Video Query Processing in the VDBMS Testbed for Video Database Research

Walid Aref
Moustafa Hammad

Ann Christine Catlin
Ihab Ilyas
Thanaa Ghanem

Ahmed Elmagarmid
Mirette Marzouk

Purdue University
West Lafayette, IN 47906 USA

ABSTRACT
The increased use of video data sets for multimedia-based applications has created a demand for strong video database support, including efficient methods for handling the content-based query and retrieval of video data. Video query processing presents significant research challenges, mainly associated with the size, complexity and unstructured nature of video data. A video query processor must support video operations for search by content and streaming, new query types, and the incorporation of video methods and operators in generating, optimizing and executing query plans. In this paper, we address these query processing issues in two contexts, first as applied to the video data type and then as applied to the stream data type. We first present the query processing functionality of the VDBMS video database management system as a framework designed to support the full range of functionality for video as an abstract data type. We describe two query operators for the video data type which implement the rank-join and stop-after algorithms. As videos may be considered streams of consecutive image frames, video query processing can be expressed as continuous queries over video data streams. The stream data type was therefore introduced into the VDBMS system, and system functionality was extended to support general data streams. From this viewpoint, we present an approach for defining and processing streams, including video, through the query execution engine. We describe the implementation of several algorithms for video query processing expressed as continuous queries over video streams, such as fast forward, region-based blurring and left outer join. We include a description of the window-join algorithm as a core operator for continuous query systems, and discuss shared execution as an optimization approach for stream query processing.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *multimedia databases, query processing*.

General Terms
Algorithms, Management, Performance, Design.

Keywords
Continuous query, query processing, rank-join algorithm, stream processing, video database, window-join algorithm.

1. INTRODUCTION
The VDBMS video database management system was designed to support a full range of functionality for video as a well-defined abstract database data type, with the goal of providing video-based applications with all the powerful functionality generally provided by database management systems [1,2]. In particular, VDBMS supports query processing for content-based query, search and retrieval of video data. An efficient high-dimensional indexing mechanism was implemented in VDBMS to handle searching against video content, and new operators were defined for video query processing, such as nearest neighbor search and query by sample image for processing similarity queries. The VDBMS query processor has been designed to consider video methods and operators in generating, optimizing, and executing query plans. Additional supporting components were developed to complete the system, including a stream manager, query-based buffer management policies for effective real-time streaming [14], and pre-processing tools to generate visual features and other video metadata for the content representation used in video query processing [12]. The VDBMS framework ensures that video-based applications are provided with full video data processing functionality.

We have extended the VDBMS concept of the video data type (VDT) to handle general data streams, and the capabilities of VDBMS have been advanced to support a new VDBMS stream data type (SDT). This includes the development of a new stream manager to operate as an interface between outside stream-producing devices and internal processing, a StreamScan operator, operators for handling continuous queries, and support for multiple continuous query optimization and execution. The underlying framework for stream data processing incorporates novel stream management and query processing mechanisms to support the online acquisition, management, storage, non-blocking query, and integration of data stream sources. Requirements of our stream processing framework include a data-driven execution model (push and pull-based evaluation), support for both continuous and snapshot queries, admission control mechanisms, scalability in both number of streams and number of queries, prioritization of both streams and queries, maintenance of data stream summaries, and the ability to perform data mining over streams. Key components include the query processing interface for source streams, the stream manager, the stream buffer manager, non-blocking query execution and a new class of join algorithms for joining multiple data streams constrained by a sliding time window.

If we now define video data as a sequence of consecutive image frames, then video data can be viewed as a data stream, where each data item represents a single image frame. Video query processing can then be regarded as an example of stream query processing and we can express video streaming operations (such as the blurring of specific regions of frame content and fast forwarding) as continuous queries over data streams. From this viewpoint, video processing emerges as an application of stream data processing. Numerous complex operations over video data can be expressed as continuous queries over streams. In fact, with increasing research in online video analysis and online feature extraction, a wealth of information can be streamed in parallel with the stream of video frames.

There are some important advantages to expressing video processing operations as queries over data streams. These include: 1) space efficiency. Some video applications deliver processed forms of videos to users. Instead of storing multiple versions of videos, where each is customized to meet user-based restrictions, the user requirement can be expressed as a continuous query. The query is executed on one version of the video to produce an appropriately processed video for each user class. That is, query execution accesses a single stored version of the video, and the output of query processing is transferred to the user, 2) flexibility. Simple combinations of query operators produce different views of the same video and 3) scalability. Sharing of operations can be exploited so that multiple users can be supported at the same time.

In this paper, we address VDBMS video query processing in two contexts, first as applied to the *video data type* and then as applied to the *stream data type*. In section 2, we describe VDBMS query processing for the VDT and present VDBMS query operators that implement the rank-join and stop-after algorithms operating on video as a VDT. In section 3, we present the stream query processing framework of VDBMS. Section 4 presents three new VDBMS stream query processing algorithms that operate on video as an SDT, and Section 5 describes a new stream query operator for shared execution of window joins. Lastly we discuss shared execution, an optimization approach for stream query processing.

2. VDBMS QUERY PROCESSING FOR THE VIDEO DATA TYPE

2.1 The Query Processor

The VDBMS object relational database manager extends the Predator open source system [25], which has been modified extensively to provide full video query processing capability. The modifications and adaptations are based on the development and integration of video as a fundamental abstract database data type. Key extensions include high-dimensional indexing, video store and search operations, and new video query types. The extensions required major changes in many database system components since traditional methods for handling data retrieval cannot be easily extended to support the meaningful query processing and optimization of video, including online customized video views, content-based queries, video content control during streaming, and data abstraction.

VDBMS adopted the features approach in querying video by content. Visual and semantic descriptors that represent and index video content for searching are extracted during video pre-processing. The video, its indices and metadata descriptors are then stored in the database. The high-dimensional feature vectors generated by video pre-processing presented serious indexing and searching difficulties in the execution and optimization of feature-based queries [20], hence VDBMS incorporated the GiST [17,27] implementation of the SR-tree as the high-dimensional index [3,4], and modified the query-processing layer of Predator to access this index. The vector ADT was added for all feature fields, and an instance of the GiST SR-tree is used as the access path in feature matching queries. The multi-dimensional indexing structure manages the high-dimensional feature vectors that are produced by visual feature extraction and used in image similarity searches.

2.2 The Rank-Join Algorithm

Consider stored video metadata that describes low-level visual features such as color histogram, texture and edge orientation. The features are extracted for each video frame during pre-processing and stored in separate tables in the database. Each feature is then indexed using a high-dimensional index for faster query response. If a user is interested in the k video frames most similar to a given query image based on color, the database system should rank the frames according to their similarity to the color information extracted from the given image, and present only the k most similar frames to the user. The database system can use the high-dimensional index to perform an efficient nearest-neighbor search [21] and produce the nearest k neighbors. We call this simple ranking query a single-feature or a single-criteria ranking query, and no joins are required to answer the query. A database system supporting approximate matching merely ranks the tuples according to how nearly they match the query image.

A more complex similarity query occurs when a user is interested in finding the k most similar frames to a given query image based on both color and texture. In this case, the database system must obtain a global ranking of frames based on both color and texture similarities to the query image. We refer to this type of query as a multi-feature ranking query. Unlike for single feature ranking queries, it is not clear with multi-criteria ranking how the database system should combine the individual rankings of the individual criteria, even if the notion of approximate matching is supported [11,24]. In current database systems, the only way to evaluate the query in the previous example is as follows: First, the feature tables are joined on the tuple key attributes. Then, for each join result, the similarity between the tuple features and the query features are quantized and combined into one similarity score. Finally, the results are sorted on the computed combined score to produce the top-k results. Two expensive major operations are involved: joining the individual inputs and sorting the join results. When using traditional join operators to answer a ranking query, an execution plan with a blocking sorting operator on top of the join is unavoidable. If the inputs are large, the cost of this plan can be prohibitively expensive.

We have developed a practical, binary, pipelined rank-join query operator, NRA-RJ [19], which determines an output global ranking from the input ranked video streams based on a score function. Our algorithm extends Fagin's optimal aggregate ranking algorithm [11] by assuming no random access is available on the input streams. We created a new VDBMS query operator that encapsulates the rank-join algorithm in its GetNext() operation, and each call to GetNext() returns the next top element from the ranked inputs. The output of NRA-RJ thus serves as valid input to other operators in the query pipeline, supporting a hierarchy of join operations and integrating easily into the query processing engine of any database system. The incremental and pipelining properties of our aggregation algorithm are essential for practical use in real-world database engines, and our new operator will help in implementing this type of join in ordinary query plans.

The GetNext() operation is the core of the rank-join operator. The internal state information needed by the operator consists of a priority queue of objects encountered thus far, sorted on worst score in descending order. GetNext() is binary, although this restriction is merely practical, and the algorithm holds for more than two inputs. Our most significant modification to the original aggregate ranking algorithm is that we can handle ranges of scores, instead of requiring the inputs to have exact scores for each object. This modification allows for pipelining the algorithm. The modified algorithm first checks if another object can be reported from the priority queue without violating the stopping condition, and if not, moves deeper into the input streams to retrieve more objects. In each call to GetNext(), the current depth of the caller is passed to the operator. This extra information assures synchronization among the pipeline of NRA-RJ operators.

2.3 The Stop-After Algorithm

Because of its pipelined nature, NRA-RJ does not specify the number k of desired results, and we need a way to limit the output of the similarity queries. The number of reported answers to k in NRA-RJ is limited by applying the Stop-After query operator [9,10], which is implemented in VDBMS as a physical query operator Scan-Stop. This is a straightforward implementation of Stop-After, and appears on top of the query plan. The Scan-Stop does not perform any ordering on its input.

3. VDBMS QUERY PROCESSING FOR THE STREAM DATA TYPE

Since videos can be considered as long sequences of frames delivered over time, one can model video as a *stream* of frames. With this view of video data, a multitude of fine-grained and incremental video operations can be introduced. Whereas the offline and bulk processing of video is widely deployed to process stored videos, the incremental and frame-level processing of video would be advantageous in scenarios such as the following:

- Video is delivered on-line as an infinite stream where the responsiveness of video processing is important, as in the tracking of moving objects in surveillance applications.

- Storage space is limited and it is not feasible to keep multiple copies of the same video (the original video and the processed versions). In this case the processing of video upon request and streaming the resulting, or processed, video is considered a space-efficient approach. An example is delivery video based on different qualities of service [5].

In this section we describe a general model for data streams, and introduce video streams as an example application. We follow this by presenting stream query operations and their applications on video streams. Finally, we provide a brief description of the interface between query operations and the underlying streams

3.1 Stream Data Model and Stream Query Operations

We consider a stream to be an infinite sequence of data items, where items are appended to the sequence over time and items in the sequence are ordered by a timestamp. Accordingly, we model each stream data item as a tuple < *v, t*> where *v* is a value (or set of values) representing the data item content, and *t* is the time at which this item joined the stream. The data content *v* can be a single value, a vector of values or NULL, and each value can be a simple or composite data type. Time *t* is our ordering mechanism, and the time stamp is the sequence number implicitly attached to each new data item. The time stamp may be assigned to the data item at its source or at the query processor [26]. As an example application of the stream model, the single data item in a video stream can be defined as < *frame, t*>, where *frame* is an abstract data type representing frame content and *t* is the timestamp assigned at the query processor. Note that the frame data type includes different attributes such as FrameID, size, type (I, P or B frames for MPEG video), headers and binary content. Our model can easily integrate Frame-level physical features by storing a foreign key to the features table described in Section 2.1.

Some of the traditional SQL operations, such as selection and projection, have semantics similar to the relational model when applied to the processing of data streams. Selection operations select stream data items that satisfy a predicate condition (Boolean expression) much the same as selection in the relational model. Projection is also similar to its relational model equivalent, where a mapping function is repeatedly executed for each stream data item. These two operations are directly applicable to video processing when viewing video as streams of frames. As an example, a selection operation could select frames that satisfy the selection condition: “select I-Frames from the video stream” and a projection operation function LowResolution() could be applied to every frame to produce video streams with reduced (lower resolution) quality. This may be important for applications which stream video through network links with slow bandwidth.

The binary form of the join operation finds the correlated items in two data sources. For a binary join, a data item from one source (the outer) is compared against all data items in the other source (the inner) to produce the matching pairs. This definition is clearly applicable when the data sources are non-streams or if the inner-stream is a non-stream data source. For all stream data sources, iteration on all items on the inner stream is not possible since the stream is assumed to be infinite. Therefore, a restricted form of the join referred to as window-join [16] is used for joining two stream data sources. For the window-join, only part of the data stream (a window) is considered for the join. In this paper, we consider a sliding window join that is defined in terms of time units. However, other representations of window-join are also applicable, such as the landmark window and tuple-count window. In the video stream processing application domain, the following types of joins are applicable:

- Joining a video stream with a non-stream data source, for example when searching for matching frames between a video stream generated by a monitoring camera and a stored database of images.

- Window-join for two data streams, for example when tracking objects that appear in video data streams from two monitoring cameras. The objects are identified in each data stream and the maximum time for the object to travel through the monitoring devices defines an implicit time window for the join operation.

A special type of the join operation is the outer-join, where tuples from left, right or either streams are always produced as output, regardless of whether they satisfy the join condition.

3.2 A Stream Interface to Query Processing

We developed and integrated the abstract stream data type into the VDBMS video database system to represent source data types with streaming capability, and VDBMS was modified to accommodate stream processing. Any stream-type must provide interfaces for InitStream(), ReadStream(), and CloseStream(). In order to collect data from the streams and supply them to the query execution engine, we developed the stream manager as a new component. The stream manager registers new stream-access requests, retrieves data from the registered streams into its local buffers, and supplies data to be processed by the query execution engine. Running as a separate thread, the stream manager schedules the retrieval of tuples in a round robin fashion. To interface the query execution plan to the stream manager, we introduce a StreamScan operator to communicate with the stream manager and receive new tuples as they are collected by the stream manager. A similar operator to the StreamScan is also introduced in [8] and [22] for stream query interfaces.

4. VIDEO PROCESSING AS CONTINUOUS QUERIES OVER STREAMS

As described in the previous section, we define video as a stream of frames residing within the database. Video data is stored in a video stream table VideoStream with the schema (VideoID, Frame, Timestamp). The Frame attribute is a complex data type with additional attributes and manipulation functions. Frame attributes include the frame number FrameNum, the FrameType to identify the frame as I, P or B, and the frame binary data. A sample user-defined function to support streaming is PacketizeStream(), which augments a streamed frame with the necessary headers for final display.

The VideoStream table stores video as a stream data type (SDT). The SDT stores special information about the video, such as identifier, stored location, type, size, etc. At query execution time, the SDT generates a stream of frames that correspond to the stored version of the video. Only one view of the table exists at request time. It contains the SDT as a virtual table with the actual contents (tuples) of the data stream. The query communicates with a video driver to retrieve the video frame by frame and produce an appropriately processed video.

4.1 Fast-Forward

Fast forwarding a streaming video to the end-user is a simple example of video query processing expressed as a continuous query over the video data stream. In the following example query, the stream of frames for the video titled "HeartSurgery" is filtered by the selection predicate VS.frame.type = I_FRAME. Only I frames are streamed out of the query, with the result that the video is displayed to the end-user in fast forward mode:

```
SELECT VS.frame.PacketizeStream()              FROM
VideoStream VS                              WHERE
VS.frame.type = "I_FRAME" and
VS.VideoID = "HeartSurgery";
```

The next section describes an extremely useful and significantly more complex example of video processing which operates as a continuous query over a stream of video frame data.

4.2 Video Access Control during Streaming

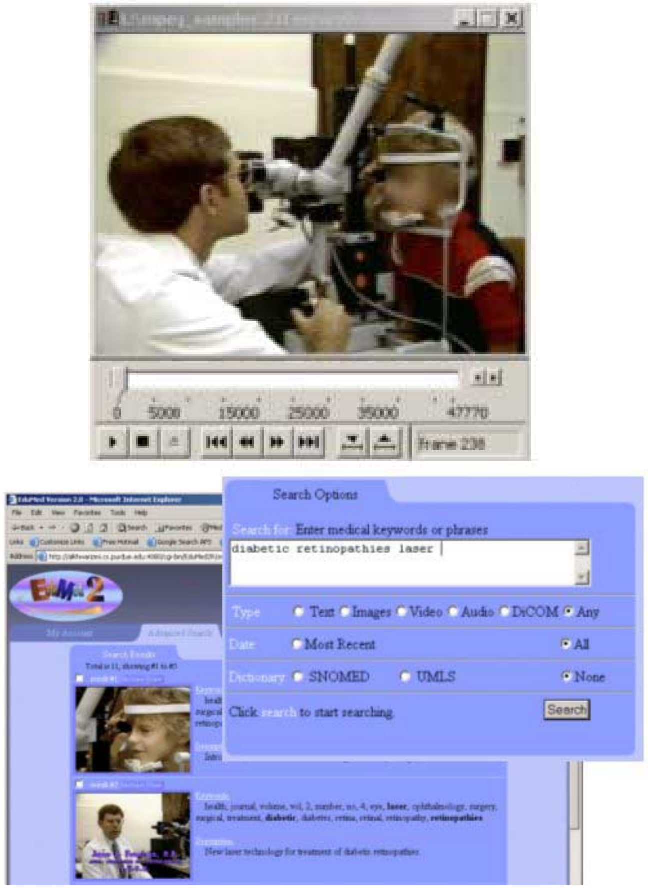
To provide customized views for a video according to user needs or specific access criteria, video stream operators can be used as access control mechanisms that apply real-time constraints on delivery video data streams [6]. We have developed an access control operator that hides areas of the video frame based on content during streaming. Query processing determines the authorized portions of each video frame that a user can receive, and alters the frames according to the user authorization and the video content description. A video-based application for medical education videos might use this mechanism to protect patient privacy, e.g., to blur the faces of patients during streaming to end-users who are not authorized to know the identity of patients. Granularity control is exercised over rectangular areas of a video frame which are associated with specific objects, such as a face or names and addresses.

In traditional databases, access control can easily be expressed in terms of a query (view) over the restricted tables. VDBMS follows this approach: when a user submits a query to retrieve a video containing the specified object, VDBMS will generate a continuous query to hide that object in all video frames by blurring the area in which the object appears. In the following query example, the ObjectTrajectory, defined by its appearance in the video frames, is determined beforehand and stored as minimum bounding rectangles (MBR), along with the frame number and video information. This information is stored in a relational table as (VideoID, FrameNum, MBR). The system submits the following query to stream the altered video stream to the user:

```
SELECT VS.frame.BlurFrame(OT.MBR)
FROM   VideoStream VS, ObjectTrajectory OT WHERE
VS.frame.FrameNum                      LEFT OUTER
JOIN OT.FrameNum and                   VS.VideoID =
"HeartSurgery" and              VS.VideoID =
OT.VideoID;
```

The query involves a Left Outer Join (described in the next section) between the video stream and the frames in the ObjectTrajectory. While streaming frames, if the current frame number is found in the ObjectTrajectory, the MBR of the frame is streamed along with the frame data to upper levels in the query pipeline. Other wise, a null MBR is streamed with this frame. The final projection of the query, BlurFrame(OT.MBR), blurs the frame if the MBR is not null. The result of the query is a streaming video where objects defined in the ObjectTrajectory are blurred.

An example application is shown in Figure 1. In the top image, a patient’s face is blurred during streaming since the user is not authorized to view it. The client interface to the VDBMS medical video library is shown in the bottom image. The client generates an access control query based on the user’s authorization level.



Search for: Enter medical keywords or phrases

```
diabetic retinopathies laser
```

Type: ○ Text ○ Images ○ Video ○ Audio ○ DiCOM [✓ Any]
Date: ○ Most Recent [✓ All]
Dictionary: ○ SNOMED ○ UMLS [✓ None]
Click search to start searching.

Keywords: health, journal, volume, vol, 2, number, no, 4, eye, **laser**, ophthalmology, surgery, surgical, treatment, **diabetic**, diabetes, retina, retinal, retinopathy, **retinopathies**

Description: New laser technology for treatment of diabetic retinopathies

Figure 1. Content-based access control for streaming video.

Fast forward can be combined with the access control mechanism to provide fast forwarding through a video for which some end-users have restricted access. The query can be expressed in SQL form as follows:

```
SELECT bVS.bframe.PacketizeStream()          FROM
(                                      SELECT
VS.frame.BlurFrame(OT.MBR) as bframe FROM
VideoStream VS, ObjectTrajectory OT WHERE
VS.frame.FrameNum                      LEFT OUTER
JOIN OT.FrameNum and         VS.VideoID =
"HeartSurgery" and              VS.VideoID =
OT.VideoID;                             ) AS bVS
WHERE  bVS.bframe.type = "I_FRAME";
```

Clearly, access control queries will never be generated by the end-user. Instead, a client interface which has incorporated the access control mechanism will construct the query for execution. In our example query, useful optimization techniques can be applied, such as pushing the selection predicate, bVS.frame.type = "I_FRAME", down in the query plan. In this case P and B frames would not be processed for blurring and the query execution time will be reduced. In plan (a) of Figure 2, the selection is pulled above the join and many tuples are unnecessarily joined. In plan (b) the selection is pushed down before the join so that unnecessary tuples are filtered out before the join.

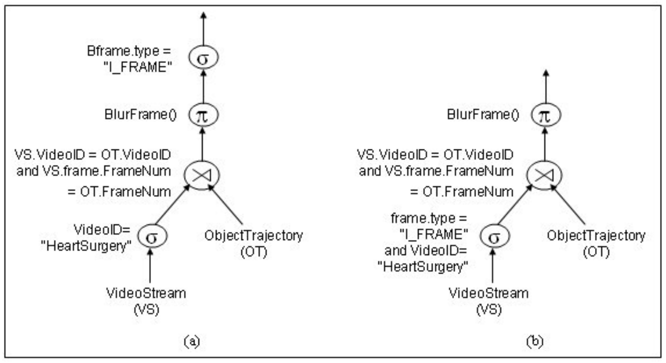

Bframe.type = "I_FRAME"
σ
BlurFrame()
π
VS.VideoID = OT.VideoID and VS.frame.FrameNum = OT.FrameNum
⋈
VideoID= "HeartSurgery"
σ
ObjectTrajectory (OT)
VideoStream (VS)
(a)
BlurFrame()
π
VS.VideoID = OT.VideoID and VS.frame.FrameNum = OT.FrameNum
⋈
frame.type = "I_FRAME" and VideoID= "HeartSurgery"
σ
ObjectTrajectory (OT)
VideoStream (VS)
(b)

Figure 2. Optimizing select operator placement.

4.3 Implementation of the Left Outer Join

We briefly describe our implementation of the left outer join algorithm. Query execution in VDBMS uses the "iterator model." A tuple iterator is set up on the highest plan operator, tuples are retrieved one-by-one, and each plan operator accesses its children by setting up iterators on them. A graph is created for the execution phase, and each node in this graph corresponds to a physical algorithm. The Left Outer Join is constructed as a node in the execution graph.

The Left Outer Join operator is implemented as a simple tuple nested loop join with specialized code that allows it to return additional output records. The Left Outer Join operator class contains information about the join which is kept alongside all GetNextRecord() function calls, including an index to the array that specifies from which relation (outer or inner) to get the next item and a counter that keeps the number of records from the inner relation that match the current outer relation tuple. GetNextRecord() takes a record from the outer relation, opens a scan on the inner relation, and passes through the inner relation record by record. When a record matches the join condition with the current outer record, the record is returned. After a scan on the inner relation is complete, the counter is checked. If it equals zero, a new record is constructed with the fields corresponding to the outer tuple extracted from the current outer tuple, along with the fields corresponding the inner tuple set to NULL, and this record is also returned.

5. THE WINDOW-JOIN

In this section we describe an algorithm for implementing the sliding window-join operation. As described in Section 3, the window-join (W-join) operation has many practical applications for the video stream model. An example SQL query that includes W-join is the tracking of objects that appear in multiple data streams from multiple cameras. For an object Obj that requires w time units to travel between two monitoring cameras, the query is posed[1]:

```
SELECT A.Obj                          FROM
Camera1 A, Camera2 B            WHERE similar
(A.Obj, B.Obj)          WINDOW w
```

[1] For tracking objects using more than two video cameras, the W-join can be expressed as a multi-way join between the streams from each video camera. Our W-join algorithm is presented as a multi-way W-join.

where similar() is a user-defined function that determines when two objects captured by different cameras are similar. This function can be considered as an equality predicate on object identifiers, A.Obj = B.Obj.

We identify four forms of the W-join: binary, path, graph, and clique (which include a special case referred to as uniform clique window join). In this paper, we focus on the uniform clique W-join defined as follows:

> *Given n data streams and a join condition (a Boolean expression on the tuples' values), find the tuples that satisfy the join condition and that are within a sliding time window of length w units from each other.*

We now present an algorithm for the backward evaluation form (BEW-join) of our W-join. The BEW-join provides low response time for slower input data streams. We have developed algorithms for the other forms of W-join as well, such as the path, graph and non-uniform clique-join [16].

We describe the BEW-join using an example W-join among five streams, A, B, C, D, and E, as shown in Figure 3. The five streams are joined together using a single window constraint of length *w* that applies between every two streams. For illustration, we assume that only the *black* dots (tuples) from each stream satisfy the join predicate in the WHERE clause: equality over objectID. The W-join maintains a buffer for each stream and we assume that the *vertical* bold arrow is currently pointing to the tuple about to be processed, e.g., Stream *A* is about to process the new tuple a2. The figure illustrates the positions of the tuples as they arrive over time, newer tuples are to the left of the stream and older tuples are to the right. There is no restriction that if the algorithm is processing a tuple t_{new} from one stream, then all tuples from the *other* streams that have earlier timestamps must have been processed. The order of scanning the streams is arbitrary and without loss of generality, we assume the order is: A, B, C, D, E, A, etc.

Figure 3 depicts the status of the algorithm as it begins to process tuple a2 from Stream A. First, a window of length 2w centered at a2 is formed. The algorithm iterates over all tuples of Stream B which are within the window of tuple a2. These tuples are shown inside the rectangle over B. b4 satisfies the join predicate and is located within the window of a2. The period is modified (reduced) to include a2, b4 and all tuples within w of both of them. This new period is used to test tuples in Stream C, and is shown as a rectangle over Stream C in Figure 3 (a).

The process of checking the join condition is repeated for tuples in C. Since tuple c3 satisfies the join predicate and also lies inside the rectangle, a new period is calculated that includes tuples a2, b4, c3 and all tuples that are within w of all of them. This period is shown as a rectangle over Stream D. In Stream D, d2 satisfies the join predicate and is located within the rectangle formed by a2, b4, c3. A new period is formed which includes the previous tuples and any further tuples within w of all of them. This period is shown as a rectangle over Stream E. The step is repeated for Stream E, and the 5-tuple, <a2, b4, c3, d2, e2> is reported as output. The algorithm recursively backtracks to consider other tuples in Streams D, then C and finally B. The final output 5-tuples in the iteration that starts with tuple a2 are: <a2, b4, c3, d2, e2>, <a2, b3, c3, d2, e2>, <a2, b3, c1, d2, e2>, <a2, b2, c3, d2, e2> and <a2, b2, c1, d2, e2>, respectively. While iterating over stream D, tuple d1 is located at distance more than w from all the last tuples in streams A, B, C, E, (tuples a2, b4, c3, and e2). Future tuples will all have timestamps greater than the timestamps associated with tuples a2, b4, c3, e2, thus tuple d1 can not be part of any future W-join. As a result tuple d1 can be safely dropped from the join buffer of stream D.

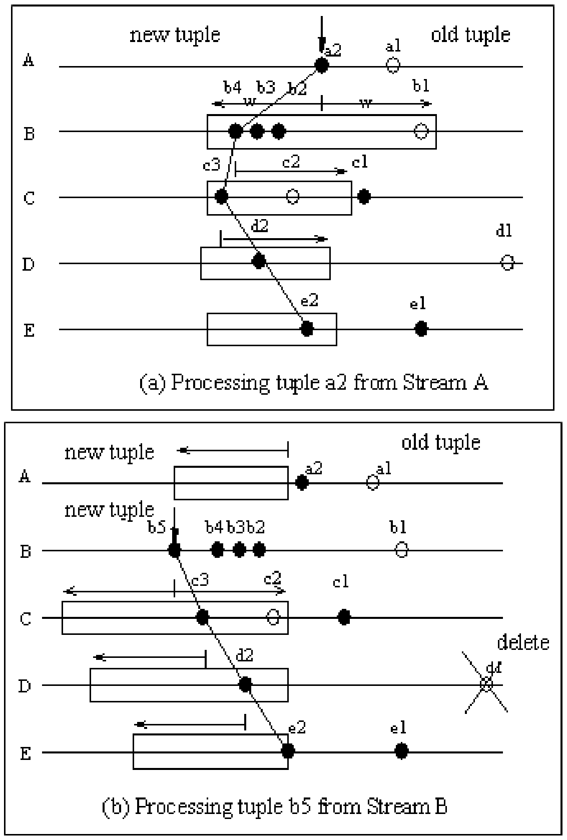


Figure 3. The BEW Join

After finishing with tuple a2, the algorithm starts a new iteration using a different new tuple (if one exists). In the example of Figure 3, we advance the pointer of Stream B to process tuple b5. This iteration is shown in 3(b), where periods over Streams C, D, E and A are constructed, respectively. This iteration produces no output, since no tuples join together in the constructed rectangles.

The algorithm never produces spurious duplicate tuples, since each iteration starts with a new tuple for the join (i.e., the newest tuple from a stream). Since, the output tuples of this iteration must include the new tuple, it is clear that duplicate tuples cannot be produced.

6. SHARED EXECUTION OF CONTINUOUS QUERIES

Centralized stream processing system must be able to support hundreds of concurrent users posing continuous queries over data streams and consuming system resources for extended periods of time. Exploiting shared execution for these queries will significantly improve system scalability. This is especially important if the sharing is performed for expensive and commonly used operators. One example of such an operator is the window-join. However sharing of the window-join is not straightforward, especially if the queries are interested in different windows over the data streams. In

[15] we investigated different approaches to scheduling shared binary window-joins over data streams. We introduced two new scheduling approaches, the shortest-window-first and the maximum-query-throughout, for a shared window-join, and we compared their performance with the largest-window-only scheduling technique. The performance of the algorithms with respect to reduced response time is more prominent when the streams possess bursty arrival rates.

Although the algorithms for shared window-joins target general data streams, these problem are extremely important in the video streaming domain. Since video stream are usually encoded in a variable rate streams, traffic is likely to provide a bursty arrival of frames from a video stream. Furthermore, when the outcome of the video query is expected to be streamed as a new video, the basic assumption is that the underlying query operations should have low, almost real-time response time. Therefore, proposed algorithms for video operations should optimize the response time. Finally, for processing on-line feeds of video streams, concurrent queries are expected to share their resources, and shared execution is becoming increasingly important.

7. CONCLUSION

Video-based applications require strong video database support, including efficient methods for handling content-based query and retrieval of portions of video data. A video query processor should support video-based operations for search by content and streaming, new video query types, and the incorporation of video methods and operators in generating, optimizing and executing query plans. In this paper, we described VDBMS database support for video query processing in two contexts: first as applied to the video data type and then as applied to the stream data type. The VDBMS query capability was designed to support a full range of functionality for video processing, based on the development and integration of video as an abstract database data type (VDT). We described two query operators for the VDT which implement the rank-join and stop-after algorithms. We then considered video data as streams of consecutive image frames, and expressed video query processing as continuous queries over video data streams. The stream data type (SDT) was developed and integrated into VDBMS, and system functionality was extended to support general data streams. From this viewpoint, we presented an approach for defining and processing streams, including video, through the new VDBMS query execution engine. We described the implementation of several algorithms for SDT video query processing, such as fast forward, region-based blurring and left outer join, which were expressed as continuous queries over video streams. We also described the window-join algorithm and shared execution over data streams as core operations for continuous query systems.

8. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grants IIS-0093116, EIA-9972883, IIS-0209120 and by the NAVSEA/Naval Surface Warfare Center, Crane.

9. AUTHOR INFORMATION

Author phone numbers and email addresses are as follows: Walid Aref 765-494-1997, aref@cs.purdue.edu. Ann Christine Catlin 765-494-4465, acc@cs.purdue.edu. Ahmed Elmagarmid 765-494-1998, ake@cs.purdue.edu. Moustafa Hammad 765-494-4359, mhammad@cs.purdue.edu. Ihab Ilyas 765-496-6348, ilyas@cs.purdue.edu. Mirette Marzouk 765-494-6020, marzouk@cs.purdue.edu. Thanaa Ghanem 765-494-6020, ghanemtm@cs.purdue.edu.

10. REFERENCES

[1] Aref, W., Catlin, A.C., Elmagarmid, A., Fan, J., Hammad, M., Ilyas, I., Marzouk, M., and Zhu, X. A video database management system for advancing video database research. In *Proc. of the Int Workshop on Management Information Systems*. Nov 2002. Tempe, Arizona.

[2] Aref, W., Catlin, A.C., Elmagarmid, A., Fan, J., Guo, J., Hammad, M., Ilyas, I., Marzouk, M., Prabhakar, S., Rezgui, A., Teoh, S., Terzi, E., Tu, Y., Vakali, A. and Zhu, X. A distributed server for continuous media. In *Proc. of the 18th Int Conf on Data Engineering*. Feb 26-Mar 1 2002. San Jose, California.

[3] Beckmann, N., Kriegel, H., Schneider, R. and Seeger, B. The R* -tree: an efficient robust access method for points and rectangles. *SIGMOD Record, ACM Special Interest Group on Management of Data*, 19(2): pp. 322-331. 1990.

[4] Berchtold, S., Böhm, C., Jagadish, H., Kriegel, H-P. and Sander, J. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proc. of the 16th Int Conf on Data Engineering*. San Diego, CA. pp. 577-588. February 2000.

[5] Bertino, E., Elmagarmid, A. and Hacid, M-S. Quality of service in multimedia digital libraries. *SIGMOD Record*. 30(1), pp. 35-40, March 2003.

[6] Bertino, E., Hammad, M., Aref, W. and Elmagarmid, A. An access control model for video database systems. In *Proc. of the 9th Int Conf on Information and Knowledge Management*. pp. 336-343. Nov 2000.

[7] Bertino, E., Samarati. P. and S. Jajodia. An extended authorization model. *IEEE Trans. on Knowledge and Data Engineering*. 9(1). pp. 85-101. 1997.

[8] P. Bonnet , J. E. Gehrke and P. Seshadri. Towards Sensor Database Systems. In *Proc. of the 2nd Inter Conf on Mobile Data Management*. Jan 2001.

[9] Michael J. Carey and Donald Kossmann, On saying “Enough already!" in SQL, In *Proc. CK SIGMOD*. Tucson, Arizona. May 1997.

[10] Michael J. Carey and Donald Kossmann, Reducing the Braking Distance of an SQL Query Engine, *In Proc. CK’98 VLDB*. New York, August, 1998.

[11] Fagin, R., Lotem, A. and Naor, M. Optimal aggregation algorithms for middleware. In *Proc. PODS’01* Santa Barbara, CA. May 2001

[12] Fan, J., Aref, W., Elmagarmid, A., Hacid, M., Marzouk, M. and Zhu, X. Multiview: Multi-level video content representation and retrieval. *Journal of Electrical Imaging,* Vol. 10, No. 4, pp. 895-908, October 2001.

[13] Guntzer, U., Balke, W-T. and Kiessling, W. Optimizing multi-feature queries for image databases. In *Proc. Of 26th Int Conf On Very Large Databases*. Cairo, Egypt. pp. 419-428. September 10-14 2000.

[14] Hammad, M., Aref, W., and Elmagarmid, A. Search-based buffer management policies for streaming in continuous media. In *Proc. of the IEEE Int Conf on Multimedia and Expo.* Lausanne, Switzerland. August 26-29, 2002.

[15] Hammad, M., Franklin, M., Aref, W. and Elmagarmid. A. Scheduling for shared window joins over data streams. In *Proc. of the 29th Int Conf on Very Large Data Bases*. 2003

[16] Hammad, M., Aref, W. and Elmagarmid. A. Stream Window Join: Tracking Moving Objects in Sensor-Network Databases. In *Proc. of the 15th SSDBM Conf.* Jul 2003.

[17] Hellerstein, J., Naughton, J. and Pfeffer, A. Generalized search trees for database systems. In *Proc. of 21st Int Conf on Very Large Data Bases*. Zurich, Switzerland. September 11-15, 1995.

[18] Ilyas, I. and Aref, W. SP-GiST: An extensible database index for supporting space partitioning trees. *Journal of Intelligent System.* 17(2-3). pp. 215-235. 2001.

[19] Ilyas, I, Aref, W, and Elmagarmid, A. Joining ranked inputs in practice. In *Proc. of the 28th Int Conf on Very Large Data Bases.* Hong Kong, China. 2002.

[20] Ilyas, I. and Aref, W. An extensible index for spatial databases. In *Proc of the 13th Int Conf on Statistical and Scientific Databases*. Virginia. July 2001.

[21] Katayama, N. and Satoh, S. The SR-tree: An index structure for high dimensional nearest neighbor queries. *SIGMOD Record, ACM Special Interest Group on Management of Data*, 26(2). 1997.

[22] S. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. of the SIGMOD Conf.* 2003.

[23] Nepal, S., Ramakrishna, M. Query processing issues in image (multimedia) databases. In *Proc. of the 15th Int Conf on Data Engineering*. Sydney, Australia. pp. 22-29. March 23-26, 1999.

[24] Natsev, A., Chang, Y-C., Smith, J., Li, C-S. and Vitter, J. Supporting incremental join queries on ranked inputs. In *Proc. of 27th Int Conf on Very Large Data Bases*. Rome, Italy. 2001.

[25] Seshadri, P. Predator: A resource for database research. *SIGMOD Record.* 27(1). pp. 16-20. 1998.

[26] R. T. Snodgrass. Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann, 2000.

[27] Thomas, M., Carson, C., and Hellerstein, J. Creating a Customized Access Method for Blobworld, In *Proc of the 16th Int Conf on Data Engineering*. San Diego, CA, March 2000