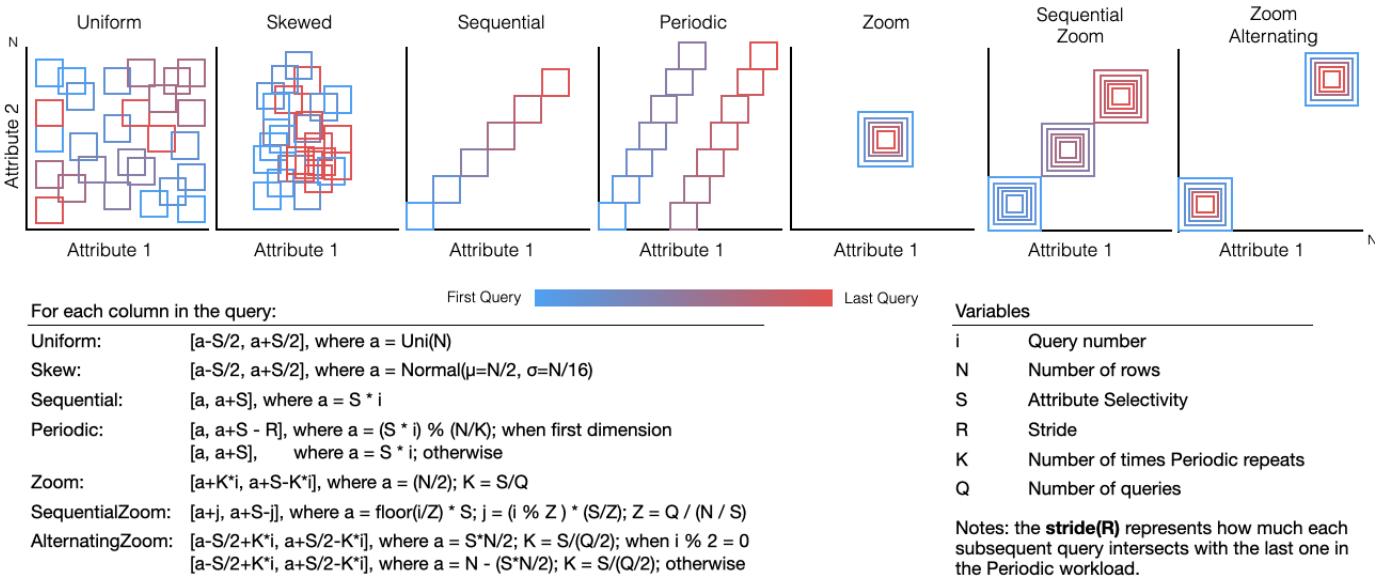


Fig. 4: Visual representation of the different synthetic workloads.

described in 19 dimensions. The workload consists of 100 queries constructed by bio-informaticians.

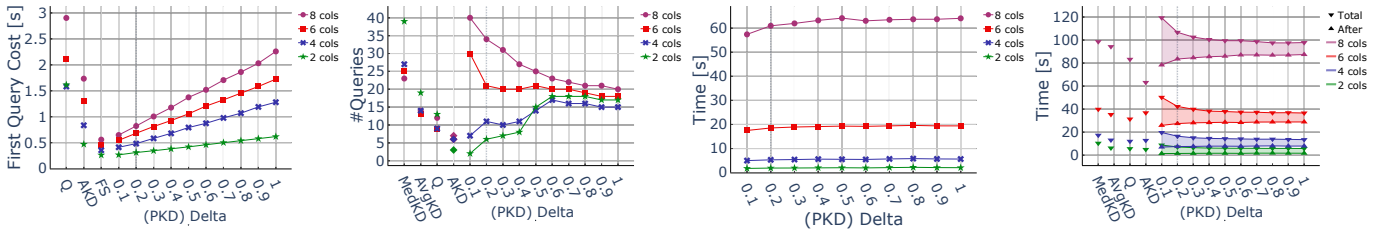
Uniform. It follows a uniform data distribution for each attribute in the table, consisting of 4-byte floating-point numbers in the range of $[0, N)$, where N is the experiment's number of tuples. We use eight different synthetic workloads in our performance comparison, similar to the ones described in Holanda et al. [7] but extended for the multidimensional case. Figure 4 depicts a two-dimensional example of these workloads with the mathematical formulas used to generate them. In addition to these workloads, we propose a new one, called *shifting*. The shifting workload represents a common scenario in data exploration where the columns being queried change constantly (e.g., the data scientist executes ten queries on three columns, which leads him to investigate other three columns, and so forth). When generating a synthetic workload, we take as parameter the overall query selectivity σ. To keep σ constant, regardless of the number d of dimensions used, we set the per-dimension selectivity with d dimensions to $\sigma_d = \sqrt[d]{\sigma}$; e.g., for $\sigma = 1\%$, we get $\sigma_2 = 10\%, \sigma_4 = 31\%, \sigma_6 = 46\%, \sigma_8 = 56\%$.

B. Impact of delta (δ) on Progressive KD-Tree

The parameter δ defines a percentage of the total amount of our data that is pivoted per query. If $\delta = 0$, no indexing is performed, hence only full scans are executed, and the index will never converge. On the other hand, if $\delta = 1$, the creation phase completes in the first query, with the data fully pivoted once in the first dimension. In this section, we explore how δ impacts our index in terms of the burden on the first query, how many queries it takes for the index to pay-off when compared to a full scan, how much time it takes until full index convergence, and the impacts on cumulative time for the entire workload. We use a uniform data set and workload, with 30 million rows, $d \in \{2, 4, 6, 8\}$ columns, and 1000 queries with 1% selectivity. We test with multiple δ values, ranging from 0.1 to 1. Where applicable, we compare Progressive KD-Tree (PKD) with Adaptive KD-Tree (AKD), QUASII (Q), Average/Median KD-Tree (AvgKD/MedKD), Full Scan (FS). Both Average and Median KD-Tree are built using the attribute order given by the table schema.

First Query. The first query cost is the cost of fully scanning the data with addition of copying and pivoting a δ-fraction of the data. Figure 5a depicts the first query cost over varying δ for multiple columns. With Progressive KD-Tree, the cost increases linearly as we increase δ, and hence the amount of indexed data, with the impact being larger the more columns are involved, i.e., the more data needs to be copied. With $\delta = 0$, the first query merely performs a Full Scan. The first query cost for Adaptive KD-Tree is about the same as for Progressive KD-Tree with $\delta \in [0.6, 0.7]$. The first query cost of QUASII is significantly higher that those of both Adaptive and Progressive KD-Tree due to the more intensive refinement work of QUASII. For Average KD-Tree and Median KD-Tree, the first query costs grow linearly with the number of columns. We omit them from Figure 5a as bulding the entire index is far more expensive than any query shown there.

Pay-off. In this experiment, we define pay-off as the number q of queries required until investing in incrementally building the Progressive KD-Tree pays off compared to performing only full scans without indexing, i.e., the smallest q for which $\sum_{i=0}^{q} t_{i,progKD} \leq \sum_{i=0}^{q} t_{i,FScan}$. Figure 5b depicts the pay-off for multiple dimensions. While a small δ limits the indexing impact over the initial queries, it also limits and the indexing progress. For workloads with high per-column selectivity, this results in the queries being capable of taking advantage of the little index progress early on. However, in a workload with a low per-column selectivity (e.g., with 8 columns, we need a per-column selectivity of 56% to yield an overall query selectivity of 1%), this results in the queries not being able to take advantage of the indexing early on. For example, with $\delta =$



(a) First query cost. (b) #Queries until Pay-off. (c) Time until Convergence. (d) Cumulative time (1000 queries).

Fig. 5: Impact of δ on Progressive KD-Tree.

0.1 it takes 10 queries to pivot the first node fully. Since in our experiment, we use a uniform data set, and the Progressive KD-Tree uses averages as pivots, that results in a pivot that partitions the data on two pieces with approximately 50% of the total data. In the case of an 8 dimensional workload with per-column selectivity of 56%, the workload is not able to take advantage of the index for the first 10 queries. Hence, the initial queries always perform index creation and full scans, resulting in a higher pay-off when compared to lower per-column selectivities. Furthermore, a higher δ reduces the limitation on the index progress, creating an index that can boost queries early on and diminishing the number of queries for the pay-off. Focusing on only the minimal indexing for the given workload, Adaptive KD-Tree pays-off as early as the quickest variant of Progressive KD-Tree ($\delta = 0.1$).

Convergence. The Convergence is defined as the time, in seconds, it takes for the Progressive KD-Tree to fully index the data and achieve the same query performance as the Average KD-Tree. Figure 5c depicts the convergence for multiple dimensions. The time to converge increases with the number of dimensions, because the average query time also increases. However, since δ determines a percentage of the data that is indexed per query, the number of dimensions has no impact on number of queries to converge. For example, with $\delta = 0.1$ the number of queries to converge is about 100, independent on the number of columns.

Total Response Time. In Figure 5d, downward-pointing triangles ("Total") mark the cumulative times to execute the entire workload of 1000 queries, while upward-pointing triangles ("After") mark the cumulative times for only the tail of the workload after the index is fully built and used for optimal query performance, i.e., no further index refinement is performed. The shaded range between both indicates the cumulative time until the index is fully built. Progressive KD-Tree takes at most 103 queries to converge to a full index with $\delta = 0.1$, or even as a mere 10 queries with $\delta = 1$. Consequently, 90% ($\delta = 0.1$) to 99% ($\delta = 1$) of the 1000 queries in the workload benefit from the fully-built index, accounting for the majority of the cumulative execution time due to their number rather than per-query time. Only between 1% ($\delta = 1$) and 10% ($\delta = 0.1$) of the workload contribute to progressively constructing the index. For the non-progressive techniques, we only show the "Total" workload time, without breaking it down into before and after convergence. Adaptive KD-Tree and QUASII never converge in this experiment, while Average KD-Tree and Median KD-Tree converge with the first query by design. Overall, with $\delta \geq 0.2$, Progressive KD-Tree yields about the same total workload time as the non-progressive techniques. Only in the 8-dimensional scenario, QUASII and Adaptive KD-Tree outperform Progressive KD-Tree.

Picking a Delta (δ). For exploratory data analysis, our indexes must not impose a high burden over the initial queries, while still paying off their investments quickly and preferably converging fast and presenting a low total cost. Taking these objectives in mind, we select a $\delta = 0.2$ for our performance comparisons. It offers a sharp decrease in total cost and convergence when compared to $\delta = 0.1$, without a significant increase in cost in the first query.

C. Performance Comparison

In the remainder of the experimental section, we will focus on comparing the performance of our three proposed indexes, the Adaptive KD-Tree (AKD), the Progressive KD-Tree (PKD), and the Greedy Progressive KD-Tree (GPKD) with the state-of-the-art. In particular, we compare it with QUASII (Q) and two KD-Tree full-index implementations, the Average KD-Tree (AvgKD) that uses the average value of pieces as pivots and the median KD-Tree (MedKD) that uses the median values as pivots. We also test a Full Scan (FS) implementation using candidate lists as the baseline.

We verify four main characteristics that are desirable in indexing approaches for multidimensional exploratory data analysis. (1) The first query cost. (2) The number of queries executed so the investment performed on index creation pays-off. (3) The workload robustness. (4) The total workload cost. To evaluate our indexes we execute all workloads as described in Section IV-A.

We execute the real workloads as given. For the Synthetic workloads, we generate $d = 8$ dimensions, with 300 million tuples for Uniform, Skewed, SequentialZoom, and 50 million tuples for all others. All queries have $\sigma = 1\%$ overall selectivity, while the per-dimension selectivity for all columns is $\sigma_8 = 56\%$. The only exception is the sequential workload, where we only generate two dimensions with $\sigma_2 = 0.1\%$. This is because with the sequential workload, query ranges must not overlap; with more than two attributes the per attribute selectivity is too big, and using query selectivity $\sigma = 1\%$ would yield only

		MedKD	AvgKD	Q	AKD	PKD(.2)	GPKD(.2)	FS
50M	Unif(8)	20.20	12.46	5.11	3.07	1.36	1.36	0.91
	Skewed(8)	20.23	12.48	6.25	3.49	1.26	1.26	0.82
	Zoom(8)	20.28	12.68	6.13	3.24	1.32	1.31	0.84
	Prdc(8)	20.17	12.42	6.99	6.94	0.99	1.00	0.60
	SeqZoom(8)	19.98	12.42	5.23	2.90	1.42	1.41	0.93
	AltZoom(8)	20.18	12.43	6.98	6.93	0.99	1.00	0.60
	Shift(8)	20.20	12.46	5.11	3.07	1.36	1.36	0.91
	Seq (2)	15.88	8.30	4.01	0.68	0.26	0.26	0.19
Real	Power	1.52	0.83	0.33	0.23	0.08	0.08	0.06
	Genomics	2.58	2.62	1.25	0.99	0.27	0.27	0.03
	Skyserver	14.31	6.84	1.19	0.63	0.36	0.35	0.26
300M	Unif(8)	146.72	83.91	37.25	20.93	8.17	8.17	5.47
	Skewed(8)	146.80	84.01	43.06	21.24	7.94	7.96	5.12
	SeqZoom(8)	146.87	84.36	35.93	18.08	8.84	8.83	6.41

TABLE II: First query response time (Seconds).

		MedKD	AvgKD	Q	AKD	PKD(.2)	GPKD(.2)
50M	Unif(8)	22.19	13.57	11.12	6.83	31.41	22.88
	Skewed(8)	23.67	14.42	9.90	5.44	36.06	28.06
	Zoom(8)	31.25	18.54	6.19	3.26	39.50	30.19
	Prdc(8)	22.00	13.47	7.08	7.09	29.14	22.53
	SeqZoom(8)	21.22	13.20	5.27	2.91	32.00	24.39
	AltZoom(8)	21.53	13.15	8.12	7.57	19.15	26.46
	Shift(8)	2094.98	1319.28	1085.27	26.34	1152.43	1263.61
	Seq (2)	15.89	8.30	4.07	51.17	1.93	7.62
Real	Power	1.79	0.96	0.81	0.41	1.04	1.80
	Genomics	6.41	6.49	9.06	6.09	16.16	17.69
	Skyserver	14.32	6.84	1.24	0.75	2.91	9.40
300M	Unif(8)	154.82	87.70	74.92	40.52	197.89	160.04
	Skewed(8)	159.33	88.26	65.96	32.97	229.73	180.63
	SeqZoom(8)	151.92	91.32	36.17	18.17	185.14	155.27

TABLE III: Pay-off (Seconds).

10 disjoint queries, hence, we decrease overall selectivity to $\sigma = 0.0001\%$ which yields 1000 disjoint queries.

We use $size\_threshold = 1024$ tuples as a minimum partition size for all indexes. Unless stated otherwise, all progressive indexing experiments use an interactivity threshold equal to the first query cost of Progressive KD-Tree with $\delta = 0.2$.

First Query. Table II depicts the first query cost of all algorithms on all workloads. The Median KD-Tree and the Average KD-Tree present the highest times on the first query since they create a full index when we query a group of columns for the first time. The Median KD-Tree usually presents a higher cost since finding the median of a piece is more costly than finding the average value. The adaptive indexes are up to one order of magnitude cheaper than the full indexes since they only index a focused region necessary to answer the query. QUASII has a more aggressive partitioning algorithm than the Adaptive KD-Tree (for example, in the first query of the uniform workload the Adaptive KD-Tree creates 161 nodes while QUASII creates 13,480) and, thus, ends up being a factor 2 slower in the first query evaluation. Finally, both progressive indexing solutions have the same time on the first query, since they execute it with the same δ. They impose the smallest burden on the first query and are up to one order of magnitude faster than the adaptive indexing solutions.

Pay-off. Table III depicts the time it takes for the investment spent on index creation to pay-off when compared to a full scan only scenario. For the full index approaches, the Average KD-Tree presents a smaller pay-off than the Median KD-Tree due to a lower cost on index creation while maintaining a similar cost on index lookup. In the adaptive solutions, the Adaptive KD-Tree has the lowest pay-off, not only when compared to QUASII but overall, this is a direct result from its core design of only indexing the pieces necessary for the executing query, while QUASII performs a more aggressive refinement strategy that increases its pay-off. The Adaptive KD-Tree has the worst pay-off in the sequential workload, which represents its worst-case scenario. Finally, the progressive solutions present the highest pay-off in general, however it is important to notice that we picked our δs optimizing for a low burden in the first query. Since most experiments are with 8 columns, as depicted in Figure 5b to optimize for a low pay-off we would need to use larger δs. One can notice, that the progressive solutions perform the best on the sequential workload, due to the low number of columns benefiting from the small δ. One can notice that for the Shift(8) workload, no algorithm besides the Adaptive KD-Tree pays-off due to the low number of queries executed before shifting the columns we are looking into.

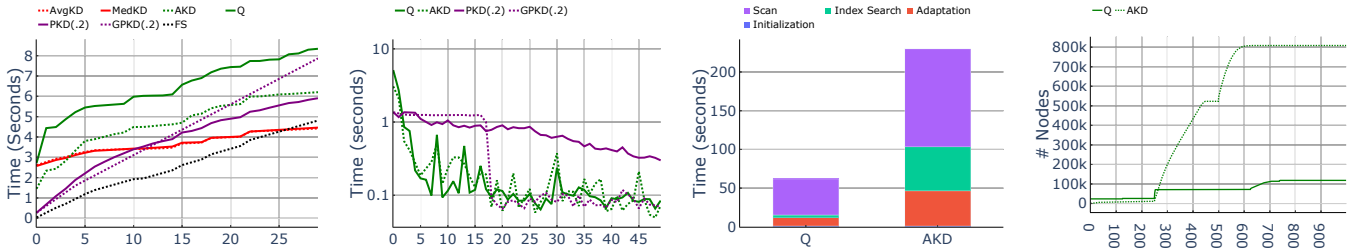
		Q	AKD	PKD(.2)	GPKD(.2)
50M	Unif(8)	6E-01	2E-01	9E-02	1E-03
	Skewed(8)	8E-01	2E-01	8E-02	2E-03
	Zoom(8)	7E-01	2E-01	8E-02	1E-03
	Prdc(8)	1E+00	9E-01	4E-02	6E-04
	SeqZoom(8)	5E-01	2E-01	1E-01	2E-03
	AltZoom(8)	1E+00	9E-01	8E-02	6E-04
	Shift(8)	2E+00	9E-01	3E-02	1E-03
	Seq (2)	3E-01	3E-03	1E-03	8E-05
Real	Power	3E-03	1E-03	6E-04	3E-05
	Genomics	2E-01	6E-02	1E-02	9E-04
	Skyserver	4E-02	8E-03	4E-03	2E-04
300M	Unif(8)	3E+01	1E+01	4E+00	3E-02
	Skewed(8)	4E+01	9E+00	3E+00	3E-02
	SeqZoom(8)	3E+01	6E+00	4E+00	5E-02

TABLE IV: Query time variance (smaller is better).

Figure 6a depicts the cumulative response time of the first 30 queries in the Genomics Benchmark. When compared to full indexes, both adaptive and progressive indexes take longer to pay-off and achieve full index response time. This is due to the full indexes having a low first query cost as discussed in the first query sub-section.

Robustness. To calculate the robustness we check the variance in per-query cost, for the first 50 queries or up to full index convergence. For full indexes, the variance is 0, because it fully converges in the first query. Table IV depicts the robustness of all adaptive and progressive algorithms. The Adaptive KD-Tree is as robust as QUASII. The progressive indexing solutions are the most robust options, with up to 3 orders of magnitude lower variance than the adaptive indexing approaches, with the Greedy Progressive KD-Tree always being the most robust, with a constant per-query cost until convergence due to its cost model adaptive δ (Fig. 6b).

Total Response Time. Table V depicts the total response time of all benchmarks. The Progressive indexing approaches have a very similar response time when compared to the



(a) Cumulative response time. Genomics, first 30 queries.

(b) Per query response time. Uniform(8), first 50 queries.

(c) Time breakdown. Periodic(8).

(d) Index size per query. Periodic(8), first 1000 queries.

Fig. 6: Total Response Time, Per-Query Cost, and Index Size Comparison.

		MedKD	AvgKD	Q	AKD	PKD(.2)	GPKD(.2)	FS
50M	Unif(8)	109.7	101.4	95.6	74.3	122.6	109.9	857.5
	Skewed(8)	147.6	138.3	107.6	43.1	160.8	151.1	856.6
	Zoom(8)	52.0	40.9	11.4	7.1	58.5	51.6	687.1
	Prdc(8)	85.8	73.6	61.9	229.9	93.3	86.4	807.7
	SeqZoom(8)	31.0	24.2	8.2	4.5	46.6	34.1	499.6
	AltZoom(8)	44.0	34.2	18.9	22.4	53.4	48.3	747.0
	Shift(8)	2095.0	1319.3	1085.3	775.5	1152.4	1263.6	885.5
	Seq (2)	15.9	8.3	6.0	102.9	7.8	7.6	332.6
Real	Power	26.0	24.4	24.6	31.3	25.0	24.7	164.6
	Genomics	10.9	10.9	10.6	7.3	16.2	17.7	16.1
	Skyserver	16.0	14.1	6.9	12.0	10.7	10.4	20186.5
300M	Unif(8)	468.8	366.9	422.9	352.0	558.4	472.7	5423.8
	Skewed(8)	581.9	399.8	521.0	195.2	674.9	595.9	5367.1
	SeqZoom(8)	183.0	122.5	48.7	24.5	277.3	186.0	3221.2

TABLE V: Total response time (Seconds).

		MedKD	AvgKD	Q	AKD	PKD(.2)	GPKD(.2)	FS
Unif(2)	First Query	15.94	8.35	2.89	1.05	0.55	0.54	0.52
	PayOff	16.05	8.40	5.56	1.63	1.94	8.18	-
	Convergence	-	-	*	*	9.68	7.78	-
	Robustness	-	-	0.20	0.02	0.01	0.00	-
	Time	19.08	11.49	10.76	9.34	12.75	11.24	425.34
Unif(4)	First Query	17.13	9.56	3.14	1.65	0.83	0.82	0.65
	PayOff	17.33	9.66	5.80	3.26	4.65	11.40	-
	Convergence	-	-	*	*	14.47	10.66	-
	Robustness	-	-	0.20	0.08	0.03	0.00	-
	Time	25.27	17.72	17.13	18.32	22.32	19.39	614.59
Unif(8)	First Query	20.20	12.46	5.11	3.07	1.36	1.36	0.91
	PayOff	22.19	13.57	11.12	6.83	31.41	22.88	-
	Convergence	-	-	*	*	38.02	21.34	-
	Robustness	-	-	0.60	0.20	0.09	0.00	-
	Time	109.69	101.41	95.59	74.27	122.60	109.90	857.54
Unif(16)	First Query	45.10	36.99	29.19	10.85	2.07	2.05	1.30
	PayOff	223.96	173.06	50.65	35.64	183.21	185.68	-
	Convergence	-	-	*	*	96.14	74.17	-
	Robustness	-	-	20.00	3.00	0.03	0.08	-
	Time	1054.69	1023.24	461.45	260.02	1026.44	1029.89	1258.90

TABLE VI: Performance difference on Uniform benchmark with different number of attributes.

full indexes, due to their design characteristics prioritizing robustness and convergence over total response time, that is reinforced by the low δ picked for the experiments. Adaptive indexing always has the lowest total response time, due to their high focus on refining pieces requested by the currently executing query, with the Adaptive KD-Tree presenting the fastest results for the majority of the workloads. The exception is for highly skewed workloads (e.g., Alternating Zoom and SkyServer), which is due to QUASII's extra refinement paying-off almost immediately, and in the Periodic and Sequential Benchmarks. Figure 6c presents the total cost breakdown of the Periodic benchmark for both adaptive indexes. QUASII presents a lower adaptation, scan, and index search costs, indicating that our index is constantly performing refinement with the queries taking almost no benefit from it. Figure 6d depicts the number of created nodes throughout the workload execution, one can notice the sudden increases on node number over 250 and 500 nodes, due to the restart of the periodic pattern. This particular workload causes the Adaptive KD-Tree always to visit unrefined pieces on the following queries, causing its high refinement costs and the insertion of many nodes. The latter causes a high cost for index lookup search.

The Sequential benchmark emulates the worst case scenario for the Adaptive KD-Tree, where the KD-Tree ends up almost equal to a linked list. This happens due to blindly adapting using the query predicates and because the KD-Tree has no self-balancing mechanism. The Shifting benchmark also presents a peculiar result, where the only index with a response time that is faster than the full scan is the Adaptive KD-Tree with its workload-dependent refinement approach quickly paying off for such a small window of queries.

D. Impact of Dimensionality

In this section, we evaluate how the number of dimensions affects the performance of each technique. We experiment with a uniform workload of 1000 queries with 1% selectivity, on a uniform data set with 2, 4, 8 and 16 columns. Table VI depicts the first query cost, time to pay-off, time until convergence, robustness, and total execution time for each index. Similar to the results presented in the previous section, the Average KD-Tree has the upper hand in terms of total cost and number of queries until pay-off, while the Progressive KD-Trees are the most robust with a predictable convergence. One can notice that as the number of dimensions increases, the difference in total time and pay-off between the Adaptive Indexing solutions and the Progressive Indexing increases drastically. This happens due to the convergence principle of progressive indexing, which causes it to behave similarly to a full index.

E. Full Scan Exceeding the Interactivity Threshold

Figure 7 depicts the behavior of the Adaptive KD-Tree (AKD), the Progressive KD-Tree (PKD), and both options for the Greedy Progressive KD-Tree, with a fixed number of queries as input (GPFQ) and a fixed penalty (GPFP).

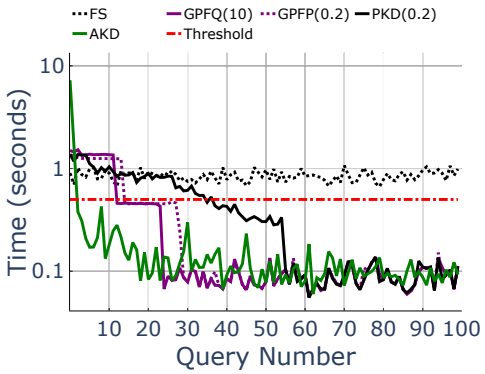


Fig. 7: Adaptive and Progressive KD-Tree with scans costs exceeding the interactivity threshold; first 100 queries.

For this experiment, we set our interactive threshold to 0.5s, approximately half the cost of a full scan. AKD performs the necessary indexing as a pre-processing step during the first query. Hence its first query is one order of magnitude more expensive than a full scan. Due to this investment, all remaining queries are under the threshold. PKD starts with the user-provided δ of 0.2 and gradually reaches a scan cost below the interactivity threshold. At that point, it calculates a new δ', which gradually converges to a full index. Both GPFQ and GPFP have similar behavior, they start on a cost higher than the interactivity threshold, have a sudden drop to the threshold cost, and later one more drop until full convergence. For GPFQ, this first drop happens after ten queries, as requested by the user, at the expense of slightly higher first query costs than GPFP. GPFP uses an indexing penalty of $\delta = 0.2$, and only drops once pieces are small enough, slightly later than GPFQ.

V. Conclusions & Future Work

In this paper, we extended existing work on multidimensional adaptive indexing by introducing three new indexing algorithms, one adaptive and two progressive ones. We showed that our algorithms are superior when compared with the state-of-the-art multidimensional indexing in a variety of real and synthetic workloads. In general, we notice that the Adaptive KD-Tree is the fastest solution due to its minimum indexing property: Indexing only what is strictly necessary to answer the query. Both Progressive KD-Tree's present the lowest penalty on the initial queries, with the Greedy Progressive KD-Tree yielding the fastest convergence and best robustness. In general, which technique to use depends on the properties desired by the user, if the ultimate goal is the total cost, the Adaptive KD-Tree is the algorithm of choice. However, in exploratory data analysis, where we want to keep the impact on initial queries low and we want a constant query response time without performance spikes, Greedy Progressive KD-Tree is the algorithm of choice.

As future work, we point out the following directions:

Adaptive/Progressive Table Partitioning: A similar reorganization strategy can be extended for the original table's data instead of creating a secondary index structure. This would increase the usability of the data reorganization since the multidimensional indexes will suffer from tuple reconstruction costs when accessing non-indexed tuples.

Approximate Adaptive/Progressive Indexing: Keeping query times interactive becomes a larger challenge the larger the data sets get. To truly achieve interactive times also with huge data sets, adaptive/progressive indexing would need to be integrated with approximate query processing, and construct the index while accessing samples of the data. The advantage is that the further the index is progresses, the more precise the approximation would be.

Acknowledgment

This work was funded by the Netherlands Organisation for Scientific Research (NWO), projects "Maintenance prediction for industries" (Nerone), "Data Mining on High-Volume Simulation Output" (Holanda) and "SIMCTIC/Min. Comunicações" (Almeida).

References

[1] Thibault Sellam, Emmanuel Müller, and Martin Kersten. Semi-Automated Exploration of Data Warehouses. In *CIKM*, pages 1321–1330, 2015.
[2] Zhicheng Liu and Jeffrey Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Trans. Vis. Comput. Graph.*, 20:2122–2131, 2014.
[3] Douglas Comer. The Difficulty of Optimum Index Selection. *TODS*, 3(4):440–445, 1978.
[4] Nicolas Bruno. *Automated Physical Database Design and Tunning*. CRC-Press, 2011.
[5] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. Database Cracking. In *CIDR*, volume 3, pages 1–8, 2007.
[6] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The uncracked pieces in database cracking. *PVLDB*, 7(2):97–108, 2013.
[7] Pedro Holanda, Mark Raasveldt, Stefan Manegold, and Hannes Mühleisen. Progressive indexes: indexing for interactive data analysis. *PVLDB*, 12(13):2366–2378, 2019.
[8] 1000 Genomes Project Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.
[9] Alexander S Szalay, Jim Gray, Ani R Thakar, Peter Z Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. The SDSS skyserver: public access to the sloan digital sky server data. In *SIGMOD*, pages 570–581, 2002.
[10] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. Quasii: query-aware spatial incremental index. In *EDBT*, pages 325–336, 2018.
[11] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
[12] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, 1990.
[13] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.
[14] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Comm. of the ACM*, 18(9):509–517, 1975.
[15] Tilmann Zäschke, Christoph Zimmerli, and Moira C. Norrie. The ph-tree: a space-efficient storage structure and multi-dimensional index. In *SIGMOD*, pages 397–408, 2014.
[16] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. *CoRR*, 2019.
[17] Stefan Sprenger, Patrick Schäfer, and Ulf Leser. Multidimensional range queries on modern hardware. In *SSDBM*, pages 4:1–4:12, 2018.
[18] Ahmed M Aly, Ahmed R Mahmood, Mohamed S Hassan, Walid G Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. Aqwa: adaptive query workload aware partitioning of big spatial data. *PVLDB*, 8(13):2062–2073, 2015.
[19] Sheng Wang, David Maier, and Beng Chin Ooi. Fast and adaptive indexing of multi-dimensional observational data. *PVLDB*, 2016.
[20] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.