Query Processing in the MultiMedia Visual Information Seeking Environment: A Comparative Evaluation

Stacie Hibino and Elke A. Rundensteiner
Electrical Engineering & Computer Science Department, Software Systems Research Laboratory
The University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122 USA
hibino@eecs.umich.edu, rundenst@eecs.umich.edu

Abstract

Although much research has been conducted in the area of multidimensional range queries, we examine this problem from a new perspective. In particular, our goal is to support the processing of *incremental* multidimensional range queries specified via our temporal visual query language (TVQL), a direct manipulation visual query interface. In this paper, we describe the details and context of our problem, emphasizing the need to support temporal browsing and examining the characteristics of temporal data. We present a simple but new array-based index structure called our k-Array and its bucket-based counterpart, the k-Bucket. We then describe the series of experiments over several temporal queries that we ran to compare the k-Array and k-Bucket with basic methods such as the linked array and other popular bucket-based methods such as the grid file and k-d tree. Our results show that the k-Bucket performs the best overall, even though it is subject to projection effects. The other methods also perform fairly competitively, but only under certain conditions.

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 those for business (e.g., inventory databases or real estate applications—"show me all homes with 4-5 bedrooms that cost between $200K and $250K dollars") or scientific analysis. Dynamic query (DQ) filters and visual information seeking (VIS) provide a novel approach to filtering data through multidimensional range searching [2]. 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 also been shown to aid users in trend searching [1].

While the problem of processing multidimensional range queries has been examined by several researchers [6, 7, 8], it now must be re-examined within the needs and constraints of this new visual query paradigm based on dynamic queries and VIS. For example, in order to preserve and support the notion of browsing within VIS (i.e., so that the visualization of results is dynamically updated *as* the user manipulates any query filter), we want to optimize and give priority to the search cost over other costs such as updates. In particular, we assume that the data set remains frozen for the index structure used (this is very realistic as the data may be, for instance, a collection of video events being used for the purpose of analysis). When and if updates do occur, the index structure can be rebuilt. Even when rebuilding the index is time-consuming, it can be done off-line and/or overnight—an approach commonly taken by large text-based information retrieval systems. While this increases the cost of updates, it decreases query processing costs which are critical to preserving the VIS paradigm.

Another important characteristic of VIS is that queries are specified *incrementally*. That is, users specify successive queries by refining their current query. 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). These types of queries must thus also be processed 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.

In previous work on analyzing various main memory data structures for dynamic queries, Jain and Shneiderman [15] used both an analytical model and experimental results to determine 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 simple but new customized index structures referred to as our *k-Array* and *k-Bucket* indexes (see Section 4.2), 2) examining additional index structures such as an optimized grid file [14], 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 (since previous work focused on main memory models and analysis [15]). The goal of this work was to optimize query processing for our temporal visual query language (TVQL), a direct manipulation interface which uses specialized DQ filters for posing and browsing temporal relationships [13, 10]. Thus, in addition to our contributions on processing DQs in general, we also present our results on examining clustering techniques and skewed data distributions more specifically related to temporal data.

This paper is divided into seven additional sections. In Section 2, we review our temporal visual query language (TVQL). In Section 3, we present parameters and assumptions for processing dynamic queries and in Section 4, we provide descriptions of index structures for multidimensional range queries. Section 5 describes our experimental design while Section 6 presents our experimental results. In Section 7, we discuss related work and finally in Section 8, we present our conclusions and future work.

2. TVQL: TEMPORAL VISUAL QUERY LANGUAGE

TVQL is part of our integrated MultiMedia Visual Information Seeking (MMVIS) system[1], an interactive visualization environment based on extending VIS technology for video analysis [10]. Our TVQL is a visual, direct manipulation approach towards specifying queries over video by non-experts. In MMVIS, users select two subsets of video events via subset selection palettes. They can then use TVQL to "query" temporal relationships between the subsets of events. For example, suppose we have video of an Olympic two-man beach volleyball tournament. If subset A was set to select all events when players come in contact with the ball (e.g., player P1 digging, setting, hitting, or blocking the volleyball, player P2 digging, etc.) and subset B was set to select all video events representing both successful and unsuccessful side-out and point plays, then TVQL could be used to specify queries such as "show me all places where players *start at the same time as* a side-out or point play." Such a query would allow us to easily compare and contrast the frequency in which the different players *initiate* both successful and unsuccessful plays, since TVQL is tightly coupled to a temporal visualization (TViz) of results [10, 11]. In TViz, bars are drawn between the various types of A and B events to indicate the presence and frequency of the temporal relationship between them. In the above query, a thick bar between player P1 digging and a point play would indicate that P1 frequently received and successfully passed the first ball on a point play. As users manipulate the TVQL filters, the relationship bars between A and B events appear and disappear, and grow and shrink, giving immediate feedback to the users and allowing them to see the one-to-one correspondence between adjusting their temporal query and the update to the visualization of results. Thus, because TVQL supports this notion of temporal browsing, users can simply adjust any slider to incrementally update their temporal query and explore relationships such as how often players *participate* in the different types of plays, how often they *end* at the same time as various plays, etc.

While a formal specification of TVQL can be found elsewhere [13], we review its basic principles here as needed for the remainder of this paper. Given two events A1 (○—●) and B1 (▭) with nonzero duration, there are thirteen possible primitive temporal relationships between them: *before, meets, during, starts, finishes, overlaps,* the symmetric counterparts to these six relationships, and the *equals* relationship [3]. Although there are four pairwise relationships between temporal starting and ending points of the events (e.g., start A1 - start B1), only one to three of these relationships are required to define any one temporal primitive (see Figure 1). TVQL provides a separate DQ filter for each of the defining endpoint difference relationships, thereby allowing users to specify any one of the temporal primitives. More importantly, this set of temporal query filters supports users in 1) specifying combinations of similar primitives (i.e., *temporal neighborhoods* [9], equivalent to selecting a series of adjacent cells such as a row, column, or grid from Figure 1) and 2) *browsing* temporal relationships (since users can directly manipulate the filters with no specific query in mind).

[1]A working prototype of TVQL and MMVIS has been fully implemented and a formal demonstration of the system was presented at CHI'96 [12].

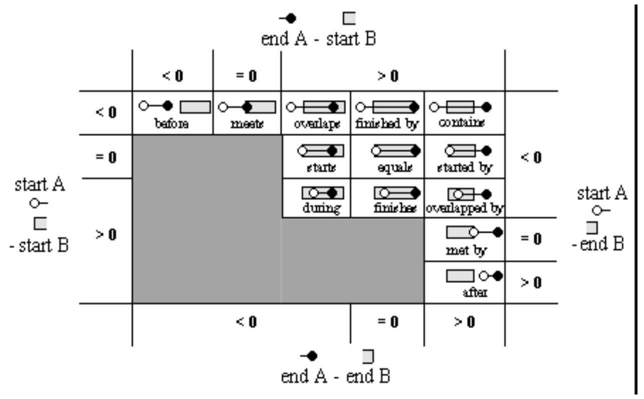


Figure 1. Relationships between temporal primitives and the four defining endpoint difference relations.

Based on the previous volleyball scenario, Figure 2 illustrates how users could ask the query "show me how often players *start* both successful and unsuccessful side-out and point plays." Note that in addition to the temporal query filters, TVQL is further enhanced with qualitative descriptive labels along the top and side, and our dynamic temporal diagrams along the bottom. The labels allow users to "read" the relationship specified and the diagrams provide visual confirmation of the temporal primitive(s) specified (though not quantitative values as in the filters).

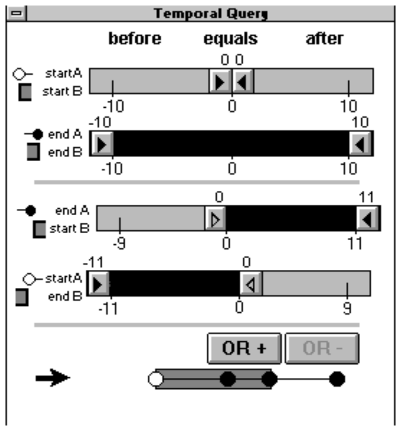


Figure 2. TVQL palette. This query specifies all events of type A that start at the same time as B events but can end before, at the same time, or after B events.

3. BACKGROUND ON PROCESSING TVQL QUERIES

3.1 Basic DQ Characteristics

The problem of processing dynamic queries in TVQL can be characterized as a multidimensional range query problem in which queries are *incrementally* specified. More specifically, given a set of DQ filters, we have:

- each DQ filter represents one attribute of the data set,
- 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 using direct manipulation to adjust and select valid attribute ranges for each DQ filter and is incremental 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 each attribute 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. In the case of TVQL, an item is an (a_i,b_j) temporal pair with $a_i \in A$ and $b_j \in B$. Each item pair has one startA-startB value, one endA-endB value, etc. and can thus be sorted into one slot of each of the corresponding four temporal query filters. A given temporal pair is then in the solution set when each of its four temporal endpoint differences is in the selected range of each of the corresponding temporal query filters.

3.2 TVQL Parameters

Before we present our approach to TVQL processing, 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, . . . , tic_z}	\\ set of evenly spaced valid slider thumb value positions
`dtIncr`	= delta increment: min amount you can change an endpoint value of a slider 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	
`slotCount`	= number of slots for the given filter = (2*(maxVal - minVal)/dtIncr) + 1	

Given:

`V`	= {v_1, v_2, . . . , v_s}	\\ set of video documents
`Ann(`v_p`)`	= {va_1, va_2, . . . , va_T | T = number of annotations}	\\ annotations for video v_p

Users select A and B subsets via subset selection query palettes:

A = {a_1, a_2, . . . , a_m | $m \leq T$, $a_i \in$ Ann(v_p)}	\\ first subset of video annotations
B = {b_1, b_2, . . . , b_n | $n \leq T$, $b_j \in$ Ann(v_p)}	\\ second subset of video annotations

Based on the subset selected, a new database of temporal pairs (TPairs) is formed as follows:

$$\text{TPairs} = \{(a_i, b_j) \mid a_i \in A,\ b_j \in B,\ a_i.\text{end} - b_j.\text{start} \geq \text{dTfo.minVal AND } a_i.\text{start} - b_j.\text{end} \leq \text{dTof.maxVal}\},$$

where:

`dTfo` =	temporal DQ filter for specifying difference between end of a_i and the start of b_j, and
`dTof` =	temporal DQ filter for specifying difference between start of a_i and the end of b_j.

Users explore temporal relationships by using TVQL to query the TPairs database. We define:

`T` = total # of records in annotation collection Ann(v_p), i.e., the data set size from which subsets are selected
`N` = # of (a_i, b_j) pairs in TPairs ($\leq T^2$) from which pairs that meet temporal relationship criteria are selected
`k` = number of attributes or dimensions used for query specification (for TVQL, k=4)

3.3 TVQL Query Processing

Given that users *incrementally* specify and update queries by directly manipulating any DQ filter thumb, we must essentially be able to find the "nearest neighbor" when processing these queries. Given a double-thumbed query filter such as those used for TVQL, 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. These types of manipulations can be represented as:

```
dqManip(dqi, thumbID, action, <direction>)
```

where

`dq`$_i$	// identifies DQ filter being adjusted,
`thumbID` ∈ `{LEFT, RIGHT}`	// indicates which DQ filter thumb was manipulated,
`action` ∈ `{MOVE, FILL, UNFILL}`	// indicates the type of manipulation made, and
`direction` ∈ `{LEFT, RIGHT}`	// indicates the direction the DQ thumb was moved, if applicable.

Figure 3 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 3a, the DQ filter is selecting the range 0 ≤ values < 1. This corresponds to including slots 6 and 7 of the internal representation shown in Figure 3b.

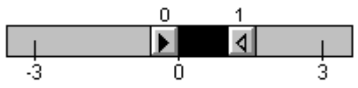


Figure 3.a. Sample DQ filter. (minVal = -3, maxVal = 3, dtIncr = 1)

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

Figure 3.b. Corresponding internal representation where the double-lined border around slots 6 and 7 corresponds to the selected DQ filter range in Figure 3.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 `count` represents the number of valid attribute values for pr_q, then pr_q is in the solution set when `count=4` (or, in general, when `count=k`, the number of dimensions).

4. INDEX STRUCTURES FOR MULTIDIMENSIONAL RANGE QUERIES

This subsection summarizes existing and new index structures for multidimensional range queries. The key features for each structure are highlighted and a description of how incremental search is accomplished is given.

4.1 Strawman Index Structures

These base case index structures are simple and straightforward to both implement and search.

Sequential List. A sequential list is simply a list of all items in the data set. Search is performed by scanning once through all items of the list to collect those which fall within all of the specified query ranges for each dimension. Search time is enhanced by keeping a sorted list.

Grid. A grid is a k-dimensional hyper-rectangular matrix representing every possible combination of every unique normalized value (i.e., based on `slot` value rather than attribute value) in each dimension. Searching a grid is thus reduced to simple lookups. If `slotCount` is the same for each of the `k` DQ filters (i.e., dimensions), then the size of the grid index would be $\texttt{slotCount}^{\texttt{k}}$. In skewed data distributions, much of the grid index may be empty. Since the size of the matrix remains fixed, its storage cost may be much higher than necessary. An advantage of a grid for DQs, however, is that locating the nearest neighbors in any dimension as required when incrementally adjusting query filters can be easily calculated and accessed. (Note that for a k-dimensional range query, the nearest neighbors in any dimension are represented by a `k-1` hyper-rectangle of the full matrix.)

4.2 Item- and Array-based Index Structures

The item- and array-based index structures are simple sparse matrix representations. They are essentially based on projecting data items from k-dimensional space onto `k` separate 1-dimensional arrays, one for each individual dimension, and associating a `count` with each item. Search is accomplished by accessing items in the valid ranges of each dimension and updating their counts. When a `count` changes from `k-1` to `k`, an item is added to the solution, and when it changes from `k` to `k-1`, it is deleted from the solution set. The advantage of these structures are that 1) overall storage size of the index is likely to be small (especially as `k` and/or `slotCount` increase), since the size of the array-based index structures is linear with respect to N rather than exponential with respect to slotCount, as in the case of the grid index, 2) 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 encapsulated within the 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.

Linked Array. In the linked array [18], we have a set of `k` arrays:

$$\texttt{lArray = \{indexArray}_1\texttt{,..., indexArray}_i\texttt{,...,indexArray}_k \mid 1 \le i \le k\}$$

where each array has a corresponding size `slotCount`$_i$. Each slot of each array points to the ID of the first item in its slot. The linked array also has an array of data item information, `lArrayInfo[1..N]`. Each entry in `lArrayInfo` contains `k next` pointers and the `count` associated with the given item. The index structure is created by sorting each data item into its corresponding slot for each of the k dimensions. If a slot is empty, then the slot is set to the current item ID. If the slot already contains a first item ID, then the `next`$_i$ pointer of the current ID is set to the first item ID of the slot (i.e., `lArrayInfo[itemID].next[i] = indexArray`$_i$`[slotNum]`), and the slot is set to the current ID (i.e., the current ID is inserted at the front of the list of items for that slot). In the case of DQs, all item counts are set to `k`, since we start by initializing 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 the data set size increases.

k-Array. We propose our k-Array as an optimization to the linked array structure, especially when secondary storage is required. In the k-Array, we also have a set of k arrays:

$$\text{kArray} = \{\text{indexArray}_1, ..., \text{indexArray}_i, ..., \text{indexArray}_k \mid 1 \le i \le k\}$$

In the k-Array, however, rather than keeping an array of data item information such as `lArrayInfo[]`, we instead replicate item IDs for each dimension and separate out their counts, using only the minimum number of bits necessary to specify each `count` (e.g., for `k=4`, only 3 bits per item ID are needed). We do this by setting:

`indexArray`$_i$ `[slotNum]` = # of items in slot
and storing: `itemCounts[N]` = array of counts for each item // stored with minimum # of bits required

We also reserve sufficient buffer space to keep the indexArrays in main memory, if possible, along with the slotNum, page number, and page offset corresponding to the current position of each query filter thumb. The list of data item IDs are stored on disk, in the order in which they occur in the indexArrays. Thus, each slot of each array, along with the left and right query thumb information "points" to a list of item IDs, rather than to entries of lArrayInfo[]. While the k-Array does not save much space over the linked array (primary savings is obtained by reducing the space required for the counts), the replication of IDs in place of next pointers allows us to store information for each slot *contiguously*. The basic structure of our k-Array is depicted in Figure 4.

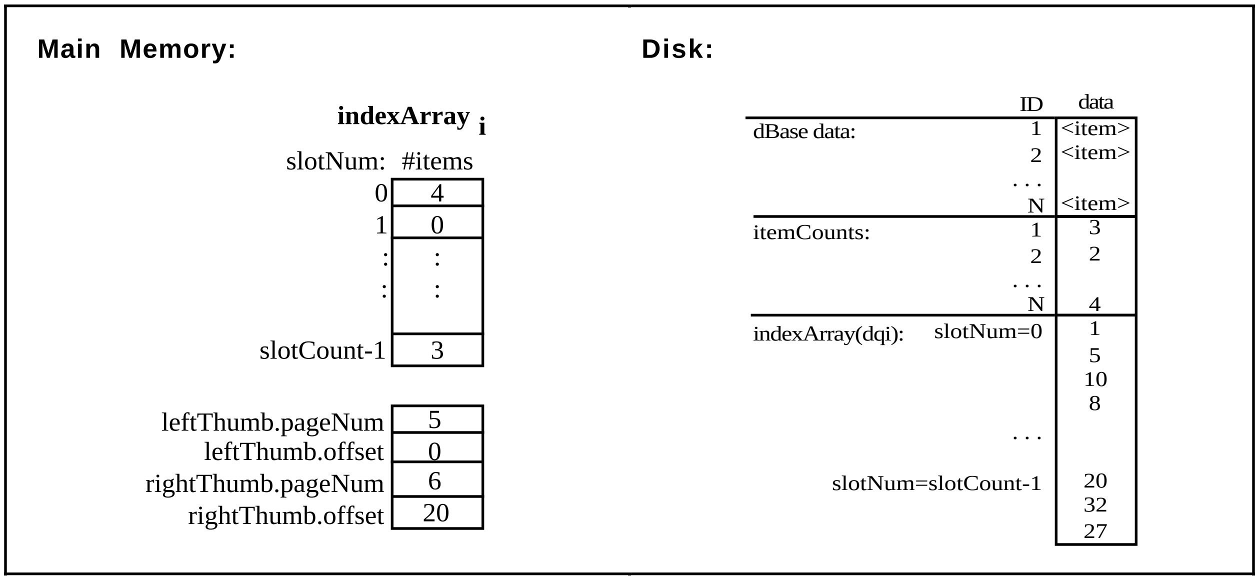


Figure 4. Disk-based version of the k-Array approach.

Our k-Array processes each incremental query specified through a dqManip as follows:

- get the slotNum, page number and page offset for the thumbID of the dq filter specified
- using the thumbID, action, and direction (if specified), determine whether the nearest neighbor to update is dq_i.thumbID.slotNum - 1 or dq_i.thumbID.slotNum + 1
- using the thumbID, action, and direction (if specified), determine whether the manipulation corresponds to expanding or contracting the selected attribute range
- loop from 1 to slotSize, doing the following:
  - fetch the next item id
  - fetch its count
  - if removing the current slot from the selected attribute range, then

```
            - update the item's count by decrementing it
            - if new count = k-1, then fetch actual data item
              and remove it from the solution set
    - otherwise we are adding the current slot to the selected attribute range, so:
            - update its count by incrementing it
            - if new count = k, then fetch actual data item
              and add it to the solution set
```

Our experimental results in Section 6 show how our k-Array performs much better than the linked array.

Discussion of Array-Based Methods. In the array based methods—the linked array and k-array (as well as the linked bucket and k-Bucket described below), items or buckets are projected from k-dimensions onto each of the k-arrays and counts are updated to see if particular data meet the query constraints (these counts must be updated independent of whether or not they effect 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 5), and the array-based methods are forced to update counts for a long-arm of the L even when they will not affect the final solution. For example, if in Figure 5, we had $2 \leq X \leq 4$ and $Y = 3$ and we extended the range of X to include the first row, then the counts for all black tiles in that row would have to be updated, even though only the tile in row Y=3 will be added to the solution set.

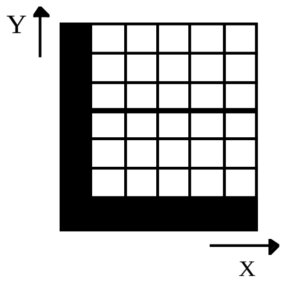


Figure 5. L-shaped distribution in a two-dimensional data set.

4.3 Bucket-Based Structures

Linked Bucket and k-Bucket. While the linked array and k-array index structures were originally designed as item-based structures, it turns out that they can easily be converted into bucket-based methods. If a *bucket* is defined as the smallest unit of search (i.e., equivalent to the normalized placeholders in the full grid matrix), then when a data item is in the solution set, its bucket is also in the solution set and vice versa. Similarly, 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 where we have two houses—$house_1$(3 bdrms, $200K) and $house_2$(3 bdrms, $200K). If $house_1$ is in the solution set, then `#bdrms=3` and `cost=$200K` must each be included in the selected ranges of their corresponding dimensions. If these values are included in the selected ranges, then their corresponding `bucket(#bdrms=3, cost=$200K)` must also be in the solution set. Thus, we see that knowing $house_1$ is in the solution also guarantees that $house_2$ is in the solution set. If items were grouped by bucket (and we only kept the non-empty buckets), then we could easily apply the linked array and k-array indexing techniques to buckets, rather than to individual data items. Counts would be associated with each bucket so that when a bucket's `count` changed from `k-1` to `k`, the bucket—including all of its data items—would be added to the solution. We call these improved structures the *linked-bucket* and *k-bucket* index structures. Our experimental results confirm that this notion of compressing the data index based on bucket information does indeed improve the efficiency of query processing (see Section 6).

Grid File. The original grid file [15] was designed to be a hierarchical grid. At the top level, the full grid matrix is divided into k-dimensional hyper-rectangular subregions. That is, the full matrix is converted into a more coarse matrix representation. This super matrix (also know as the *region 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 valid values. A super-bucket can be defined in terms of its near and far vertices and contains a list of the non-empty bucket IDs that exist within the region defined by the super-bucket. (Note that although the original grid file was item-based, it could easily be converted to be bucket-based (where a *bucket* is defined as above, rather than restricted by page size), since buckets and items are essentially indistinguishable during search). The maximum number of IDs within a super-bucket is constrained by the disk page size. Thus, when the grid file super matrix is being 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.

Search is accomplished by using the region directory to first check for overlapping regions between the specified query and the super-buckets. If a super-bucket overlaps the query region, then it is fetched and its list of non-empty buckets is sequentially searched. This sequential search is inexpensive given that the whole list is guaranteed to be in memory and hence no further page faults will be required. 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 super-matrix, and one to access the data bucket). In addition to this principle, however, the grid file was also designed to improve the efficiency of range queries with respect to all attributes. This is accomplished by providing two levels of clustering—one for clustering the data items into buckets and another for clustering the buckets into super-buckets.

In our experimental studies on multidimensional range queries, we are using an optimized version of the grid file proposed by Hinterberger et al. [14]. In the optimized grid file, the region directory is converted from a matrix representation into a list of super-buckets. In this way, the optimized grid file could actually 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 have two primary effects: 1) the size of the region directory is reduced, and 2) unnecessary disk accesses due to inflated regions are eliminated, thereby reducing overall search time.

An advantage of the grid file is that it provides a sparse matrix representation that is not subject to the projection effects of the above array-based methods. The drawback of the grid file is that it can become increasingly difficult to optimize the super-bucket convex regions as the number of dimensions increases.

k-d Tree. The k-d tree is a multidimensional binary search tree [5]. It is a divide-and-conquer approach where each level of the tree divides the data set (i.e., set of non-empty buckets) based on the median value for a particular key. The tree cycles through each dimension with increasing level number (i.e., level 0 splits the tree on dimension 1, level 2 splits the tree on dimension 2, etc.). While the original k-d tree stored data in the internal as well as external nodes, optimized k-d trees store data only at the leaves of the tree. In addition, the k-d tree can be further optimized by using "adaptive partitioning," whereby discriminator keys for each level do not have to be chosen cyclically, but can rather be selected based on the dimension with the widest range of valid values in the sub-partition (similar to how the grid file is partitioned). This optimization can be accomplished by saving the discriminator ID with each internal node.

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.

4.4 Discussion of Index Structures

Table 1 summarizes the estimated storage and search costs for the various structures used in our experimental evaluation. Since $B \leq N$, the bucket-based indexes should have smaller storage overhead than the item-based linked array and k-Array. The search costs are more difficult to compare and contrast and we expect them to change under different conditions. Thus, our experimental evaluation is designed to test various situations such as variations in data set size, number of items found, and projection effects.

Table 1. Estimated Storage and Search Costs for Processing Multidimensional Range Queries for Various Index Structures (N=total # of records, B=# of non-empty buckets, k=# of dimensions, F=number of records found)

Index Structure	**Storage Cost S(N, B, k)**	**Search Cost Q(B, k)**
Linked Array	O(Nk)	O(Nk/slotCount)
k-Array	O(Nk)	O(Nk/slotCount)
Linked Bucket	O(Bk)	O(Bk/slotCount)
k-Bucket	O(Bk)	O(Bk/slotCount)
Grid File	O(Bk)	$O(B^{1-1/k} + F)$
k-d Tree	O(Bk)	$O(B^{1-1/k} + F)$

5. EXPERIMENTAL DESIGN

Assumptions. We assume that once A and B subsets have been selected, the TPairs database is formed and remains frozen. This allows us to optimize temporal browsing over the temporal events in the TPairs database. Should the cost for creating the TPairs database require too large an overhead, we can address this problem by providing support for off-line processing and storage of additional TPairs databases based on other A/B subsets.

Since a given TPairs database remains frozen, the indexing structure is static and can be constructed based on fore-knowledge of the data distribution to further optimize the incremental query processing. Since the cost to build the index is only done once, we do not consider the preprocessing index creation cost in our analysis.

Simulation Environment. While we have experimented with real-world video in our case study [11], our simulations were based on analyzing a hypothetical 30-minute video in order to be able to vary the distribution of A and B events. 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 the A and B events and the extreme values of the temporal query filters, which ranged from -10 seconds to 10 seconds for the startA-startB and endA-endB filters. In the results below, we tested data sets of size 500 A events by 500 B events, 1000 by 500, 1000 by 1000, 1000 by 1500 and 1000 by 2000. These had corresponding TPairs data set sizes of approximately 6000, 12000, 23600, 36000, and 47000. The estimates of subset sizes for a 30-minute video were based on real video data collected by CSCW researchers [20] and used in our MMVIS case study [11].

We compare our k-Array and k-Bucket index structures to the Linked Array, its Linked Bucket counterpart, an optimized Grid File [14], and a k-d Tree. All index structures are implemented in C++ and have been run on top of both UNIX and MacOS operating systems. In this paper, we focus our results on the use of 2K pages and a buffer of 100 pages. Although this is a relatively small buffer size by today's technology standards, we do not expect the general trends and shapes of our evaluation graphs to significantly change when we increase the data set size (e.g., to process longer or more than one video) as well as the buffer size. Thus, we assume the results below can be used as a predictive model for processing larger detests within an increased memory buffer.

Query Descriptions. In the next section, we present results from running simulations for ten different incremental TVQL queries. These queries were designed to explore the advantages and limitations of the various index structures as well as to mimic real world queries (i.e., to imitate how a real user might temporally browse the data or pose specific temporal queries—see [11] for more details). A summary of the nine queries is presented in Table 2. We assume that 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 2).

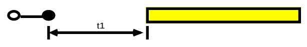
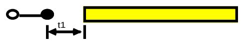
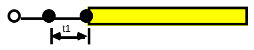
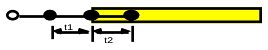
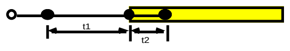
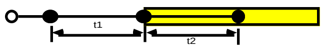
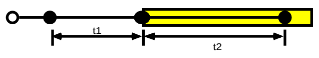
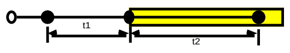
Query Rationale. Queries Q1 and Q2 were designed to test the processing of single increment user manipulations when a large portion of the database is currently selected. These types of queries are most likely the most expensive to process. Queries Q5 and Q6 are one-increment queries that were designed to examine the effect of processing queries when only a small portion of the database is currently selected—a situation similar though not the same as the best case scenario. The best case scenario essentially represents 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, since it would have just been previously processed. Interestingly enough, we see similar best case scenario effects in Query Q7.

Queries Q3, Q4, and Q8 were included as intermediary queries, as in a continuous sequence of inquiry, and help illustrate how users might use TVQL to pose real queries as described in the sample scenario below. Queries Q9 and Q10 were designed to test the k-bucket approach under projection effects as described below. The reader should note that while 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 also used in our experiments to test incremental query specification *as* the user adjusts any filter thumb.

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 are evaluating video data of a group of students discussing a science project. Subset A represents places in the video where Annie is speaking and Subset B represents places in the video where 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, and successfully do so by Q3. Once they specify this query, they want to refine it, to only view the situations where Annie speaks up to two seconds before Billy, and 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 may decide to see if there are also situations where Billy interrupts Annie (i.e., when Annie has not finished speaking when Billy starts speaking, as in Q6). In Q7 and Q8, the researchers widen the range of values for which Annie continues speaking even though Billy does not stop his interruption, and finally in Q9, they can examine the extent to which Annie may attempt to persevere through Billy's interruptions.

Note how together, the series of queries will 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).

Table 2. Summary of the nine TVQL queries used in our simulations.

Q#	User manipulation	Temporal Semantics	Temporal Diagram & Quantitative Info	Soln. Size (%N)
Q1	right-click the right thumb of dTfo filter to exclude end of range, dTfo.max	dTfo < dTfo.max	[all temporal relationships remain selected]	56%
Q2	drag right thumb of dTfo filter to the left by one ticVal	dTfo < dTfo.max - dtIncr	[all temporal relationships remain selected]	51%
Q3	drag right thumb of dTfo filter to 0	dTfo < 0	 0 < t1 ≤ |dTfo.min|	23%
Q4	drag left thumb of dTfo filter to -2	-2 ≤ dTfo < 0	 0 < t1 ≤ 2	5%
Q5	right-click the right thumb of dTfo filter to include 0	-2 ≤ dTfo ≤ 0	 0 ≤ t1 ≤ 2	7%
Q6	drag right thumb of dTfo filter to 2	-2 ≤ dTfo ≤ 2	 0 ≤ t1 ≤ 2, 0 ≤ t2 ≤ 2	11%
Q7	drag left thumb of dTfo filter to -6	-6 ≤ dTfo ≤ 2	 0 ≤ t1 ≤ 6, 0 ≤ t2 ≤ 2	21%
Q8	drag right thumb of dTfo filter to 6	-6 ≤ dTfo ≤ 6	 0 ≤ t1 ≤ 6, 0 ≤ t2 ≤ 6	27%
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%
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%

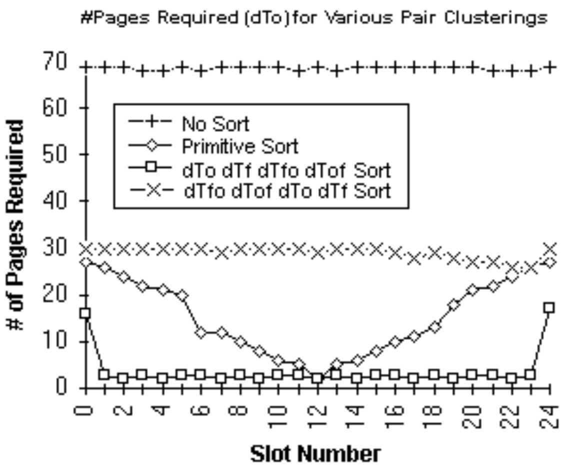
6. EXPERIMENTAL RESULTS

6.1 Clustering the Data

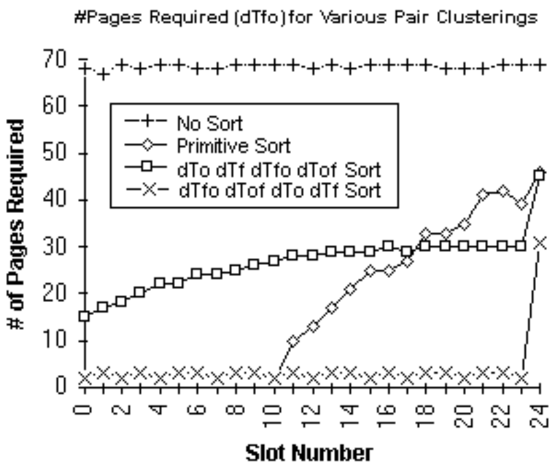
Disk-based query processing is highly sensitive to the size of objects and the order (clustering) in which the objects are stored on disk. Reducing the amount of information to be maintained and improving object clustering reduces the number of page faults required during query processing. A unique opportunity in the case of TVQL and other DQs that we propose to take advantage of is the fact that the multidimensional range queries are specified *incrementally*. In this section, we present results from exploring different clustering strategies for the purpose of processing TVQL queries.

Consider the case of processing a two-dimensional DQ query using a two-dimensional matrix. Each item in the data set can be sorted into one tile (i.e., normalized bucket) of the matrix. Specifying a query is equivalent to selecting a row, column, or rectangular region of the matrix. As described above, for every bucket in the selected region, every item in the bucket is in the solution. Thus, we know that we should cluster data items by bucket. However, what we don't know is how the buckets should be clustered on disk (e.g., whether they should be ordered by one attribute key, by a median, etc.). As users adjust the DQ filters, their manipulations correspond to adding/removing all or part of a row or column. If the buckets were clustered by row, page faults would be reduced in one direction, but not the other.

In the case of our k-Array method, data items are not sorted into buckets created by the k keys, but are instead sorted by their IDs into slots of each of the k arrays. The change in an item's count determines whether or not it is in one of the buckets being added or removed. While we can sort the actual data items using a nested loop over the k keys, this would result in clustering the data into buckets but would probably skew the optimization in favor of the first key (i.e., this still results in the problem of not necessarily optimizing the order in which buckets are stored). In the case of TVQL, one important observation is that adjusting a temporal query filter results in moving within a temporal primitive, or adding/deleting the neighbor of a primitive. Thus, we propose to sort the TPairs database first by temporal primitive and then within the primitives by the unbounded keys and/or the keys which determine constraints on the other keys (e.g., first by endA-startB for the *before* temporal primitive).



a) Comparison for dTo (startA-startB) query filter.



b) Comparison for dTfo (endA-startB) query filter.

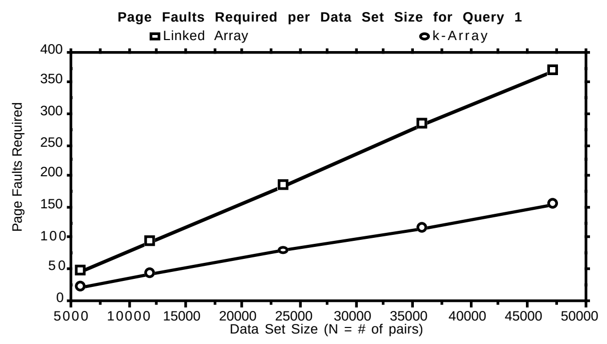
Figure 6. Number of pages required for various clusterings.

Figure 6 compares three sort-based clustering methods—our single sort by temporal primitive (called *tPrimSort*) and two nested sorts—with no sorting. The figure includes the number of pages that would be required if we had to fetch every item in a given slot of a particular query filter (the dTo (startA-startB) query filter in Figure 6a and the dTfo (endA-startB) query filter in Figure 6b). The figures are based on a data set size of over 10,000 pairs and a 2K page size. Examining these graphs indicates that: 1) any of the sorting strategies is better than no sorting at all, 2) using a nested sort does skew the distribution to be the most optimal for one dimension (e.g., the dTo-dTf-dTfo-dTof sort potentially requires the fewest number of page faults for dTo query filter manipulations), but not all dimensions, and 3) on average, the primitive sort requires the fewest number of page faults overall. Based on these results, we have adopted the *tPrimSort* sorting strategy for clustering the TPairs data and analyzing our array-based methods, as well as for improving performance for all bucket-based methods tested.

6.2 Evaluating the Array-Based Methods

6.2.1 The Linked Array vs. the k-Array

Our k-Array was designed to be an improvement over the linked array. In Figure 7, our simulation results comparing these methods for processing two sample TVQL queries indicate that the performance of the linked array degrades considerably for larger data sets, while the performance of the k-Array is fairly stable (i.e., it degrades gracefully). This is true for both the worst-case (Query 1) and better-case (Query 6) scenarios, but is seen more dramatically in the worst case scenario (note the difference in scale between the two graphs).



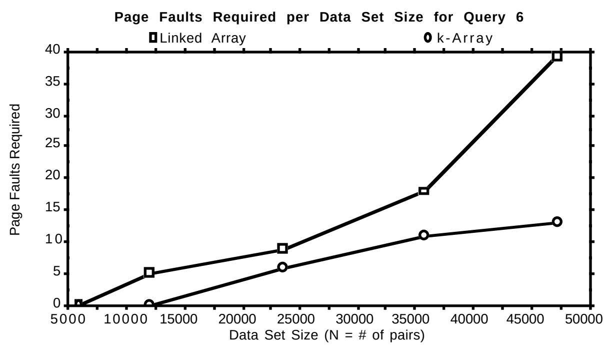


Figure 7. Number of page faults required for processing two sample TVQL queries for various data set sizes: comparison between linked array and k-Array methods. Query 1 represents a worst-case scenario while Query 6 represents a better-case scenario.

6.2.2 Optimizing Array-Based Techniques by Converting them to Bucket-Based Techniques

Our next set of simulations were designed to explore the impact of converting item-based methods to bucket-based ones. Figure 8 compares the number of page faults required for using the linked array and k-Array item-based methods and their bucket-based counterparts—the linked bucket and k-Bucket methods. Again, the results are provided for processing worst- and better-case sample queries (Q1 and Q6) over a range of data set sizes. The graphs indicate that the bucket-based methods do perform better than the item-based methods and that in particular, the linked bucket provides a significant improvement over its item-based counterpart, the linked array. The k-Bucket requires fewer page faults than the k-Array, but the increase in efficiency is not as dramatic as the conversion of the linked array to the linked bucket. The linked bucket now presents a competitive alternative to both the k-Array and the k-Bucket methods and we thus continue to evaluate its performance in the next section.

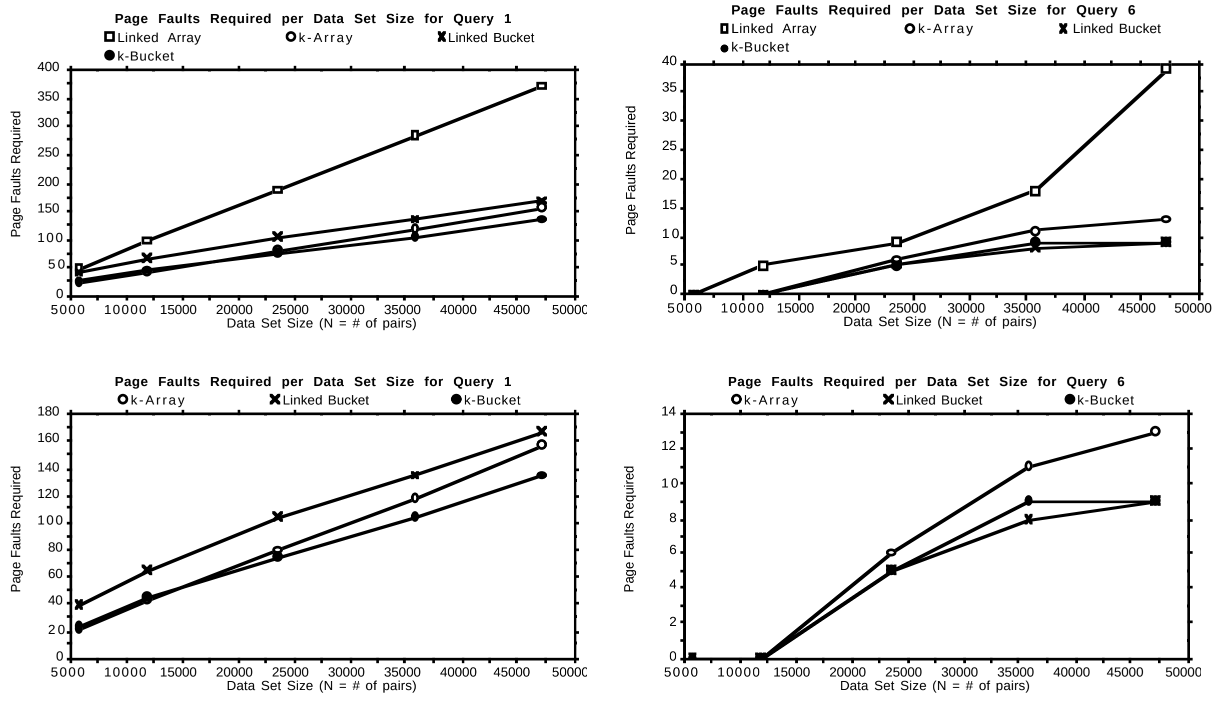


Figure 8. Number of page faults required for processing two sample TVQL queries for various data set sizes: comparison between linked array, k-array, linked bucket and k-bucket methods. Q1 is a worst-case scenario query while Q6 represents a better-case scenario. The top two graphs show all four indexing methods while the bottom two provide additional detail by removing the linked array plot and zooming in on the remaining methods.

6.3 Comparing Bucket-Based Methods: k-Bucket vs. Linked Bucket vs. Grid File vs. k-d Tree

6.3.1 Comparison over Data and Solution Set Size

Figure 9 presents the results for processing all ten of the TVQL queries summarized in Table 2. Note that the results for query Q7 are not included since Q7 never requires any page faults for these methods. Although this section is designed to compare the bucket-based methods, the k-Array is also included to indicate that it too performs fairly competitively, even when compared to other bucket-based methods. Note also 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.

If we let F be the number of items found and N the number of items in the TPairs data set, then we can use Figure 9 along with Table 2 to 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, Q4, and Q8). Most notably, the k-Bucket performs better than other approaches in the worst-case query scenarios as seen in the graphs for Q1 and Q2. That is, the k-bucket outperforms the other methods for large F and large N.
- the *k-d tree* does fairly well for the four remaining queries depicted in Figure 9. These queries are characterized by a smaller F. The k-d tree performs poorly in queries Q1, Q3 and Q8. These queries are characterized by a larger F (note the difference between preceding and corresponding values of the solution size in Table 2). These results confirm that the k-d Tree is particularly sensitive to F.
- the *grid file* also performs poorly for large N and large F (see Q1 and Q2). However, it does perform best or as one of the better methods in queries Q6, Q3 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 is a much less predictable method.
- the *linked bucket* is characterized in many graphs with a sharp upward trend as the data set size increases (e.g., Q2, Q3, Q5, Q8, Q9). It performs more competitively when F is fairly small (e.g., Q6 and Q10). Overall, the linked bucket is better suited for smaller data sets and smaller F. This result confirms the corresponding main memory evaluation of the linked array [15].
- the *k-array* performs surprisingly well for an item-based method, being the worst performer in only three of the nine queries (Q4, Q6, and Q10). It performs most obviously poorest in Q10, under a projection effect which is described in more detail below. The advantage of the k-Array is that it has a relatively small setup time, since once items have been clustered, they can be sorted into the k-arrays in one pass through the data.

6.3.2 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 counts to determine query results. As mentioned above, we would expect these types of methods to be particularly sensitive to L- or T-shaped projection effects. Queries Q9 and Q10 were designed to examine such effects. The results in Figure 9 confirm that the array-based methods are indeed sensitive to projection effects—the linked array is more highly affected in Q9, while the k-Array is more obviously a poorer performer in Q10. Note that while the k-Bucket does not perform the best in the projection effect situation, it performs much closer to the other methods in this situation than they do to the k-Bucket outside of this situation. This is seen when comparing and contrasting the scales for the different types of graphs. In Q10, the k-Bucket only requires a handful (i.e., less than five) of additional page faults while in Q1, k-Bucket requires a minimum of twenty page faults less than the other bucket-based methods.

In Q1, the query 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 2), 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 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-Array and k-Bucket methods actually perform better than the other bucket methods (the array-based linked bucket does not perform better since it is more sensitive to large F and large N).

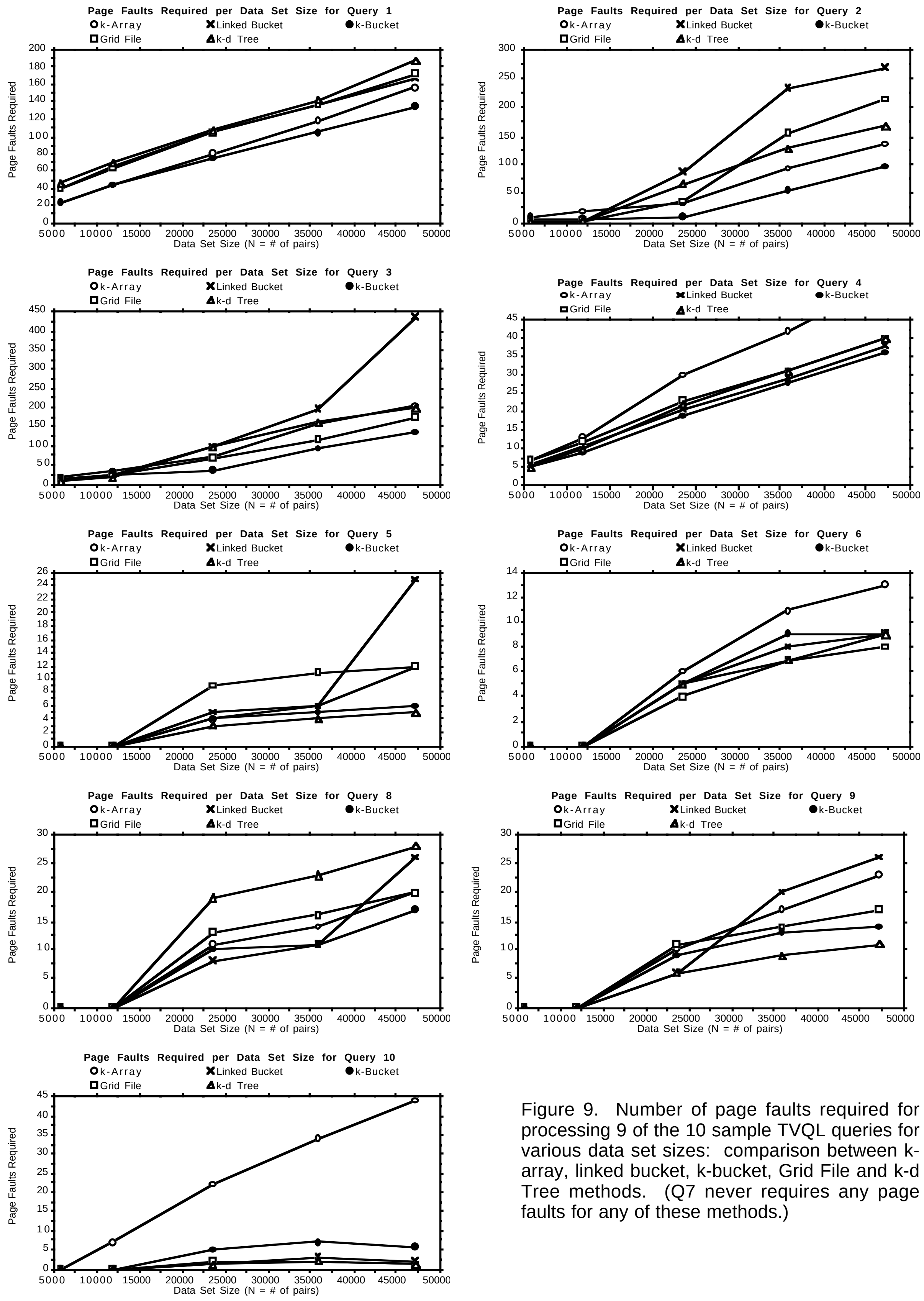


Figure 9. Number of page faults required for processing 9 of the 10 sample TVQL queries for various data set sizes: comparison between k-array, linked bucket, k-bucket, Grid File and k-d Tree methods. (Q7 never requires any page faults for any of these methods.)

6.3.3 Incremental vs. Non-incremental Queries

Figure 10 compares the performance of the various index methods for specifying incremental vs. non-incremental queries for a data set of N ≈ 12,000. 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.

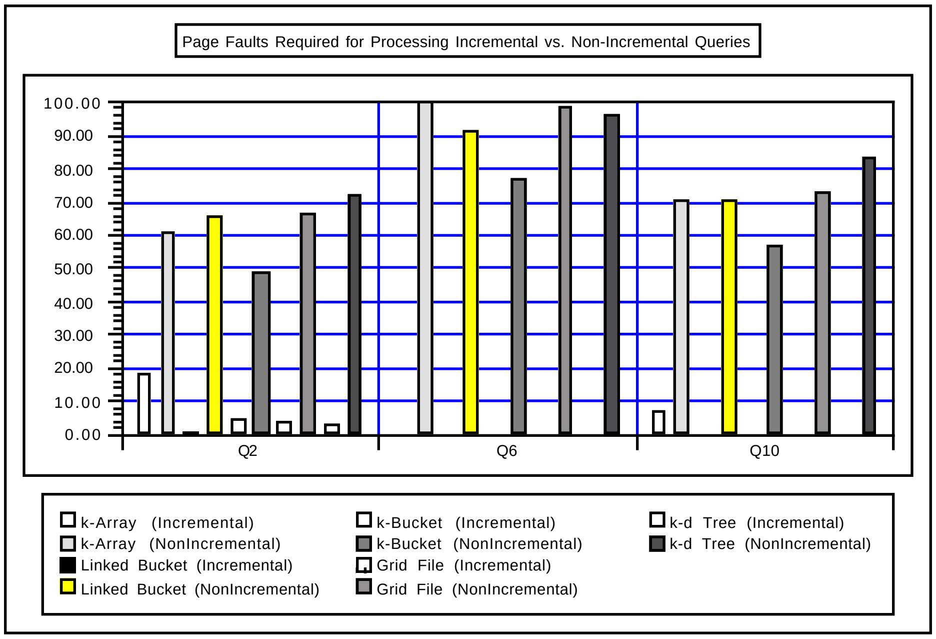


Figure 10. Number of page faults required for processing a sample of incremental queries vs. processing the same queries from scratch (results from queries Q2, Q6 and Q10): comparison between k-Array, Linked Bucket, k-Bucket, Grid File and k-d Tree. (N ≈ 12,000.)

Even in the case of a relatively small data set, all methods perform better for incremental queries. In particular, in the data set shown in Figure 10, most of the indexing methods require *zero* page faults for queries Q6 and Q10, since all information required for incrementally processing the query is still in the buffer. This is not the case for the non-incremental queries. We expect the comparison between incremental and non-incremental queries for larger sets to be similar in shape. That is, given that the number of page faults required for incremental queries increases with data set size, we expect the number of page faults for non-incremental queries to also increase with data set size.

7. DISCUSSION AND RELATED WORK

Research in temporal queries has typically focused more on historical (discrete) databases rather than on databases of temporal media (e.g., [23]). They focus on *discrete* changes entered into the databases as an effect of changes made in the real world (e.g., John broke his leg on April 10, 1994) rather than changes in continuous, dynamic media such as video. 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 using our tPrimSort method 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 [16]. 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., [17]). 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 two criteria of optimizing processing over updates and processing queries incrementally are critical for preserving the browsing paradigm in our MMVIS environment. 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 newly proposed k-Array and k-Bucket indexing techniques to one base-case index (the Linked Array [18] and its Linked Bucket counterpart) as well as to the 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, 21] 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) we decided to evaluate the simpler k-d tree and grid file structures before evaluating more complex ones such as the r-tree.

8. CONCLUSION AND FUTURE WORK

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. In our evaluation, also experimented with various clustering options and identified one that is particularly suitable for storing temporal data pairs. This strategy, called tPrimSort, is primarily based on sorting buckets of the pairs by temporal primitives.

We 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, and in fact dramatically so for the case of the linked array to linked bucket conversion. We then compared the bucket methods for a series of TVQL incremental queries. We compared and contrasted the linked bucket, the k-Bucket, the grid file and the k-d tree. Our results showed that the k-Bucket performed the best overall, but that it was indeed subject to projection effects. In addition, the other methods also performed fairly well under various conditions. In particular, the k-d tree is sensitive to F, the number of items found, but performs fairly well when F is smaller and the k-d tree is not subject to projection effects. The linked bucket performs well for small data set sizes, but performance degrades fairly rapidly with increases in data set size. While the grid file performed fairly well in half of the queries, its performance behavior was not as predictable as for the other methods.

We identified the potential for L-shaped skewed data distributions for temporal pair data sets and analyzed the structures based on this skewed distribution. In particular, we examined the projection effects for the array-based methods and were able to determine that performance of these methods (i.e., the k-Array, linked bucket and k-Bucket) does decrease in the projection effect situation. However, 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 showed how the index structures all performed better for processing queries incrementally rather than recalculating the full solution from scratch, and that the k-Bucket performed better overall for both types of queries.

In the future, we plan to enhance our analysis of TVQL query processing by exploring other multidimensional range query indexing techniques as well as including a comparison of the various methods for specifying disjunctive multidimensional range queries.

ACKNOWLEDGMENTS. This work was supported in part by UM Rackham Fellowship, NSF NYI #94-57609, and equipment support from AT&T. The k-d tree was originally implemented by Monal Sonecha and Jialei Jin.

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, pp. 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. and 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. MM96 Hibino, S. and Rundensteiner, E. (in press). “MMVIS: Design and Implementation of a Multimedia Visual Information Seeking Environment.” To appear in *ACM Multimedia’96 Conference Proceedings.*
11. IMMIR Hibino, S. and Rundensteiner, E. (in press). “Interactive Visualizations for Temporal Analysis: Application to CSCW Multimedia Data.” To appear in *Intelligent Multimedia Information Retrieval* (Mark Maybury, Ed.).
12. Hibino, S. and Rundensteiner, E. (1996). “MMVIS: A MultiMedia Visual Information Seeking Environment for Video Analysis,” *CHI’96 Conference Companion*. ACM Press, 15-16.
13. Hibino, S., and 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, and P.B. 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. *Proc. of the Workshop on Advanced Visual Interfaces*. NY: ACM, 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 Transactions on Knowledge and Data Engineering*, 5(4), 682- 694.
18. Knuth, D.E. (1973). *The Art of Computer Programming, Vol 3: Sorting and Searching*, Addison-Wesley.
19. Nievergelt, J., Hinterberger H., and Sevcik, K.C. (1984). The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. on Database Systems*, 9(1), 38-71.
20. Olson, J., Olson, G., and Meader, D. (1995). What mix of audio and video is important for remote work. *CHI’95 Conf. Proc*. NY: ACM, 362-368.
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.