Querying Video Libraries*

Eenjun Hwang, V.S. Subrahmanian
Department of Computer Science
Institute for Advanced Computer Studies
Institute for Systems Research
University of Maryland
College Park, MD 20742.
{hwang, vs}@cs.umd.edu

Abstract

There is now growing interest in organizing and querying large bodies of video data. In this paper, we will develop a simple SQL-like video query language which can be used not only to identify videos in the library that are of interest to the user, but which can also be used to extract, from such a video in a video library, the relevant segments of the video that satisfy the specified query condition. We investigate various types of user requests and show how they are expressed using our query language. We also develop polynomial-time algorithms to process such queries. Furthermore, we show how video-presentations may be synthesized in response to a user query. We show how a standard relational database system can be extended in order to handle queries such as those expressed in our language. Based on these principles, we have built a prototype video retrieval system called VIQS. We will describe the design and implementation of VIQS and show some sample interactions with VIQS.

1 Introduction

Recent years have seen a spectacular increase in the ability to deliver video-on-demand (VOD) services to customers over electronic networks. In most such VOD systems, the user specifies a simple request of the form "I'd like to view the movie `The Rope`" and the VOD system's task is limited to retrieving the movie from its resident disk location and delivering it to the customer. This task, though, may be far from trivial, involving subtle multiplexing issues and efficient usage of network bandwidth.

In contrast, our aim in this paper is to be able to retrieve appropriate *segments* of video from a video-library. Thus, for example, a user may (informally) ask queries such as: `Find me a segment of video where a party occurs.` This informally phrased query admits a number of possible interpretations:

*This work was supported by the Army Research Office under Grant Nr. DAAL-03-92-G-0225 and by the Air Force Office of Scientific Research under Grant Nr. F49620-93-1-0065, and by ARPA/Rome Labs contract F30602-93-C-0241 (ARPA Order Nr. A716).

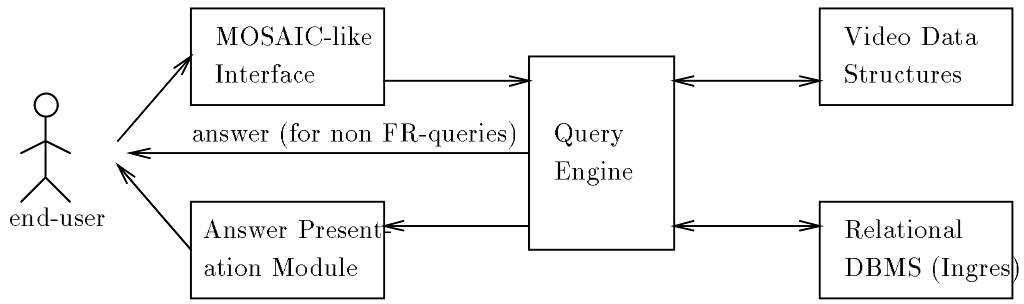

Figure 1: Architecture of the VIQS System

- For instance, a party occurs in frames 33 through 56 of the movie, `The Rope`. Should this *entire* segment of video be included in the solution or is it enough to show the user any subsegment?
- Surely, the user should be able to specify that he would like to see at most say 10 frames of video that satisfy the specified query. For example, the user may not wish to see 30 minutes of video, he may initially just want to take quick peek at the relevant portions and reserve the right to view the rest of the video at his leisure.
- Third, there may be several segments of videos (either from the same movie, or from different movies) that satisfy a given query. The user should have the ability to specify that he wishes to view "a little bit" (e.g. a few frames) or "all" of one or each of these video segments.

Before proceeding any further, we present a "birdseye" view of the overall VIQS architecture so that the reader may have a global picture of how the different sections of this paper fit together. The user interacts with VIQS through a MOSAIC-like graphical user interface (described in Section 6). This interface allows the user to express queries in an SQL-like language that we have developed. The language itself is described in Section 3. This query language allows the user to retrieve answers in textual form that may require access to video-data as well as to retrieve relevant video segments from one or more movies. Queries involving retrieval of video segments are a special class of queries called *frame-request* queries (or FR-queries, for short). Using FR-queries, the user may specify the lengths of the segments that he wishes to see, or he may ask the system to generate a maximal segment (or segments) that satisfies his query. To facilitate this, the query language contains various primitive operations that apply to *sets* of video-segments. The query engine executes these queries using a formal model of video data structures defined by Adalı et. al. [1]. This paper builds on the data structures defined in [1], and adds two fundamentally new contributions: first, it presents a unified query language within which video data can be accessed – this query language includes temporal modalities; second, this query language provides various algebraic operations to compose video-segments together. The answer presentation module (described in Section 4) of VIQS provides a unified paradigm within which different video-segments retrieved in response to a user query, may be merged together to form a single cohesive body of video. In this paper, in addition to the specification of the query language, we develop algorithms that compute the aforementioned algebraic operations on sets of video-segments.

2 Preliminaries

In [1], the authors develop a formal model of video data and devise a storage scheme using modified spatial data structures. In this section, we present a quick overview of the video model developed in [1].

As is well known, whenever a movie is played on a VCR, the particular point in the movie that is currently being displayed may be captured by a number (e.g. 1155) on the VCR monitor. This number may be viewed as a frame number of the video. Furthermore, any video contains a number of *objects* of interest (e.g. the different characters appearing in the movie, different items such as guns, dogs, etc. are all objects) as well as a number of *activities* (e.g. weddings, murders, parties, etc.) of interest. Furthermore, an *event* is a kind of activity – for example, the activity `wedding` may have two events in the movie – one referring to the wedding of John and Mary, while the other refers to the wedding of Ed and Lisa. These two events share the same activity type, i.e. *wedding*, but have two distinct sets of participants in that event. Associated with any activity type is a set of *roles* – for example, the activity type `wedding` may have two associated roles – *groom* and *bride*. In any event, there are certain *players* in these roles – for instance, in the event `Wedding of John and Mary`, the role *groom* is played by John, the role *bride* is played by Mary. Adalı et. al. [1] develop a formal model of the above informal description of video-information and develop specialised data structures to store such information.

Every object and every event that occurs in a movie occurs during certain *segments* of the movie. Thus, for instance, the event `Wedding of John and Mary` may occur between frames 1133 and 1147. We would represent this frame-sequence as the left-closed right-open interval $[1133, 1148)$ that denotes the set $\{x \mid 1133 \leq x < 1148\}$ where x ranges over the integers. In general, an object/event may occur in multiple frame sequences. For instance, the object John may occur in frame-sequences $[1078, 1101), [1133, 1148)$ and $[1198, 1225)$. Adalı et. al. [1] use this intuition to argue that associated with each object/activity/event, is a set of frame-sequences. Each frame sequence can be viewed as a line-segment; hence, associated with any object/activity/event, is a set of line segments. They enhance an existing data structure, called the segment tree (cf. Samet [11]) and use that as a basis for querying. This data structure is described below. Each node in a segment tree represents an interval. The root represents the entire interval associated with the movie (e.g. $[0, 1200)$). The interval associated with a node is the disjoint union of the intervals associated with its children. The `start` and `finish` fields associated with a node represent the interval; the `objlist` associated with a node shows all objects that occur in *all* frames in the interval associated with a node. The `actlist` is similarly defined.

```
type treenode = record of
     start      : integer;
     finish     : integer;
     objlist    : ^objnode;
```

```
     actlist    : ^actnode;
     lchild     : ^treenode;
     rchild     : ^treenode;
     end;

type objnode = record of
     objid : integer
     next  : ^objnode
     end

type evtnode = record of
     evtid : integer
     next  : ^evtnode
     end
```

The OBJECTARRAY: The object array is an array whose i'th element denotes video object number i. Associated with any element of this array is an *ordered linked list* of pointers to nodes in the frame segment tree.

The EVENTARRAY: As in the case of OBJECTARRAY, the event array stores a linked list of pointers to nodes in the frame segment tree for each different event. Events are uniformly numbered from 1 to N. For each event, we store the activity type, the team and the list of tree nodes in the segment tree.

The ACTIVITYARRAY: This is another index which simply stores for each activity type the list of events of that type. This will facilitate queries about a certain activity type as opposed to a certain event of that type. Similarly, a ROLEARRAY simply lists the name of the roles.

3 Query Language

In the preceding section, we described, albeit briefly, a data structure developed by Adalı et. al. [1] to index video data. In this section, we will show how this data structure can be used to facilitate the execution of queries. We will enumerate examples of such queries and show how these queries can be answered using our data structure. Before describing the query language, we observe that queries to video data may involve answers having two forms – the answer to a query may only include textual data (e.g. `What is the name of the actor playing the role of the murder victim in the movie, ''The Rope''?` – even though this query involves accessing video data, the answer is just a string. In contrast, the query `Find the video-segments in ''The Rope'' where a murder takes place` asks for relevant video-footage. Thus, queries are of two types – those that involve textual answers and those that involve returning video-segments to the user. Queries of the latter kind are called *frame-request (FR)-queries* (we will define them formally later). Below, we present some general mathematical definitions pertinent to FR-queries.

3.1 Frame Sequences

A frame sequence $[i, j)$ where $1 \leq i < j \leq n$ is said to satisfy a query Q iff for each $i \leq k < j$, frame k satisfies query Q.

A frame sequence $[i, j)$ is said to maximally satisfy a query Q iff

1. frame sequence $[i, j)$ satisfies Q and
2. there are no other frame sequence $[i', j')$ where $i' < i$ or $j < j'$ satisfying Q.

Given a query, there may, in general, be several frame sequences that satisfy the query. Furthermore, the length of each frame sequence satisfying the query might be different. For example, the query *Find all frame-sequences in which a brown dog appears* may be satisfied by frame sequences $[5, 11)$ as well as $[56, 90)$. The user may or may not wish to view the entire set of frame sequences associated with a query. For example, some frame sequences may be very long, and it may be extremely tedious and time-consuming for the user to have to watch the whole frame-sequence. Hence, we need some method to control the length of frame sequence that will be presented to the user.

In order to address this problem, we allow the user to parametrize frame sequence variables. The purpose of such parametrized frame-sequence variables is to allow the user to specify, in his query, a length of the presentation that he wishes to view. The parameters that can be used in the query are shown below.

- 1: choose a single frame from the frame sequence.
- k: choose a frame sequence of length k from the frame sequence
- *: choose a maximal frame sequence.

3.2 Syntax of Query Language

The syntax of the query language in this paper is similar to SQL. Its general structure is as follows.

```
     FIND result_object_type[parameter]
/* the object type returned by the query */
     FROM video_name [frame_list]
/* the video name and frame sequence list */
     [WHERE condition_list]
/* criterion for selecting frame sequences */
```

In the FIND clause, the type of the object which will be returned as a result of the query is specified. In our index scheme and query language, we support three different types.

- **Frames :** frame sequences satisfying the query are returned.

- **Objects :** objects that appear in the set of frame sequence specified in the query are returned.
- **Activities :** activities that occur in the set of frame sequence specified in the query are returned.

Every video present in a video-library is not only indexed by the data structures shown above, but may also have related information about the video stored in a relational DBMS. The relational database(s) may contain information relevant to the movie itself and/or information about objects in the movie. Our query language supports accessing not only the data stored in the specialized AVIS data structures, but also supports accessing data stored in these affiliated relational DBMSs.

In the FROM clause of a query, we may indicate any specific video name stored in the database on which we want to execute a query. In the context of query processing, the video name provides a pointer to the entire frame-segment tree associated with that video. Depending on the application, we may need to restrict the range of frame sequences for which the index data are accessed and searched. For example, we need to know all the objects appearing in the specific set of frame sequences. If the user is familiar with that movie, then by specifying its range, the query can be executed more efficiently than just from the scratch.

There are two ways that can be used to restrict the range. In the first way, the user may specify a set of frame sequences directly in the query. This set of specified frame-sequences may either have been explicitly specified by the user, or may have been obtained as the result of a previously executed query. The second possibility is to specify any query that returns a set of frame sequences as a result.

The WHERE clause specifies the properties we want frame sequences to have. In our scheme, a body of video data is characterized by the events and the objects that occur within that body of video data. Therefore, in order to select the frame sequences that we want, it is sufficient to specify conditions on the objects and events that we wish to see.

- Object condition: **obj has (not)** object_name
- Activity condition: **act has (not)** activity_name[:player list]

The object condition specifies the list of objects that the user thinks are relevant to filtering the frame sequences. In the above specification, the keyword **obj** refers to the set of objects appearing in the corresponding frame sequence. Similarly, the operator **has** tests whether the input object appears in the **obj** list associated with that frame sequence. If the object appears in the list of objects associated with a frame (or frame sequence), then the condition **obj has** object_name evaluates to true.

Likewise, the activity condition plays a similar role in query processing. As activities have greater structure than objects (i.e. they have associated players in specified roles, etc.) in this case, we can specify more detailed conditions on the activities using the list of players appearing in the event.

3.3 Frame-Request Queries

A special case of the query format occurs when the `result_object_type` is `frame`, i.e. the user is interested in finding frames. These queries are of the form:

```
FIND frames[parameter]
FROM video_name [frame_list]
[WHERE condition_list]
```

Unlike other types of queries that return symbol/textual answers, frame-request queries require presenting the user with a list of frames that satisfy a given query. However, consider a simple query of the form

```
FIND frames[*]
FROM Oliver Twist
WHERE obj has 'Fagin'.
```

Suppose Fagin was present in frames $[8, 16)$ and frames $[29, 32)$. This means that the set of frames in which Fagin appears is $\{8, 9, 10, 11, 12, 13, 14, 15, 29, 30, 31\}$. However, this same set can also be represented by the three intervals $[8, 11), [11, 16), [29, 32)$ – the reader will easily see that many other representations are possible. Yet, we would like representations to be compact, cohesive chunks of data. Even more complex scenarios may occur when, for instance, we wish to ask a query of the form

```
FIND frames[*]
FROM Oliver Twist
WHERE obj has 'Fagin' AND act has robbery.
```

Here, we may have robberies occurring in frames $[13, 19)$ and $[41, 49)$ – hence, the frames that satisfy this query are frames $\{13, 14, 15\}$. Though computing this set is easy, compactly representing large sets of this kind may not be easy. In order to facilitate reasoning about sets of frame-sequences, we now introduce some special definitions. These apply only to the case of queries of the form `FIND frames`$[*]$ `FROM` $< -, - >$ `WHERE` $< -, - >$.

An answer A to a frame-request query is { f | f is a frame and satisfies the condition}. Namely, an answer consists of all the frames k such that frame k satisfies the query.

However, as is clear from the above examples, the answer A to a query may be a very large set. It certainly makes sense to represent such sets in a compact form. Below, we define the notion of a *presentation* of an answer – a presentation is a compact representation of an answer.

A presentation, PRES(A), of an answer A is a set of frame sequences $f_1, \ldots, f_n$ such that

1. $f_i \cap f_j = \phi$ for all $i \neq j$

2. $f_1 \cup \ldots \cup f_n = A$

3. there is no frame sequences f_i, f_j such that f_i = [a,b) and f_j = [b, c)

Whenever we answer a query of the form `FIND frames`[$*$]..., the answer should be presented according to the definition above. In particular, this means that even if the query has a conjunctive condition in the `WHERE` clause, the set of frame-sequences corresponding to each conjunct must be integrated together so as to satisfy the conditions described above. Suppose now that A_1, A_2 are answers to two queries Q_1, Q_2. Then the answer to the conjunction of queries Q_1, Q_2 is $A_1 \cap A_2$. We would like to define algorithms which, instead of taking A_1, A_2 as input, take instead, `PRES`(A_1), `PRES`(A_2) as input, and return as output, `PRES`($A_1 \cap A_2$). Of course, this can be done by brute-force, but this would be inefficient. In Section 4, we show how such algebraic operations on presentations may be efficiently implemented.

3.4 Types of Queries

We will now define different types of queries and show how they can be expressed in the VIQS query language. Not only will we show how they can be represented in the VIQS query language, we will also outline how these queries may be evaluated by traversing the data structures described in Section 2.

3.4.1 Elementary object query

This is a query of the form: "Find all the video frames where a given object appears from the set of frame sequences."

Example: "Find all video frame sequences in the movie “The Rope” where Brandon appears."

```
FIND frames[*]
FROM Rope
WHERE obj has 'Brandon'
```

Method: This query can be processed by first finding the entry of the object (e.g. Brandon) in question in the OBJECTARRAY using a hash table index into the OBJECTARRAY. Then, follow the pointers in the frames field, creating a set of frame sequences corresponding to the start and end points of the tree nodes pointed by this field, and finally merge the frame sequences (as described in Section 4) to obtain a valid presentation of the answer.

3.4.2 Elementary activity query

This is a query of the form: “Find all the video frames in which events of a given activity type occur.”

Example: “ Find all video frame sequences in the movie “The Rope” where someone is murdered.”

```
FIND frames[*]
FROM Rope
WHERE act has 'murder'
```

Method: The query may be answered by first locating all the events corresponding to the activity type (e.g. murder) given in the WHERE clause. This can be done by looking into ACTIVITYARRAY using a hash table, and then following the events field which has event id into the EVENTARRAY. The set of frame sequences for all such events are obtained one by one by following the links in the EVENTARRAY and collecting these frame sequences into a set. These sets of frame sequences are then merged into a final solid set to give the final answer.

3.4.3 Detailed activity query

This is a query of the form: ”Find all video frames in which one of a given set of events occurs, where the events are specified by the activity type and the roles of specific objects involved in this activity.”

Example: ”Find all the video frames in which Rupert is given a clue by Philip.”

```
FIND frames[*]
FROM Rope
WHERE act = 'finding clue':finder = 'Rupert':
      giver = 'Philip'
```

Method: The query can be solved in a manner similar to the Elementary activity query, except that the search to locate relevant events is more complex. For this case, we first locate (using a hash table), the given activity type in ACTIVITYARRAY. Then all the events linked to the entry are followed checking to see whether teams contain all of the necessary players. Then, for all those events, the link to the frame segment tree is followed, forming a solid set of frame sequences. Finally, all these sets are merged to give the final answer.

3.4.4 Object occurrence query

This is a query of the form: ”Find all objects that occur in a given set of frame sequence.” The frame sequences are specified directly or by another query returning frame sequences as result.

Example: ”Find all the objects that are present in frame sequence [5,40).”

```
FIND objects
FROM Rope[5:40]
```

Method: The query may be solved by searching the frame segment tree starting from the root for the given set of frame sequences. If the node being visited intersects with any of the frame sequences in the set, all the objects stored in this node are added to the output set of objects, and then both the left and the right children of the node are visited. It is possible to split the set of frame sequences for the children so that only those sequences that may possibly intersect with the corresponding child are included in that call.

3.4.5 Activity occurrence query

This is a query of the form: "Find all the activities that occur in a given set of frame sequences."

Example: "Find all the activities that occur in the frame sequence where Rupert appears."

```
FIND activities
FROM Rope[] =
    ( FIND frames[*]
      FROM Rope
      WHERE obj has 'Rupert' )
```

Method: In this query, the activity search itself is very similar to the object search. The key difference is that the input frame sequences are specified by another query. Hence, this query can be solved in two stages. In the first stage, execute the elementary object query specified in the FROM clause to get the result frame sequences where Rubert appears. In the second stage, search the segment tree for each frame sequence given in the first stage, collecting activities as in the case of object occurrence queries.

3.4.6 Conjunctive query

This is a query in which the `WHERE` clause involves a conjunction of conditions. Thus far, all the queries have had only one type of condition in the `WHERE` clause. In a conjunctive query, we can connect them using logical connectives to compose more complicated conditions in the WHERE clause. Executing this query will involve the decomposition of the query into elementary subqueries, and then compose the results of these sub-queries to form an answer to the overall query.

Example: Find all the frames where people are eating in the place where a chest can be seen."

```
FIND frames[*]
FROM Rope
WHERE obj has 'chest' and act has 'eating'.
```

Method: The algorithms depends on the specific query posed. But the general rule to execute the conjunctive queries is to decompose conditions into simpler ones. For example, this query can be decomposed into an elementary object query and activity query each of which can be executed using the algorithm mentioned above. Finally, we intersect these frame sequences to obtain the answer[1].

4 Composing Presentations

Complex queries like conjunctive queries can be executed by decomposing the selection conditions into elementary, atomic ones. Processing an elementary atomic condition C_1 produces a presentation, $\texttt{PRES}(A_1)$, of the answer A_1 of the selection condition C_1. Similarly, processing an elementary atomic condition C_2 produces a presentation, $\texttt{PRES}(A_2)$ of the answer A_2 of the selection condition C_2. The answer to the conjunctive query:

```
FIND frames[parameter]
FROM video_name [frame_list]
WHERE C1 & C2.
```

is $A_1 \cap A_2$. Hence, as shown in Figure 2, one way to compute the presentation $\texttt{PRES}(A_1 \cap A_2)$ is to compute $A_1 \cap A_2$ and then compute a presentation of $A_1 \cap A_2$. However this is inefficient because when computing elementary frame-request queries (such as C_1, C_2), our algorithms yield a presentation (such as $\texttt{PRES}(A_1), \texttt{PRES}(A_2)$). Surely, there must be a way to *directly combine* $\texttt{PRES}(A_1)$ with $\texttt{PRES}(A_2)$ to obtain $\texttt{PRES}(A_1 \cap A_2)$? Figure 2 shows a diagrammatic rendering of the situation. The dashed-lines in the figure show solutions that have already been computed. The dotted line shows what we would *like* to compute. The two bold-lines show how we would like to compute $\texttt{PRES}(A_1 \cap A_2)$ using $\texttt{PRES}(A_1)$ with $\texttt{PRES}(A_2)$ directly.

A diagram similar to that in Figure 2 applies when we are interested in considering the *disjunction* of conditions C_1 and C_2. In that case, we would like to directly combine $\texttt{PRES}(A_1)$ with $\texttt{PRES}(A_2)$ to obtain $\texttt{PRES}(A_1 \cup A_2)$.

4.1 Computing Intersections of Presentations

In this section, we present an algorithm that takes two presentations $\texttt{PRES}(A_1)$ and $\texttt{PRES}(A_2)$ as input (each represented as an array) and returns $\texttt{PRES}(A_1 \cap A_2)$ as output.

Algorithm 1 : Composing an intersection of two frame sequences

Input : Two integer arrays ARR1 and ARR2 of size 2n and 2m respectively which consists of the start and finish point of maximal frame sequences $fs1$ and $fs2$.

[1] Conjunctive query optimization strategies will be considered in detail in a future paper.

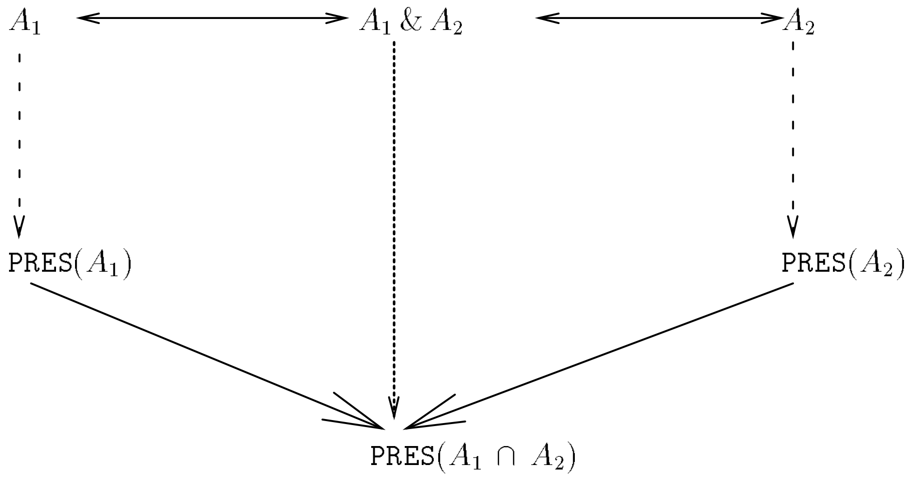

Figure 2: Composing Presentations

Output : An answer presentation $fs1 \bigotimes fs2$ corresponding to the set of frame $\{(f_1 \cup \ldots \cup f_n) \cap (f'_1 \cup \ldots \cup f'_m)\}$

```
int st, fi,       /* start and end of each frame in the result */
    sp[1], sp[2], /* variables pointing to array index */
    dt[1], dt[2], /* variables for entering and exiting segment */
    line_num;     /* number of lines in consideration */

initialize  line_num to zero ;
initialize  sp[1] and sp[2] to one ;
initialize  dt[1] and dt[2] to one ;

set st to the min (ARR1[sp[1]], ARR2[sp[2]]) ;

if (ARR1[sp[1]] == ARR2[sp[2]])
          line_num = line_num + dt[1] + dt[2] ;
          increase sp[1] and sp[2] by one ;
          dt[1] = -dt[1] ; dt[2] = -dt[2] ;
     else
          set i to sp of min(ARR1[sp[1]],ARR2[sp[2]]) ;
          line_num = dt[i] ;
          dt[i] = -dt[i] ;
          increase sp[i] by one ;

while (sp[1] <= 2n and sp[2] <= 2m)
     if (line_num == 2)
          set fi to the min(ARR1[sp[1]],ARR2[sp[2]]) ;
          output pair of (st, fi);
          if (ARR1[sp[1]] == ARR2[sp[2]])
               line_num = line_num + dt[1] + dt[2] ;
```

```
            dt[1] = -dt[1] ;  dt[2] = -dt[2] ;
            increase sp[1] and sp[2] by one ;
        else
            set i to sp of min(ARR1[sp[1]],ARR2[sp[2]]) ;
            line_num = line_num + dt[i] ;
            dt[i] = -dt[i] ;
            increase sp[i] by one ;
    else
        set st to the min(ARR1[sp[1]],ARR2[sp[2]]) ;
        if (ARR1[sp[1]] == ARR2[sp[2]])
            line_num = line_num + dt[1] + dt[2] ;
            dt[1] = -dt[1] ; dt[2] = -dt[2] ;
            increase sp[1] and sp[2] by one ;
        else
            set i to sp of min(ARR1[sp[1]],ARR2[sp[2]]) ;
            line_num = line_num + dt[i] ;
            dt[i] = -dt[i] ;
            increase sp[i] by one ;
```

To see how the algorithm works, we show a figure where each frame sequence is represented by a line segment and the whole movie is represented by a whole line. In figure 3, the numbers specified on the line indicate how many input lines are under consideration on that segment. For example, the line segment with value 0 means that any frame in that segment should not be in the presentation. Similarly, the line segment with value 2 indicates that all frames in this segment are contained in both input frame sequences. In the intersection operation, any segment with value 2 should be in the presentation. The algorithm is easily generalized to a presentation where we have n conjuncts instead of just 2. In that case, any segment with value n should be in the presentation.

Geometrically speaking, one may think of the algorithm as working by sweeping a vertical line across the horizontal lines shown in Figure 3. The vertical sweep line stops whenever either a frame-sequence is being "entered" (i.e. when the sweep line encounters a new frame sequence) or when a frame sequence is being "exited" (i.e. when the sweep line leaves behind a frame-sequence it was in previously). The algorithm describes how one keeps track of the frame-sequences in the two presentations being intersected so as to create the composite presentation consisting of the intersection of the input frame sequences.

To see how this algorithm works, consider the following example.

Intersection Example: Suppose we consider the movie, "The Rope" and assume that it has 160 frames and that:

- Rupert appears in frames $[8, 24), [61, 75)$ and $[111, 132)$, and
- David appears in frames $[11, 19), [70, 79), [91, 97)$ and $[128, 135)$.

Consider now the intersection query where we wish to find all frames in which both Rupert

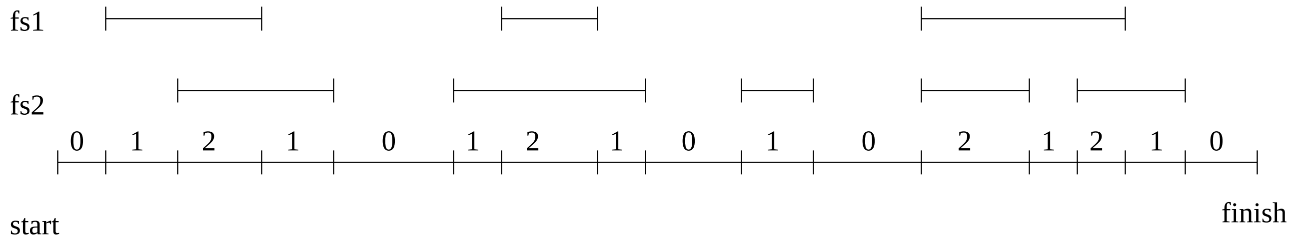

Figure 3: Intersection of two frame sequences

and David appear. This query involves computing the *intersection* of the two presentations given above. In this case:

$$\begin{aligned} \texttt{ARR1} &= [8, 24, 61, 75, 111, 132]. \\ \texttt{ARR2} &= [11, 19, 70, 79, 91, 97, 128, 135]. \end{aligned}$$

Initially, $\texttt{ARR1}[\texttt{sp}[1]] = 8$ and $\texttt{ARR2}[\texttt{sp}[2]] = 11$. The algorithm first compares 8 and 11. As $8 < 11$, `t` increments $\texttt{sp}[1]$, adds `dt[1]` to `line_num` and changes the sign of `dt[1]`. It also stores 8 into `st` as a possible start point of a frame sequence in the output presentation. The next elements to be compared are 24 and 11. As $24 > 11$, the algorithm increments `sp[2]`, adds `dt[2]` to `line_num` and changes the sign of `dt[2]`. Also, as a new possible start point, it updates `st` to 11. Now, it is given 19 and 24. However, as $\texttt{line\_num} = 2$, it sets `fi` to the minimum of these two numbers (in this case, 11) and outputs `(st,fi)` as one of the frame-sequences in the intersection. The role of the `dt[i]` variable is to indicate entering and exiting line segments. $\texttt{dt}[1]$ indicates the status associated with the first presentation, while `dt[2]` indicates the status associated with the second presentation. Whenever $\texttt{dt}[\texttt{i}] = 1$, it means that the sweep line is not currently "within" a line from the `i`'th presentation; when $\texttt{dt}[\texttt{i}] = -1$, this means that the sweep line is currently "within" a line from the `i`'th presentation (for $\texttt{i} = 1, 2$). When entering a new line segment, `line_num` is incremented by one. The same process is repeated as the sweep line moves across the presentations, producing the final answer presentation [11,19),[70,75),[128,132).

It is easy to see that the intersection composition algorithm works in time $O(max(m, n))$ where m and n are the number of of frame-sequences in the two input presentations, i.e. the algorithm is linear in the size of the two inputs.

4.2 Computing Unions of Presentations

In this section, we present an algorithm that takes two presentations $\texttt{PRES}(A_1)$ and $\texttt{PRES}(A_2)$ as input (each represented as an array) and returns $\texttt{PRES}(A_1 \cup A_2)$ as output.

Algorithm 2 : Composing a union of two frame sequences

Input : Two integer arrays ARR1 and ARR2 of size 2n and 2m respectively which consist of the start and finish point of maximal frame sequences $fs1$ and $fs2$.

Output : An answer presentation $fs1 \bigoplus fs2$ corresponding to the set of frame $\{(f_1 \cup \ldots \cup f_n) \cup (f'_1 \cup \ldots \cup f'_m)\}$

```
int st, fi,       /* start and end points of each frame in the result */
    sp[1], sp[2], /* variables pointing array index */
    dt[1], dt[2], /* variables for entering and exiting line segment */
    line_num;     /* number of lines in consideration */

initialize  line_num to one ;
initialize  sp[1] and sp[2] to one ;
initialize  dt[1] and dt[2] to one ;

set st to the min (ARR1[sp[1]], ARR2[sp[2]]) ;

if (ARR1[sp[1]] == ARR2[sp[2]])
          line_num = line_num + dt[1] + dt[2] ;
          increase sp[1] and sp[2] by one ;
          dt[1] = -dt[1] ; dt[2] = -dt[2] ;
     else
          set i to sp of min(ARR1[sp[1]],ARR2[sp[2]]) ;
          line_num = dt[i] ;
          dt[i] = -dt[i] ;
          increase sp[i] by one ;

while (sp[1] <= 2n and sp[2] <= 2m)
     if (line_num == 0)
          output pair of (st, fi) ;
          set st to the min(ARR1[sp[1]],ARR2[sp[2]]) ;
          if (ARR1[sp[1]] == ARR2[sp[2]])
               line_num = line_num + dt[1] + dt[2] ;
               dt[1] = -dt[1] ;  dt[2] = -dt[2] ;
               increase sp[1] and sp[2] by one ;
          else
               set i to sp of min(ARR1[sp[1]],ARR2[sp[2]]) ;
               line_num = line_num + dt[i] ;
               dt[i] = -dt[i] ;
               increase sp[i] by one ;
     else
          set fi to the min(ARR1[sp[1]],ARR2[sp[2]]) ;
          if (ARR1[sp[1]] == ARR2[sp[2]])
               line_num = line_num + dt[1] + dt[2] ;
               dt[1] = -dt[1] ; dt[2] = -dt[2] ;
```

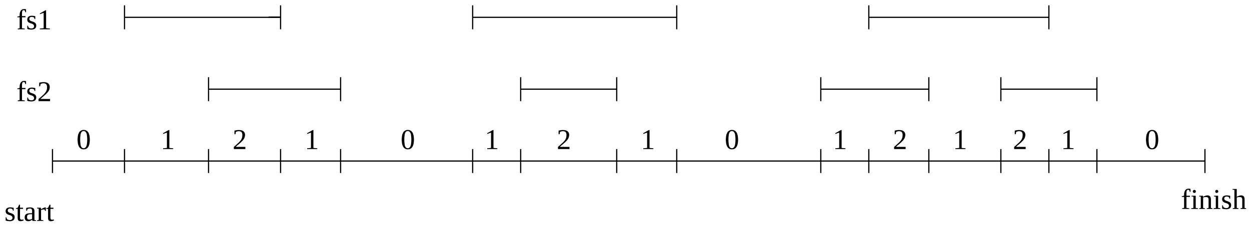

Figure 4: Union of two frame sequences

```
                increase sp[1] and sp[2] by one ;
          else
                set i to sp of min(ARR1[sp[1]],ARR2[sp[2]]) ;
                line_num = line_num + dt[i] ;
                dt[i] = -dt[i] ;
                increase sp[i] by one ;

if (sp[1] < 2n)
     output remaining frames in ARR1 with st, if any ;
else if (sp[2] < 2m)
     output remaining frames in ARR2 with st, if any ;
```

The algorithm considers all the start and finish points of the frame sequences given as input. Whenever it enters a new frame sequence(meets start point), it increments `line_num` variable by one. Also, whenever it exits a frame sequence(meets finish point), it decrements `line_num` variable by one.

The algorithm works in exactly the same way as the intersection algorithm described earlier. The key difference is that any segment of the movie that has a value greater than 0 is included in the union. This is true even if n elementary atomic conditions are involved, not just two.

Union Example: Let us return to the Intersection Example presented earlier, and consider instead, the query *Find all frames in which either Rupert or David* appeared. In this case, we need to apply the union composition algorithm to the two presentations described in the Intersection example. The union composition algorithm works as follows: Initially, $\texttt{ARR1}[\texttt{sp}[1]] = 8$ and $\texttt{ARR2}[\texttt{sp}[2]] = 11$. First, it compares 8 and 11. As $8 < 11$, it increments `sp`[1], adds `dt`[1] to `line_num` and changes the sign of `dt`[1]. It also stores 8 in `st` as a start point for a frame-sequence in the unioned-presentation. Unlike the intersection algorithm, the union algorithm considers the next point encounted by the sweep line as a possible finish point. The next elements that should be compared are 24 and 11. As $24 > 11$, it increments `sp[2]`, adds `dt[2]` to `line_num` and changes the sign of `dt[2]`. Also, as a possible finish point, it stores 11 in `fi`. It will repeat this process till $\texttt{line\_num} = 0$, when it can output one frame-sequence in the answer presentation. In this example, [8,24) could be the first such frame-sequence in the answer presentation. For the next pair of elements, it will continue this comparison, constructing the answer presentation on the fly. The final presentation would be [8,24),[61,79),[91,97),[111,135).

Like the intersection composition algorithm, the union-composition algorithm also works in time $O(max(m, n))$ where m and n are the number of of frame-sequences in the two input presentations, i.e. the algorithm is linear in the size of the two inputs.

4.3 Computing Complements of Presentations

We observe that the set $\{\&, \vee, \neg\}$ is a complete set of logical connectives, i.e. all boolean operations can be expressed in terms of these three connectives. Suppose a user wishes to express an FR-query of the form

```
FIND  frames[parameter]
FROM  video_name [frame_list]
WHERE obj has not obj_name
```

This query aks the user for all frames (within the specified parameters) that do not contain a given object. Similar queries can be expressed for situations where activities are missing.

Example: "Find all frame-sequences in the movie, The Rope, in which Rupert does not appear." This query can be expressed as:

```
FIND  frames[*]
FROM  rope
WHERE obj has not 'Rupert'
```

Our indexing method allows us to compute frames in which Rupert appears, but no automatic indexing is available for the latter. In order to compute this query, we first find a presentation of the answer, PRES(A), of all frame sequences where Rupert appears, and then we need to compute PRES($\overline{A}$), i.e. we need to be able to find a presentation of the complement of A – however, instead of taking A as input to the algorithm, we need to work with a *presentation* of A. For this, all we need to do is to use the same algorithm as before, except that now, we must return all segments (cf. Figure 5) that are marked with 0.

Complement Example: Let us return to the Intersection Example presented earlier, and consider instead, the query *Find all frames in which Rupert was not present.* In this case, we need to apply the complement composition algorithm to the presentation associated with Rupert given in the Intersection example. The complement composition algorithm works as follows: First, it checks to see if the first point is the first frame of the movie. If so, `st` is set to the next element. If not, `st` is set to the first frame of the movie. Then it will make up an answer presentation when crossing the start point of each frame segment (which is actually the `fi` value) setting the next element as next `st` value of next

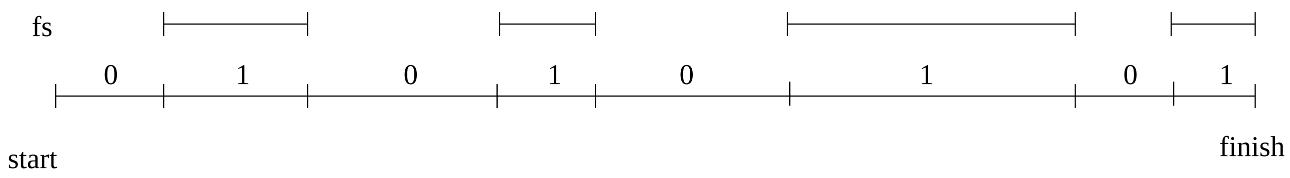

Figure 5: Complement of a frame sequence

presentation. For example, given the first frame sequence, the final presentation would be [1,8),[24,60),[75,110),[132,160).

The complement-composition algorithm described here works in time O(n) where n is the number of frame-sequences in the input presentation, i.e. the algorithm is linear in the size of the input.

5 Relation-coupled query

In the previous section, we defined various types of queries, developed a query language to express such queries, and also developed algorithms to efficiently process those queries. However, the query language presented thus far falls short of the ideal in a number of ways.

Suppose we consider a user who asks the following types of queries:

(Query 1) Find out which actor played Rupert in "The Rope."

(Query 2) Present a video-clip consisting of 5 frames each from each movie other than "The Rope" in which this actor has starred.

(Query 3) Find out which actors and which actresses in "The Rope" also act in "Rear Window" and for each such actor and actress, show a frame-sequence of 5 frames or less from "Rear Window" in which that actor and actress appear together.

The data structures of Adalı et. al. [1] are not adequate to express the information requested in the above queries. However, there is no need to reinvent the wheel – data such as which actor appeared in which role in which movie is typically likely to be stored in a relational database management system. Query 1 above requires the ability to formulate a simple relational query and can therefore be straightforwardly expressed in a language such as SQL.

In contrast, Query 2 is somewhat more complex. It requires:

- executing Query 1 (relational query)
- identifying other movies in which this actor has appeared (relational query), and finally
- executing a `FIND frames[5]` $\cdots$ query that finds frames from *each of the movies identified in the preceding step.* In other words, this requires the ability to *iteratively* find 5 frames each from each movie identified in step 2 above and concatenating the segments thus identified.

Query 3 is even more complex. It requires identifying pairs of actors and actresses who appear in two movies, and then iterating on this pair, finding frames where both appear together.

In the rest of this section, we will augment the VIQS query language so as to support all the above types of queries.

A *relation-coupled FR-query* takes a set as an argument and creates an answer presentation by evaluating each element of the set. Each element of the set is used to get a sequence of video frames using the algorithms described earlier in the paper. The general form of a relation-coupled query is shown below.

```
/* this command establishes a linkage between a variable  */
/* and set data, so in the subsequent query, the variable */
/* name is used to indicate the set data.                 */
SET set_var TO set_object_expression

/* set iteration operator where var is used to represent  */
/* each element in the set per iteration                  */
FOREACH(FORALL) var IN set_object(set_var)
    FIND frames
    FROM video_data_name    /* index searching operation  */
    WHERE condition_clause
```

The SET clause establishes a connection between a variable and a set of data items. It is like a variable declaration with an initialized value in a standard programming language. The variable will be used in the following queries. The set expression to which a `set_var` is initialized may be constructed by an SQL query. Another way to specify set elements is to explicitly enumerate them within a pair of set-braces.

Example: Consider the relations `actor` and `sex` described below.

Name	Movie	Role
James Stewart	Rope	Rupert
James Stewart	Rear Window	John
James Stewart	The Trouble with Harry	Donald
Farley Granger	Rope	Philip
Farley Granger	Rear Window	Douglas
John Dall	Rope	Brandon
John Dall	The Trouble with Harry	Ed
Joan Chandler	Rope	Janet
Joan Chandler	Rear Window	Anne
Constance Collier	Rope	Mrs. Atwater
Constance Collier	The Trouble with Harry	Mrs. Atwater

`actor`

Name	Sex
James Stewart	male
Farley Granger	male
John Dall	male
Joan Chandler	female
Constance Collier	female

`sex`

Suppose we wish to find all actors/actresses who have acted in The Rope and place the result in a variable X. The `SET` statement may then be used in the following way.

```
SET X TO (SELECT Name
          FROM   actor
          WHERE  Movie = Rope)
```

The `SET` construct is relatively simple and has only been included for pedagogical completeness. On the other hand, the constructs FOREACH and FORALL are set-iterating constructs which execute the subsequent query for each element in the set. The set is either a set variable declared in the SET clause or a set object itself specifed directly. The difference between them is the way they compose the answer presentation. FOREACH construct essentially computes the union of all the frame sequences returned, whereas the FORALL constructs their intersection.

Example (Query 2): Let us examine query 2. The FOREACH construct may be used to compose a solution to this query as follows.

```
SET MOVIES TO ( (SELECT A.Movie
                 FROM actor A, actor B
                 WHERE A.Name = B.name AND
                       B.Movie = 'Rope'  AND
                       B.Role = 'Rupert')
                 MINUS
                (SELECT DISTINCT Movie
                 FROM actor
                 WHERE Movie = 'Rope') )

FOREACH X IN MOVIES
   FIND frames [5]
   FROM X
```

Example (Query 3): In a similar vein, Query 3 may be expressed as follows:

```
SET ACTOR  TO ( (SELECT A.Name
                 FROM actor A, sex S
                 WHERE S.Sex = 'male' AND
                       S.Name = A.Name AND
                       A.Movie = 'Rope')
                 INTERSECT
                (SELECT A.Name
                 FROM actor A, sex S
                 WHERE S.Sex = 'male' AND
                       S.Name = A.Name AND
                       A.Movie = 'Rear Window' ) )

 SET ACTRESS TO ( (SELECT A.Name
                   FROM actor A, sex S
                   WHERE S.Sex = 'female' AND
                         S.Name = A.Name AND
                         A.Movie = 'Rope')
                   INTERSECT
                  (SELECT A.NAME
                   FROM actor A, sex S
                   WHERE S.Sex = 'female' AND
                         S.Name = A.Name AND
                         A.Movie = 'Rear Window' ) )


 FOREACH X IN ACTORS
    FOREACH Y IN ACTRESS
       FIND frames [5]
       FROM Rear Window
       WHERE obj has X AND obj has Y
```

We now characterize the meaning of the `FOREACH` and `FORALL` constructs in much greater detail. We assume that we have a set $\{e_1, e_2, \ldots, e_n\}$ specified in the construct. For each element e_i, the index searching operation returns a presentation, S_i, of a set of frame sequences as the result. At this stage, the FOREACH query returns an answer presentation consisting of $\bigcup_{i=1}^{n} S_i$. In contrast, the FORALL query returns the answer presentation $\bigcap_{i=1}^{n} S_i$.

5.1 Foreach relation-coupled query

This is a query of the form: "Given a set of objects, find a sequence of video frames where at least one of the set elements appears. " In general, this query can be expressed in the form

```
SET VAR TO set_valued_expression
FOREACH X in VAR
   FIND frames
   FROM video
   WHERE conditions.
```

Example: Consider the query "Find videos, from "The Rope", of all people who have acted in both "The Rope" and "Rear Window"."

```
SET SOL TO ((SELECT Name
             FROM   actor
             WHERE  Movie = Rope)
            INTERSECT
            (SELECT Name
             FROM   actor
             WHERE  Movie = Rear Window)).

FOREACH X IN SOL
   FIND frames
   FROM Rope
   WHERE obj has X.
```

Method: The query can be solved by first executing the database query contained in the `SET` clause on the `actor` relation defined earlier on in the paper. Then store the result in a temporary file named `SOL`. Now, for each element in the `SOL` file, we execute the subsequent elementary query which returns a sequence of video frames. Finally, we combine those video frames using the union algorithm described earlier so as to generate an answer presentation. During the iteration, the variable connected to the set goes through the set, executes the subsequent query and unions the resulting video frames to the currently accumulated video frames.

Using SQL, we can handle more complicated queries. For example, the query shown below uses join operation.

Example: "Show video-clips, from "The Rope", of all female actors (i.e. actresses) who have acted in both "The Rope" and "Rear Window"." This query may be expressed as follows.

```
SET SOL1 TO (SELECT Name
             FROM   actors A, sex S
             WHERE  A.Name = S.name AND A S.sex = female)
            INTERSECT
            (SELECT Name
             FROM   actors
             WHERE  Movie = Rear Window)
```

```
            INTERSECT
            (SELECT Name
             FROM   actors
             WHERE  Movie = Rope).

FOREACH X IN SOL1
   FIND frames
   FROM Rope
   WHERE obj has X.
```

Method: The query can be solved by first executing the join operation on the `actors` and `sex` relations. The result is stored in a temporary file named `SOL1`. For each element in `SOL1`, execute the subsequent query composing a sequence of video frames as an answer presentation using the*union composition* algorithm described earlier.

5.2 Forall presentation query

This is a query of the form: "Given a set of objects, find all the video frames where all the objects in the set appear together."

Example: "Find all frames (if any) in the movie "The Trouble with Harry" where all actors who appeared in both "The Rope" and "Rear Window" appear together." This query may be expressed as follows, where `SOL` is as defined earlier.

```
FORALL X IN SOL
   FIND frames
   FROM The Trouble with Harry
   WHERE obj has X.
```

Method: For each element of `SOL`, execute the subsequent object query for the movie "The Trouble with Harry" to get a set of video frames where the element in `SOL` appears. Finally, we intersect all these sets of video frames (using the *intersection* composition algorithms described in this paper, to get a final answer presentation.

6 Implementation

All the algorithms described in this paper have been implemented in a prototype system called VIQS ("Video Querying System") at the University of Maryland. VIQS currently contains about 3500 lines of C code, which includes a parser for the language described in this paper, as well methods to compose a presentation. The system also includes a graphical user interface, constructed using the Tcl/Tk toolkit. The system currently runs on Unix workstations under X-Windows.

When the user invokes VIQS, a screen comes up (cf. Figure 6). The user first specifies the video database that he is interested in by using the `Database` button seen in Figure 6. He may then type in his query in the appropriate form in the specified windows. The query is executed when the user presses the `Go` button. Figure 6 shows a specific query requesting all frames of the movie "The Rope" where Brendon appears and there is a conversation going on. This query requires the use of the intersection-composition algorithms described earlier in the paper. Initially, the bottom window of Figure 6 ("Result of Query Execution") remains empty. However, after the query is executed, the presentation of the answer is listed in this bottom window. In this case, the answer to the query consists of several frame-sequences. Ten of these frame-sequences are shown in the bottom window – others may be viewed using the scroll-bar on the right. The user may click on any one of these frame sequences – when he does so, the relevant frame sequences are composed together into thumb-nail sketches and displayed to the user. Figure 6 shows the situation when the user has requested that the frame-sequence $[24, 30)$ be displayed.

Figure 7 shows VIQS working on a `FOREACH` query that uses, within a `FOREACH` statement, the result of a query similar to that in Figure 6 with some restrictions on where the frames may be selected from (i.e they should lie between frames 1 and 130). The query asks for all frame-sequences where either Rupert or Phillip is present and a conversation is going on.

Figure 8 shows the system working on a `FORALL` query. In effect, this query asks for all frame-sequences where Rupert, Phillip and Brendon and all present together. However the `FORALL` construct works as a loop, first finding the frame-sequences in which Rupert appears, then finding the frame-sequences in which Phillip appears, composing these two presentations together using the intersection-composition algorithm; the system then computes the frame-sequences where Brendon appears, and composes this presentation (using the intersection-composition algorithm) with the presentation composed earlier.

7 Related Work

Over the last couple of years, there has been a small, but noticeable, spurt of activity in the area of video databases. The primary aim of this paper is to develop techniques by which video may be organized and queried. Three works that are closely related are [10], [5] and [7],

Oomoto and Tanaka [10] have defined a video-based object oriented data model, OVID. They take pieces of video, identify meaningful features in them and link these features. They also outline a language called VideoSQL for querying such data. One of the key advantages of the VIQS query language is that it can allow the user to specify how many frames he would like to see in a presentation. In addition, the operations `FORALL` and `FOREACH` are new and allow the user to synthesize meaningful presentations. Finally, our methods of composing presentations are also new and novel.

Gibbs et. al. [5] study how stream-based temporal multimedia data may be modeled using object based methods. However, concepts such as roles and players, the distinction between activities and events, and the integration of such video systems with other traditional database systems are not addressed.

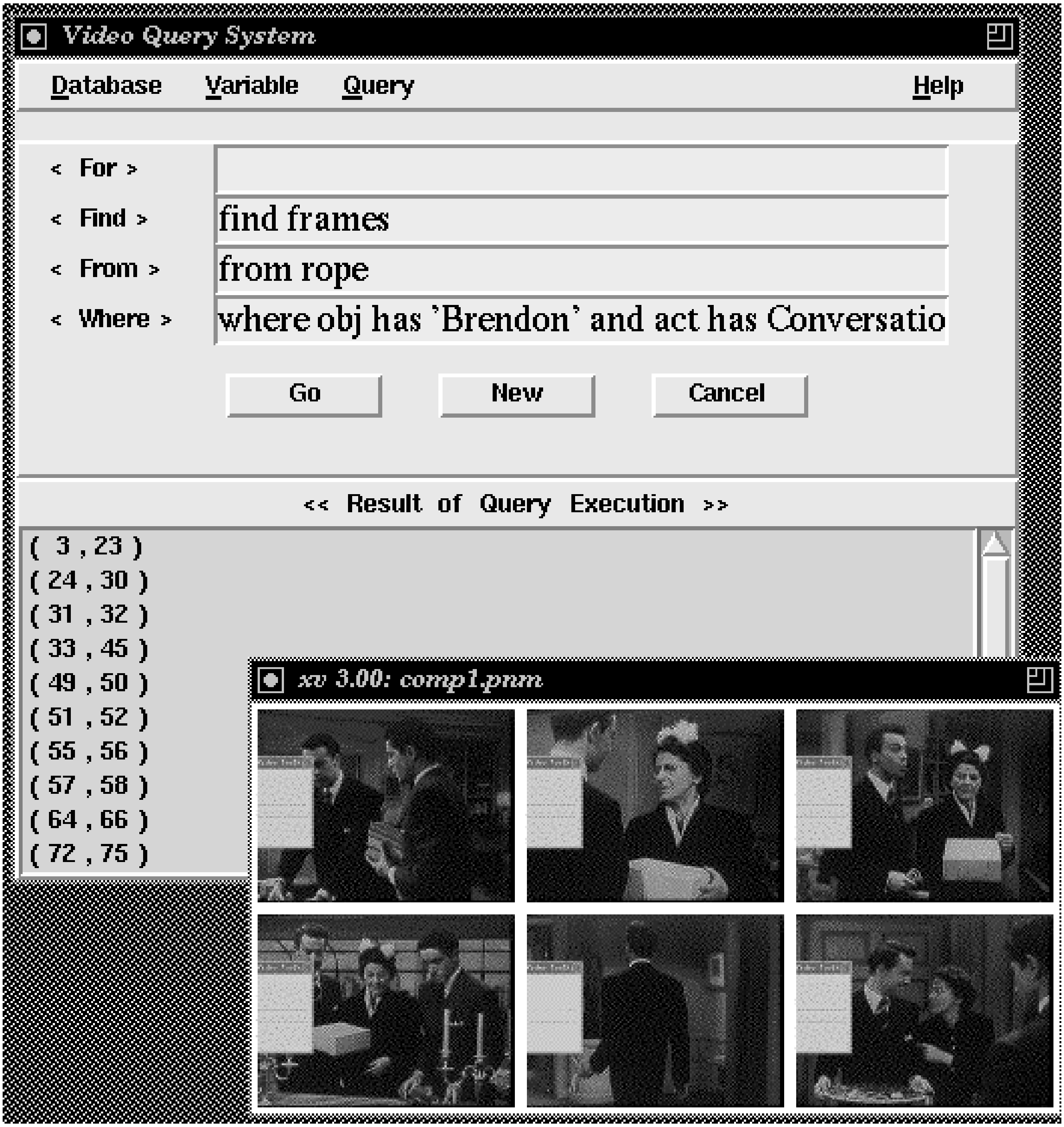

Figure 6: VIQS on a Simple Conjunctive Query

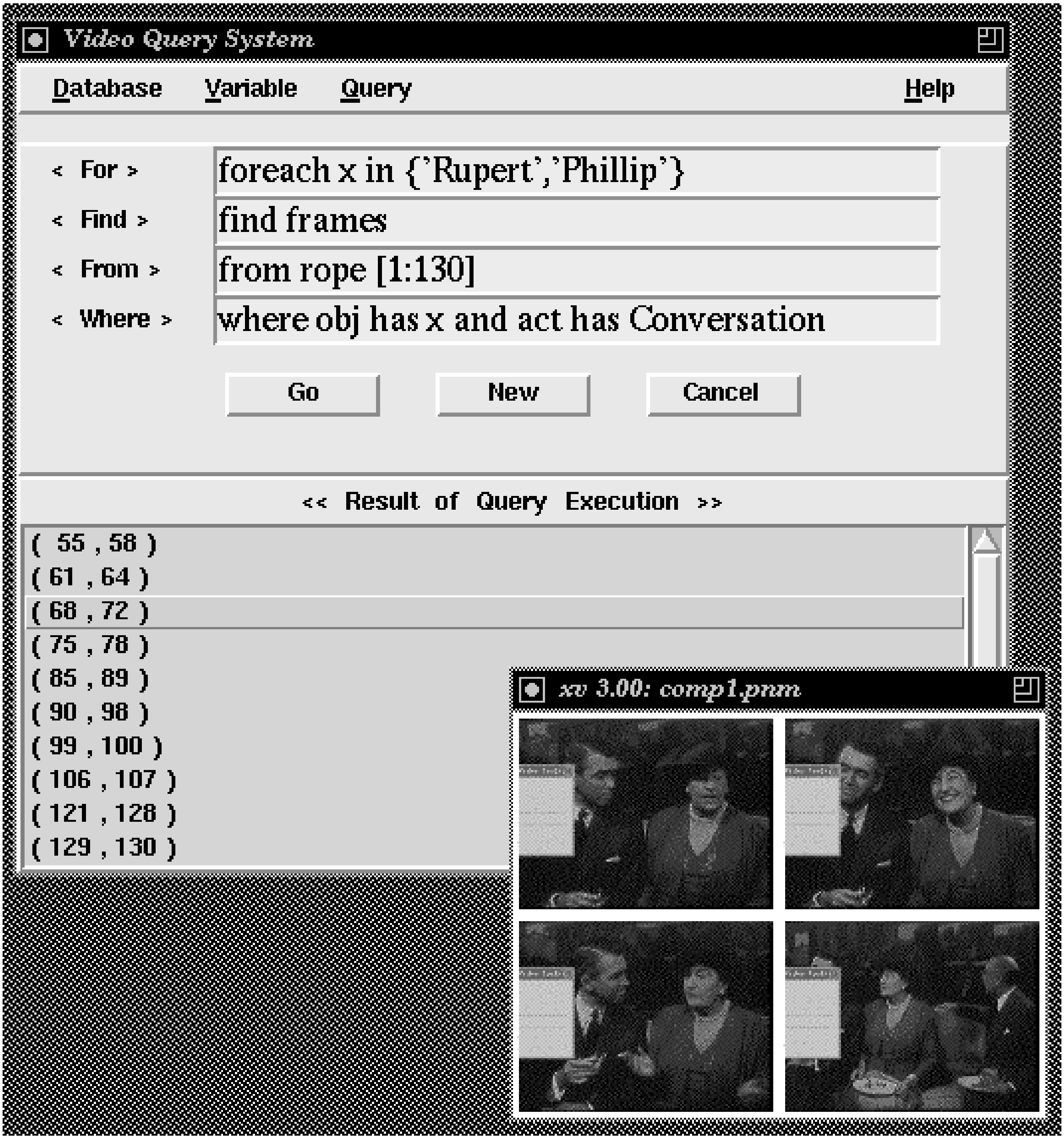

Figure 7: VIQS on a (Complex) Foreach Query

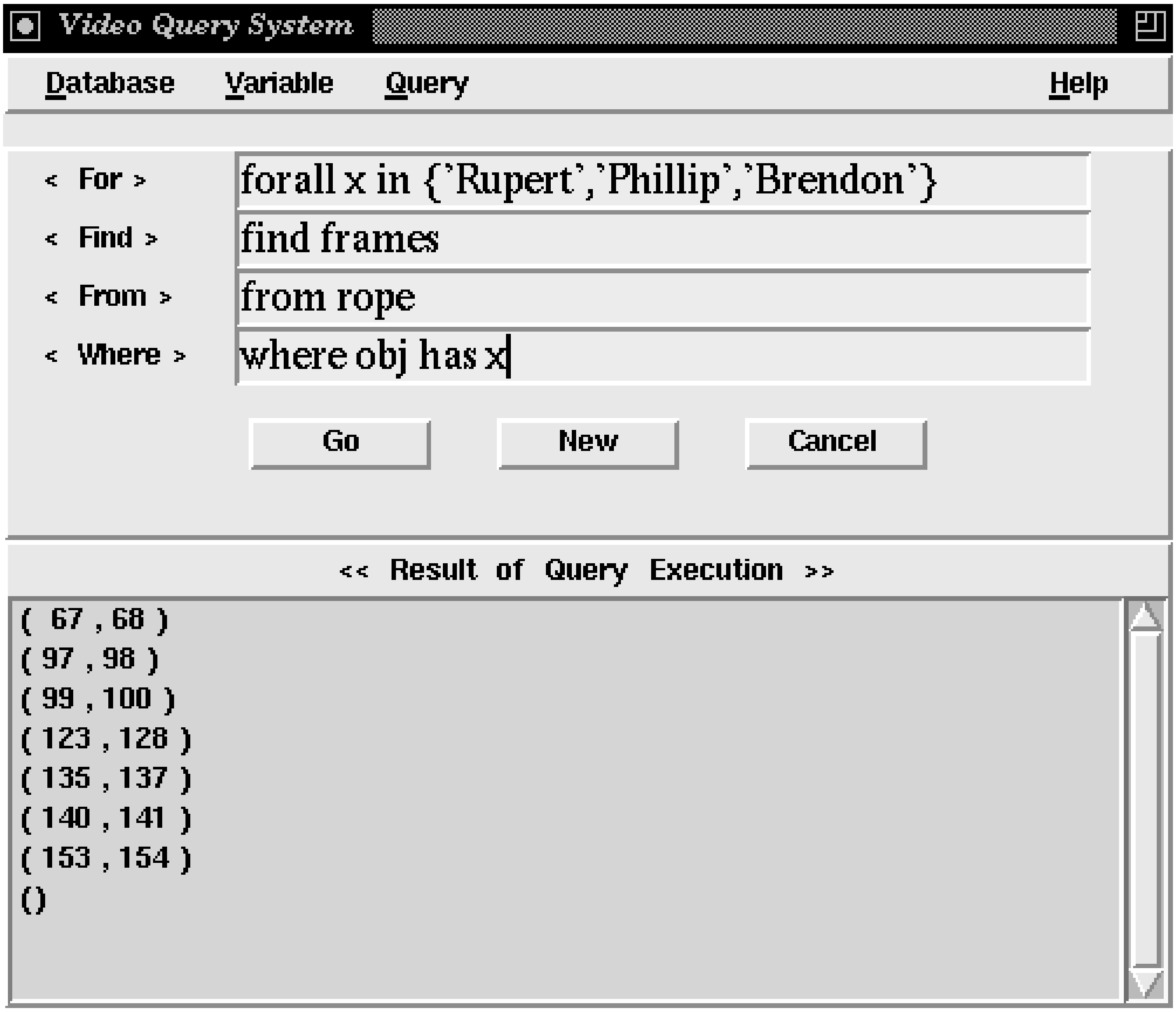

Figure 8: VIQS on a (Complex) Forall Query

Hjelsvold and Midtstraum [7] develop a "generic" data model for capturing video content and structure. Their idea is that video should be included as a data type in relational databases, i.e. systems such as PARADOX, INGRES, etc. should be augmented to handle video data. In particular, they study temporal queries. However, they have no way of composing video-presentations together, nor do they have any constructs similar to our iterative constructs. Additionally, one of the innovations in our approach is the use of well studied spatial (rather than temporal)' data structures, suitably modified, to query video data.

Arman et. al. [2] develop algorithms that can operate on *compressed* video directly – they can identify scene changes by performing certain computations on DCT coefficients in JPEG and MPEG encoded video. Their effort complements ours neatly in the following way: their algorithms can identify, from compressed video, frame sequences that are of interest, and the objects/roles/events of these frame sequences can be stored using the indexing structures of Adalı et. al.[1], and subsequently queried using the VIQS query language.

Other work on video includes work by Davenport et. al. [4] who argue that segmenting video should not be done at the frame level. This is consistent with our rendition – segmenting video at the frame level corresponds to a well-known data structure called the *unit* segment tree (cf. Samet [11]) which is just like the segment tree described here except that leaves always must represent unit intervals, i.e. intervals of the form $[i, i+1)$. In contrast, by using segment trees instead, we allow leaves to have whatever granularity is needed to best represent the content of the video being annotated.

8 Conclusions

With the advent of the information superhighway, there is now a spectacular amount of data available across computer networks. As the bandwidth of these wide area networks increases, a vast array of video data is likely to become accessible to authorized users. For example, museums and learned societies (e.g. National Geographic) possess large video library archives that one may reasonably expect to become publicly available not too long from now.

As such video data becomes more and more widely accessible, the need to efficiently index this data becomes more and more significant. In this paper, we have developed schemes that allow frame-segment based retrieval of large video databases. Davenport et. al. [4] have argued persuasively against the development of indexing schemes that index each and every frame of video; the reason for this is that many thousands of contiguous frames may often denote a single event of interest, and in such cases, repetitive representation of this data, once for each frame, is likely to lead to a tremendous waste of storage space. In this paper, we have proposed a frame-sequence based approach of storing video-data so that this problem is circumvented.

Additionally, we have proposed a high-level video-language that has several positive features: first, the language is an SQL-like language that is very easy to use. Second, the language allows users to retrieve video-segments from a video archive without worrying about low-level implementation details. Third, we have developed algorithms to implement this language that support *succinct, cohesive presentations* of video data based on queries expressed in our language. These algorithms take complex boolean queries and compose presentations together efficiently – all the algorithms described in this paper can be executed in linear time. Finally, we have provided special, intuitive language constructs (`FOREACH` and `FORALL`) that allows a user query to specify complex presentations based on iterative concatentations of elementary presentations computed within the iterative loop. Finally, the entire VIQS system based on the principles articulated in this paper has been implemented at the University of Maryland.

References

[1] S. Adali , K.S. Candan, S.-S. Chen, K. Erol and V.S. Subrahmanian. (1995) *AVIS: Advanced Video Information Systems*, submitted for journal publication. Also available via the WWW at http://www.cs.umd.edu//projects/hermes/publications/abstracts/avisdsqp.html.

[2] F. Arman, A. Hsu and M. Chiu. (1993) *Image Processing on Compressed Data for Large Video Databases*, First ACM Intl. Conf. on Multimedia, pps 267–272.

[3] A. Brink, S. Marcus and V.S. Subrahmanian. (1995) *Heterogeneous Multimedia Reasoning*, to appear in: IEEE Computer.

[4] G. Davenport, T.A. Smith and N. Pincever. (1991) *Cinematic Primitives for Multimedia*, IEEE Comp. Graphics and Applications, July 1991.

[5] S. Gibbs, C. Breiteneder and D. Tsichritzis. (1994) *Data Modeling of Time-Based Media*, Proc. 1994 ACM SIGMOD Conf. on Management of Data, pps 91–102.

[6] S. Gibbs and D. Tsichritzis. (1994) *Multimedia Programming: Objects, Environments and Frameworks*, ACM Press/Addison Wesley.

[7] R. Hjelsvold and R. Midtstraum. (1994) *Modeling and Querying Video Data*, Proc. 1994 Intl. Conf. on Very Large Databases, pps 686–694, Santiago, Chile.

[8] M. Iino, Y.F. Day and A. Ghafoor. (1994) *An Object-Oriented Model for Spatio-Temporal Synchronization of Multimedia Information*, Proc. 1994 Intl. Conf. on Multimedia Computing and Systems, pps 110-120, IEEE Press.

[9] S. Marcus and V.S. Subrahmanian. (1994) *Multimedia Database Systems*, to appear in: "Multimedia Databases: Research Issues and Directions" (eds. S. Jajodia and V.S. Subrahmanian), Springer-Verlag, to appear.

[10] E. Oomoto and K. Tanaka. (1993) *OVID: Design and Implementation of a Video-Object Database System*, IEEE Trans. on Knowledge and Data Engineering, 5, 4, pps 629–643.

[11] H. Samet. (1989) *The Design and Analysis of Spatial Data Structures*, Addison Wesley.

[12] R. Weiss, A. Duda and D.K. Gifford. (1994) *Content-Based Access to Algebraic Video*, Proc. 1994 Intl. Conf. on Multimedia Computing and Systems, pps 140–151, IEEE Press.