Processing Incremental Multidimensional Range Queries in a Direct Manipulation Visual Query Environment

Stacie Hibino*
Bell Labs/Lucent Technologies
1000 E. Warrenville Road
Naperville, IL 60566
hibino@research.bell-labs.com

Elke A. Rundensteiner
Computer Science Department
Worcester Polytechnic Institute, 100 Institute Road
Worcester, MA 01609-2280 USA
rundenst@cs.wpi.edu

Abstract

We have developed a MultiMedia Visual Information Seeking (MMVIS) Environment designed to support an integrated approach to direct manipulation temporal querying and browsing of temporal relationship results. In this paper we address the optimization of queries specified via our visual query interface. Queries in MMVIS are incrementally specified and continuously refined multidimensional range queries. In this paper, we present our k-Array index structure and its bucket-based counterpart, the k-Bucket, as new indexes optimized for processing these direct manipulation queries. In an experimental evaluation comparing our k-Array and k-Bucket solutions to alternate techniques from the literature, we show that the k-Bucket performs generally equal or better than the other techniques and is the best overall approach for such environments.

1. Introduction

As large databases become more widely available to everyday users, new tools are needed for quickly and easily processing and filtering this information based on new paradigms such as direct manipulation querying. *Range searching* is an important filtering technique for many applications such as business, decision support, GIS, scientific analysis, etc. A sample query to a real estate database with ranges over the two dimensions of the number of bedrooms and the price is "show me all homes with 3-5 bedrooms that cost between $200K and $250K dollars." Dynamic query (DQ) filters and visual information seeking (VIS) provide a novel approach to filtering data through multidimensional range searching [2, 12]. In this framework, users specify range queries through direct manipulation of DQ filters (i.e., buttons and sliders) while a visualization of results is *dynamically* updated. This approach supports users in *posing specific queries* without requiring them to memorize the syntax and semantics of a query language, and in *browsing* the database without having a specific query in mind. It has also been shown to aid users in trend searching [1, 10].

While the problem of processing multidimensional range queries has been studied before [6, 7, 8], we now re-examine it within the needs and constraints of the VIS query paradigm. In order to *dynamically update* the visualization of results *as* users manipulate DQ filters, query processing must be extremely efficient. We thus give priority to optimizing search costs over update costs. We assume that the data set remains frozen, and that the index structure can be built once and offline. This is realistic for many applications as the data may be, for instance, a collection of video events to be analyzed [11] or static GIS map data that changes infrequently.

Another important characteristic of VIS is that queries are specified *incrementally*. That is, users specify successive queries by refining their current query via direct manipulation of sliders. There is no jumping back and forth between totally unrelated query formulations—only "sliding" from the current query to the next one (and seeing changes in the display while adjusting any query filter). In our studies, we have found it beneficial to thus process these types of queries incrementally—processing *changes* to query results rather than recalculating the full solution from scratch. In this light, we postulate that index structures which exploit the notion of "nearest neighbor" might prove to be more efficient than those which do not provide support for accessing data within "close proximity" of the current result set.

The only work we are aware of on processing queries in a VIS environment is [15], in which an analytical model and experimental results were used to analyze various main memory data structures. They determined that 1) in *uniformly distributed* data, a linked array performs well for small data sets, while a grid array (i.e., matrix) works well for larger data sets; and 2) in *skewed* data distributions (focusing on data skewed along the diagonal), tree structures such as the k-d tree [4, 5] and the quad tree [4, 8] had much better performance than the grid array. The k-d tree was recommended over the quad tree, however, since it is simpler to construct when ranges of each dimension in the database are different sizes.

In this paper, we extend previous work on processing DQs by 1) developing new index structures (*k-Array* and *k-Bucket*) customized for VIS query processing, 2) examining existing indexes such as the linked array [18], linked bucket, k-d tree [4], and grid file [14] for their applicability, 3) using bucket information to compress the size of the index structures, and 4) conducting experiments on the performance of these structures when secondary storage is required. Our results show the k-Bucket to be the *best overall performer* for large and small data set sizes under a variety of buffer conditions.

* This work conducted while author was at the University of Michigan.

The goal of this work was to optimize query processing for our temporal visual query language (TVQL), a novel direct manipulation interface using specialized DQ filters for posing and browsing temporal relationships [13, 12].

Overview. This paper is divided into six additional sections. Section 2 provides background on DQ environments and processing DQs. Section 3 introduces our k-Array and k-Bucket indexes and identifies existing indexes for multidimensional range queries used in our comparative studies. In Sections 4 and 5, we then present our experimental design and results. We discuss related work in Section 6 and our conclusions in Section 7.

2. Background

2.1 Direct Manipulation Query Environments

Research results in this paper have been driven by the optimization of MMVIS, an interactive visualization environment based on extending VIS technology for video analysis [12, 10]. In MMVIS, users manipulate TVQL, a visual query language for specifying temporal relationship queries between subsets of video data. For example, given video of an Olympic two-man beach volleyball game, if we select a subset A of all types of player action events (e.g., Kiraly setting the volleyball, Steffes serving, etc.) and a subset B of error events, we can then use TVQL to specify queries such as "show me how often player actions *start and end at the same time as* errors" (Figure 1). We can use such a query to compare the frequency and types of errors made by different players, since TVQL is dynamically linked to a visualization of results [11, 12], giving users immediate feedback as they adjust any temporal query filter.

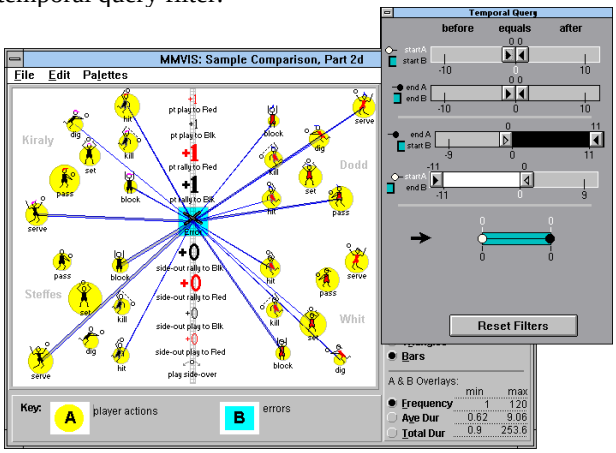


Figure 1. Sample MMVIS application. In this 2-man beach volleyball scenario, the TVQL temporal query palette specifies a query for examining situations when all A events (i.e., individual player actions highlighted with circular overlays) start and end at the same time as B events (i.e., errors indicated by the square highlighter). The bars between the A and B event icons indicate the existence and strength of the temporal relationship specified.

Given two events A1 (○—●) and B1 (▭) with nonzero duration, there are thirteen primitive temporal relationships between them (e.g., *before*, *meets*, etc. [3]), defined by four pairwise relationships between temporal starting and ending points of the events (e.g., start A1 - start B1). TVQL provides a separate DQ filter for each of the four defining pairwise relationships [13]. E.g., the top DQ of the TVQL temporal query palette in Figure 1 allows users to specify the startA-startB relationship. TVQL supports users in 1) specifying combinations of similar primitives (i.e., *temporal neighborhoods* [9]) and 2) *browsing* temporal relationships (since users can manipulate filters with no specific query in mind).

2.2 Processing Dynamic Queries

Basic DQ Characteristics. Processing DQs is a multidimensional range query problem in which queries are *incrementally* specified. Given k DQ filters, we have:

- each DQ filter represents an attribute of the data set,
- all possible slider thumb positions are non-overlapping (i.e., they correspond to disjoint values of the domain of the DQ attribute range),
- all attribute values of the data set are assumed to be discrete singletons (vs. set values or interval ranges),
- each item in the data set can be placed into one and only one "slot" of each DQ filter (see *Parameters*),
- a query corresponds to adjusting desired attribute ranges via directly adjusting one DQ thumb at a time,
- a query is incrementally specified since a new query is specified by adjusting the previously specified query,
- a query is incrementally processed *as* each filter is adjusted, rather than after all ranges have been set, and
- an item in the data set is in the solution iff each value of its k attributes is in the selected range of the corresponding DQ filter (conjunction).

Parameters. We now define the terms and parameters used. Let each DQ slider be characterized by:

`minVal, maxval` = min and max values of slider range

`ticVals` = {tic_1, tic_2, ..., tic_z | tic_1 = minVal, tic_z = maxVal}
// evenly spaced possible slider thumb value positions

`dtIncr` = tic_{i+1} - tic_i = constant, $1 \le i < z$,
// delta increment: min amount user can change an endpoint
// value of a DQ selected range when *moving* a DQ thumb

`slot` = a DQ filter has an internal normalized *slot* for every unique position of a slider thumb (see Figure 2)

`slotCount` = `(2*(maxVal-minVal)/dtIncr)+1`
// number of slots for the given filter.

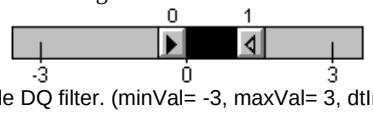


Figure 2.a. Sample DQ filter. (minVal= -3, maxVal= 3, dtIncr= 1)

ticVal	-3		-2		-1		0		1		2		3
slotNum	0	1	2	3	4	5	6	7	8	9	10	11	12

Figure 2.b. Internal representation of a sample DQ filter where the double-lined border around slots 6 and 7 corresponds to the selected DQ filter range in Figure 2.a.

Figure 2 presents a sample DQ filter (`minVal=-3, maxVal=3, dtIncr=1`) and its corresponding internal

representation. It has `slotCount = 2*(3 - ¯3)/1 + 1 = 13`. There are more slots than tics because a filter thumb can be *closed* or *open* to include or exclude an endpoint value, respectively. In Figure 2.a, the DQ filter is selecting the range 0 ≤ values < 1. This corresponds to including slots 6 and 7 in Figure 2.b.

Each DQ slider represents a dimension (i.e., attribute) in the database and we define:

`k` = number of attributes used in query specification, and
`N` =total number of items in the database.

For TVQL, `k=4`, and `N`=number of items in the temporal pairs (`TPairs`) database, where `TPairs` is formed based on user-specified A and B subsets of video events.

Query Specification. Given that users *incrementally* specify queries by manipulating one of the `2k` DQ filter thumbs, we essentially need to find the "nearest neighbor" when processing these queries. Given a double-thumbed DQ filter, there are two types of user manipulations for specifying a query: 1) move a DQ thumb or 2) toggle the fill of a DQ thumb from filled to unfilled and vice versa. We represent these as:

`dqManip(dq`$_i$`, thumbID, action, param),`
where `dq`$_i$ // DQ filter being adjusted,
`thumbID` ∈ `{LEFT, RIGHT}` // DQ thumb manipulated,
`action` ∈ `{MOVE, TOGGLE}` // type of manipulation,
`param` ∈ `{LEFT, RIGHT, FILL, UNFILL}`
// parameter for given action.

3. Index Structures

k-Array. We have designed the k-Array index to meet the characteristics required by dynamic query processing. The *k-Array* is an array-based index structure that realizes a sparse matrix representation by projecting data items from the k-dimensional space onto `k` separate 1-dimensional index arrays, one for each individual dimension (i.e., one `indexArray`$_i$ for each dimension `i`). Similar to others (e.g., [18]), we associate a count `matchCount` with each item to keep track of the number of dimensions for which the item is included in the corresponding selected ranges of the dimensions. An item is in the solution set when its `matchCount=k` (the number of dimensions). Search is accomplished by accessing items in the selected ranges of each dimension and updating their match counts. An item's `matchCount` is incremented/decremented when the item is added/removed to the selected range of any attribute. When the `matchCount` of an item `x` changes from `k-1` to `k`, `x` is added to the solution, and when it changes from `k` to `k-1`, `x` is deleted.

The conceptual representation of the k-Array index depicted in Figure 3 is based on a small real estate database with `k=2` attributes—the Cost of houses and number of bedrooms (#BR); a `slotCount=7` for each attribute; and `N=15` data items (houses in the database). An `indexArray`$_1$ is used for the Cost attribute, `indexArray`$_2$ for #BR, and the user selected slots of the arrays are shaded. The ID of each house object is replicated `k` times and a separate copy of the ID is sorted into its corresponding slot of each of the index arrays. An item's `matchCount` keeps a running tally of the number of matched dimensions for that item and thus must not be replicated as in the case of the item IDs. We thus store `matchCounts` by itemID in a centrally located separate array (see `matchCounts` array on right side of Figure 3). All houses with corresponding `matchCount=k=2` are included in the solution set.

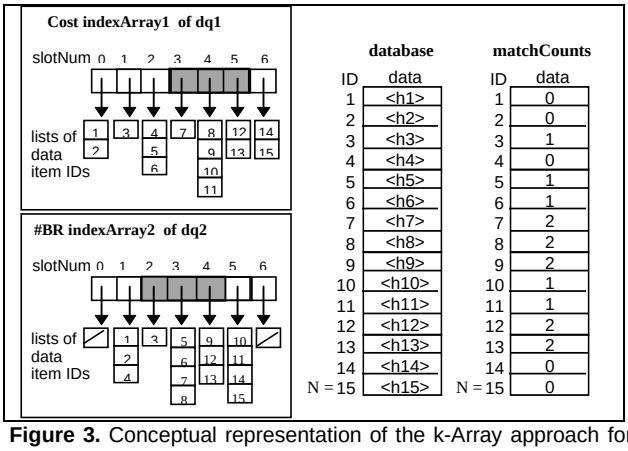


Figure 3. Conceptual representation of the k-Array approach for a real estate example (k=2, slotCount=7, N=15).

In the disk-based version of the k-Array, we modify the conceptual representation of Figure 3 to compress and cluster data to minimize the number of disk pages to be accessed during query processing. The disk-based k-Array still creates `indexArrays` by replicating the ID of each item `k` times, once for each dimension, and by maintaining a list of IDs corresponding to each slot of each `indexArray`. However, since we will unlikely be able to keep the ID lists for all slots of all `indexArrays` in main memory for a large data set or for many dimensions, we store these `indexArrays` (i.e., the `indexArrays` on left side of Figure 3) on disk. Since we assume the TPairs database remains frozen during temporal browsing and since item IDs are replicated, we can further enhance query processing by *contiguously* storing the list of data item IDs for each slot of all `indexArrays`.

In order to access the `indexArrays` from disk, we create a new structure of auxiliary index array information, referred to as `iaInfo`. One `iaInfo`$_i$ is created for each corresponding `indexArray`$_i$ (`1 ≤ i ≤ k`), where each `iaInfo`$_i$ contains:

// # of items in each slot of `indexArray`$_i$:
`numItems[0:slotCount-1]`

// info on each of its DQ filter thumb's position, including slot #
// of thumb and disk "pointer" (disk page # + offset) to access
// info in `indexArray`$_i$ from disk:

// `LEFT` pointer accesses 1st item of `LEFT.slotNum`:
`LEFT.slotNum,LEFT.pageNum,LEFT.offset`

// `RIGHT` pointer accesses last item of `RIGHT.slotNum`:
`RIGHT.slotNum,RIGHT.pageNum,RIGHT.offset`

We assume that when possible, sufficient buffer space is reserved to keep each $iaInfo_i$ in main memory. The amount of memory required for holding all `iaInfos` in main memory is `k*(slotCount + 6) * sizeof(int)`. Thus, for `k=4`, we can hold `iaInfos` with each `slotCount` $\leq$ 122 slots in a single 2K page. When the space required to hold `iaInfos` in main memory is too large, they will be stored on disk.

In addition to the `indexArrays`, the disk-based k-Array also stores the `matchCounts` array on disk in an ordered, contiguous manner. Since the maximum `matchCount` of any data item is fixed, and always $\leq$ `k`, we can use a fixed sized bit representation to compress the space required for this data. In the case of TVQL where `k=4`, the `matchCount` will range from 0 to 4 and only 3 bits per item ID are needed.

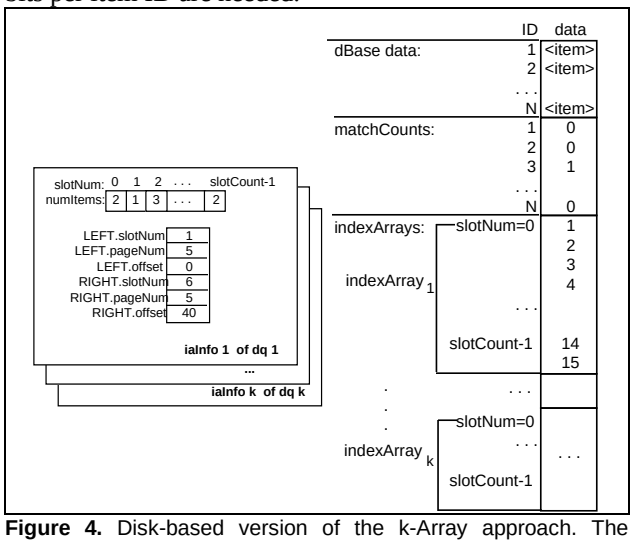


Figure 4. Disk-based version of the k-Array approach. The `iaInfo` data on the left side is held in main memory, while remaining data on the right side are stored on disk.

In addition to the actual `N` data items stored on disk, the disk-based `k-Array` (Figure 4) includes:

// set of `iaInfos` for accessing `indexArrays` from disk;
// kept in main memory when possible:

$$iaInfos = \{..., iaInfo_i, ... \mid 1 \leq i \leq k\}$$

// array of match counts for each item, stored on disk:

`matchCounts[1:N]`

// `indexArrays` stored contiguously on disk.:

$$indexArrays = \{..., indexArray_i, ... \mid 1 \leq i \leq k\}$$

We can use the disk-based k-Array to process each incremental query specified via a user's `dqManip(dq`$_i$`, thumbID, action, param)`, where $1 \leq i \leq k$, as follows:

- use $iaInfo_i$ to get current slot #, page # and offset for `thumbID` of the dq_i filter specified
- initialize `idSize` (disk read amount when fetching itemID)
- using the `thumbID`, `action`, and `param` specified by the user's `dqManip`, determine `targetSlot` (new internal slot position for `thumbID` of dq_i), `direction` (direction that filter thumb is moved), and `rangeAction` (whether the range is expanded or contracted)
- determine `startSlot` and `endSlot` to process
- loop `slotNum` from `startSlot` to `endSlot`:
  - get `slotSize` of next slot to process
  - loop itemCount from 1 to `slotSize`:
    - fetch next `itemID` in slot `slotNum` of $indexArray_i$ from disk
    - if removing `slotNum` from selected attribute range:
      - fetch, decrement, & update `matchCount` from disk
      - if new `matchCount` is `k-1`, then fetch data item from disk and *remove* it from solution
    - if adding `slotNum` from selected attribute range:
      - fetch, increment, & update `matchCount` from disk
      - if new `matchCount` is `k`, then fetch data item from disk and *add* it to solution
    - update `pgNum, pgOffset` to point to next `itemID`
    - update `thumbID` "pointer" information in $iaInfo_i$

k-Bucket. We define a *bucket* as the smallest unit of search in terms of the granularity at which sliders can be moved to constrain values along one dimension (i.e., equivalent to normalized placeholders in the full matrix index). When a data item is (not) in the solution set, its bucket is also guaranteed (not) to be in the solution set and vice versa. We refer to indexes that group items together by bucket as *bucket-based structures*. We can convert the item-based k-Array into a bucket-based *k-Bucket*, by grouping data items by bucket (keeping only non-empty buckets) and applying the k-Array indexing technique to buckets, rather than individual items. Match counts are then correspondingly associated with a bucket.

Alternative Index Structures. We compare our k-Array and k-Bucket index structures to the linked array [18], its linked bucket counterpart, an optimized grid file [14], and a k-d tree [5]. The *linked array* [18] is an array-based method using `k` index arrays. However, it is a pointer-based method where each linked array entry (one per data item) contains `k` $next_i$ pointers and the `matchCount` associated with the given item. The *linked bucket* is derived by applying the linked array method to buckets rather than individual items. A *grid file* [19] consists of a grid directory and `k` linear scales for representing a hierarchical grid, dividing the full matrix into k-dimensional hyper-rectangular subregions. The optimized grid file proposed by [14] that we use converts the region directory from a matrix into a list of super-buckets and "deflates" the bounds of each super-bucket to exclude any empty buckets along the edges of the hyper-rectangular region. The *k-d tree* is a multidimensional b-tree [5] where each level of the tree divides the data set (i.e., set of non-empty buckets) based on the median value for one of the `k` dimension keys. The optimized k-d tree that we use only stores data only in tree leaves.

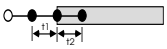
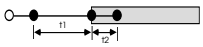
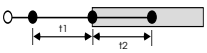
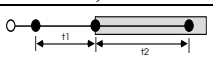
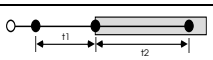
4. Experimental Design

Assumptions. In this work, we give higher priority to browsing (i.e., *processing* incremental queries) rather than maintaining updates and thus do not include the cost of building the indexes in our analysis. We also assume that queries are *specified and processed incrementally*.

Simulation Environment. Our data set corresponds to a hypothetical 60-minute video based on average parameters taken from real video data. A and B subsets of video events were randomly generated and we varied density by controlling the A to B subset size ratio. The TPairs database was then formed from these subsets. We tested a variety of data set sizes, including: 1000 A events by 1000 B events, 2000 by 1000, 2000 by 2000, 2000 by 3000, 2000 by 4000, 2000 by 5000, and 2000 by 6000. These had corresponding TPairs data set sizes ranging from about 12,000 to 143,000 pairs and data storage requirements ranging from 290 kilobytes to 3.4 megabytes. All index structures were implemented in C++ and run on top of UNIX and the MacOS. In this paper, we study the performance of the index structures under various buffer size conditions, varying page size (2K, 4K, 8K) and number of pages (100, 200, 400).

Query Descriptions.

Table 1. Description of the ten TVQL queries used in our studies (N=data set size; F=# of items found; dTfo = endA-startB filter).

Q#: User Manipulation; [Temporal Semantics for filter range selected]	Resulting Temporal Diagram and Quant. Information	Soln. Size (%N)	F (%N)
Q1: right-click right thumb of dTfo filter to exclude end of range, dTfo.max; [dTfo.min ≤ dTfo < dTfo.max]	[all relationships remain selected]	56%	44%
Q2: drag right thumb of dTfo filter to the left by one ticVal; [dTfo.min ≤ dTfo < dTfo.max - dtIncr]	[all relationships remain selected]	51%	5%
Q3: drag right thumb of dTfo filter to 0; [dTfo.min ≤ dTfo < 0]	 0 < t1 ≤ |dTfo.min|	23%	28%
Q4: drag left thumb of dTfo filter to -2; [-2 ≤ dTfo < 0]	 0 < t1 ≤ 2	5%	18%
Q5: right-click the right thumb of dTfo filter to include 0; [-2 ≤ dTfo ≤ 0]	 0 ≤ t1 ≤ 2	7%	2%
Q6: drag right thumb of dTfo filter to 2; [-2 ≤ dTfo ≤ 2]	 0 ≤ t1 ≤ 2, 0 ≤ t2 ≤ 2	11%	4%
Q7: drag left thumb of dTfo filter to -6; [-6 ≤ dTfo ≤ 2]	 0 ≤ t1 ≤ 6, 0 ≤ t2 ≤ 2	21%	10%
Q8: drag right thumb of dTfo filter to 6; [-6 ≤ dTfo ≤ 6]	 0 ≤ t1 ≤ 6, 0 ≤ t2 ≤ 6	27%	6%
Q9: drag right thumb of dTfo filter to one slot short of end of slider; [-6 ≤ dTfo ≤ dTfo.max - dtIncr]	 0 ≤ t1 ≤ 6, 0 ≤ t2≤ dTfo.max-dtIncr	31%	4%
Q10: drag right thumb of dTfo filter to end of slider (dTfo.max); [-6 ≤ dTfo ≤ dTfo.max]	 0 ≤ t1 ≤ 6, 0 ≤ t2 ≤ dTfo.max	32%	1%

Since no known benchmark queries exist for DQs or incremental multidimensional range queries, we created a set of test queries (Table 1) designed to 1) explore the advantages and limitations of the various indexes, and 2) mimic real world user queries as in [11]. A rationale for choosing these test queries is given in [10], but omitted here due to limited space. In Table 1, we describe each test query by column: 1) its query number, the users' manipulation of TVQL to specify the given query, and the temporal semantics of the newly specified filter range; 2) a visual representation of the specified query including the resulting temporal diagram and quantitative information; 3) the size of the query's full solution presented as a percentage of data set size `N`; and 4) `F` (`F` ≤ `N`), the number of items found in the difference set between the current and previously specified query.

5. Experimental Results

Evaluating the Array-Based Methods. When comparing the number of page faults required for using the array-based index structures—the linked array, the k-Array, and their bucket-based counterparts, we found that 1) performance of the linked array degrades considerably as data set size increases, 2) performance of the k-Array is fairly stable for small to medium data set sizes (N < 71,500), 3) bucket-based methods perform much better than their item-based counterparts, especially when converting the linked array to a linked bucket, and 4) the linked bucket presents a competitive alternative to the k-Array and the k-Bucket methods, whereas the linked array does not [10]. We thus exclude the linked array from the remainder of our performance evaluations.

Comparing Indexes Over the Sample Queries. In this section, we compare the various indexes for processing the sample TVQL queries of Table 1. We use a (100, 4K) main memory buffer to compare the performance of the index structures for small, but not overly adverse buffer conditions. Figure 5 presents the results for processing two of the ten sample TVQL queries of Table 1: a) query Q3, a sample worst-case scenario and b) query Q6, a sample better case scenario. The scales are *not* the same on all graphs—they have been kept to a minimum to provide as much detail as possible during comparison. Details on all ten sample queries are available in [10].

Table 2 summarizes results for processing all ten sample queries when using the (100, 4K) buffer size. In columns 2 and 3, we repeat the solution set size (%N) and the size of the items found in the difference set F (also in terms of %N). We summarize the results by indicating the minimum and maximum page faults required for the largest N (columns 4 and 5), the differences between the maximum and minimum page faults required (column 6), and the best and worst index structures (columns 7 and 8).

Table 2 indicates that `F` is greatest for queries Q1, Q3, and Q4, and smallest for Q10, Q5, Q6, and Q9.

Figure 5. Number of page faults required for processing queries Q3 and Q6 for various data set sizes (buffer = 100, 4K pages).

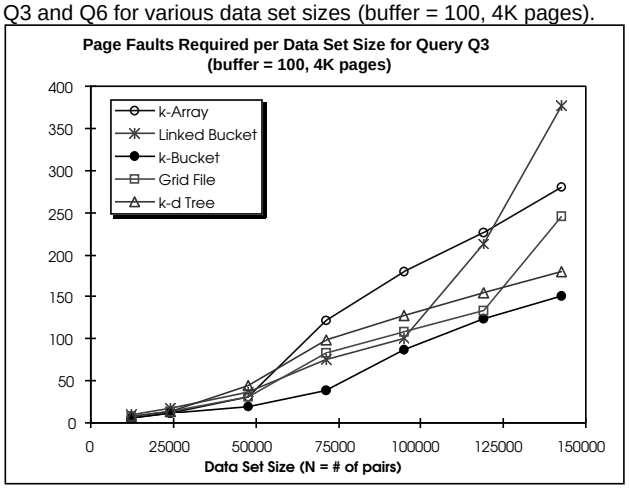


Figure 5.a. Query Q3, a sample worst-case scenario query.

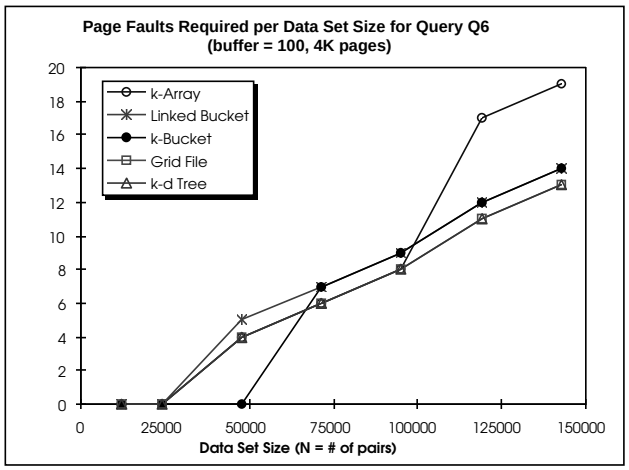


Figure 5.b. Query Q6, a sample better case scenario query.

While F is relatively small for Q2, we expect processing of Q2 to be more expensive than Q8 since in Q2, larger ranges are selected for each attribute and hence the hyperrectangular difference region to be searched is also bigger. Using Table 2 and referring back to the sample graphs in Figure 5, we can identify the following trends:

- the *k-Bucket* outperforms the rest of the methods in five of the nine queries (Q1, Q2, Q3, Q5, and Q8). Most notably, the k-Bucket performs better than other approaches in the worst-case query scenarios as shown by the results for the *most expensive* queries Q1, Q2, and Q3. Also, the k-Bucket outperforms the other methods for large F and large N (see queries Q1, Q3).
- the *k-d tree* is the best performer for the four queries characterized by the smallest value of F: Q5, Q6, Q9, and Q10. The k-d tree performs poorly in queries Q1 and Q8 and also performs less competitively in Q3 for N < 100,000. Queries Q1 and Q3 are characterized by a larger F. These results confirm that the k-d tree is particularly sensitive to F. The poor performance of the k-d tree for query Q8 is somewhat surprising since F is relatively small for that query.
- the *grid file* also performs poorly for large N and large F (e.g., Q1 and Q3). However, it does perform best in queries Q6 and Q10. Q6 and Q10 are characterized by a smaller F, but this trend does not generally hold true for the grid file as it does not perform as well for Q5. Although the grid file performs fairly competitively in half of the queries, it appears to be a much less predictable method.
- the *linked bucket* is characterized in many graphs with a sharp upward trend as the data set size increases (Q2, Q3, Q8, Q9). It performs most competitively in queries Q4 (where it is best) and Q1 (second best). Overall, the linked bucket is fairly competitive for smaller data sets (N < 75,000), but quickly degrades as the data set size increases. This result confirms the corresponding main memory evaluation of the linked array [15].
- the *k-Array* behaves similarly to the linked bucket in that it is most sensitive to the data set size N. It performs relatively competitively for half of the queries (Q1, Q3, Q5, Q7, Q8) for smaller N (N < 50,000 and in some cases, for N < 70,000). The k-Array is not, however, the worst performer for all types of queries, as it outperforms the grid file and linked bucket in Q2 and does better than the linked bucket in Q3. The k-Array is most notably useful when N is small and when the bucket to item ratio is small. When the latter ratio is large, it is much more efficient to use the k-Bucket.

Table 2. Summary of query processing results presented in Figure 5. (N=data set size; F=number of items found; kA=k-Array; kB=k-Bucket; LB=linked bucket; GF=grid file; KD=k-d tree)

Q#	Soln. Size (%N)	F (%N)	Page Faults Required for Largest N			Index Performance at Largest N	
			min	max	max - min	best	worst
Q1	56%	44%	172	231	59	kB	kA
Q2	51%	5%	136	246	110	kB	LB
Q3	23%	28%	150	376	226	kB	LB
Q4	5%	18%	54	64	10	LB	kA
Q5	7%	2%	7	24	17	kB/ KD	kA
Q6	11%	4%	13	19	6	GF/ KD	kA
Q7	21%	10%	0	0	0	-	-
Q8	27%	6%	18	33	15	kB	kA
Q9	31%	4%	12	33	21	KD	kA
Q10	32%	1%	2	68	66	GF/ KD	kA

Comparing Indexes Over Various Buffer Sizes. Assuming that each of the ten queries is just as likely to occur as any other, we can create a summary graph of page faults required per data set size for all queries by adding the page faults required for all ten queries for each data set size of each type of index structure. Figure 6 presents summary graphs for two representative sample buffer sizes out of the nine combinations of 100, 200, and 400 number of buffer pages with 2K, 4K, and 8K buffer page sizes that we examined. In Table 3, we provide a summary over all nine buffer sizes for a large data set size (N ≈ 143,000), highlighting the best (*), worst (-), and second worst (4, i.e., 4th out of 5) index performers.

Figure 6. Summary of page faults required per data set size (sum of all queries by index structure).

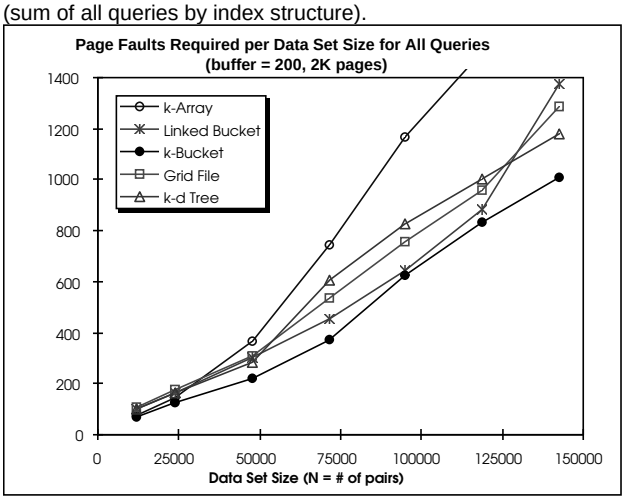


Figure 6.a. buffer = 200, 2K pages.

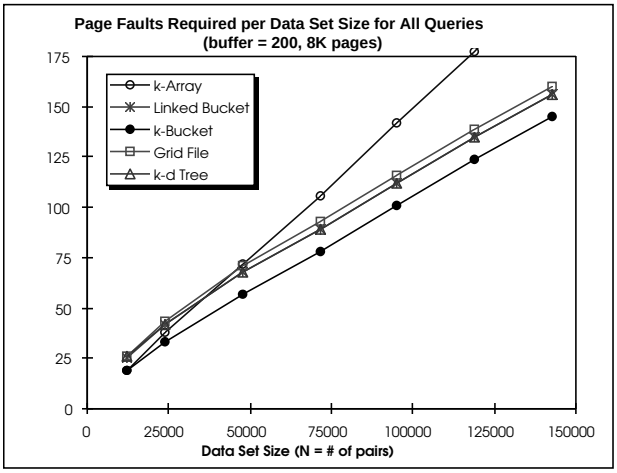


Figure 6.b. buffer = 200, 8K pages.

Table 3. Additional cost (in # of pages) for each index structure for a large data set size, based on the difference from the minimum (min.) number of page faults used.(* = best performer, - = worst performer; ^{4} = 4th performer; data set size N≈ 143,000)

Buffer		Index Structure					
# Pgs	Page Size	k-Array	Linked Bucket	k-Bkt	Grid File	k-d Tree	min.
100	2K	1329 -	1089 ^{4}	432	739	0 *	1357
200	2K	811 -	363 ^{4}	0 *	273	167	1011
400	2K	792 -	145	0 *	162 ^{4}	144	584
100	4K	402 -	360 ^{4}	0 *	224	69	572
200	4K	398 -	21	0 *	34 ^{4}	32	343
400	4K	140 -	23	0 *	28 ^{4}	24	288
100	8K	246 -	12	0 *	44	68 ^{4}	176
200	8K	74 -	11	0 *	15 ^{4}	11	145
400	8K	66 -	11	0 *	15 ^{4}	11	145

The results in Figure 6 and Table 3 show that the k-Bucket is the best overall performer for all data set sizes and all buffer variations except for the smallest buffer size of (100, 2K) pages. Our results show that in this smallest buffer size, the k-Bucket performs best for small to medium sized data set sizes (up to about N = 80,000), after which it performs second best only to the k-d tree [10]. The k-Array, however, is the worst overall performer for all but the smallest buffer size. This further confirms the benefit of bucket- over item-based methods.

Examining the graphs in Figure 6, we see that the k-Array is only competitive for relatively small data set sizes (N < 50,000). This is likely due to a smaller number of items to buckets ratio at smaller data set sizes, so that the overhead for storing bucket information may cancel gains in processing the query based on buckets.

When investigating the effect of increasing data set size within a *fixed* buffer size, we find that the k-Bucket is the best overall performer per data set size for all buffer sizes except for the case of processing large data set sizes within the smallest (100, 2K) buffer size. When we increase the size of the main memory buffer by keeping the number of pages constant and varying page size from 2K to 4K to 8K in size, the k-Bucket remains the best overall performer and the linked bucket shows the largest gains over the k-Array, grid file, and k-d tree. The linked bucket is the worst or second worst performer in the smallest buffer size of (100, 2K) pages, but ultimately becomes the second best performer in the case of the (100, 8K) buffer size. Similarly, when we keep the buffer page size a constant and increase the main memory buffer by varying the number of pages from 100 to 200 to 400 pages, we again find that the k-Bucket remains the best overall performer and that the linked bucket shows greater gains over the k-Array, grid file, and k-d tree.

The magnitude of page faults required for the smallest buffer size (see first row of Table 3) indicates that many of the indexes "starve" under adverse conditions of processing a very large data set in a very small buffer size of (100, 2K). In this situation, the k-d tree outperforms the other indexes for large data set sizes by a large margin. Thus, the overall results indicate that the k-d tree should be recommended for the conditions of processing a very large data set (N > 80,000) using a very small buffer size of (100, 2K), and that under all other buffer conditions, the k-Bucket should be recommended.

6. Discussion and Related Work

While recent research in temporal queries over continuous data has emerged [21], such work has not focused on the notion of temporal browsing. Although research in multidimensional range queries has been ongoing for years, even recent reviews of current techniques indicate that there is not one multi-attribute method that has become the standard of choice [16]. As the number of dimensions increases, node splitting and data clustering continue to be challenging problems. In addition, many researchers continue to look at the problem of handling updates and keeping balanced structures,

particularly with tree-based data structures (e.g., [17]). In our work, we have examined the problem from a different perspective—one of optimizing query processing over updates and processing queries incrementally. These criteria are critical for preserving the browsing paradigm in MMVIS. To our knowledge, these types of queries (i.e., DQs) have only been examined by [15], and their evaluation was only based on main memory processing, rather than taking the need for secondary storage into consideration. We have, however, taken advantage of previous work, by comparing our k-Array and k-Bucket indexing techniques to a base-case index (the linked array [18] and its linked bucket counterpart) and to well-established and popular indexing structures such as the grid file [19, 14] and k-d tree [4, 5].

Although the quad tree, r-tree and their many variants [4, 8, 20] are also popular indexing structures for multidimensional range queries, we chose not to evaluate these structures since (1) results from [15] lead to the recommendation of the k-d tree over the quad tree due to the fact that the k-d tree performs very similarly but has other more desirable features such as ease of constructing the index, and (2) both fall into the category of tree type structures so we expect them to have similar results.

7. Conclusion

In order to preserve the notion of interactive browsing for trend analysis, dynamic queries—which are *incremental* multidimensional range queries—must be processed as efficiently as possible. In this paper, we presented results from running a series of experiments to evaluate index structures for processing dynamic queries posed via our temporal visual query language (TVQL [13]). We introduced a new array-based indexing method called our k-Array and its even more efficient bucket-based counterpart the k-Bucket.

When comparing the k-Array, linked bucket, k-Bucket, grid file and k-d tree over a series of incremental queries processed using a (100 page, 4K) buffer, we found that the k-Bucket performs the best overall. Comparing the performance of these same methods over various buffer sizes, based on a buffer page size of 2K, 4K, or 8K, and a buffer page count of 100, 200, or 400 pages, we found that the k-Bucket is the best overall performer for processing all data set sizes under all buffer conditions except for the one extreme case when processing *large* data set sizes in the *smallest* (100, 2K) buffer size. In this exception case, the k-d tree is the best performer. The k-Array is the worst overall performer for larger data set sizes for all buffer sizes. The linked bucket has greater gains over the grid file and k-d tree with increase in the number of pages as well as with the increase in page size.

Acknowledgments. Special thanks to Monal Sonecha and Jialei Jin who implemented the initial k-d tree used in these studies.

References

1. Ahlberg, C., Williamson, C., & Shneiderman, B. (1992). Dynamic Queries for Information Exploration: An Implementation and Evaluation. *CHI'92 Conference Proceedings*, NY:ACM Press, 619-626.
2. Ahlberg, C., & Shneiderman, B. (1994). Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays. *CHI'94 Conference Proceedings*, NY:ACM Press, 619-626.
3. Allen, J.F. (1983). Maintaining knowledge about temporal intervals. *CACM*, 26(11), 832-843.
4. Beckley, D.A., Evens, M.W., Raman, V.K. (1985). Multikey Retrieval from K-d Trees and Quad Trees. *ACM SIGMOD Proceedings*, 291-301.
5. Bentley, J. (1975). Multidimensional Binary Search Trees Used for Associative Searching. *CACM*, 18(9), 509-517.
6. Bentley, J.L. & Friedman, J.H. (1979). Data Structures for Range Searching. *Computing Surveys*, 11(4), 397-409.
7. Bentley, J.L. and Maurer, H.A. (1980). Efficient Worst-Case Data Structures for Range Searching. *Acta Informatica* 13, 155-168.
8. Flajolet, Gonnet, Puech, & Robson. (1993). Analytic Variations on Quadtrees. *Algorithmica*, 10, 473-500.
9. Freksa, C. (1992). Temporal reasoning based on semi-intervals. *AI*, 54(1992), 199-227.
10. Hibino, S. (1997). *MultiMedia Visual Information Seeking*. University of Michigan PhD dissertation.
11. Hibino, S. & Rundensteiner, E. (1997). "Interactive Visualizations for Temporal Analysis: Application to CSCW Multimedia Data." To appear in *Intelligent Multimedia Info. Retrieval* (Mark Maybury, Ed.).
12. Hibino, S. and Rundensteiner, E. (1996). MMVIS: Design and Implementation of a Multimedia Visual Information Seeking Environment. *ACM Multimedia'96 Conference Proceedings*, NY:ACM Press, 75-86.
13. Hibino, S., & Rundensteiner, E. A. (1996). "A Visual Multimedia Query Language for Temporal Analysis of Video Data," *Multimedia Database Systems: Design & Impl. Strategies* (Nwosu, Thuraisingham, & Berra, Eds.). Norwell, MA: Kluwer Academic Publishers, 123-159.
14. Hinterberger, H., Meier, K.A., Gilgen, H. (1994). Spatial Data Reallocation Based on Multidimensional Range Queries, 228-239.
15. Jain, V. & Shneiderman, B. (1994). Data Structures for Dynamic Queries: An Analytical and Experimental Evaluation. *Proceedings of the Workshop on Advanced Visual Interfaces.* NY: ACM Press, 1-11.
16. Lomet, D. (1992). A Review of Recent Work on Multi-attribute Access Methods. *SIGMOD Record*, 21(3), 56-63.
17. Nakamura, Y., Abe, S., Obsawa, Y. Sakauchi, M. (1993). A Balanced Hierarchical Data Structure for Multidimensional Data with Highly Efficient Dynamic Characteristics. *IEEE TKDE*, 5(4), 682-694.
18. Knuth, D.E. (1973). *The Art of Computer Programming, Vol 1: Fundamental Algorithms.* Addison-Wesley.
19. Nievergelt, J., Hinterberger H., & Sevcik, K.C. (1984). The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. on Database Systems*, 9(1), 38-71.
20. Samet, H. (1990). *The Design & Analysis. of Spatial Data Structures.* Reading, MA:Addison-Wesley.
21. Seshadri, P., Livny, M., and Ramakrishnan, R. (1994). Sequence Query Processing. *1994 ACM SIGMOD Conference Proceedings.* NY: ACM, 430-441.