To Appear in the *ACM Multimedia'96 Conference Proceedings* (November 1996).

MMVIS: Design and Implementation of a MultiMedia Visual Information Seeking 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 Rd.
Worcester, MA 01609-2280 USA
rundenst@cs.wpi.edu

ABSTRACT. In our new paradigm for video analysis, we advocate the use of interactive visualizations where users can *browse* video data in search of temporal trends by specifying temporal queries via direct manipulation. In this paper, we describe the design and implementation of our MultiMedia Visual Information Seeking (MMVIS) system that successfully realizes this exploratory approach to temporal analysis. We present our design goals and decisions, including the design specifications of our subset selection query interface, our direct manipulation temporal visual query language (TVQL), and our temporal visualization (TViz) of results. We also present our strategies for implementing MMVIS—focusing in particular on our overall system architecture and the TVQL query processor. Finally, we briefly review a case study using real CSCW data and preliminary results of a user study, used to validate the utility of TVQL and MMVIS.

KEYWORDS
Interactive visualizations, video analysis, temporal visual query language, multimedia system design.

1. INTRODUCTION

Previous approaches to video analysis have emphasized video annotation and coding over sophisticated analysis techniques (e.g., [6, 10, 21]). That is, by providing tools to simplify the process of creating annotations and more importantly for consistently coding relationships between objects or events in the data, these researchers have reduced the analysis process to specifying selection-type queries over the pre-coded data. For example, in a video of a design meeting setting, researchers could use previous video analysis systems to annotate all occurrences when "P1 is digressing." Then, to find out how often P1 is digressing, they can pose the query, "Select annotations where person=P1 ^ action=digressing" and then count the number of results they retrieve. The problem with such an approach is that 1) it requires users to code relationships *a priori*, limiting analysis to the pre-coded relationships and 2) it can be difficult to abstract related temporal information (e.g., it could be cumbersome for users to find out how often P1 *starts* or *ends* a digression).

In contrast to these approaches, we propose a new paradigm for video analysis involving the use of interactive visualizations in which users *browse* the data in search of temporal trends. Rather than requiring users to pre-code relationships, we have simplified the annotation process to that of coding atomic actions and events. Our system then *enhances the analysis process* by allowing users to explore temporal relationships between subsets of such annotations using simple mouse manipulations. For this, we have developed a temporal visual query language (TVQL [13]—see review in Section 2.3) composed of integrated temporal query filters that support the specification of complex temporal queries using a direct manipulation paradigm. TVQL facilitates temporal exploration of the video data by allowing users to continuously specify and incrementally refine individual as well as combinations of similar temporal relationships. Thus, in our approach, users would code "P1 Talking" and "Digression" as separate annotations. Using our dynamic temporal query filters, they could then examine all types of temporal relationships between P1 talking and digressions, including how often P1 starts, participates in, or ends a digression.

In our MMVIS system, the results of TVQL queries are presented in a temporal visualization, called TViz (Section 2.4). The visualization presents the temporal relationships between subsets by abstracting relationships between individual event instances into trends over a given time interval. Some of the abstraction strategies we have explored include frequency counts, total duration, and average duration. The visualization is user-customizable on-the-fly, allowing users to highlight subsets according to their current focus of temporal analysis. In addition, our temporal visualization is *dynamically* updated as temporal filters are adjusted. This provides immediate feedback on temporal trends as a function of the type of temporal relationship (e.g., "does Joe always interrupt Mary the moment she starts talking" versus "does Joe interrupt Mary only after she has been talking continuously for at least five minutes?") or as a function of the type of selected event subsets (e.g., "does Joe interrupt only the female members of the meeting who are speaking for more than five minutes and not the male members?").

†This work was completed while author was at the University of Michigan.

The combination of specialized query filters and visualizations forms our integrated MultiMedia Visual Information Seeking (MMVIS) environment. In this paper, we describe the design and implementation of our MMVIS system. We present the system design of primary interface components of MMVIS—namely, our visual query language including the subset query and TVQL query palettes and our temporal visualization, TViz. We also present our strategies for implementing MMVIS, focusing in particular on our overall system architecture and on the TVQL query processor. In order to preserve the notion of interactive browsing for trend analysis, the visual queries must be processed as efficiently as possible. Thus, the query processor is a critical component of the system and hence we discuss it in more depth in this paper.

We describe a new index structure (the k-Array method) and an associated query processing strategy we have developed for processing TVQL queries. Our preliminary experimental evaluation of the query processor reported in this paper demonstrates the effectiveness of our index strategy for handling multidimensional range queries specified in an *incremental* fashion—the main mode of interaction with MMVIS. Lastly, a brief report on our case study using real CSCW data and preliminary results from a user study illustrate the utility of TVQL and MMVIS.

This paper is divided into five additional sections. In Section 2, we present our system design—describing subset query palettes for dynamic subset selection, reviewing our temporal visual query language (TVQL), and presenting our temporal visualization (TViz). In Section 3, we describe our system implementation, beginning with the system architecture, presenting our annotation model, and discussing the underlying query processing technique. This is followed by evaluation and discussion of the efficiency and utility of our approach in Section 4. In Section 5, we discuss related work, and finally in Section 6, we present our conclusions.

2. SYSTEM DESIGN

2.1 System Requirements: A New Visual Paradigm for Video Analysis

The goal of MMVIS is to provide a new paradigm for video analysis—one in which users can temporally and continuously *explore* data in search of *temporal relationships and trends*. In this new paradigm, users should be able to use the following process:

1. Select two subsets of the video data.
2. Query for temporal relationships between subsets via specialized temporal query filters.
3. Review a visualization of results for temporal trends.
4. Customize visualization for further clarification, if desired, and go to 3.
5. Go to 2 to incrementally adjust relative temporal query or go to 1 to select new subsets.

In designing MMVIS, we chose to realize this new paradigm for video analysis by extending existing work in Visual Information Seeking (VIS) [2]. In VIS, users can browse a database of information through direct manipulation of buttons and sliders. These buttons and sliders represent query filters with which users can select a range of desired values for each attribute of the database. For example, in a real estate application [2], users may wish to select all houses with two to four bedrooms that cost between \$100K and \$300K. If they had a slider for the number of bedrooms and another slider for the cost, then they could use simple mouse moves to adjust these sliders to specify the desired ranges for each attribute. The sliders act as query filters to the data and are *dynamic* in that a visualization of results is dynamically updated *as* users adjust each filter. In addition, the filters are dynamically linked to one another to prevent invalid queries. For example, if users only wanted to spend \$250K and the largest \$250K house only had 3BR, then when users adjust the cost query filter to reduce the cost, the number of bedrooms filter would automatically be updated to reduce the largest number of bedrooms to 3BR. This indicates that all four bedroom houses cost more than \$250K.

This use of dynamic query (DQ) filters provides us with an easy-to-use *visual* paradigm for formulating and posing questions. Because a visualization of results is dynamically updated as users adjust any query filter, queries are *incrementally* specified and refined and users see the direct correlation between adjusting values of query parameters and the corresponding display of results.

Another system requirement for MMVIS is that it should be general enough to handle a variety of multimedia data typically characterized by *dynamic, spatio-temporal characteristics* (e.g., video, real-time data, etc.). We accomplish this by using annotations to abstract spatio-temporal information from the original media and then analyzing the annotation collection. In the context of this paper, we explain this approach using video data as an example. Section 3.2 describes our data model for these video annotations.

Although the VIS paradigm is suitable for our goals, the current VIS framework is not designed for selecting multiple subsets nor for handling spatio-temporal characteristics of multimedia data. Thus, in order for MMVIS to support the desired new paradigm for video analysis, several extensions to VIS are required:

- *subset query palettes* with multi-selection list filters for specifying multiple subsets of different types of events (e.g., Subset A = “all person P1 talking events” and Subset B = “all person P2 talking events”),
- *specialized temporal query filters* (i.e., a temporal visual query language (TVQL [13])—Section 2.3) for exploring temporal relationships between the subsets formed, and
- user-customizable *spatio-temporal visualizations* (e.g., TViz—Section 2.4) for highlighting the occurrence of the selected subsets and frequency of specified relationships.

The design of each of these components is described in more detail in the remainder of this section.

2.2 Selecting Multiple Subsets

In MMVIS, each annotation (i.e., event in the database) is typically characterized by name, action, receiver, and category. Thus, in order for users to select a subset, they need to be able to select from these alphanumeric characteristics. Although alphasliders [1] could have been used to dynamically select singleton items from each attribute, this would have prohibited the selection of several items for any given attribute domain. The requirements for subset selection then are to support:

- selection of multiple items for any attribute domain,
- consistency in the dynamic query interface:
  - query filtering via direct manipulation,
  - dynamic updating of the display as users manipulate any query filter,
  - dynamic updating of other query filters to indicate interrelationships between attributes, and
- concurrent selection of multiple subsets.

In designing a solution to multiple subset selection, we chose to provide a separate subset query palette for each subset. Each subset palette contains lists of data attributes from which to select (see Figure 1). We developed a *multi-select list box dynamic query filter* (called an *mList DQ*) for dis-/continuous selection of multiple values from attributes with categorical (i.e., discontinuous) data. Selected items of a list are ORed, and the results from each list are then ANDed together.

In each mList DQ on the subset palette, users can simply click on an item to toggle its selection on and off. As in the standard DQ filters, these mList DQs are interrelated to one another so that the de-/selection of items in one mList can automatically affect the de-/selection of items in the other mLists. Thus, if an item is deselected in one mList, and an item in another mList is only related to that deselected item, then it too would be deselected. For example, in Figure 1, the name Nil is only associated with the actions NonVerbal and Transcript. Since Transcript has already been deselected, deselecting NonVerbal from the Action mList would automatically deselect Nil from the Name mList.

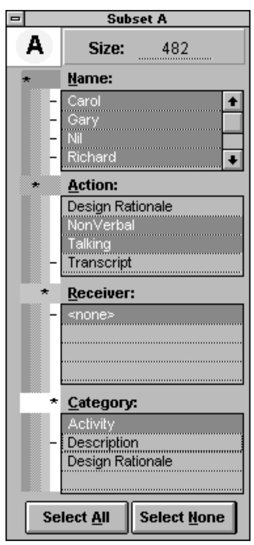

Figure 1. Sample subset query palette.

2.3 TVQL: Temporal Visual Query Language

2.3.1 Requirements for a Temporal Query Language

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 2).

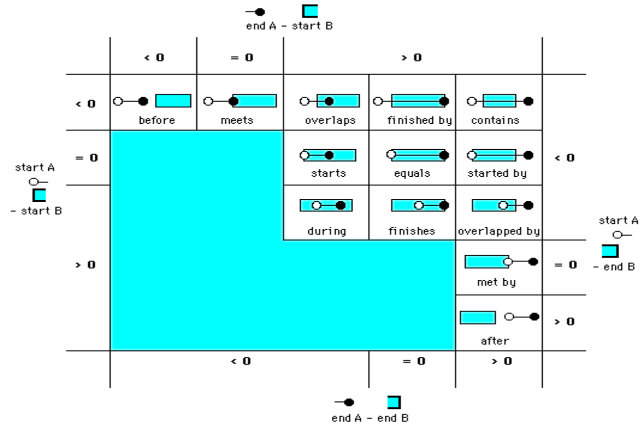

Figure 2. Relationships between temporal primitives and the four defining endpoint difference relations.

A general temporal query language must be able to specify any one of these primitives. However, it is also desirable to specify combinations of the primitives (e.g., to see how often events start at the same time but may end at different times, corresponding to combining the *starts*, *started by*, and *equals* primitives). Rather than providing arbitrary combinations of such relationships, we support users in selecting *similar* primitives (i.e., *temporal neighborhoods* [8], equivalent to selecting a series of adjacent cells such as a row, column, or grid from Figure 2).

A set or combination of temporal relationships between two events forms a *temporal neighborhood* if it consists of relations that are path-connectable conceptual neighbors. Two primitive temporal relationships between two events are *conceptual neighbors* if a continuous change to the events (e.g., shortening, lengthening, or moving the duration of the events) can be used to transform either relation to the other (without passing through an additional primitive temporal relationship) [8]. Thus, the *before* ( ) and *meets* ( ) relations *are* neighbors, because we can move the ending point of A from before the start of B to its start without specifying any additional primitive relationship. On the other hand, the *before* ( ) and *overlaps* ( ) relations are *not* neighbors. This is because we cannot move the ending point of A past the starting point of B without first passing through the *meets* relation.

2.3.2 TVQL Design

While a formal specification of our temporal visual query language (TVQL) can be found elsewhere [13], we review its basic design principles here. In order to define a temporal query interface capable of specifying any individual primitive temporal relationship, we designed TVQL to be a collection of four temporal query filters—one filter for each of the defining endpoint difference relationships described in Section 2.3.1 and depicted in Figure 2. More importantly, this design also allows us to capture *temporal neighborhoods* [8]. In this way, not only can users browse for temporal relationships between two subsets, but they can browse in a *temporally continuous* manner. More specifically, our TVQL interface allows users to explore within and between primitives as well as within and between temporal neighborhoods. Figure 3 presents our TVQL [13, 14]. As in the standard double-thumbed slider query filters [2], the thumbs are manipulated to select the endpoints of a range, and a filled or open arrow thumb indicates when the endpoint of a range is included or excluded respectively.

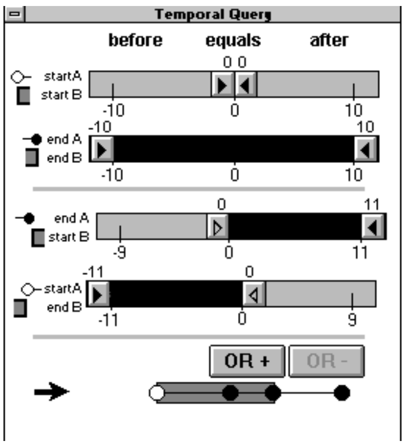

Figure 3. TVQL palette. This query specifies all events of type A that start at the same time as events of type B (and may end before, after, or at the same time as B events).

To enhance the TVQL user interface, we have incorporated qualitative descriptive labels along the top and side and our dynamic temporal diagrams along the bottom of the palette. 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 given by the filters). If subset A specified person P1 and subset B specified all Plan design rationales, then Figure 3 illustrates how users could ask the query "show me how often person P1 starts at the same time as a Plan starts." The descriptive labels can be used to "read" the top query filter as "start A equals start B." The relationship between the temporal ending points is unconstrained as indicated by the selection of all values in the second (i.e., endA-endB) query filter. This is also reflected in the temporal diagram, which indicates that the end of A (represented by a filled circle) can be before, equal to, or after the end of B. The benefit of this direct manipulation design is that it supports specifying *particular* queries as well as *browsing* for temporal trends. That is, users can simply drag DQ thumbs with no particular query in mind and when an interesting visualization appears, they can look at the temporal diagram to see which temporal query was specified.

Similar to standard DQ filters, our temporal DQ filters are bound to one another to prevent the specification of invalid queries. As users adjust one query filter, the other filters are automatically updated accordingly. In the case of Figure 3, the user only has to set the filter thumbs of the top startA-startB query filter to 0. The bottom two filters are automatically constrained as indicated.

2.4 TViz: Temporal Visualization of Results

Our first goals in designing the temporal visualization (TViz) of results for MMVIS were to maintain the VIS paradigm by 1) presenting the results in a visual format (in contrast to a text-based table of numbers) and 2) tightly coupling the visualization of results to the DQ filters. In this way, users can see the correlation between adjusting temporal parameters and the corresponding visualization of temporal relationships. In addition to these goals, we identified several other requirements for TViz:

- provide and/or preserve context as much as possible,
- highlight temporal occurrences of members of the event subsets (e.g., highlight relative frequency, average or total duration of the various types of events), and
- aggregate and highlight the strength of temporal relationships between subset members and specified by TVQL (i.e., in contrast to timelines, where information about temporal relationships is distributed).

These types of visualization extensions go beyond the original VIS [2], where the results were typically displayed as an enhanced scatterplot.

Our overall approach has been to provide a main window of representative annotation icons for context, and to use transparent overlays to highlight temporal occurrences (i.e., selected subsets) and relationships between them. In addition, we provide some options to allow users to tailor visualizations according to their needs and preferences.

2.4.1 The Main MMVIS Window

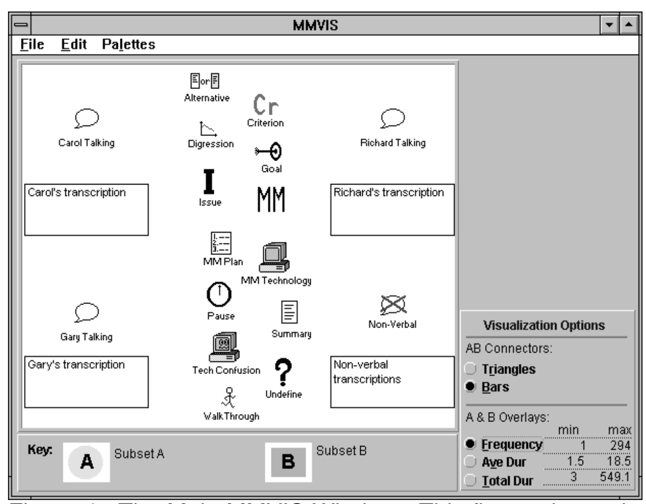

Figure 4. The Main MMVIS Window. This figure shows the main MMVIS window before any subsets have been selected or any temporal queries have been specified.

Figure 4 presents the main MMVIS window. This window is divided into three areas: the primary visualization area, the key below the visualization area, and visualization options in the lower right corner of the window. Initially, the *visualization area* only contains icons representing the different types of annotations in the database (in Figure 4, icons from the CSCW case study are displayed). By default, these annotations are spatially placed according to their corresponding first occurrence in the video and are used to provide context for TViz. Users can move icons according to their preference. In the case study data set [20], we found that subjects never changed positions (e.g., walk across the screen) in the video, so there was a one-to-one correspondence between where talking icons were placed and where the corresponding speakers appeared on the video. In this way, the annotation placement formed a spatial abstraction of the video context for the case study.

The *key* below the visualization emphasizes that subset A events will be highlighted with yellow transparent circle overlays while subset B events will be highlighted with blue transparent square overlays (see also Figure 6). The *visualization options* allow users to customize the view according to their needs and preferences. More details on the subset highlighters and visualization options are provided below.

2.4.2 Visualization of the Selected Event Subsets

The subset selection highlighters of TViz are designed to support users in comparing temporal occurrences of different types of events within the context provided by the main MMVIS window. We accomplish this by using transparent overlays to visually indicate the current subset as it is being de-/selected. Members of subset A are highlighted by yellow circles, while members of subset B are indicated by blue squares. The radius of these transparent overlays represents either relative frequency, average duration, or total duration, customized according to the user's preference.

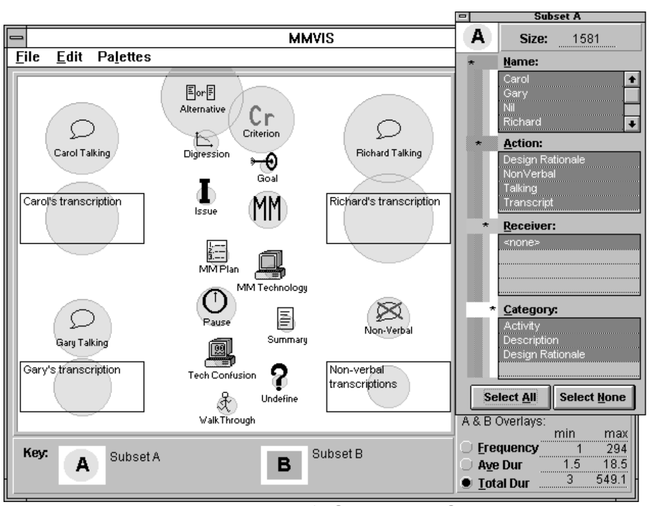

Figure 5. Visualization of Selected Subsets. In this example, Subset A is set to all types of annotations and Subset B is set to none. Transparent circle overlays highlight the types of A events selected. The relative size of the circular highlighters provides a comparison of *total duration*. Other viewing options include *frequency* and *average duration*.

Figure 5 presents an example where the user has set Subset A to all types of annotations and Subset B to none. By doing so, the user can then switch between visualization options to gain an overview of the data and to compare the differences between relative frequency, average duration, or total duration of all types of events. Figure 5 provides a comparison based on total duration. In it, we can see, for example, that the total time that Richard speaks is longer than that of Carol or Gary.

2.4.3 Visualization of the Temporal Relationships

Once users have specified A and B subsets, they can then use TVQL to specify a temporal query. As they manipulate the TVQL filters, users see connectors between the centers of A and B events appear and disappear, grow and shrink, thereby indicating the existence and strength of the temporal relationship currently specified. The base width of the connector denotes the relative frequency that the temporal relationship occurs. TViz is based on the visualizations used by Olson et al for temporal sequences [20].

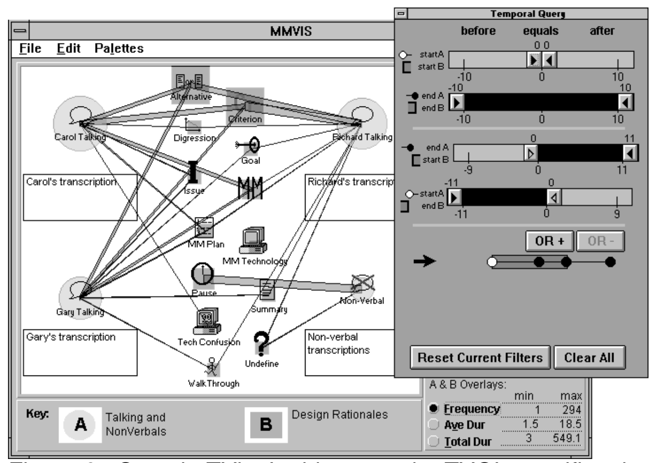

Figure 6. Sample TViz. In this example, TVQL specifies the *all starts* temporal query, where events start at the same time but may end at the same or different times. The AB connectors in TViz are displayed as *bars,* and the width of these bars indicates the relative frequency that the *all starts* relationship occurs between each type of AB pair.

Figure 6 shows an example where the user has specified the *all starts* temporal query, where events start at the same time but may end at the same or different times. In this example, the thick bar between NonVerbal and Pause indicates that these types of events frequently start at the same time. Similar to the overview visualizations, the temporal relationship connectors are also user-customizable, being viewable as triangles or bars. Figure 6 shows the view by *bars*.

3. IMPLEMENTATION

3.1 System Architecture

Figure 7 presents the MMVIS system architecture, illustrating how the design specifications described in Section 2 are integrated with the underlying system components, including the Database Manager. In MMVIS, users have access to basic Annotation Tools to code the original media (e.g., video) by either directly creating annotations within the system or importing annotation information from other, perhaps automated, systems (e.g., [18]). These annotations, which abstract atomic objects and events taking place in the media, are passed through an Annotation Processor on to the Database Manager and are stored in an underlying database. The Annotation Processor translates user specifications of annotation parameters into annotation objects to be stored in the database.

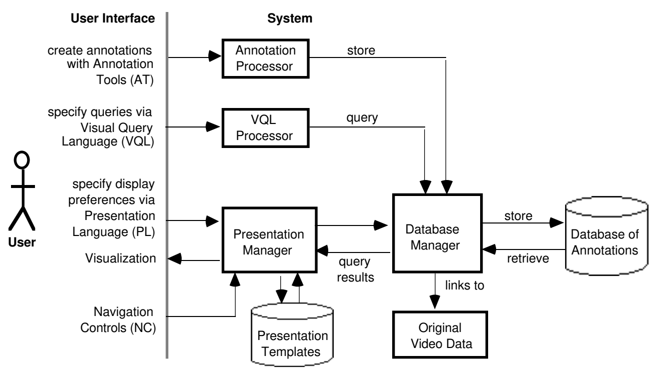

Figure 7. MMVIS System Architecture.

Once the annotation collection has been formed, users can then explore and analyze the video through iteratively specifying queries using a Visual Query Language (VQL; i.e., the subset selection and TVQL query palettes described in Sections 2.2 and 2.3) and by reviewing the visualization (e.g., TViz, Section 2.4) of results presented. In order to maintain the notion of browsing to the users, the visualizations must be updated in real-time and thus queries must be processed as quickly as possible. This is accomplished through the VQL Processor which is described in Section 3.3. The VQL Processor sends messages to the Database Manager which then passes updates of the solution set to the Presentation Manager. The Presentation Manager takes these updates of the query results, along with any user-defined display preferences and updates the visualization. Users can view the visualization as it changes and visually scan the final results to look for data trends. If no trends are found, they can use the Presentation Language (PL) to clarify the visualization, the Navigation Controls to further explore query results, or the VQL to incrementally adjust the query.

MMVIS has been implemented in a Windows-based multimedia PC environment, using a ToolBook interface to a database library. The subset selection query palettes, TVQL, VQL Processor, Database Manager, and a primitive Presentation Manager for displaying and updating TViz have all been incorporated into the current working system. The Annotation Tools are currently limited to importing existing databases of video events, but tools for creating video annotations directly within the existing framework are being developed. In addition, we are also investigating the use of alternative visualizations to extend the collection of Presentation Templates from which users can select. Details of the Database Manager and our VQL query processing strategy are presented below.

3.2 Database Manager

The Database Manager uses our annotation model to store and retrieve information from the annotation collection. Below, we highlight the basic characteristics and structure of our annotation model. Recall that annotations are used to abstract spatio-temporal and conceptual information about objects and events taking place in the video. Figure 8 summarizes our basic annotation data model.

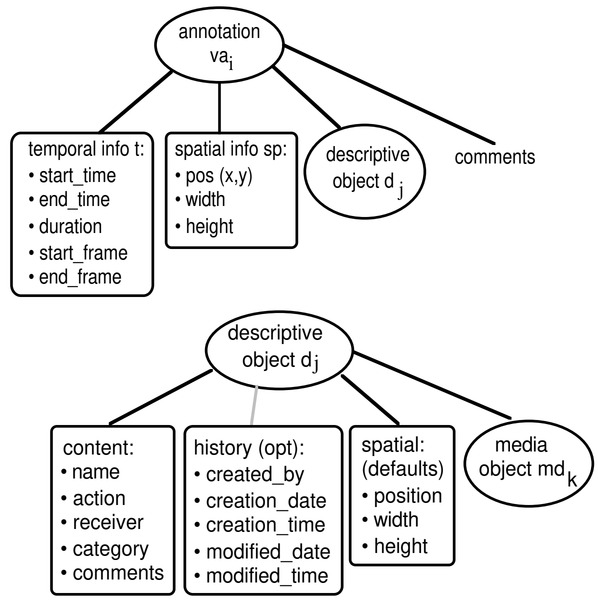

Figure 8. Overview of Annotation Data Model

The descriptive objects are separate from the annotations so that several annotations can reference the same descriptive object. This provides consistency and improves efficiency during annotation creation by allowing users to link directly to previously specified descriptive objects rather than having to specify the same information over and over again. The same is true for the relationship between the descriptive object and the media object.

The Database Manager stores information on basic annotation, descriptive, and media objects in a dBase IV format while the actual views of the objects are realized directly within the application. The t.start_frame and t.end_frame attributes of an annotation object allow us to obtain direct access to the corresponding video segment.

3.3 Query Processing in MMVIS

3.3.1 Basic DQ Characteristics

The problem of processing temporal dynamic queries 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 the following characteristics:

- each DQ filter represents an 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 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,
- 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 the real estate scenario, 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, each (a_i,b_j) temporal pair has one startA-startB value, one endA-endB value, etc. and can thus be sorted into one slot of each of the corresponding temporal query filters. A given temporal pair is then in the solution set when each of its temporal endpoint differences is in the selected range of each of the corresponding temporal query filters.

3.3.2 Terms and 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}
// evenly spaced slider 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}
// annotations for video v_p

Users select A and B subsets via subset selection DQs:

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

Based on the subset selected 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, ..., pr_q, ..., pr_N
| pr_q=(a_i, b_j), $a_i \in$ A, $b_j \in$ B,
(a_i.end − b_j.start ≥ dTfo.minVal
AND a_i.start − b_j.end ≤ 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)
`N` = # of (a_i, b_j) pairs in TPairs ($\le T^2$) from which pairs that meet temporal relationship criteria are selected
`k` = number of attributes or dimensions used for query specification (for TVQL, k=4)

Figure 9 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 9a, the DQ filter is selecting 0 ≤ values < 1. This corresponds to including slots 6 and 7 of the internal representation shown in Figure 9b.

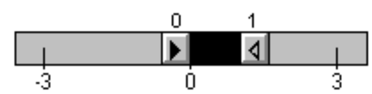

Figure 9.(a) Sample DQ filter (minVal = -3, maxVal = 3, dtIncr = 1).

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

Figure 9.(b) Corresponding internal representation where the double-lined border around slots 6 and 7 corresponds to the selected DQ filter range in Figure 9.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, 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` (i.e., startA-startB) query filter, then pr_q would go into slot 6 of the filter. Recall that pr_q is only in the solution set when each and every one of its attribute values are valid (i.e., when each is included in the selected range of its corresponding attribute DQ filter). Another way to think of this is that if `count` represents the number of *valid* attribute values for pr_q, then a temporal pair pr_q is in the solution set when `count=4` (or, in general, when `count=k`).

3.3.3 DQ Filter Manipulations

Since users *incrementally* specify and update queries by directly manipulating any DQ filter thumb, an important goal in identifying an indexing scheme would be to use one with efficient support for finding the "nearest neighbor" when processing these incremental user queries. Given a double-thumbed query filter such as those used for TVQL, there are two types of 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(dq_i, thumbID, action, <dir>)

where

dq_i // DQ filter being adjusted,

thumbID ∈ {LEFT, RIGHT}

// which DQ filter thumb was manipulated,

action ∈ {MOVE, FILL, UNFILL}

// type of manipulation made, and

dir ∈ {LEFT, RIGHT}

// optional parameter to specify direction the
// DQ thumb was moved, if applicable.

Query processing in TVQL must thus be able to effectively process such sets of manipulations. This includes identifying items affected by the manipulation, updating appropriate auxiliary structures such as counts, and updating the solution set by passing information on TPairs items that should be added/removed from the solution to the Presentation Manager via the Database Manager.

3.3.4 k-Array: An Array Based Method

One of the simplest methods of indexing multidimensional data is to use a k-dimensional matrix representation. While such a matrix is simple to build and easy to use, it is often disregarded due to its high storage costs, which for dynamic queries is on the order of O($\mathtt{slotCount}^k$). Another disadvantage of the full matrix-based approach is that in a data set with a skewed data distribution, many buckets (e.g., individual tiles in the two-dimensional matrix case) will be empty—while others will be very full. Note that the four dimensions of our TVQL language, namely the four temporal query filters, are highly interdependent—as illustrated in Figure 2. This indicates that any TPairs database may have a skewed distribution. Hence, the simple full matrix method is not as applicable a technique for TVQL query processing.

In order to reduce the storage required for indexing the data, to avoid empty buckets as much as possible, and to optimize performance, we thus set out to develop an alternative strategy for TVQL query processing. While several methods for processing multidimensional range queries exist (e.g., [4, 19]), we felt that the requirements and constraints of our TVQL query processing problem were different enough from previous work to warrant additional investigation. More specifically, our goal was to identify an indexing method for processing *incremental* multidimensional range queries over temporal data, giving higher priority to processing queries over updating data.

Our proposed *k-Array* approach is an array-based method generally applicable to DQ query processing—while of course for our TVQL language we set k=4. Given:

dBase = {$item_1$, $item_2$, . . . , $item_N$}

// with unique_IDs {1, 2, . . . , N}
// for TVQL, dBase = TPairs

DQ = {dq_1, dq_2, . . . , dq_k}

// set of DQ filters

kArray
= {indexArray(dq_1), indexArray(dq_2),
. . . , indexArray(dq_k)}

The basic idea is to keep an index array of size slotCount for each DQ filter. Each position in the index array has a one-to-one correspondence to the slotNum of the respective DQ filter. Thus, each item in the database can be sorted into one and only one position of each index array. This means that every item in the database occurs a total of k times in the full index structure.

If count is the number of selected ranges of the index arrays in which an item occurs, then an item should be added to the solution set when its count changes from k−1 to k. Similarly, the item should be removed from the solution when its count changes from k to k−1. We only need to access the actual item when it is being added to or removed from the solution. The rest of the time (i.e., while count ≤ k−1), it is sufficient to access only the count for each item. For this reason, the counts are stored in a separate array, consecutively by item ID:

itemCounts[N] = array of data item counts

Using pointers and a main memory model, the k-Array approach is captured by Figure 10.

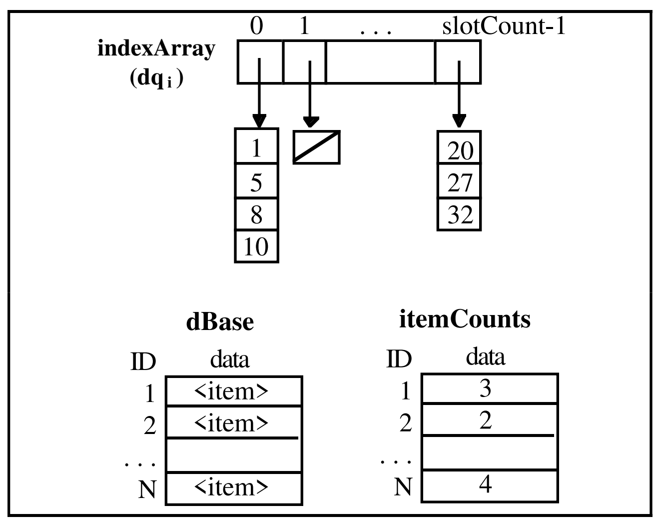

Figure 10. Main memory based k-Array approach.

In order to provide support for a fine-grained analysis of video data (e.g., analyzing the video over events that are only seconds or less than seconds long) and/or support for analyzing several videos at once, our system must be able to handle a large collection of video annotations as well as an even larger TPairs collection (given that TPairs are formed by pairwise combinations of the video annotation subsets). Ultimately, this means that the number of pairs can become too large to fit in main memory, and secondary storage would be required. For this case, we propose the following modifications to convert the main memory-based k-Array method into a disk-based structure:

- use indexArrays to indicate number of item IDs in each position (i.e., slot) rather than store a list of IDs,

- store the list of IDs on disk, in the order in which they occur in the indexArrays,
- use the minimum number of bits required to store the itemCounts on disk (for TVQL, k=4, so we need a minimum of 3 bits per itemCount),
- reserve sufficient buffer space to keep the indexArrays in main memory,
- reserve sufficient buffer space to keep the slotNum, page number (pageNum), and page offset corresponding to the current position of each query filter thumb in main memory.

The corresponding disk-based version of the k-Array is presented in Figure 11.

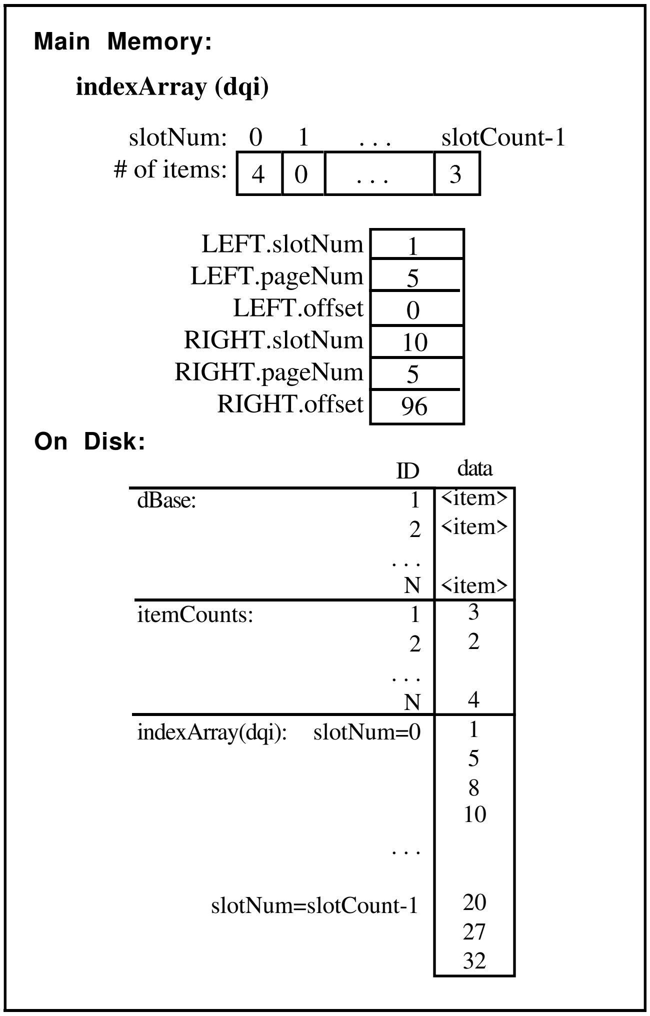

Figure 11. Disk-based version of the k-Array approach.

Our query processing strategy for our k-Array is to process each DQ user manipulation (i.e., dqManip) as follows:

- get the slotNum, pageNum and offset for the thumbID of the dq filter specified
- using the thumbID, action, and dir (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 dir (if specified), determine if 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 current slot from 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 current slot to 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

Consider the case where the left filter thumb is unfilled, and the query manipulation taken is dqManip(dq_i, LEFT, FILL, NULL). This corresponds to changing the left hand side of the specified range from

leftVal < dq_i.attributeName

to leftVal ≤ dq_i.attributeName,

meaning that the nearest neighbor is dq_i.LEFT.slotNum−1 and that we want to add that slot to the selected range. If dq_i had the corresponding indexArray in Figure 11, where dq_i.LEFT.slotNum = 1, then we would be adding all data items corresponding to slot 0 of that indexArray to the solution. Since indexArray(dq_i)[0] = 4, then we would loop over the four IDs in that slot (i.e., ID=1, ID=5, ID=8, ID=10), incrementing the count for each, and adding to the solution set the corresponding data item for each ID with a new count equal to k. Although all counts are not displayed in Figure 11, we do see that data item ID=1 has count=3. Thus, since now k would become 4, then we would fetch and add data $item_1$ to the solution set.

In general, the k-Array has a storage cost on the order of O(kN) and search cost on the order of the average slotSize—i.e., O(N/slotCount). While this search cost may not seem to be the most efficient, it is counterbalanced by the fact that all item IDs in a slot are clustered together, and minimum information is kept to maintain and cluster the corresponding counts. Given a 2K page buffer size, this means that page faults in TVQL could be reduced to one page fault per every 512 item IDs fetched from the indexArrays and one page fault per every 5461 itemCounts.

3.3.5 Processing Subset Selections

Processing subset selection queries over the annotation collection is similar to processing TVQL queries over TPairs, though much simpler. Since the annotation set is assumed to be frozen at query time, the annotations can be presorted by descriptive ID (d_ID). In addition, TViz only requires information on the frequency, average duration, and total duration for each subset of annotations with a given descriptive ID (vs. information on individual annotations). Thus, this information can be calculated ahead of time and stored directly with the respective subset highlighter objects (i.e., transparent yellow circle and blue square overlays).

Users manipulate the subset selection DQ filter lists to select subsets by d_ID. De-/selecting a d_ID then corresponds to hiding or showing the corresponding highlighter. We use a k-Array to process these incremental subset selection queries. Note that the k-Array supports access to "nearest neighbor" slots (as required by incremental selection of *continuous* ranges in the temporal query filters) as well as direct access to any slot of any DQ filter, which is desired in the case of subset selection based on *discontinuous* ranges of attributes.

4. EVALUATION AND DISCUSSION

We have divided evaluation of the MMVIS environment into two components—efficiency and usability. In Section 4.1, we present some preliminary experimental results evaluating the efficiency of our k-Array query processing strategy. The usability and utility of MMVIS are being evaluated through user studies and through a case study applying MMVIS to the analysis of real video data. In Section 4.2, we summarize the results from our case study as well as preliminary results from our TVQL user study.

4.1 Efficiency Evaluation: TVQL Processing

The linked array is another array-based query processing approach to multidimensional range queries which works well for smaller sized databases but does not scale up very well [16]. Our k-Array was designed to improve on the linked array method while maintaining a simple index structure and incremental update strategy. In particular, we were interested in improving the efficiency for working with large data sets, where secondary storage is required. In this section, we present preliminary results comparing performance of the linked array with our k-Array.

4.1.1 Assumptions

We assume that once A and B subsets have been selected, the TPairs database is formed and remains frozen. Since no deletions, insertions or updates are made to the database, the indexing structure is static and can be constructed based on fore-knowledge of the data distribution. Our ultimate goal is to process queries as quickly as possible (and updates are assumed to be rare). Since the cost to build the index is only done once, we do not consider the preprocessing index creation cost in our analysis.

4.1.2 Linked Array vs. k-Array Methods

The linked array is similar to the k-Array in that data items are sorted into slots of each of the k arrays representing each query filter. In the linked array, however, each slot holds the first item of a linked list of items for that slot, rather than information for accessing the slot's contents as one sub-array. That is, in the linked array, rather than storing the IDs to each item k times (as we propose for the k-Array method), the IDs are stored along with their counts and a set of k links, one for each `next` data pointer in the same slot of each dimension. These linked array items (i.e., the `count` and `k next` pointers) are stored sequentially on disk according to `ID` and can be accessed via page offsets.

Our proposed k-Array is an improvement over the linked array in that item IDs in each slot are clustered together. Figure 12 compares the number of page faults required for using the linked array vs. the k-Array methods for processing a sample TVQL query. The graph indicates 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). We have compared our k-Array to the linked array for a series of TVQL queries and consistently find the same result—the k-Array outperforms the linked array, especially as the data set size increases [11].

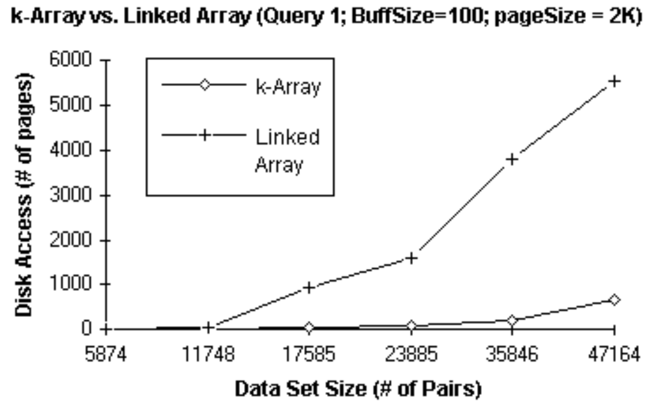

Figure 12. Number of page faults required for processing a sample TVQL query for different data set sizes: comparison between linked array and k-Array methods.

Figure 13 compares the performance of the two approaches for specifying incremental vs. non-incremental queries. A query is *incremental* when it is processed as an *update* to the previous query rather than as a new query calculated from scratch. Both methods perform better overall for incremental queries, but the k-Array still outperforms the linked array under both conditions.

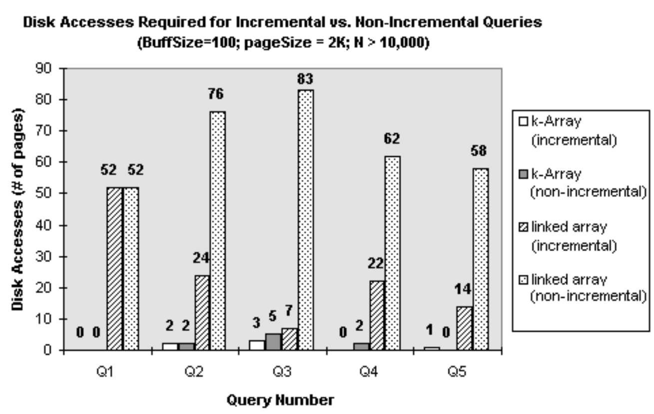

Figure 13. Number of page faults required for processing a series of incremental queries vs. processing the same queries from scratch: comparison between linked array and k-Array methods.

Our work on evaluating the TVQL Processor is ongoing and we have recently completed enhancements to the k-Array, converting it to a bucket-based method called the k-

Bucket [11]. We have compared the k-Array and k-Bucket to the linked array and other popular indexing structures such as the k-d tree [4, 5] and grid file [15, 19]. Thus far, our findings indicate that although the k-Bucket is not the most efficient under all conditions, it is the best performer overall [11].

4.2 Usability Evaluation

4.2.1 CSCW Case Study

Our case study using the exploratory approach to video analysis provided in MMVIS illustrates the utility of our system and approach [12]. In the case study, we use MMVIS to analyze real video data collected as part of a research study on computer-supported cooperative work (CSCW) [20]. The case study shows how our approach is not limited to analyzing temporal sequences studied by the original researchers, but can also be used to examine *any* type of temporal relationship—parallel, overlapping, or sequential. It also illustrates how our interactive approach simplifies the process of discovering temporal trends.

The series of sample temporal queries in the case study, for example, demonstrates how our tools can be used to investigate who in the study might have emerged as the leader of the group, even though no leader was assigned. We proposed a number of (temporal query) questions to investigate this and then used MMVIS to pose them and explore the results. We looked at the frequency and average duration of Talking events to see who talked more frequently and who spoke for longer periods of time than others. We examined the relationship between when individuals spoke and when digressions took place to see who initiated, who participated, and who ended digressions. TViz not only highlighted this information, but also highlighted the frequency (i.e., strength) of these temporal relationships. The overall results of the case study led us to some interesting observations, some of which were desirable and expected and some of which could lead to more in-depth analysis.

4.2.2 TVQL and MMVIS User Interface Studies

We are in the process of running a user interface study to evaluate TVQL compared to a forms-based query interface. While our study is not yet complete, our preliminary findings indicate that 1) users can interpret TVQL queries, 2) users can specify temporal queries using TVQL, though they have better qualitative than quantitative accuracy, and 3) users can easily understand the temporal diagrams and in fact most often quote the diagrams as the most beneficial part of the interface. Once the user interface study is complete and it raises some suggestions for improving TVQL, we plan to redesign TVQL to increase its usability. In this light, our goal is to improve the interface while still maintaining the power of the language—most notably, the power to browse within and between both temporal primitives and temporal neighborhoods, thereby preserving the notion of temporal browsing.

5. RELATED WORK

Previous video annotation and analysis systems, such as VideoNoter [21], require users to *pre-code* temporal relationships rather than allowing them to discover the relationships by coding atomic information and posing temporal queries. Other systems, such as Media Streams [6], have focused more on novel approaches to creating and finding individual annotations rather than analyzing relationships between them. While systems such as Timelines [10] and Media Streams [6] do provide some query support, their visual presentations of the annotations are restricted to text- and timeline-based displays. In contrast, MMVIS provides a new, integrated approach to temporal analysis where users can use a direct manipulation approach, called TVQL, customized for visually browsing for temporal relationships between events in the video data and review an aggregated display of results in TViz.

Although other extensions to dynamic query filters and VIS have been explored [7, 9], these extensions primarily focus on aggregation extensions to the interface. While these aggregation techniques could be incorporated into our system to enhance the formation of subsets, they do not address the temporal and relative exploratory needs of video analysis.

Previous research on processing multidimensional range queries focuses on processing queries *once* rather than processing dynamic, direct manipulation queries. Hence, the problem we are addressing is different from conventional multidimensional range query processing in that our input to the query processor corresponds to sets of *incremental* (continuous) query updates typically generated by adjusting one query filter at a time. Work by [16] comes closest to our TVQL query processing approach in that they analyze various structures for dynamic queries in general. However, the researchers focus their attention on main memory rather than disk-based methods. In this paper, we have presented an array-based indexing structure customized for incremental query processing. Advantages of our proposed strategy include not only its simplicity in implementation but also its efficiency of processing TVQL queries compared to alternative approaches, such as the linked array method.

6. CONCLUSION

In this paper, we presented details on the design and implementation of an interactive visualization environment for video analysis. In our MultiMedia Visual Information Seeking (MMVIS) environment, a visual query language is tightly integrated with a temporal visualization of results. The environment provides a new paradigm for video analysis, one in which users can explore temporal trends via interactively browsing for temporal relationships. In this paper, we presented the system design of the primary interface components of MMVIS — namely, our visual query language including the subset query and TVQL query palettes and our temporal visualization TViz.

In order to preserve the notion of interactive browsing for trend analysis, the visual queries must be processed as efficiently as possible. Thus, the query processor becomes a critical component of our system implementation. We thus have developed and fine-tuned a novel index data structure (called the k-Array method) and associated query processing strategy for handling TVQL queries. Our analysis indicates that the k-Array performs much better than the linked array as the data set size increases, and that it is particularly well suited to handle incrementally specified queries — the main mode of interaction with MMVIS.

In the future, we plan to enhance the analysis of TVQL query processing, including a comparison of the various alternate processing methods specified in the literature for specifying disjunctive multidimensional range queries. We will also continue our ongoing effort on conducting user studies to test the usability and conceptual understanding of the TVQL interface as well as the utility of the MMVIS system for identifying temporal trends.

ACKNOWLEDGMENTS

This work was supported in part by UM Rackham Fellowship, NSF NYI #94-57609, and equipment support from AT&T. Special thanks to Judy Olson for permission to use the sample data set.

REFERENCES

1. Ahlberg, C., & Shneiderman, B. (1994). The Alphaslider: A Compact and Rapid Selector. *CHI'94 Conference Proceedings*. NY:ACM Press, 365-371.
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. Davis, M. (1993). Media Streams: An Iconic Language for Video Annotation. *Telektronik 4.93: Cyberspace, 89*(4), 59-71.
7. Fishkin, K. and Stone, M.C. (1995). Enhanced Dynamic Queries via Movable Filters. *CHI'95 Conference Proceedings*. NY:ACM Press, 415-420.
8. Freksa, C. (1992). Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54, 199-227.
9. Goldstein, J. & Roth, S. (1994). Using Aggregation and Dynamic Queries for Exploring Large Data Sets. *CHI'94 Conference Proceedings*. ACM Press, 23-29.
10. Harrison, B.L., Owen, R., & Baecker, R.M. (1994). Timelines: An Interactive System for the Collection of Visualization of Temporal Data. *Proc. of Graphics Interface '94.* Canadian Info. Processing Society.
11. Hibino, S. and Rundensteiner, E. "Processing Incremental Multidimensional Range Queries." University of Michigan Tech. Report (in preparation).
12. 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.).
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. Hibino, S. & Rundensteiner, E. (1995). A Visual Query Language for Identifying Temporal Trends in Video Data, *Proc. of the 1995 International Workshop on Multi-Media Database Management Systems*. Los Alamitos, CA: IEEE Society Press, 74-81.
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. Mackay, W. E. (1989). EVA: An experimental video annotator for symbolic analysis of video data. *SIGCHI Bulletin*, 21(2), 68-71.
18. Nagasaka, A. and Tanaka, A. (1992). Automatic Video Indexing and Full-Video Search for Object Appearances. *Visual Database Systems, II* (E. Knuth and L.Wegner, Eds.). Elsevier Science Publ., 113-127.
19. Nievergelt, J. & Hinterberger H. (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. Roschelle, J., Pea, R., & Trigg, R. (1990). VIDEONOTER: A tool for exploratory analysis (Research Rep. No. IRL90-0021). Palo Alto, CA: Institute for Research on Learning.