Progressive Indexes

Timbó Holanda, P.T.

CHAPTER 4

Multidimensional Progressive Indexing

1 Introduction

As seen in the previous chapter, techniques like Adaptive Indexing [36, 50] and Progressive Indexing (Chapter 3) aim to alleviate the index construction issue on exploratory workloads by creating partial unidimensional 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 unidimensional workloads. For instance, the 1000 genomes project [16] has human genetic information, the Power data set[1] that contains sensor information from a manufacturing installation, and the SkyServer data set [56] which maps the universe, are some of many examples that perform multidimensional filtering.

Pavlovic et al. [45] published a study on multidimensional adaptive indexes, initially testing a Space-Filling Curve strategy, where multiple dimensions are mapped to one dimension. They used unidimensional adaptive indexing techniques on top of the created map. However, the first queries' indexing burden was too high, making this approach unfeasible for interactive times. They later propose the QUASII index, a d-level hierarchical structure that similarly partitions the data as the coarse granular index strategy [50]. When accessing one piece, the data is continuously refined until all pieces are smaller than a given size threshold. This strategy is much more efficient

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

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 chapter introduces two novel algorithms to tackle the problem of multidimensional adaptive indexing under exploratory data analysis. (a) 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. (b) 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.

1.1 Contributions

The main contributions of this chapter are:

- 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 chapter.

1.2 Outline

This chapter is organized as follows. In Section 2, we investigate related research that has been performed on multidimensional indexes. In Section 3, we describe our novel Multidimensional Progressive Indexes. In Section 4, we perform a quantitative assessment of each of the novel methods we introduce, and we compare them with

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

the state-of-the-art on multidimensional adaptive indexing under three real workloads and eight synthetic workloads. Finally, in Section 5, we draw our conclusions.

2 Related Work

In the previous chapter we discussed how the selection of which indexes to create has been a long-standing problem in database automatical physical design. However, the selection of the indexes is just part of the problem. Another equally important problem is selecting which data-structure to use since each structure is catered to different workload patterns and data distributions. Multidimensional access methods can be distinguished between point access methods (PAMs) and spatial access methods (SAMs) [18]. Typically, PAMs aim at databases storing only points with support to spatial searches on them, like KD-Trees, PH-Tree, and flat structures. The term point refers to both locations in space and point objects stored in the database. SAMs, like R-Trees and Z-Ordering, aim at extended objects (e.g., polygons in geographic databases) while, like PAMs, also storing points [55].

In this section, we briefly discuss the state-of-the-art multidimensional index structures.

2.1 Multidimensional Data Structures

R-Tree [24]

The R-Tree 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 [5], for read-mostly workloads that balances the rectangle coverage and reduces overlapping.

VA File [61]

The VA File 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 [7]

The KD-Tree 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 [63]

The PH-Tree 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.

This approach is advantageous in data sets where data points are not evenly spread and share many prefixes. Otherwise, spread out data with large number of dimensions increases the number of nodes and the depth of the prefix tree, which also increases the space requirements and the lookup time.

Flood [42]

Flood 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. [55]. 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 our progressive solution. 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.

2.2 Adaptive/Progressive Index

In this section, we discuss the state-of-the-art adaptive indexing techniques that produce multidimensional indexes.

Space Filling Curve Cracking [45]

Space Filling Curve Cracking 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 unidimensional adaptive/progressive indexing techniques. Later on, queries also must be translated to this unidimensional mapping.

QUASII [45]

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.

Adaptive KD-Tree [43]

The *Adaptive KD-Tree* is a multidimensional adaptive indexing technique that follows the same principles as Adaptive Indexing [36]: (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

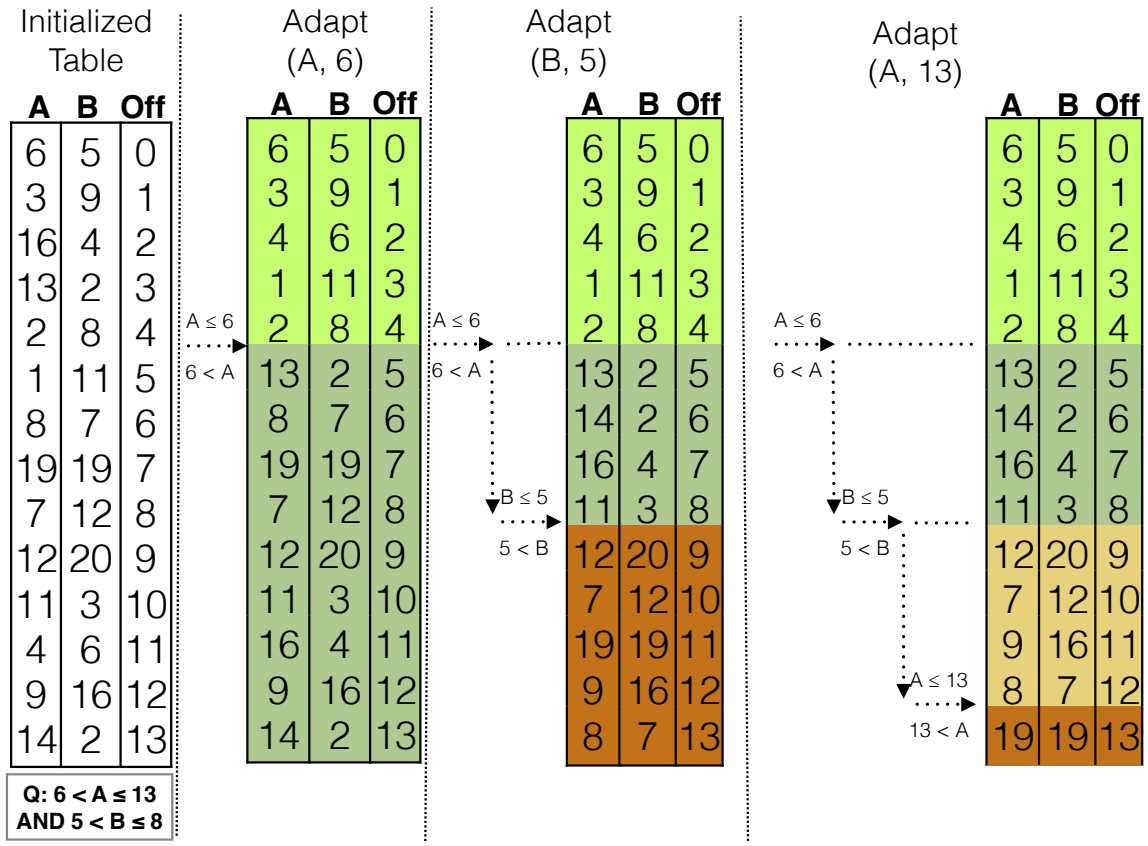

Figure 4-1: Adaptive KD-Tree: The adaptation phase with query: $6 < A \leq 13$ AND $5 < B \leq 8$, and *size_threshold* $= 4$.

columns is executed. In this phase, it creates a copy of the original table into our index table. In the *adaptation* phase, it swaps rows in the index table to partition it according to the query predicates.

Figure 4-1 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 by 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 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 $(B, 8)$.

Discussion.

Space-Filling Curve Cracking is the first attempt to perform adaptive indexing of multiple columns. However, as demonstrated by Pavlovic et al. [45], 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. The Adaptive KD-Tree has a less aggressive refinement strategy than QUASII. However, it still does not present the required fined-grained indexing to mitigate the robustness problem, as Progressive Indexing has. Finally, other techniques are self-proclaimed multidimensional adaptive indexes, like AQWA [3] and SICC [59]. 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.

3 Multidimensional Progressive Indexing

The *Progressive KD-Tree* is a multidimensional progressive indexing technique inspired by Progressive Quick-Sort (Chapter 3). 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 4 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 first dimension's average value. 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.

3.1 Data Structure

Listing 3 KD-Node for Progressive Indexing

```
1  template <typename T>
2   struct KDNode {
3      T key;
4      unsigned int discriminator_attribute;
5      struct KDNode* left_child;
6      struct KDNode* right_child;
7      unsigned int start;
8      unsigned int end;
9      unsigned int cur_start;
10     unsigned int cur_end;
11     unsigned int left_sum;
12     unsigned int right_sum;
13  };
```

Instead of using a standard KD-Tree node in our data structure, our Progressive KD-Tree uses a slightly extended KD-Tree node structure, as depicted in Listing 3. The standard elements are a key, a discriminator attribute, and two pointers for the left and right children (Lines 3-6). Since partitioning a single piece (i.e., splitting a single node) can take multiple queries, we cannot simply keep a single offset pointing to the final pivot location. Instead, we need to store the offsets marking the boundaries of the piece at hand (Lines 7-8) as well as the offsets marking the progress of the pivoting so far (Lines 9-10). Once a piece has been fully pivoted, the latter two offsets are identical and mark the pivot's location. In lines 11-12, we define the variables

that sum the value of the next to-be-partitioned dimension. We use these values to calculate the average of each piece for the next dimension, for example, the left pivot is $\frac{left\_sum}{cur\_start - start}$.

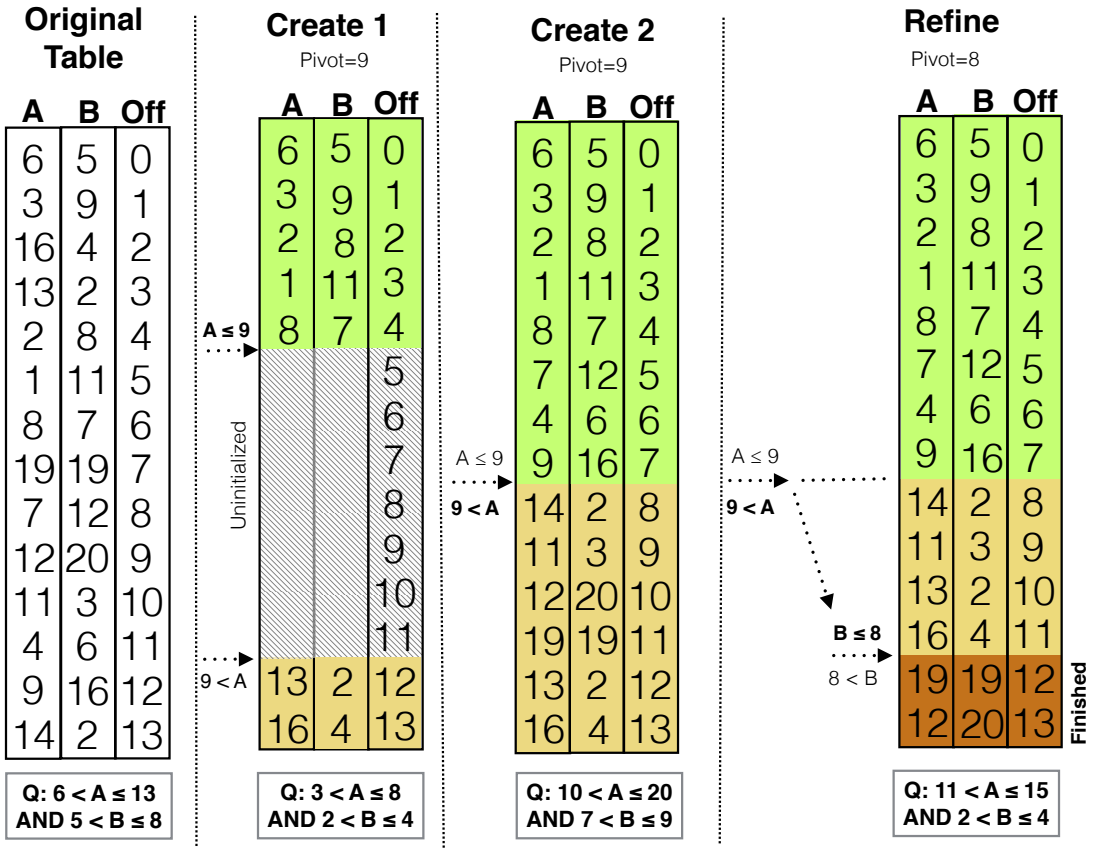

Figure 4-2: Progressive KD-Tree with index budget $\delta = 0.5$ and *size_threshold* $= 2$. Four queries submitted in the workload.

3.2 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 4-2 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 original table columns. 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 answer the query completely. 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.

Listing 4 details the creation phase. In lines 2-13, we initialize all necessary variables to compute our candidate list and store elements in the correct position related to the pivot. In lines 2-5, we select the original column, index column, and the query pivots for the dimension discriminated by the root node. In lines 6-7, we create the variables that hold the offsets of both upper and bottom indexed pieces and update the *root.cur_start* and *root.cur_end* after finishing the execution. Line 8 stores an offset to the original table that indicates the last row that was indexed. Line 9 subtracts from our budget the amount of data that will be indexed in this iteration. Lines 11-13 initialize the candidate list that will result from the creation phase and the *go_down* bit vector that for each row keeps track of whether pivoting moves that row to the top part or bottom part of the refined piece. In lines 14-24, the copied elements are indexed, inserting them to either top or bottom of the index while checking if their values match the query predicates (Lines 15 and 16). One might note that all the code is predicated. We avoid branches that could lead to non-robust (i.e., highly varying execution times) due to branch mis-predictions [48, 10]. In line 25, we omit from this listing the code that propagates the pivoting to the remaining dimensions. This code sweeps over each remaining column's respective piece and uses the *go_down* bit vector as set in line 21 to assign each value to the top part or bottom part of the refined piece. The code performs a similar operation to the one described in lines 14-24, with three main differences. First, we do not push elements into the candidate list but rather manage the ones in there while checking for matches in the next dimensions. Second, instead of the pivot comparison, we use the information in the *go_down* bit vector to place the elements in the column properly. Third, for

Listing 4 Code Snippet of the Creation Phase

```
1  template <class OPL,class OPR> create(Query &q, int& budget) {
2      col = orig_tbl.columns[root.dim];
3      idx_c = table.columns[root.dim];
4      l = q[root.dim].low
5      h = q[root.dim].high
6      low_pos = root.cur_start;
7      high_pos = root.cur_end;
8      c_pos = root.cur_start + root.end - root.cur_end;
9      n_idx = min(c_pos + budget, root.end);
10     budget -= n_idx - c_pos;
11     bit_idx = 0;
12     CandidateList cl;
13     BitVec go_down = BitVec(n_idx-c_pos);
14     for (i = c_pos; i < n_idx; i++) {
15         mtch = OPL(col[i],l) & OPR(col[i],h);
16         cl.maybe_push_back(i,mtch);
17         big_pvt = (col[i] >= root.key);
18         sml_pvt = 1 - big_pvt;
19         idx_c[low_pos] = col[i];
20         idx_c[high_pos] = col[i];
21         go_down.set(bit_idx++, sml_pvt);
22         low_pos += sml_pvt;
23         high_pos -= big_pvt;
24     }
25     ...
26     root.cur_start = low_pos;
27     root.cur_end = high_pos;
28     return cl;
29 }
```

the first dimension after the *root.dim* we update *root.left_sum* and *root.right_sum* according to the *go_down* bit vector. After indexing all dimensions, in line 26-27, we update the root offsets, and in line 28, we return the created candidate list that refers to the δ fraction of the table. Hence, the function that calls the *create* method still checks the not-yet-indexed fraction of the original table and the previously indexed bottom/top index pieces accordingly.

3.3 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, 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 4-2 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.

Query Execution

In this section, we describe how we use the Progressive KD-Tree during query execution. In the next paragraphs, we describe the two primary operations, the Index Lookup and the Piece Scan.

Index Lookup. After performing the necessary index creation 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 4-3 we depict an example of the entire search process for predicates $6 < A \leq 15$ AND $0 < B \leq 5$. The 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 root's right child 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

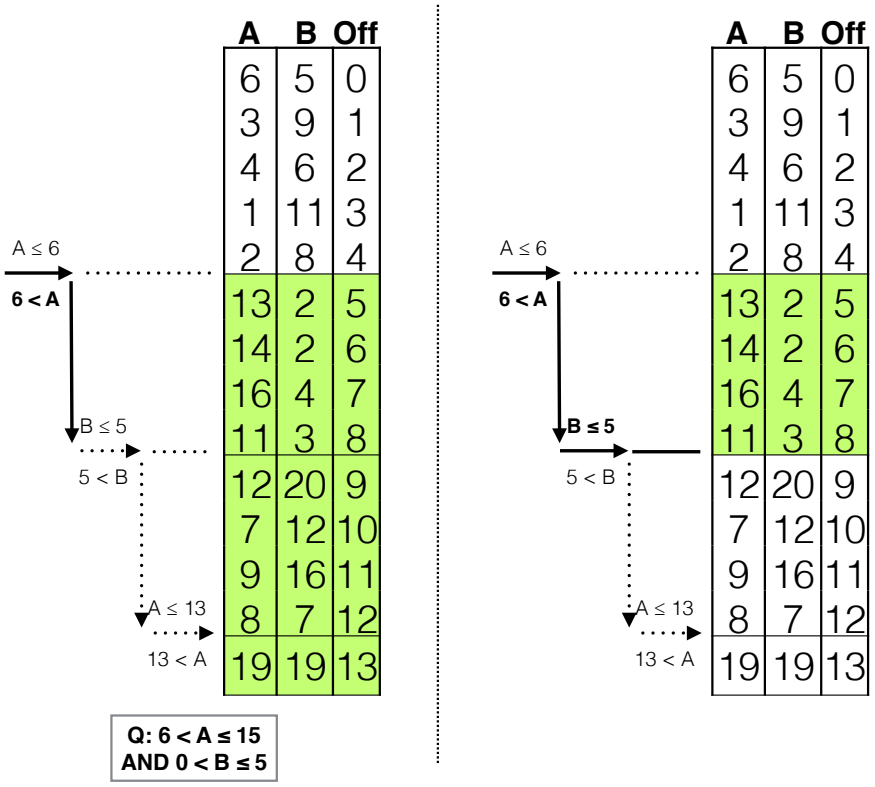

Figure 4-3: A search with predicates $6 < A \leq 15$ AND $0 < B \leq 5$ in the Adaptive KD-Tree.

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 4-3, 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 multidimensional conjunctive selection on one or more pieces. There are, in general, two ways to perform multidimensional 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 (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 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 drops below τ. Then, we calculate a δ' as in situation (1) and proceed as described above.

3.4 Greedy Progressive Indexing

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

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 4.1: Parameters for Progressive Indexing Cost Model.

$(t'_i < t_{scan} \Rightarrow t_i < t_{total})$.

While generally decreasing, t'_i, and hence t_i, can still vary significantly until the index is fully built.

We propose *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 construction or refinement does not boost 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 4.1.

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 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 height h times the cost of random page access ϕ gives the cost for traversing the KD-Tree, resulting in $t_{lookup} = h * \phi$. For the swapping of elements, we perform predicated (i.e., branch-free) swapping [10] 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 is less than τ, distribute this indexing work over x queries, and calculate how much indexing budget $t_{budget++}$ is needed 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).

4 Experimental Analysis

In this section, we provide a quantitative assessment of our proposed 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 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 indexes QUASII (Q) and Adaptive KD-Tree (AKD). We 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 our algorithms' behavior when the full scan cost is higher than the interactivity threshold.

4.1 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

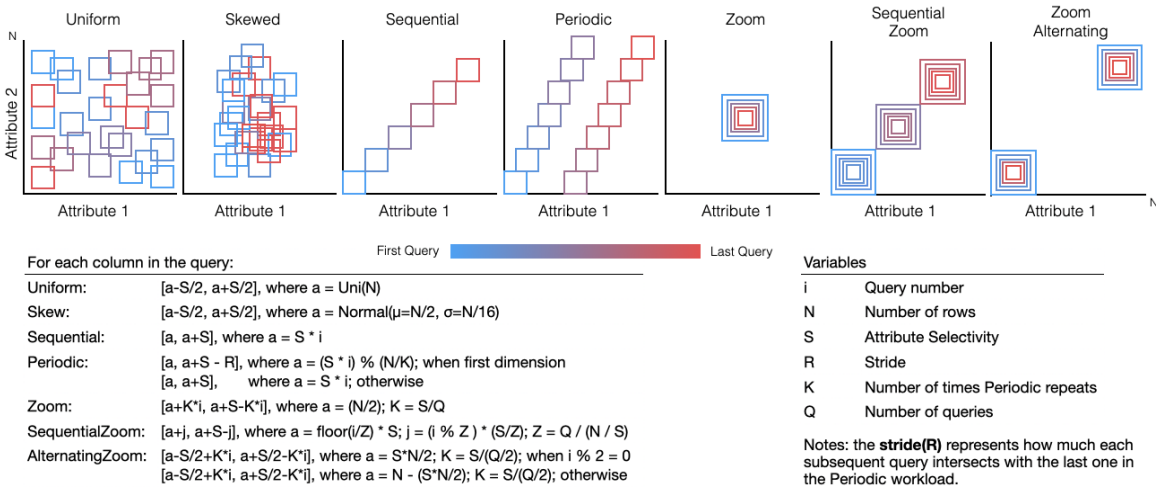

Figure 4-4: Visual representation of the different synthetic workloads.

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.

4.2 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, 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 those described in Chapter 3 but extended for

[3]https://debs.org/grand-challenges/2012/
[4]http://skyserver.sdss.org

the multidimensional case. Figure 4-4 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 the other three columns, and so forth). When generating a synthetic workload, we take as a 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 $\sigma_2 = 10\%, \sigma_4 = 31\%, \sigma_6 = 46\%, \sigma_8 = 56\%$.

4.3 Delta Impact

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 the addition of copying and pivoting a δ-fraction of the data. Figure 4-5 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 than those of both Adaptive and Progressive KD-Tree

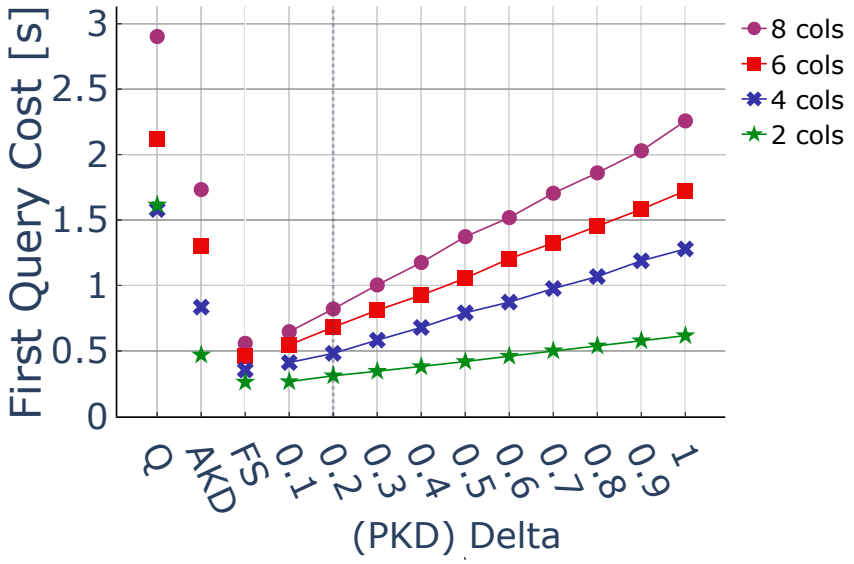

Figure 4-5: First query cost.

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 4-5 as building 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 4-6 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 $\delta = 0.1$, 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

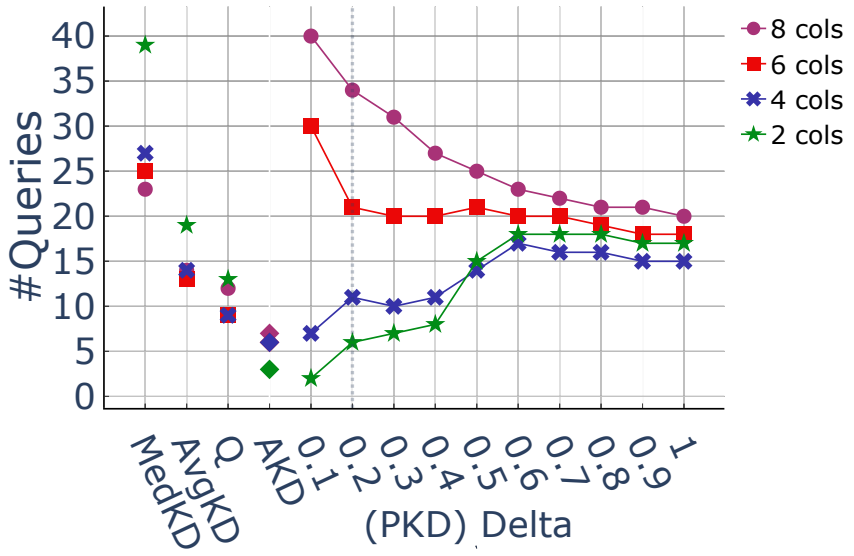

Figure 4-6: #Queries until Pay-off.

workload cannot 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. Regarding 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 4-7 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 the number of queries to converge. For example, with $\delta = 0.1$, the number of queries to converge is about 100, independent of the number of columns.

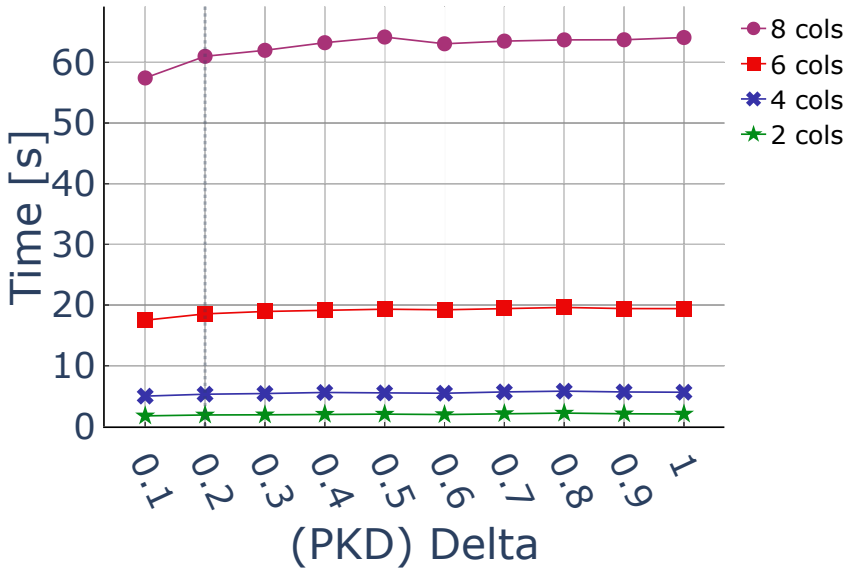

Figure 4-7: Time until Convergence.

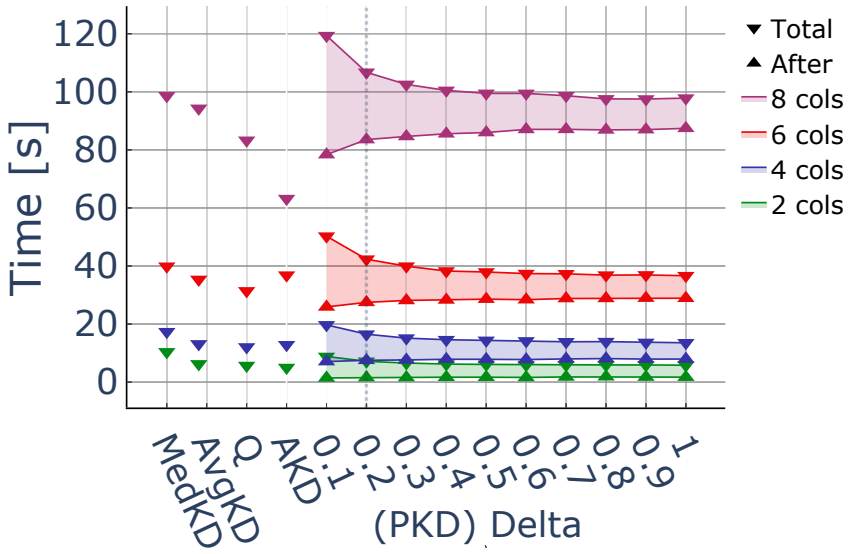

Figure 4-8: Cumulative time (1000 queries).

Cumulative Time

In Figure 4-8, 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 compared to $\delta = 0.1$, without a significant increase in cost in the first query.

4.4 Performance Comparison

In the remainder of the experimental section, we will focus on comparing the performance of 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), Adaptive KD-Tree (AKD), 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 4.2.

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 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 PKD with $\delta = 0.2$.

First Query.

		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 4.2: First query response time (Seconds).

Table 4.2 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.

		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 4.3: Pay-off (Seconds).

Table 4.3 depicts the time it takes for the investment spent on index creation to pay-off when compared to a full scan. 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 of its core design of only indexing the pieces necessary for the executing query. At the same time, 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 4-6 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. Figure 4-9 depicts

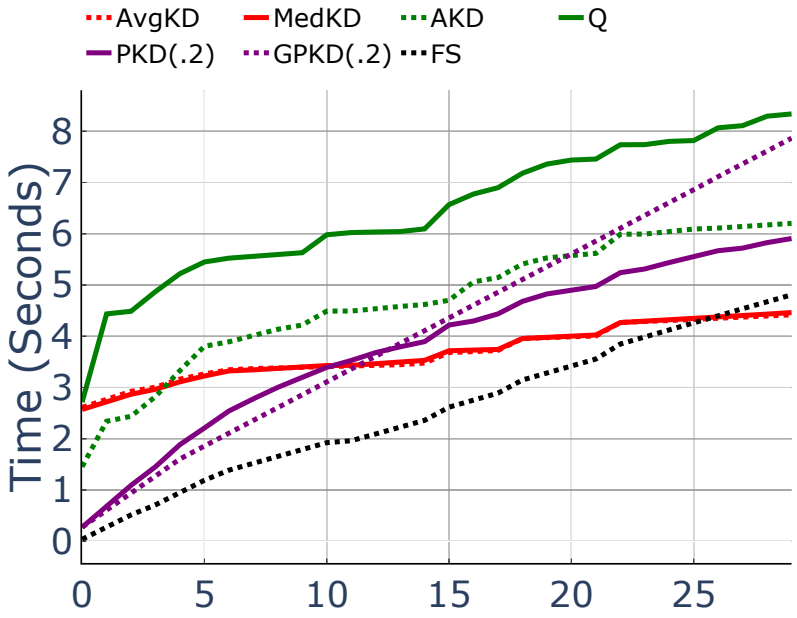

Figure 4-9: Cumulative response time.
Genomics, first 30 queries.

the cumulative response time of the first 30 queries in the Genomics Benchmark. 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 4.4 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. 4-10).

		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 4.4: Query time variance (smaller is better).

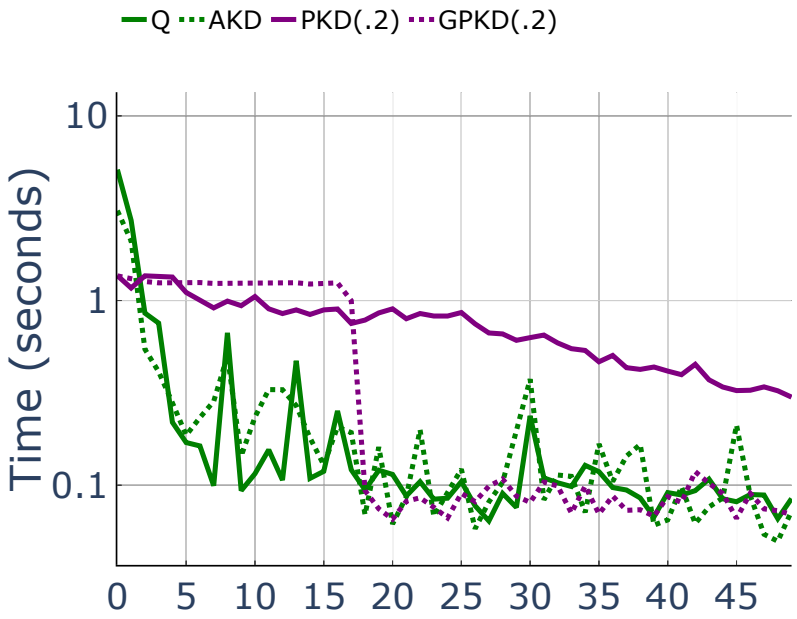

Figure 4-10: Per query response time.
Uniform(8), first 50 queries.

Total Response Time.

Table 4.5 depicts the total response time of all benchmarks. The Progressive Indexing approaches have a very similar response time compared to the full indexes due to their design characteristics prioritizing robustness and convergence over total response time, which is reinforced by the low δ picked for the experiments. Adaptive indexing always

		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 4.5: Total response time (Seconds).

has the lowest total response time due to its high focus on refining pieces requested by the currently executing query. The Adaptive KD-Tree presents the fastest results for most 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.

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. The only index with a faster response time 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.

4.5 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 4.6 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

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

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.

4.6 Full Scan Exceeding the Interactivity Threshold

Figure 4-11 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). 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

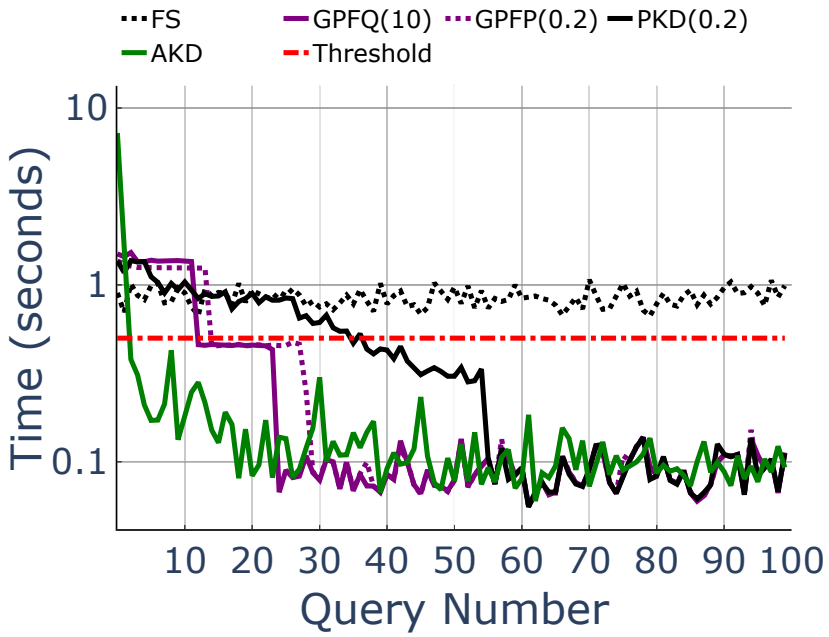

Figure 4-11: Adaptive and Progressive KD-Tree with scans costs exceeding the interactivity threshold; first 100 queries.

index. Both GPFQ and GPFP have similar behavior. They start at 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.

5 Summary

This chapter extended existing work on multidimensional adaptive indexing by introducing two new progressive indexing algorithms. We showed that our algorithms are superior compared with state-of-the-art multidimensional indexing in various real and synthetic workloads. In summary, 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 logical choice.

Up to this point in this thesis, we explored how to create uni and multidimensional

Progressive Indexes. However, these indexes assume that the data is immutable (i.e., no appends or updates happen). In the next chapter, we propose one new progressive algorithm designed to merge updates into Progressive Indexes.