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

Stacie Hibino
EECS Dept., Software Systems Research Lab
The University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122 USA
hibino@eecs.umich.edu

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 that is the first system to support an integrated approach to the direct manipulation specification of temporal queries and the browsing of temporal relationship results. In this paper we address the optimization of queries specified via our direct manipulation visual query interface. Queries in MMVIS are multidimensional range queries that exhibit features unique to such a direct manipulation query paradigm. In particular, queries are incrementally specified and continuously refined. Incremental result sets must be passed to the visualization component to provide continuous update and context, and updates of the underlying multimedia data (e.g., video) are rare. In this paper, we present a query processing solution based on an array-based index structure, the k-Array and its bucket-based counterpart, the k-Bucket, that are optimized to take advantage of these features of direct manipulation queries. We have conducted an experimental evaluation comparing our k-Array and k-Bucket solutions with alternate techniques from the literature. Our results confirm that the k-Bucket, while being among the simplest to implement, performs generally equal or better than the other techniques and is the best overall technique for such environments.

The area of data engineering most closely related to this paper is:

Temporal and Spatial Database Systems: Query Processing & Optimization.

Keywords: Visual query paradigms, direct-manipulation queries, query processing, temporal queries, multimedia databases, incremental multidimensional range queries.

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, direct manipulation query paradigms. *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 4-5 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. The advantage of this approach is that it supports users both in 1) *posing specific queries* without requiring them to memorize the syntax and semantics of a query language, and in 2) *browsing* the database to discover data without having a specific query in mind. In addition, the tight coupling of the DQ filters to the visualization of results provides dynamic feedback which has 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], it now must be re-examined within the needs and constraints of the VIS query paradigm. In order to *dynamically update* the visualization of results *as* users manipulate any dynamic query filter (i.e., to preserve browsing within VIS), query processing must be extremely efficient. We thus give priority to optimizing search costs over update costs. This is supported by the fact that we assume that the data set remains frozen, and thus the index structure can be built once and offline. This is realistic for main applications as the data may be, for instance, a collection of video events being used for the purpose of analysis 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 (i.e., range constraints) 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). As we will demonstrate, it is beneficial to thus process these types of queries incrementally—processing *changes* to the 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 identifying and accessing data within "close proximity" of the current result set.

Previously, the primary work on processing queries specified in a VIS environment is [16], 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 (in particular where data is skewed along the diagonal of the full multidimensional, hyper-rectangular matrix), 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 the 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 referred to as our *k-Array* and *k-Bucket* indexes customized for VIS query processing (see Section 3), 2) examining additional index structures including the linked array [19], linked bucket, k-d tree [4], and an optimized grid file [15], 3) using bucket information to compress the size of the index structures, and 4) comparing and contrasting the performance of various structures when secondary storage is required through extensive experimental studies. Our results show that while the k-d tree is the best performer when processing a very large data set (N > 80,000) using a very small buffer size of (100, 2K) pages (only one of the nine buffer conditions tested), our k-Bucket is the *best overall performer* for all data set sizes (large and small) under all other buffer conditions. The goal of this work was to optimize query processing for our temporal visual query language (TVQL), a novel direct manipulation interface which uses specialized DQ filters for posing and browsing temporal relationships [14, 12].

Overview. This paper is divided into seven additional sections. In Section 2, we provide background on direct manipulation query environments and processing dynamic queries. In Section 3, we introduce our k-Array and k-Bucket index structures and in Section 4, we summarize existing index structures for multidimensional range queries to which we compare our approach. In Section 5, we describe our experimental design and in Section 6 we present our experimental results. In Section 7, we discuss related work and we provide a summary in Section 8.

2. BACKGROUND

2.1 Direct Manipulation Query Environments

Research results presented in this paper have been driven by the development and optimization of MMVIS, a MultiMedia Visual Information Seeking system that provides an interactive visualization environment based on extending VIS technology for video analysis [12, 13]. In MMVIS, users use TVQL, a visual, direct manipulation 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 digging, setting, hitting the volleyball, Steffes digging, 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). Such a query allows us to easily compare the frequency and types of errors made by different players, since TVQL is dynamically linked to a temporal visualization (TViz) of results [11, 12], giving users immediate feedback as they adjust any temporal query filter. Once we have examined this characterization of user errors, we could then investigate what types of player actions lead to errors by manipulating TVQL to specify AB temporal sequences.

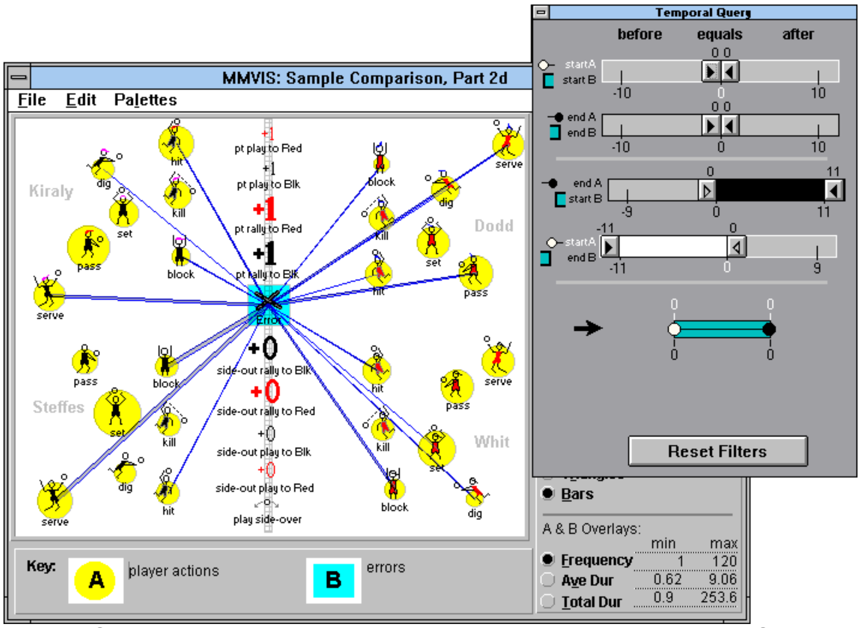


Figure 1. Sample MMVIS application. In this 2-man beach volleyball scenario, the TVQL temporal query palette specifies a query for examining situations when all A events (i.e., individual player actions highlighted with circular overlays) start and end at the same time as B events (i.e., errors indicated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

While a formal specification of TVQL can be found elsewhere [14], we review its basic principles here. Given two events A1 ( ) and B1 ( ) with nonzero duration, there are thirteen primitive temporal relationships between them (such as *before, meets, during,* 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. For example, the top DQ of the TVQL temporal query palette in Figure 1 allows users to manipulate 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

2.2.1 Basic DQ Characteristics

The problem of processing dynamic queries such as those posed within MMVIS can be characterized as a multidimensional range query problem in which queries are *incrementally* specified. Given k DQ filters, we have:

- each DQ filter represents one of the k attributes of the data set,
- all possible positions of the slider thumbs 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 (versus set values or interval ranges),

- each item in the data set can be placed into one and only one slot of each DQ filter (where a DQ filter slot is an internal normalized representation for every unique (i.e., discrete) position of a slider thumb),
- a query corresponds to adjusting and selecting desired attribute ranges via directly manipulating one query filter thumb at a time and is ultimately formed by combining the selected ranges via conjunction,
- a query is incrementally specified in that users specify a new query by adjusting their previously specified query,
- a query is incrementally processed in that it is processed as each filter is adjusted, rather than waiting until all ranges have been set, and
- an item in the data set is in the solution iff each value of the k attributes of the data item is in the selected range of the corresponding DQ filter.

In a real estate database, for example, each house in the database has a defined number of bedrooms and a specific list price. A four bedroom (4BR), $300K house would be placed into the `value=4` slot of the #BRs DQ filter as well as in the `value=$300K` slot of the Cost DQ filter. This 4BR, $300K house would be in the solution set iff `value=4` was in the selected range of the #BRs DQ filter AND `value=$300K` was in the selected range of the Cost DQ filter.

2.2.2 Parameters

Before we present our approach to processing dynamic queries, we first define the terms and parameters used. Let each DQ slider be characterized by the following parameters:

`minVal`	=	minimum value of slider range
`maxVal`	=	maximum value of slider range
`ticVals`	=	$\{tic_1, tic_2, \ldots, tic_z \mid tic_1 = minVal, tic_z = maxVal\}$ \\ set of evenly spaced possible slider thumb value positions
`dtIncr`	=	delta increment: minimum amount user can change an endpoint value of a slider's selected range when moving a slider thumb. (I.e., $dtIncr = tic_{i+1} - tic_i$)
`slot`	=	a DQ filter has an internal normalized *slot* for every unique position of a slider thumb (a separate position is used for a filled versus unfilled filter thumb—see Figure 2)
`slotCount`	=	number of slots for the given filter = `(2*(maxVal - minVal)/dtIncr) + 1.`

In our system, the `minVal`, `maxVal`, and `dtIncr` can be specified by the user at any time. However, the index structure must be recalculated when any of these values are modified. The `ticVals`, `slot`, and `slotCount` parameters are automatically calculated from `minVal`, `maxVal` and `dtIncr`.

Each DQ slider represents a dimension in the database and we define:

k = number of attributes or dimensions used for query specification (e.g., for TVQL, `k=4`),

T = total number of records in $Ann(v_p)$, the database of video annotation events,

$A = \{a_1, a_2, \ldots, a_i, \ldots, a_m \mid m \le T,\ a_i \in Ann(v_p)\}$ \\ 1st subset,

$B = \{b_1, b_2, \ldots, b_j, \ldots, b_n \mid n \le T,\ b_j \in Ann(v_p)\}$ \\ 2nd subset.

Based on these user selected A and B subsets and the constraints imposed by extreme values of the TVQL filters, a new database of temporal pairs (`TPairs`) is formed as follows:

$$TPairs = \{pr_1, pr_2, \ldots, pr_q, \ldots, pr_N \mid pr_q = (a_i, b_j),\ a_i \in A,\ b_j \in B,\ (a_i.end - b_j.start \ge dTfo.minVal \text{ AND } a_i.start - b_j.end \le dTof.maxVal)\},$$

where: `N` = number of all possible pr_q pairs in `TPairs` $(\leq T^2)$ meeting temporal relationship criteria,
`dTfo` = temporal DQ filter for difference between a_i`.end` and b_j`.start`,
and `dTof` = temporal DQ filter for difference between a_i`.start` and b_j`.end`.

2.2.3 Query Processing

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

```
dqManip(dq_i, thumbID, action, param),
```

where
```
dq_i                                  // DQ filter being adjusted,
thumbID ∈ {LEFT, RIGHT}               // DQ filter thumb that was manipulated,
action ∈ {MOVE, TOGGLE}               // type of manipulation made, and
param ∈ {LEFT, RIGHT, FILL, UNFILL}   // parameter for given action.
```

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 $0 \leq$ values < 1. This corresponds to including slots 6 and 7 of the internal representation shown in Figure 2.b.

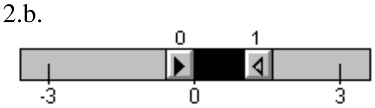


Figure 2.a. Sample DQ filter. (minVal = -3, maxVal = 3, dtIncr = 1)

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.

Given that each item in the TPairs data set has one and only one valid value for each attribute, this means in TVQL that for a given pair $pr_q = (a_i, b_j)$, the value of each attribute determines the slotNum of the corresponding query filter. For example, if pr_q`.dTo = 0` and if the above filter represented the dTo query filter, then pr_q would go into slot 6 of the filter. Recall that pr_q is only in the solution set when *all* of its attribute values are valid (i.e., when each is included in the selected range of its corresponding attribute DQ filter). In other words, if `matchCount` represents the number of valid attribute values for pr_q, then pr_q is in the solution set when `matchCount=4` (or, in general, when `matchCount=k`, the number of dimensions).

2.2.4 Query Processing Example

In a DQ-based query interface supporting queries in `k` dimensions (i.e., with one DQ filter for each of the `k` attributes), a *full matrix index* is a `k`-dimensional hyper-rectangular matrix representing every possible combination of every unique normalized value (i.e., based on `slot` number rather than attribute value) in each dimension. Processing multidimensional range queries corresponds to simple lookups of the full matrix.

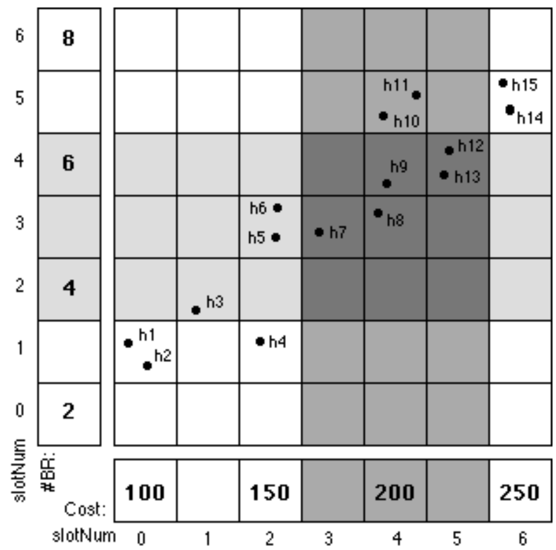


Figure 3. Sample 2-dimensional full matrix index for a real estate database (k=2, slotCount=7, N=15). In this example, the user has selected houses with 4 to 6 bedrooms that cost greater than $150K and less than $250K.

Consider the 2-dimensional full matrix index example for a real estate database presented in Figure 3. In this example, one DQ filter is provided for the cost of a house and one DQ filter is provided for the number of bedrooms (#BR). There are seven slots for each dimension, and each house in the database has been sorted into its corresponding normalized tile in the matrix. The user has selected houses with 4 to 6 bedrooms that cost more than $150K and less than $250K. The solution set is represented by the intersection of the two ranges. Thus, for this particular query, houses h7, h8, h9, h12, and h13 would be included in the solution set.

Although search is simplified in the full matrix index, the size of the matrix increases exponentially with the number of dimensions (i.e., with increasing `k`) and this storage cost may be much higher than necessary in the case of a skewed data distribution. That is, if `slotCount` is the same for each of the `k` DQ filters (i.e., dimensions), then the size of the full matrix index would be $O(\texttt{slotCount}^{k})$. In skewed data distributions, much of the full matrix index may be empty but the size of the matrix will remain fixed. The goal of alternative index structures are thus to reduce the storage cost of the full matrix index by proposing, for example, a sparse matrix representation and to do so while minimizing any increase in search costs. This is important for TVQL queries because temporal pair data inherently have a skewed data distribution.

3. THE PROPOSED INDEX STRUCTURES FOR MULTIDIMENSIONAL RANGE QUERIES

In this section, we introduce our k-Array and k-Bucket index structures as sparse matrix representations that we propose for processing incremental multidimensional range queries.

3.1 The k-Array Index Structure

We have designed the k-Array index structure to meet the specific characteristics required by dynamic query processing such as the processing of TVQL queries. The k-Array is essentially 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 $\mathtt{indexArray}_i$ for each $1 \le i \le k$, where `k` = number of dimensions). Similar to other approaches in the literature (e.g., [19]), we associate a match count `matchCount` with each item to keep track of the number of dimensions in 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 (i.e., attribute) and updating their match counts. A `matchCount` of an item is incremented/decremented when that item is being added/removed to the selected range of any attribute. When the `matchCount` of an item `x` changes from `k-1` to `k`, the item `x` is added to the solution, and when it changes from `k` to `k-1`, it is deleted.

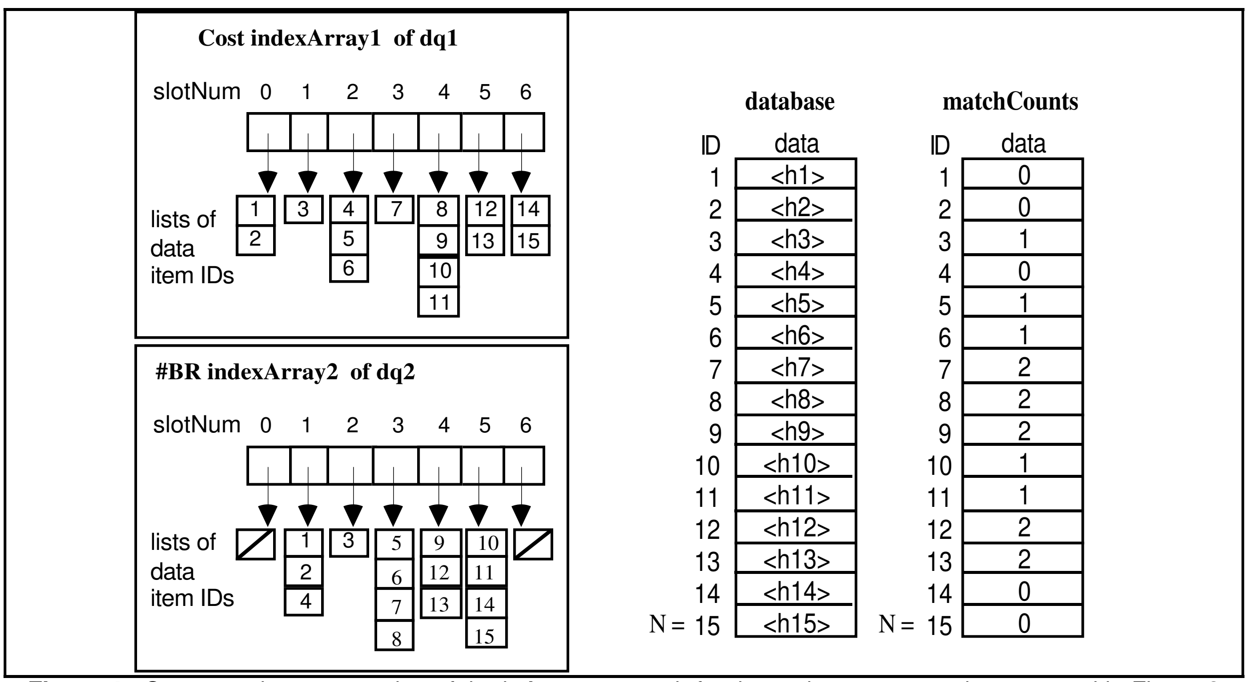


Figure 4. Conceptual representation of the k-Array approach for the real estate example presented in Figure 3 (k=2, slotCount=7, N=15). In this example, the user has selected houses with 4 to 6 bedrooms that cost greater than \$150K and less than \$250K. All houses with corresponding `matchCount =k=2` are included in the solution set.

The k-Array index organization (Figure 4) is now explained based on the real estate example presented in Figure 3. In this example, a separate `indexArray` is provided for each DQ filter corresponding to each of the `k` attributes—$\mathtt{indexArray}_1$ for the cost of houses (Cost) and $\mathtt{indexArray}_2$ for the number of bedrooms (#BR). The ID of each house object (i.e., data item in the database) is replicated `k` times and a separate copy of the ID is sorted into its corresponding slot of each of the index arrays. For example, h1 is a \$100K, 3BR house and thus its

ID is sorted into slot 0 of the cost $\texttt{indexArray}_1$ and slot 1 of the #BR $\texttt{indexArray}_2$. Since the `matchCount` of an item keeps a running tally of the number of matched dimensions for that item, it must not be replicated as in the case of the IDs of the data item. We thus store `matchCounts` by itemID in a centrally located separate array (see `matchCounts` array on right side of Figure 4).

In the disk-based version of the k-Array, we modify the conceptual representation of Figure 4 to compress and cluster data for the purpose of minimizing 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 it is unlikely that we will 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` shown on the left side of Figure 4) on disk. Since we assume the TPairs database remains frozen during temporal browsing and since item IDs are replicated (in contrast to the linked array [19] where IDs are not replicated—see Section 5), we can further enhance query processing by contiguously storing the list of data item IDs for each slot of each of the `indexArrays`.

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

```
numItems[0:slotCount-1]        // # of items in each slot of indexArray_i
// info on current position of each filter thumb of corresponding DQ filter, including slot number of DQ thumb
// and corresponding disk "pointer" (disk page number + offset) to access info in indexArray_i from disk

// LEFT pointer accesses first 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 whenever possible, sufficient buffer space is reserved to keep each $\texttt{iaInfo}_i$ 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 ≤ 122` slots in a single 2K page. When and if 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 because this data can be maintained in an ordered, contiguous manner and offsets can be used to locate the desired count. Since the maximum `matchCount` of any data item is fixed, having an upper bound of `k`, we can use a fixed sized bit representation for compressing the space required for this data. Thus, in the case of TVQL when `k=4`, the `matchCount` will range from 0 to 4 and only 3 bits per item ID are needed.

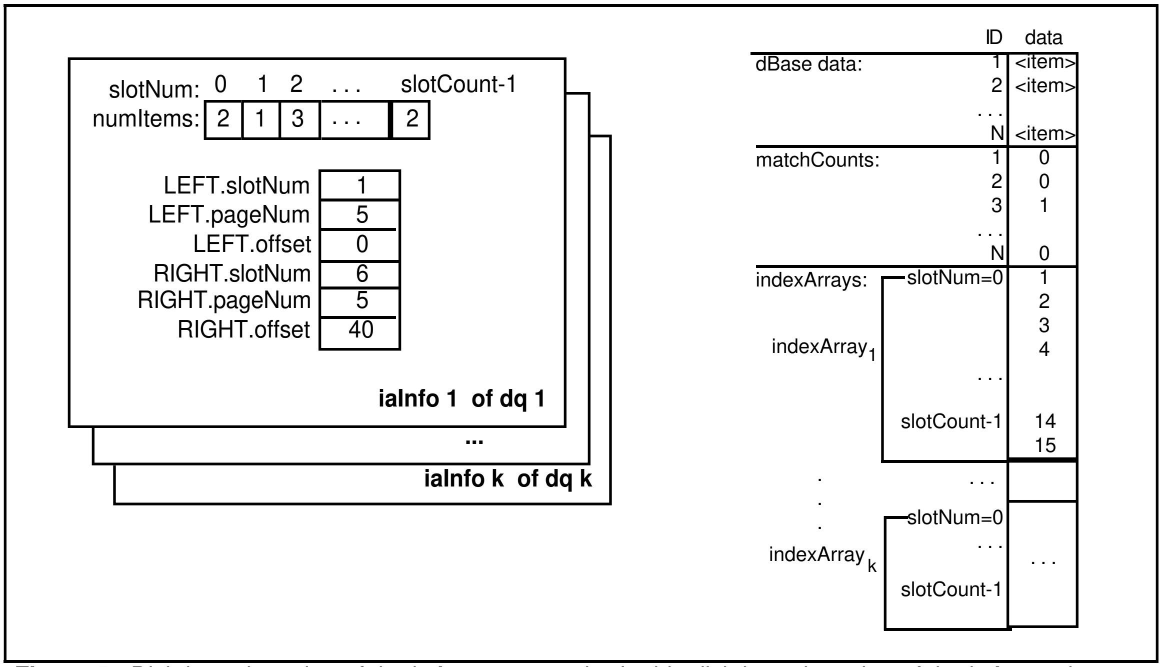


Figure 5. Disk-based version of the k-Array approach. In this disk-based version of the k-Array, the `iaInfo` data on the left side of the figure is held in main memory when possible, while the remaining data on the right side of the figure are stored on disk as indicated.

The disk-based version of the k-Array is presented in Figure 5. In addition to the actual data items (`N` = number of items) stored on disk, the disk-based k-Array formally includes:

`kArray:`
  `iaInfos` = $\{iaInfo_1, \ldots, iaInfo_i, \ldots, iaInfo_k \mid 1 \le i \le k\}$
    // set of `iaInfos` for accessing `indexArrays` from disk; kept in main memory when possible
  `matchCounts[1:N]` // array of match counts for each item, stored on disk
  `indexArrays` = $\{indexArray_1, \ldots, indexArray_i, \ldots, indexArray_k \mid 1 \le i \le k\}$
    // `indexArrays` stored contiguously on disk.

We can use the disk-based k-Array to process each incremental query specified via a users's `dqManip` (dq_i, `thumbID, action, param`) where $1 \le i \le k$ and `k` = number of dimensions, as follows:

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

3.2 The k-Bucket Index Structure

We define a *bucket* to be 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 the normalized placeholders in the full matrix index). Hence, when a data item is (not) in the solution set, its bucket is also (not) in the solution set and vice versa. For example, consider the case in the real estate database (Figure 3) where we have two houses—h5(3 bdrms, $150K) and h6(3 bdrms, $150K). If house h5 is in the solution set, then `#bdrms=3` and `cost=$150K` must each be included in the selected ranges of their corresponding dimensions. Thus, their corresponding `bucket(#bdrms=3, cost=$150K)` must also be in the solution set. Knowing h5 is in the solution then guarantees that h6 is in the solution set.

Indexes that group items together by bucket are referred to in this paper as *bucket-based structures*. While the k-Array index structure was originally designed to be item-based, it can easily be converted to be bucket-based. We accomplish this by grouping data items by bucket (keeping only non-empty buckets) and applying the k-Array indexing technique to buckets, rather than to individual data items. Match counts are associated with each bucket so that when a bucket's `matchCount` changes from `k-1` to `k`, the bucket—including all of its data items—is added to the solution. We call this improved structure the *k-Bucket* index structure and our experimental results confirm that compression based on bucket information does indeed improve query processing efficiency (see [10]).

3.3 Discussion of Index Structures for Multidimensional Range Queries

The advantages of array-based structures such as the k-Array and k-Bucket are that:

- overall storage size of the index is likely to be small even as `k` and/or `slotCount` increase, since the size of the array-based index structures is linear with respect to `N` (assuming `N >> slotCount`) rather than exponential with respect to `slotCount`, as in the case of a full matrix representation, and
- they are especially well suited for DQs since a) users only manipulate one DQ filter at a time, and in these structures, each dimension is correspondingly separated out, and b) adding or deleting nearest neighbors to the solution set does not require calculating or searching through the positions representing the `k-1` hyper-rectangle of the full matrix, since this type of information is directly accessible from the index array and encapsulated within the match counts.

On the other hand, the disadvantage of these structures is that they are subject to projection effects. This can be potentially costly in certain types of distributions such as a T- or L-shaped distribution.

In the array based methods, items or buckets are projected from k-dimensions onto each of the k separate arrays and match counts are updated to see if particular data items meet the query constraints (these counts must be updated independent of whether or not they affect the final solution set). These techniques are likely to suffer from a projection effect when, for instance, you have an L-shaped distribution in a two-dimensional data set (see Figure 6), and the methods are forced to update counts for a long-arm of the L even when they will not affect the final solution. The grid file and k-d tree index structures described in Section 4 are not subject to this type of projection effect since they identify and then search on the k-dimensional hyper-rectangle to be added rather than relying on match counts.

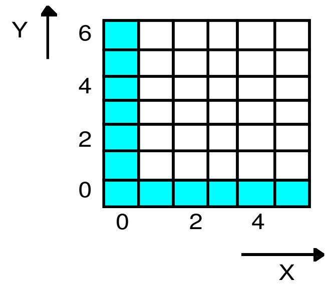


Figure 6. L-shaped distribution in a two-dimensional data set.

4. ALTERNATIVE INDEXES FOR PROCESSING MULTIDIMENSIONAL RANGE QUERIES

We compare our k-Array and k-Bucket index structures to the linked array [19], its linked bucket counterpart, an optimized grid file [15], and a k-d tree [5]. In this section, we describe these alternative index structures for query processing and discuss when a bucket-based optimization of each of the structures is possible.

Linked Array. The linked array [19] is also based on `k` index arrays—one array for each dimension. However, it is a pointer-based solution, typically used in main memory, as each linked array entry (i.e., one entry per data item) contains `k` $\texttt{next}_\texttt{i}$ pointers and the `matchCount` associated with the given item. In the case of DQs, all item counts are initially set to `k`, since we initialize the solution set to contain the full data set. While the linked array provides a sparse matrix representation, the cost of saving `k next` pointers for each data item becomes increasingly expensive as both `k` and the data set size increases. The algorithm for using a linked array index to process incremental queries is similar to the one used for the k-Array (see above) where the maintenance of `matchCounts` is used to determine when a data item should be added to or deleted from the solution set.

Linked Bucket. Similar to the way we converted the k-Array to the k-Bucket index structure, we now suggest that the linked array can also be converted into a bucket-based *linked bucket* method. That is, we can group data items by bucket, and apply the linked array indexing technique to buckets rather than to individual data items.

Grid File. A grid file [20] corresponds to a hierarchical grid, dividing the full matrix into k-dimensional hyper-rectangular subregions. The super matrix (i.e., *region* or *grid directory*) of super-buckets is created by recursively making edge-to-edge slices through the full matrix in the dimension containing the widest range of actual data values. Although the original grid file was item-based, it can easily be converted to be bucket-based (where a *bucket* is defined as above), since in DQs, buckets and items are indistinguishable during search. The maximum number of IDs within a super-bucket is constrained by the disk page size. Thus, when the region directory is constructed, the full grid matrix is continually partitioned until every super-bucket within the region directory does not contain more than the maximum number of non-empty buckets.

A set of `k` one-dimensional arrays, referred to as *linear scales,* are created to store the partition information for each attribute domain within the region directory. The grid file is most commonly known for its *two-disk access* *principle*, where a query for a *single record* can be guaranteed to be retrieved in at most two disk accesses (one to access the region directory, and one to access the super-bucket). In addition, the grid file was also designed to improve the efficiency of range queries with respect to all attributes by providing two levels of clustering—one for clustering the data items into buckets and another for clustering the buckets into super-buckets. *Incremental* search is achieved by determining the hyper-rectangular region to be added to or deleted from the solution set (rather than the region for the full solution) and then identifying the actual buckets to be added to or deleted from the solution set.

In our experimental studies, we use an optimized version of the grid file proposed by [15], where the region directory is converted from a matrix representation into a list of super-buckets. This optimized grid file can be thought of as a hierarchical sequential list. In a second optimization, the bounds of each super-bucket are "deflated" to remove any empty buckets along the edges of the hyper-rectangular region (while still maintaining a convex hyper-rectangular shape). These optimizations: 1) reduce the size of the region directory and eliminate the linear scales, and 2) eliminate unnecessary disk accesses due to inflated regions, thereby reducing overall search time.

k-d Tree. 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 we use stores data only at the leaves of the tree. Range searching the k-d tree is accomplished by recursively checking each node for overlapping regions. Search stops at non-overlapping internal nodes or overlapping leaf nodes. At the leaf nodes, a list of normalized buckets of data items are sequentially searched to determine whether or not they satisfy the query constraints. The k-d tree can be used for incremental searching by determining the hyper-rectangular region to be added to or deleted from the solution set and then using the range search method to identify the buckets to be added to or deleted from the solution set.

5. EXPERIMENTAL DESIGN

5.1 Assumptions

In this work, we give higher priority to optimizing temporal browsing (i.e., *processing* incremental multidimensional range queries) rather than maintaining updates. We assume that the index structure is static and constructed once and prior to the optimized incremental query processing. Index creation consists of one-time costs and thus are not added to our results, though we have found the build cost to be smallest for the k-Bucket and linked bucket index structures [10].

We assume that queries are specified and processed *incrementally*. Users specify queries by manipulating DQ filter thumbs as described in Section 3.3. In this way, users formulate their current query by updating parameters of their previously specified query. A query is processed incrementally in that the difference between the previous and the current query is determined and the corresponding data items are then identified and added to or removed from the solution set depending on whether the user manipulation had the effect of increasing or decreasing the selected attribute range respectively. Thus, rather than using the index structures to process the full query and identify the full solution from scratch, the indexes are used to find the data items in the difference set between the previous and current query specified. The queries examined during our simulations are thus a series of incremental queries, where a query Q_n is assumed to be specified directly by manipulating the query Q_{n-1} specification (see Sections 5.3 and 5.4). In Section 6.1, our results show that all index structures perform better when processing queries incrementally than when processing the same queries non-incrementally from scratch.

5.2 Simulation Environment

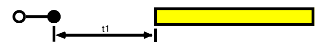
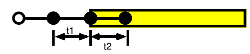
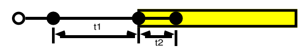
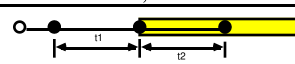
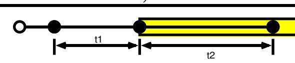
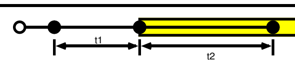
Our data set corresponds to a hypothetical 60-minute video based on average parameters taken from real video data. A and B subsets of video events were randomly generated and density was varied by varying the A to B subset size ratio. The TPairs database was then formed based on these A and B events and the extreme values of the temporal query filters (ranging from -10 seconds to 10 seconds for the startA-startB filter). 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 of approximately 12000, 24000, 48000, and 71000, 95000, 119000, and 143000 pairs and corresponding data storage requirements ranging from 290 kilobytes to 3.4 megabytes. All index structures were implemented in C++ and have been run in both UNIX and MacOS operating systems. 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).

5.3 Query Descriptions

Since no known benchmark queries exist for dynamic queries or incremental multidimensional range queries in general, we created a set of test queries (Table 1) designed to 1) explore the advantages and limitations of the various index structures, and 2) mimic real world queries (i.e., to imitate how a real user might temporally browse the data or pose specific temporal queries—e.g., see [11]).

Each test query in Table 1 is described by: its query number (column 1), the users' manipulation of TVQL used to specify the given query (column 2), the temporal semantics (new filter range specified—column 3) corresponding to the user manipulation, a visual representation of the specified query including the resulting temporal diagram and quantitative information (column 4), the size of the query's full solution presented as a percentage of data set size `N` (column 5), and `F` ($F \leq N$), the number of items found in the difference set between the current and previously specified query (last column).

Table 1. Characteristics of the ten TVQL queries used in our simulations (N=data set size; F=# of items found).

Q#	User Manipulation	Temporal Semantics (filter range selected)	Resulting Temporal Diagram & Quantitative Information	Soln. Size (%N)	F (%N)
Q1	right-click the right thumb of dTfo filter to exclude end of range, dTfo.max	dTfo.min ≤ dTfo < dTfo.max	[all temporal 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 temporal 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%

Similar to solution set size (SSS), `F` is expressed as a percentage of `N`. Note that `F` is equal to the absolute value of the difference between the SSS of the current query and the SSS of the previous query. We assume that before query Q1, the data set starts off with all items of the data set included in the solution (i.e., representing a value of 100% in column 5 of Table 1). Thus, `F` of Q1 = | 100% - (SSS of Q1) | = | 100% - 56% | = 44%; `F` of Q2 = | SSS of Q2 - SSS of Q1 | = 5%; etc. We also assume that during the main query processing evaluation presented in Sections 6.2 and 6.3, queries are processed incrementally so that query Q1 is processed by starting with the full solution set and reducing it, and that each subsequent query is processed starting from the solution provided in the previously specified query. The rationale for including each query is described below.

5.4 Justification for Query Selection

Query Rationale. Queries Q1 and Q2 test the processing of single increment user manipulations when a large portion of the database is currently selected and are most likely to be the most expensive to process. Queries Q5 and Q6 are one-increment queries for examining the effect of processing queries when only a small portion of the database is currently selected—a situation similar to the best case scenario. The best case scenario is represented by a one increment "undo" when only a small portion of the database is selected. This case is expected not to require any page faults, since only a small amount of data would need to be processed and it would already be in the buffer, having just been previously processed. Our results indicate an effect similar to the best case scenario in Query Q7.

Queries Q3, Q4, and Q8 represent intermediary queries as in a continuous sequence of inquiry, and help illustrate how users might use TVQL to pose real queries (see sample scenario below). Queries Q9 and Q10 test the k-Bucket approach under projection effects (see Sections 3.3 and 6.3). Although queries Q3, Q4, Q8 and Q9 can be specified in one separate user manipulation (i.e., by dragging one slider thumb over many increments), they represent a series of multiple one-increment queries used in our experiments to test incremental query specification *as* the user adjusts any filter thumb. Together, the series of queries test how each of the techniques perform as users browse within a temporal primitive (Q3 to Q4), between a primitive and a temporal neighborhood (Q4 to Q5), between two temporal neighborhoods (Q5 to Q6), and within a given temporal neighborhood (Q6 to Q7 to Q8 to Q9 to Q10).

Sample Scenario. To illustrate how the above series of temporal queries can be used to support real-world queries, consider the following scenario where educational researchers evaluate video data of a group of students discussing a class project. Subset A represents places in the video when Annie is speaking and Subset B represents when Billy is speaking. The researchers are interested in seeing how often Annie speaks before Billy. They start by manipulating the dTfo (i.e., endA-startB) query filter to specify the *before* temporal primitive. They can do this in a few simple manipulations, leading up to Q3. Once they specify this query, they refine it to only view situations where Annie speaks up to two seconds before Billy. They can do this with Q4. In Q5, they further refine their query to include situations when Billy starts speaking at the same time that Annie stops speaking. The researchers can then determine whether or not Billy ever interrupts Annie through Q6. In Q7 and Q8, the researchers widen the range of values for which Annie continues speaking even though Billy does not stop interrupting her, and finally in Q9, they can examine the extent to which Annie may attempt to persevere through Billy's interruptions.

6. EXPERIMENTAL RESULTS

6.1 Incremental vs. Non-incremental Queries

Figure 7 compares the performance of the various index methods for specifying incremental vs. non-incremental queries for the three queries Q2, Q3, and Q4 over a data set of N ≈ 23,600 using a small buffer of (100, 2K) pages. A query is *incremental* when it is processed as an *update* to the previous query rather than as a new query calculated from scratch. That is, the buffer effectively stays filled (hot buffer) between the execution of these incremental queries and the solution sets are incrementally updated instead of recomputed from scratch.

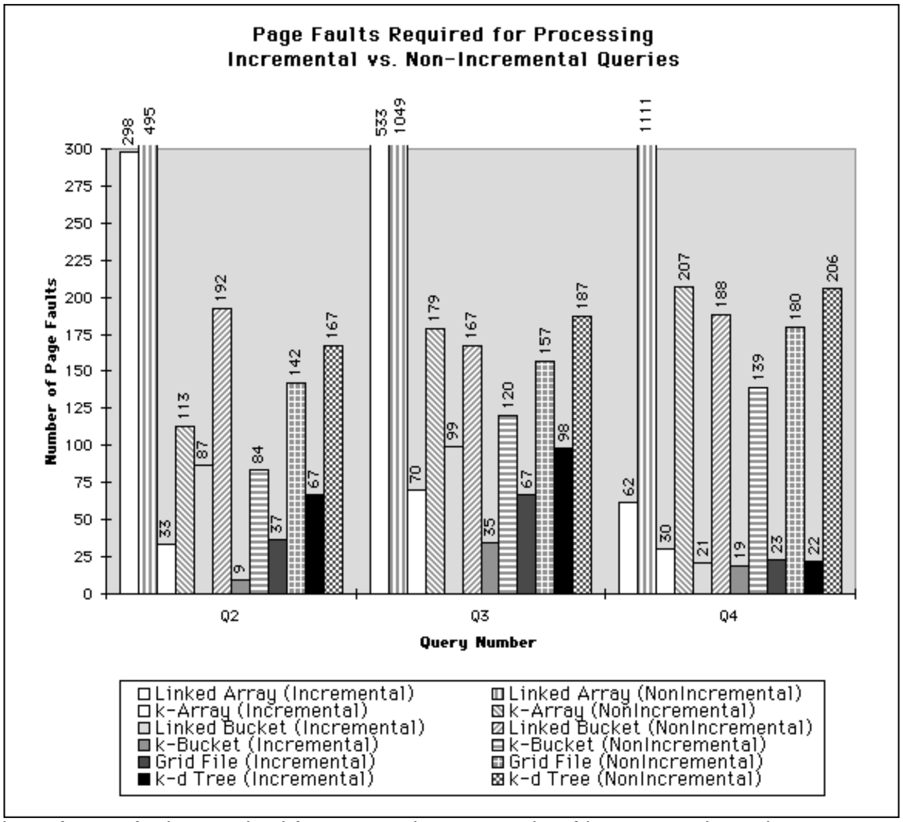


Figure 7. Number of page faults required for processing a sample of incremental queries vs. processing the same queries from scratch (results from queries Q2, Q3 and Q4): comparison between linked array, k-Array, linked bucket, k-Bucket, grid file and k-d tree (N ≈ 23,600; buffer = 100, 2K pages).

Even in the case of a relatively small data set, each method performs better by processing queries incrementally compared to using the same method to process queries non-incrementally. In addition, all incremental techniques except the linked array and linked bucket outperform the best of the nonincremental techniques for each query. In the remainder of this paper, we thus focus our evaluation on comparing the index methods based on processing queries incrementally.

6.2 Evaluating the Array-Based Methods for Incremental Processing

Our k-Array is an item-based method designed to be an improvement over the linked array. The k-Bucket and linked bucket index structures are designed to explore the impact of converting item-based methods to bucket-based ones. When we ran a series of simulations comparing the number of page faults required for using all four of the array-based index structures—the linked array, the k-Array, and their bucket-based counterparts, we found that 1) the performance of the linked array degrades considerably as data set size increases, 2) the performance of the k-Array is fairly stable for small to medium data set sizes (N < 71,500), 3) the 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 both the k-Array and the k-Bucket methods, whereas the linked array does not. Based on these results, we exclude the linked array from the remainder of our performance evaluations.

6.3 Comparing Index Methods for Incremental Processing

Comparison Over the Sample Queries. In this section, we compare the various index structures for processing the sample TVQL queries summarized in 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. We then compare the index structures under a wider variety of buffer sizes, ranging from a (100, 2K) buffer size to a (400, 8K) buffer size.

Figure 8 presents the results for processing three of the ten sample TVQL queries summarized in Table 1: a) query Q3, a sample worst-case scenario, b) query Q6, a sample better case scenario, and c) query Q9 a sample projection effect query. Note that 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].

Given F, the number of items found (to be added to or deleted from the solution set), and N, the number of items in the TPairs data set, we can use Figure 8 along with Table 1 to discuss the implications of the results for incrementally processing each type of query. Table 2 summarizes results for processing all ten sample queries when using the (100, 4K) buffer size. In columns 2 and 3 of the table, we repeat the solution set size (%N) and the size of the items found in the difference set F (also in terms of %N) that were presented in the last two columns of Table 1. We then summarize the results by indicating the minimum and maximum page faults required for the largest N (columns 4 and 5 of Table 2), the differences between the maximum and minimum page faults required (column 6), the best and worst index structures (columns 7 and 8), and other observations for each query (column 9).

Figure 8. Number of page faults required for processing 9 of the 10 sample TVQL queries for various data set sizes: comparison between k-Array, linked bucket, k-Bucket, grid file and k-d tree methods (buffer = 100, 4K pages; Q7 never requires any page faults for any of the methods).

Figure 8.a.
Query Q3, a sample worst-case scenario query.

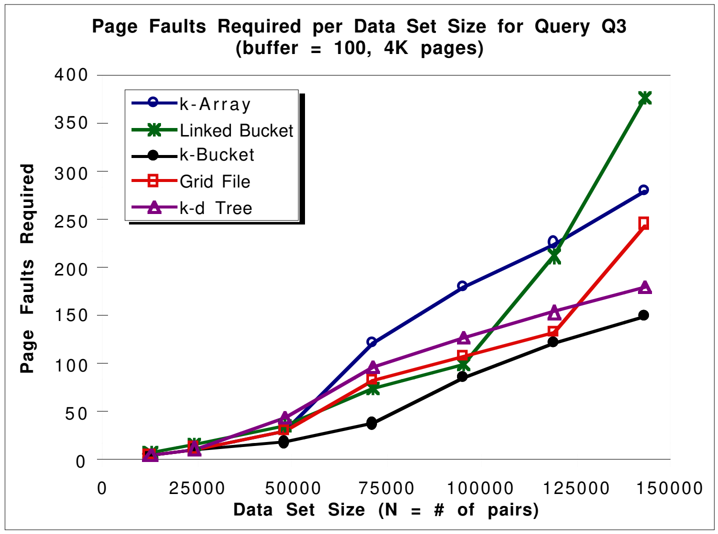


Figure 8.b.
Query Q6, a sample better case scenario query.

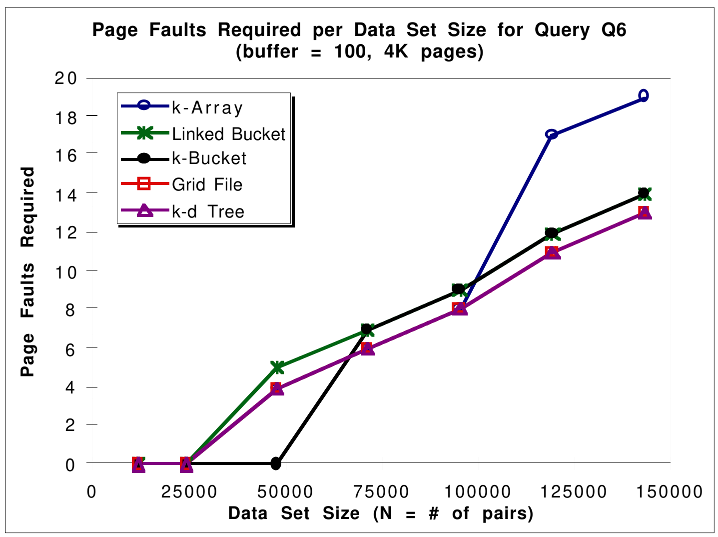


Figure 8.c.
Query Q9, a sample projection effect query.

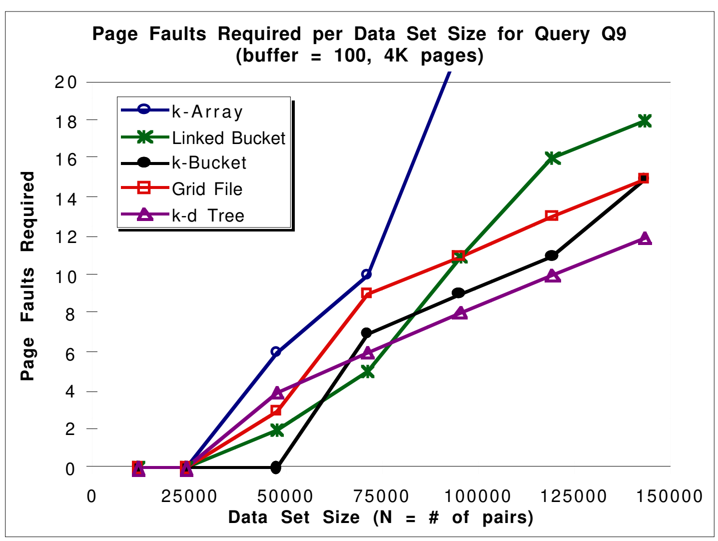


Table 2. Summary of query processing results presented in Figure 8. (N=data set size; F=number of items found; kA=k-Array; kB=k-Bucket; LB=linked bucket; GF=grid file; KD=k-d tree)

Q#	Soln. Size (%N)	F (%N)	Page Faults Required for Largest N			Index Performance at Largest N		Other Observations
			min	max	max -min	best	worst	
Q1	56%	44%	172	231	59	kB	kA	kB best; kA worst; LB and GF similar w/LB barely better; KD slightly worse than LB and GF
Q2	51%	5%	136	246	110	kB	LB	kB best; KD second best; for larger N, difference between KD and kB decreases; LB worst; GF similar to LB for larger N
Q3	23%	28%	150	376	226	kB	LB	kB best; KD second best; LB worst and exponential w/N, but fair for N < 100,000; GF fair for N < 120,000
Q4	5%	18%	54	64	10	LB	kA	LB best; kA worst; all but kA close to LB
Q5	7%	2%	7	24	17	kB/ KD	kA	kB and KD best; LB slightly worse than kB and KD; kA exponential w/N
Q6	11%	4%	13	19	6	GF/ KD	kA	kA worst; GF and KD best; kB and LB similar and next best;
Q7	21%	10%	0	0	0	-	-	-
Q8	27%	6%	18	33	15	kB	kA	kB best; kA worst and exponential w/N; LB 2nd best for N < 125,000; GF and KD similar, but GF better
Q9	31%	4%	12	33	21	KD	kA	KD best; kB 2nd best for larger N, followed by GF; kA exponential w/N; LB also degrades w/N
Q10	32%	1%	2	68	66	GF/ KD	kA	GF and KD best, then LB; kA exponential w/N; kB slightly worse than LB for larger N

Table 2 indicates that `F` is greatest for queries Q1, Q3, and Q4, and smallest for queries Q10, Q5, Q6, and Q9. Although `F` is relatively small for query Q2, we expect the processing of Q2 to be more expensive than Q8 since in Q2 larger ranges are selected for each attribute and hence the area of the hyperrectangular difference region to be searched is also bigger (refer to Section 5.4 on Query Rationale). Using Table 2 and referring back to the sample graphs in Figure 8, we can identify the following trends comparing the indexes for processing TVQL queries:

- 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 seen in the graphs for the *most expensive* queries Q1, Q2, and Q3. Also, the k-Bucket outperforms the other methods for large `F` and large `N` (refer to queries Q1 and Q3).
- the *k-d tree* is the best performer for four of the queries depicted in Figure 8 (Q5, Q6, Q9, and Q10). These are exactly the same queries that are characterized by the smallest value of `F`. 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 [16].
- 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.

Projection vs. Non-Projection Effect Queries. Recall that the array-based indexes (k-Array, k-Bucket, and linked bucket evaluated above) are created by projecting points in k-dimensional space onto each one of the dimension arrays and relying on the updating of match counts to determine query results. We expect these types of methods to be particularly sensitive to L- or T-shaped projection effects (Section 3.3) and designed queries Q9 and Q10 to examine such effects. The result in Figure 8.c indicates that the array-based methods are indeed sensitive to projection effects. The item-based k-Array is the most obviously affected, increasing exponentially with data set size. The k-Bucket, while not the best performer in these projection effect queries (see summary results of Q9 and Q10 in Table 2), provides a dramatic improvement over the k-Array. The linked bucket performs worse than the k-Bucket in Q9 but better than the k-Bucket in Q10. While the k-Bucket is not the best performer in the projection effect situation, it does perform much closer to the other methods in this situation than the other methods do to the k-Bucket in other situation. This is seen when comparing the range of page faults required for each query presented in Table 2. In Q10, the k-Bucket only requires two additional page faults while in Q1, the k-Bucket requires a minimum of nineteen page faults less than the grid file or k-d tree.

Query Q1 also includes processing data in the projection effect area (i.e., in both Q1 and Q10, dTfo is examined at its maximum filter endpoint—see Table 1), but this is not extraneous processing as in Q10, since in the case of Q1, the data in the projection effect region will actually be removed from the solution set. This is in contrast to Q10, where the data in that region will only have match counts updated, but will not affect the actual solution set size. In the situation where changes in the projection effect region affect the solution set, the k-Bucket method actually performs better than the other bucket methods (the array-based linked bucket, however, does not perform better since it is more sensitive to large F and large N).

Comparison Over Various Buffer Sizes. If we assume that each of the ten queries is just as likely to occur as any other, then 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 of these queries for each data set size of each type of index structure. Figure 9 presents summary graphs for three sample buffer sizes, representative ones out of the nine possible combinations of 100, 200, and 400 number of buffer pages with 2K, 4K, and 8K buffer page sizes that we examined.

We provide a summary over all nine buffer sizes in Table 3 for a large data set size (N ≈ 143,000). The first two columns of the table indicate the number of pages and page size for the given buffer while the last column lists the minimum number of page faults used by the best index performer. The difference from this minimum, in number of page faults, is then listed for each index in columns 3 to 7 of the table. Finally, markers are used to indicate the best (*), worst (-) and second worst (4, i.e., 4th worst) performers.

The results in Figure 9 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 (i.e., buffer = 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]. At the same time, the k-Array is the worst overall performer for all but the smallest buffer size. This further confirms the benefit of bucket-based methods over item-based ones. Examining the graphs in Figure 9, we see in particular that the k-Array is only competitive for relatively small data set sizes (i.e., for around N < 50,000). This is likely due to a smaller ratio between the number of items to the number of buckets at the smaller data set sizes. That is, when smaller data set sizes have a relatively small ratio between the number of data items and number of buckets, the overhead for storing bucket information may cancel much of the gain 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.

Figure 9. Summary of page faults required per data set size (sum of all queries by index structure): comparison between k-Array, linked bucket, k-Bucket, grid file and k-d tree methods over various buffer sizes.

Figure 9.a.
buffer = 200, 2K pages.

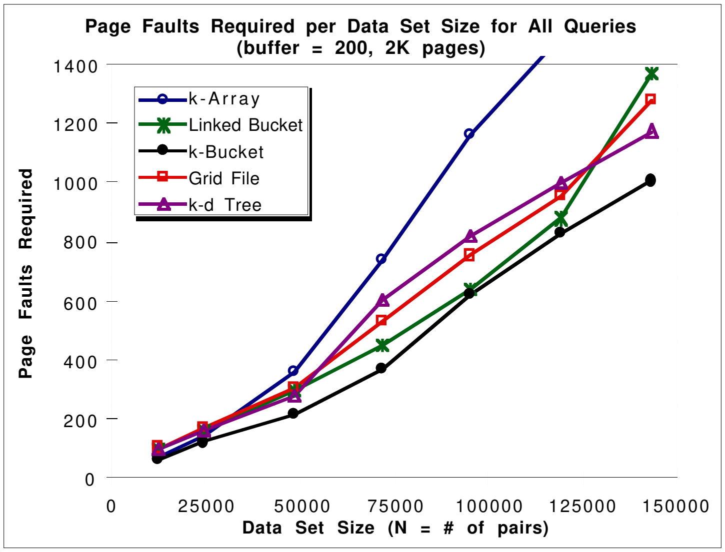


Figure 9.b.
buffer = 200, 4K pages.

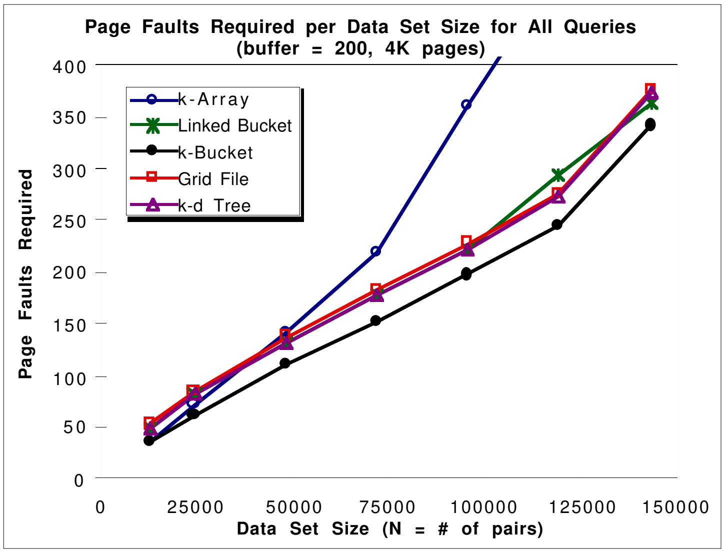


Figure 9.c.
buffer = 200, 8K pages.

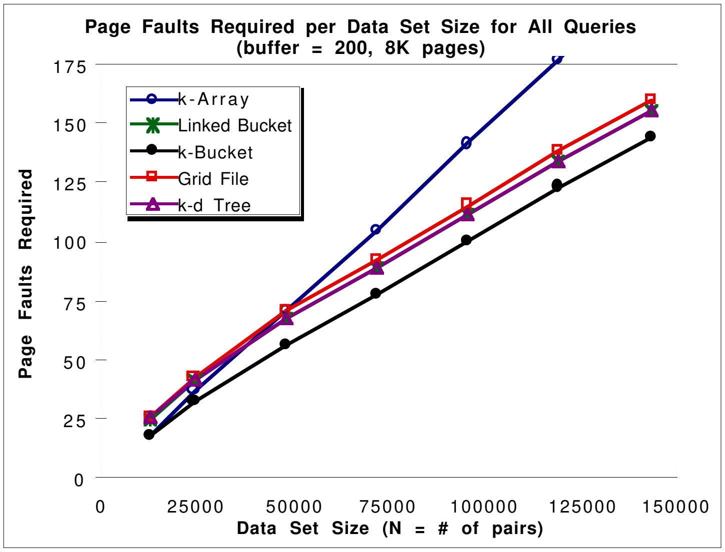


Table 3. Additional cost (in # of pages) for each index structure for a large data set size, based on the difference from the minimum (min.) number of page faults used.(* = best performer, - = worst performer; 4 = 4th performer)

Buffer # Pages	Buffer Page Size	k-Array	Linked Bucket	k-Bucket	Grid File	k-d Tree	min.
		Index Structure (Data Set Size N ≈ 143,000)					
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 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.

7. DISCUSSION AND RELATED WORK

While more recently, research in temporal queries over continuous data has emerged [22], such work has not focused on the notion of temporal browsing. In order to support temporal browsing, we have examined the processing of temporal queries over continuous temporal data as a multidimensional range query problem and have characterized the data in terms of how it can be clustered on disk (see [10]) as well as how it is inherently skewed and subject to L-shaped projection effects.

Although research in multidimensional range queries has been ongoing for quite some time now, even recent reviews of current techniques indicate that there is not one multi-attribute method that has become the standard of choice [17]. As the number of dimensions increases, node splitting and data clustering continue to be interesting and challenging problems. In addition, many researchers continue to look at the problem of handling updates and keeping balanced structures, particularly with the use of tree-based data structures (e.g., [18]). In our work, we have examined the problem from a different perspective—one where we want to optimize query processing over updates, so we assume that the data set remains frozen or that building the index structures and processing updates can be done off-line, if necessary. In particular, we explore the use of various techniques for *incremental* queries. These criteria of optimizing processing over updates and processing queries incrementally are critical for preserving the browsing paradigm in MMVIS. To our knowledge, these types of queries (i.e., DQs) have only been examined by [16], 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 newly proposed k-Array and k-Bucket indexing techniques to one base-case index (the linked array [19] and its linked bucket counterpart) as well as to well-established and popular indexing structures such as the grid file [20, 15] and k-d tree [4, 5].

Although the quad tree, r-tree and their many variants [4, 8, 21] are also popular indexing structures for multidimensional range queries, we chose not to evaluate these structures since (1) results from [16] 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.

8. CHAPTER SUMMARY

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 [14]). We introduced a new array-based indexing method called our k-Array and its even more efficient bucket-based counterpart the k-Bucket.

We first confirmed that using each index to incrementally process queries is more efficient than using the same index to calculate the query from scratch. We then showed how item-based methods such as the linked array and k-Array can easily be converted to bucket-based methods and verified through our experiments that this conversion does indeed improve performance. When comparing the k-Array, linked bucket, k-Bucket, grid file and k-d tree for a series of TVQL incremental queries processed using a (100 page, 4K) buffer, we found that the k-Bucket performs the best overall, but that it is indeed subject to projection effects. We identified the potential for L-shaped skewed data distributions for temporal pair data sets and analyzed the structures based on this skewed distribution. Our results show that in the case of the k-Bucket, there are bigger gains in using the k-Bucket in the non-projection effect situations than there are losses in the projection effect scenarios.

Finally, we compared the performance of the bucket methods and the k-Array 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. Our results showed 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 to the k-Bucket, 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. And, 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.

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 Proc*. 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.L. (1975). Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 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, P., Gonnet, G., Puech, C., Robson, J.M. (1993). Analytic Variations on Quadtrees. *Algorithmica*, 10, 473-500.
9. Freksa, C. (1992). Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54(1992), 199-227.
10. Hibino, S. (1997). *MultiMedia Visual Information Seeking*. PhD disseration.
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. (1996). “MMVIS: A MultiMedia Visual Information Seeking Environment for Video Analysis,” *CHI’96 Conference Companion*. ACM Press, 15-16.
14. Hibino, S., & Rundensteiner, E. A. (1996). “A Visual Multimedia Query Language for Temporal Analysis of Video Data,” *Multimedia Database Systems: Design and Implementation Strategies* (K. Nwosu, B. Thuraisingham, & P.B. Berra, Eds.). Norwell, MA: Kluwer Academic Publishers, 123-159.
15. Hinterberger, H., Meier, K.A., Gilgen, H. (1994). Spatial Data Reallocation Based on Multidimensional Range Queries. 228-239.
16. Jain, V. & Shneiderman, B. (1994). Data Structures for Dynamic Queries: An Analytical and Experimental Evaluation. *Proc. of the Workshop on Advanced Visual Interfaces*. NY: ACM, 1-11.
17. Lomet, D. (1992). A Review of Recent Work on Multi-attribute Access Methods. *SIGMOD Record*, 21(3), 56-63.
18. Nakamura, Y., Abe, S., Obsawa, Y. Sakauchi, M. (1993). A Balanced Hierarchical Data Structure for Multidimensional Data with Highly Efficient Dynamic Characteristics. *IEEE Transactions on Knowledge and Data Engineering*, 5(4), 682- 694.
19. Knuth, D.E. (1973). *The Art of Computer Programming, Vol 3: Sorting and Searching*, Addison-Wesley.
20. 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.
21. Samet, H. (1990). *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley.
22. Seshadri, P., Livny, M., and Ramakrishnan, R. (1994). Sequence Query Processing. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. NY: ACM, 430-441.
23. Snodgrass, R. (1992). Temporal Databases. *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space* (A.U. Frank, I. Campari, and U. Formentini, Eds.), Springer-Verlag: New York, 22-64.