Search and Discovery in Digitial Video Libraries *

W. Aref, A. Catlin, A. Elmagarmid, J. Fan,
M. Hammad, I. Ilyas, M. Marzouk and X. Zhu
Computer Sciences Department, Purdue University, West Lafayette, IN. 47906

February 1, 2002

Abstract

The most useful environments for advancing research and development in video databases are those that provide complete video database management, including (1) video preprocessing and meta-data generation, (2) video and meta-data storage management, (3) indexing and query processing, (4) buffer management, and (5) continuous media streaming. These environments support the entire process of investigating, implementing, analyzing and evaluating new techniques, thus identifying in a concrete way which techniques are truly practical and robust. In this paper we present a video database research initiative that began in June 2000, and has culminated in the successful development of a video database research platform that supports comprehensive and efficient database management capabilities for digital video libraries. The research issues involved in developing the basic system components are described, and the value of the system as a development environment is illustrated by a performance study addressing the query processing component.

1 Introduction

The proliferation of online video content confirms the value of digital video as a rich and powerful source of information. Video clips are increasingly included in news and education websites to add a spatial and temporal dimension to data. But the complexity, size and unstructured nature of digital video data guarantee that, however easily digital video can be added to enhance the presentation of information in a specific instance, the application of search and discovery to digital video databases as a means of information retrieval will remain a challenge. After a decade of research in video segmentation, analysis and indexing, no general mechanisms have been developed for producing meta-data that supports high-level content-based queries for a video with generic subject matter.

The Informedia Project [6] provides full-content search and retrieval for news and documentary videos, using combined speech recognition and image processing to transcribe, segment and index videos. Informedia has focused on news-based video content, and substantial subject-based training is required to achieve an acceptable degree of accuracy for the audio-to-text transcriptions. Virage [15] markets VideoLogger, a commercial product that uses speech and image processing to index videos and generate meta-data. Their emphasis is on web-delivery of streaming video; users enter keyword queries based on the meta-data, and Virage presents results as storyboards tied to time-stamped audio transcriptions. Virage does not handle storage of digital video; users

*This work was supported in part by the National Science Foundation under Grants IIS-0093116, EIA-9972883, IIS-9974255, EIA-9983249, the Department of the Navy under Grant N00164-00-C-0047, and the Purdue Research Foundation

store their videos outside of the system. The Photobook System [22] allows users to plug in their own content analysis procedures. WebSeek [27] builds several indexes for both images and video; indexing is based on visual features such as color, as well as non-visual features such as keywords and subject categories. VideoQ [4] supports visual features (color, texture, motion) and automatic video object segmentation/tracking for video content representation and indexing.

The most useful environments for advancing research and development in video databases are those that provide *complete* video database management, including (1) video preprocessing and meta-data generation, (2) video and meta-data storage management, (3) indexing and query processing, (4) buffer management, and (5) continuous media streaming. These *research platforms* support the entire process of investigating, implementing, analyzing and evaluating new techniques, thus identifying in a concrete way which techniques are truly practical and robust. This paper describes a video database research initiative that began in June 2000, and has culminated in the successful development of a video-enhanced database research platform that supports comprehensive and efficient database management capabilities for digital video libraries. The fundamental concept was to provide a full range of functionality for video as an intrinsic, well-defined database data type, with its own description, parameters and methods. A natural consequence is that the digital video itself must be stored within the database (not in flat files or tailored disk partitions), giving the system absolute control over every interaction with the video. This allows, for example, the secure access control of streaming video, where the video data is altered (for reasons of privacy, ownership privilege, etc.) as it streams from the database to the user [1].

The VDBMS system has been tested against more than 800 hours of medical videos. Our video tapes are obtained from Indiana University School of Medicine and cover topics in basic and continuing medical education. The medical videos are digitized, compressed into MPEG1 format, processed off-line by VDBMS to generate image and content-based meta-data, and then stored together with their meta-data in the VDBMS database. Since VDBMS search and discovery have focused on medical video data, we have developed some abstraction strategies for extracting semantic content that apply specifically to medical videos. This paper, however, describes components of the entire VDBMS system; both general and content-specific techniques are presented.

The paper begins with a functional description of the system, including the scope of the preprocessing mechanisms, query capabilities, and results presentation. We then describe the system architecture and discuss the research details of important components. The concluding section discusses the importance of VDBMS as a development environment.

2 A Functional View

Building a digital video library that supports search and discovery requires a representation and indexing structure for video data inside the database. Image- and content-based video processing is used to partition videos into meaningful segments of consecutive frames called shots. Shots are associated with visual features and/or semantic descriptors that are used to identify video content for query, search and retrieval.

A query interface should give users a window into the digital video library, allowing them to search and retrieve video shots, and hopefully discover something interesting. Many users are not familiar with the contents of the digitial library they are searching against, and they may not even know exactly what they are looking for. For example, consider a medical school faculty member who is preparing a lecture on "Laparoscopic Nephrectomy". In this case, an ideal shot for her lecture would demonstrate the use of a laparoscope during kidney mobilization and extraction. She accesses the VDBMS medical database without knowing what videos are stored there. Several

Figure 1: VDBMS query interface.

formulations for a query to locate the ideal shot would be very helpful:

1. Frame-level image similarity match on multiple features, including color, texture, motion, etc.
2. Shot-level multi-feature image similarity match, using feature aggregation across shot frames.
3. Progression through various layers of hierarchical video summaries, beginning with the highest layer, "Operation", and moving to lower layers for an increasing level of detail.
4. Keyword-based query against meta-data describing shot-level semantic content.

Items 1 and 2 are based on low-level visual features, however an optimal aggregation of the specified features might be sufficient to match the easily identifiable color, shape and texture of a human kidney. In item 1, the query is based on a similarity match between the user's "example image" of a kidney and "video frame images" from the database. In item 2, a shot-level query is used to identify those shots that contain a large number of kidney-like images, increasing the probability that the shot describes a prolonged activity involving this image. Items 3 and 4 are based on semantic features. All of these formulations are available through the VDBMS query interface. The preprocessing techniques and system components which support this functionality are described in detail in the next section; here a short discussion of the user query options and their relationship to VDBMS query processing is given.

Queries based on Visual Features. For an *image-based query*, users present an example image and query the database for images or shots "most similar" to the example based on any number and combination of visual features. Results can be frame-level (a collection of frame images with similar features) or shot-level (a collection of shots with similar aggregate features). Shot-level queries can

include camera motion features to match against, such as zoom, pan, tilt, and still. Shot-level queries can search against aggregate shot information obtained by processing (1) all frames in the shot, or (2) a set of of key frames that have been selected to identify shot content. In (2), the key frames have been selected and extracted during VDBMS preprocessing as an abstraction of shot content, using criteria that is camera-based, activity-based, or shot-based. All visual feature queries are based on results generated by image-based shot detection and image-processing techniques developed and implemented by the VDBMS research group.

Queries based on Semantic Features. For a *content-based query*, the user enters a keyword that is matched against the keyword and annotation meta-data associated with logical or content-based shots (more appropriately called scenes). Videos are partitioned into scenes and identified with text (keywords and annotations) semi-automatically, using a VDBMS visual editing tool that supports scene boundary identification and editing, along with keyword, annotation and representative key frame identification (one for each scene.) For browsing through medical videos, users can access the video content abstractions produced by the VDBMS hierarchical summarization process.

For all image- and content-based queries, the VDBMS query processor returns a ranked list of results, where the user determines the number of top-ranked results to receive. When the user requests shot-level results, a representative key frame for each shot is returned. The representative key frame is a single image that represents the shot content; this frame is selected during video preprocessing and is used only for representing shots returned as a result of shot-based queries. Users can choose to stream any or all of the shot results by clicking on the key frame.

Video Processing and the Query Interface. What research issues had to be addressed to create a system that could support such queries with an acceptable quality of service? To support the feature-based image queries, it was necessary for VDBMS to be able to perform real-time searches against the high-dimensional feature vectors resulting from the feature extraction process. Image information extracted from each frame of a video occupies vectors of cumulative dimension approaching 1000, and a half hour video contains more than 50,000 frames. The magnitude of this indexing information forced the implementation of an entirely new multi-dimensional indexing structure for image similarlity searches. The concept is based on the Generalized Search Tree (GiST) [29] implementation of the SR-tree as the indexing technique, where the nearest-neighbor (NN) search operator uses an index access path created by running an incremental NN search query on the SR-tree. New operators have been investigated and developed to support multi-feature queries, and these have been integrated into the query processing plan. The demands of continuous video streaming required a new approach to buffer management policies. Caching parts of media streams which may be referenced in the near future enhances streaming performance by reducing the number of references to disk storage and by minimizing the initial latency associated with the start of streaming. We have implemented an efficient buffer prefetching and replacement policy based on knowledge collected from the content-based search manager and the stream manager. To manage the enormous volumes of data associated with storing video data, tertiary storage such as tapes, CDs or optical disks becomes essential. We have currently integrated a CD jukebox into VDBMS, and access to this device is transparent to the user.

Finally, what about the video preprocessing techniques for extracting the representative meta-data that makes these queries possible? We have developed innovative algorithms for image-based shot detection, incorporating automatic threshold determination for selecting shot cut frames. We have applied MPEG7 standards for descriptors and description schemes, using the XML-like format to define semantic and image-based information that represents the video content. Our research

has produced novel image-processing techniques for extracting low-level visual features (color, texture, camera motion, object detection, contour-based temporal tracking. etc), including nearly all meta-data defined by MPEG7 as standard. Key frames are automatically detected and extracted, both as (1) a single frame to represent each shot for query processing and fast browsing, and (2) a collection of frames to support searches against key frame content. Hierarchical video summarization techniques have recently been introduced to the VDBMS system to support content queries. In addition, content-based text has been added to the medical video meta-data through the use of a visual scene-cut tool, which allows physicians to (1) identify and edit scene boundaries based on logical content, (2) select the key frames to reperesent the scenes, and (3) add text for keywords and annotations.

End users access the VDBMS query interface using the Windows-based client shown in Figure 1. The client connects to the VDBMS system that resides on a SUN Enterprise 450 with 4 UltraSparc-II processors running the SunOS 5.6 operating system. Users pose their query by choosing among the various options:

- Query Type: Keyword, Example Image, Motion Info or SQL statement. The SQL equivalent of the user's query is generated for all non-SQL queries.
- Query Level: Frame/Display Frame (user submits an image for matching against database frame images, and the matching frame images are returned as results), Frame/Display Shot (user submits an image for matching against agreggate features of shots from the database, and matching shots are returned as results – represented by their key frames), Shot/Display Shot (user submits a video shot, and the aggregate features of that shot are matched against agreggate features of shots from the database, matching shots are returned as results).
- Query Features: color histogram (YUV, HSV), color moment, texture edges, texture tamura, motion histogram, camera activity, etc. Users select any combination of features to match against.
- Ranking Distance: number of top-ranked results to return.

The results are presented on a sliding image "filmstrip", and users can scroll back and forth through the results using any step size. Clicking on the key frame image which represents a video shot, streams the shot directly from the database to the query interface media player.

3 A System View

The VDBMS system is built on top of an open source system consisting of Shore [25], the storage manager developed at the University of Wisconsin, and PREDATOR [24], the object relational database manager from Cornell University. The VDBMS research group investigated and developed advanced multimedia capabilities, and then integrated them into the Shore/PREDATOR base. Shore/PREDATOR was transformed from a traditional text-processing DBMS into a video database management system supporting full database functionality for the "video" as a fundamental database object. Approximately 20,000 lines of code were added to the original 144,400 lines of Shore code, and approximately 57,000 lines of code were added to the original 84,000 lines of PREDATOR code.

A diagram of the system architecture is shown in Figure 2. Components that have been added to the system include the Feature Extraction module, the Query GUI, the E-ADT Interface, the

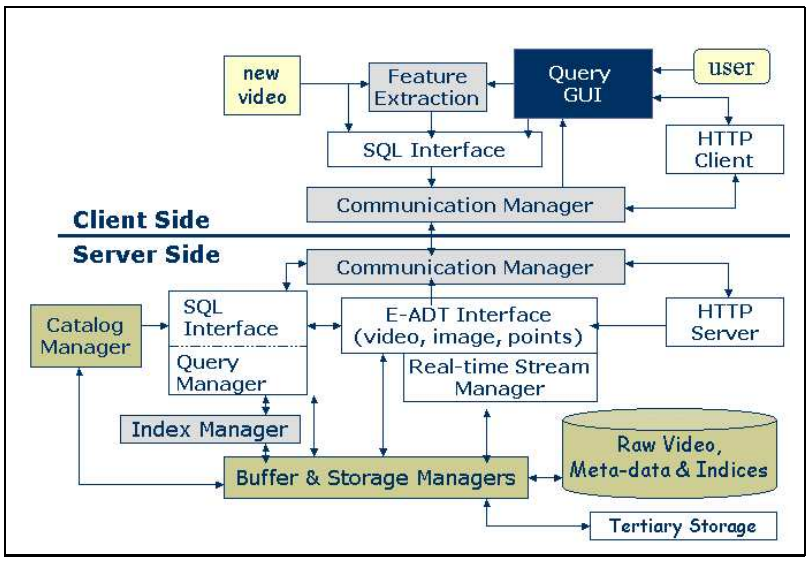


Figure 2: VDBMS system architecture

Stream Manager and the Tertiary Storage module. Components that have been modified extensively to provide video-enhanced capabilities are the Query Manager, the Index Manager, and the Buffer and Storage Managers. In the sections that follow, the algorithms and techniques that were used to develop the components of the VDBMS systems are described. Video preprocessing is treated first, as it produces the important representation and indexing information that identifies video content within the database. We also describe how MPEG7 standards are an integral part of VDBMS feature representation. We next treat the processing of user queries as it relates to (1) the implementation of new indexing schemes and (2) the investigation and development of a new query operator to handle complex multi-feature queries. The final discussion addresses our research in issues related to streaming, buffer management and storage. In particular, a stream manager layer has been implemented above the buffer manager to serve concurrent, rate-specific streams to clients and to guide the underlying buffer pre-fetching and replacement polcies. Disk and buffer management has been enhanced by the implementation of (1) different caching levels for accessing frequently referenced data, (2) segment allocation to replace the page-based approach, and (3) improved methods for managing video storage hierarchies that include buffer, disk, tertiary storage and the Internet.

3.1 Shot Detection and Key Frame Extraction

Partitioning a video into meaningful segments is the first step in creating the representation and indexing structure for each video inside the database. These segments, or shots, are the basic units for accessing, querying, browsing and retrieving video source (see Figure 3.) We need techniques that measure the "difference between consecutive frames" to identify where shots begin and end. These techniques can be based on pixels, statistics, transforms, features or histograms. In addition, since there is no magic number that identifies when successive frames are "different enough" to be considered members of different shots for *all* types of videos, the techniques must incorporate a mechanism for dynamic threshold determination. Such a mechanism can adapt automatically to different kinds of videos.

The VDBMS system uses color histogram differences between frames as its measure for shot

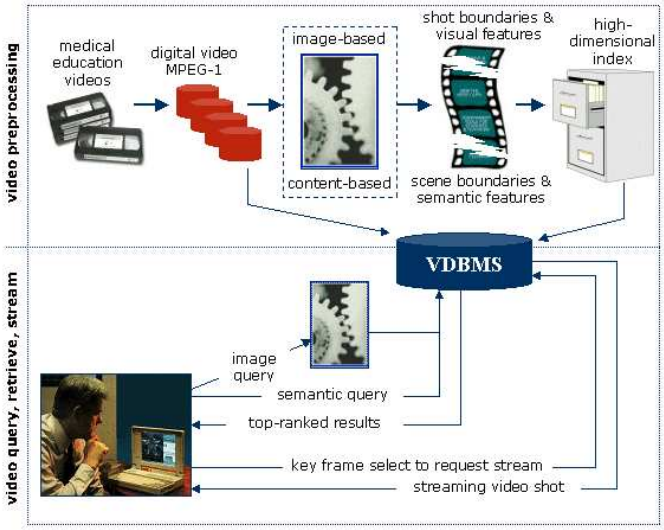


Figure 3: An overview of video preprocessing which creates the meta-data that represents and indexes video content inside the VDBMS database (top), and the end user query against the meta-data for search and discovery in the VDBMS digital video library (bottom).

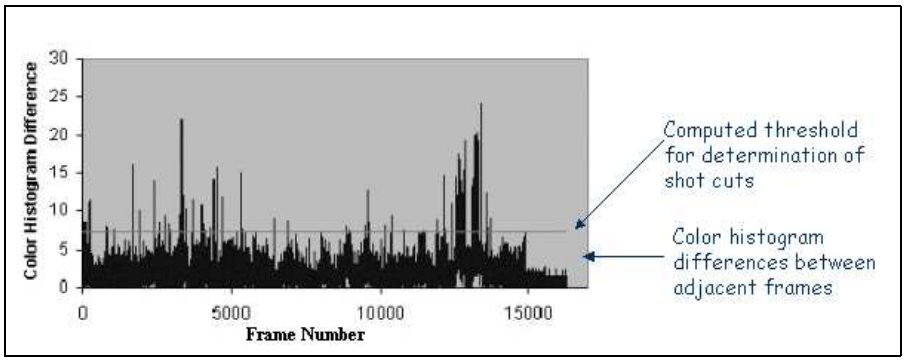


Figure 4: Pass 1 of video preprocessing computes the color histogram differences and the threshold value to use for identifying shot cuts.

cut detection. It is a two-pass process. The first pass calculates the frame differences according to the YUV values of the frame image, tracking information needed to compute the threshold value, as illustrated in Figure 4. The second pass uses the computed threshold and the frame difference values to partition the video into shots. During the second pass, the first frame of each shot (the shot cut frame) is extracted as the key frame to represent the shot for retrieval presentation. Additional key frames are extracted based on global camera motion activity and shot criteria for key frame-based queries. Finally the visual features that represent the shot content and index the video in the database are extracted from each frame.

The Shot Cut Detection Algorithm. The VDBMS shot cut detection algorithm first calculates the color histogram $H_j(k)$ for each video frame j using the YUV values of the frame image. The Y is divided into 8 intervals. U and V are divided into 2 intervals. This is mapped to a single vector

of dimension 32. The color histogram difference beween frames j and $j-1$ is computed as follows:

$$HD(j, j-1) = \sum_{k=0}^{M} \frac{[H_{j-1}(k) - H_j(k)]^2}{[H_{j-1}(k) + H_j(k)]^2} \tag{1}$$

where k is one of the M potential color levels. If $HD(j, j-1)$ is greater than an optimal threshold $\bar{T}_c$, the jth frame is selected as a shot cut. To determine $\bar{T}_c$, we first compute the probabilities for a *non-shot cut* and a *shot cut* as follows:

$$P_{nsc}(i) = \frac{f_i}{\sum_{h=0}^{T} f_h}, 0 \leq i \leq T \qquad P_{sc}(i) = \frac{f_i}{\sum_{h=T+1}^{M} f_h}, T+1 \leq i \leq M \tag{2}$$

where f_i denotes the number of frames for which the computed color histogram differences with the previous frames are equal to i and $\sum_{h=0}^{T} f_h$ represents the total number of frames for which the color histogram differences with their previous frames are in the range $0 \leq i \leq T$. Given the entropies for non-shot cut and shot cut frames, $H_{nsc}(T)$ and $H_{sc}(T)$, respectively, the optimal threshold is determined by maximizing the criteria function [2, 12] $H(\bar{T}_c) = \max \{H_{nsc}(T) + H_{sc}(T)\}$ over $T = 0, 1, \cdots, M$.

The search complexity for obtaining the optimal threshold $\bar{T}_c$ is of order $O(M^2)$ since $O(M)$ computations are required to obtain the two entropies for each element, and there are M potential elements. To reduce the search burden, VDBMS has implemented an efficient search algorithm which exploits the recursive iterations for calculating the probabilities and entropies, where the computational burden is introduced by repeated calculation of the normalized parts. The total numbers of pairs for non-shot cut parts, where the thresholds are set to T and T+1, respectively, are given by:

$$P_0(T) = \sum_{h=0}^{T} f_h \qquad P_0(T+1) = \sum_{h=0}^{T+1} f_h = P_0(T) + f_{T+1} \tag{3}$$

Thus, the recursive iteration for the corresponding entropy can be reduced to adding only the incremental part, resulting in a search burden of $O(M)$:

$$H_{nsc}(T+1) = \sum_{i=0}^{T+1} P_{nsc}(i) log P_{nsc}(i) = \sum_{i=0}^{T+1} \frac{f_i}{P_0(T+1)} log \frac{f_i}{P_0(T+1)}$$

$$H_{nsc}(T+1) = \frac{P_0(T)}{P_0(T+1)} H_{nsc}(T) - \frac{f_{T+1}}{P_0(T+1)} log \frac{f_{T+1}}{P_0(T+1)} - \frac{P_0(T)}{P_0(T+1)} log \frac{P_0(T)}{P_0(T+1)}$$

and similarly for $H_{sc}(T+1)$ using $P_1(T)$ and $P_1(T+1)$.

In the second pass of preprocessing, the frames in the video are partitioned into two classes on the basis of their color histogram differences and the optimal threshold $\bar{T}_c$: $HD(j, j-1) > \bar{T}_c$ for shot cut frames, and $HD(j, j-1) \leq \bar{T}_c$ for non-shot cut frames. The video frames between two successive shot cut frames are taken as one video shot. The visual features extraction process defined later in this section operates on the shots determined by this algorithm.

Key Frame Extraction. The first frame in each shot is the one that determines the shot cut, and this is chosen as the identifying frame for the shot. It is used for presenting shot results in response to user queries. On the other hand, the collection of key frames that represent the content abstract are extracted from each shot based on the following criteria: (1) shot-based: for each video shot, the beginning and ending frames are taken, (2) camera-based: global camera motion is an important indication of video content change; for a zoom motion, the beginnng and ending frames

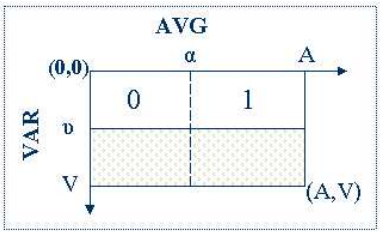


Figure 5: Scatterplot for 2D entropic thresholding.

of the zoom are selected; for a (left/right) pan or (up/down) tilt, the number of selected key frames depends on the strength of the camera motion, and (3) activity-based: moving objects are useful for identifying key frames. Camera and activity detection is based on structural video component segmentation.

Spatial Segmentation. Spacial segmentation is performed on each frame to extract the structural image components for a $4x4$ block resolution. We use the average gray level (AVG) and local variance (VAR) to indicate local detail and average properties of the frame image. In addition, we have developed a 2D entropic thresholding for determining the optimal segmentation thresholds. The AVG and VAR of block (m, n) are given by

$$AVG(m,n) = \frac{1}{16}\sum_{x=0}^{3}\sum_{y=0}^{3} I(x,y,t_n) \qquad VAR(m,n) = \frac{1}{16}\sum_{x=0}^{3}\sum_{y=0}^{3} |I(x,y,t_n) - AVG(m,n)|^2 \tag{5}$$

where $I(x, y, t_n)$ is the gray level of the pixel at position (x, y). We create a 2D scatter plot (AVG, VAR) as shown in Figure 5. There are $A \times V$ elements in the scatterplot, given that the gray level has A possible elements and the variance has V possible elements. Each element represents the occurences $f_{i,j}$ of the pair (AVG_i, VAR_j), that is, $f_{i,j}$ is the number of blocks in the image that have $AVG = i$ and $VAR = j$. Using two thresholds α and ν, the plot is partitioned into four quadrants. Since the areas (blocks) interior to homogeneous objects or backgrounds should contribute mainly to quadrants with relatively low local variance, and the edges and texture contribute mainly to relatively high local variance, this implies that quadrants 0 or 1 contain the distributions of homogeneous background and homogeneous objects. The remaining two quadrants (with high local variance) contain the distributions of blocks near domain edges and texture regions. The a priori probability of a pair (AVG_i, VAR_j) is given by $f_{i,j}$ divided by the total number of blocks in the image. Since the probabilities for homogeneous background P_B and homogeneous object P_O can be considered independent, we have:

$$P_{B_{i,j}}(\alpha,\nu) = f_{i,j}(\textstyle\sum_{h=0}^{\alpha}\sum_{k=0}^{\nu} f_{h,k})^{-1} \qquad P_{O_{i,j}}(\alpha,\nu) = f_{i,j}(\textstyle\sum_{h=\alpha+1}^{A}\sum_{k=0}^{\nu} f_{h,k})^{-1} \tag{6}$$

Given the entropies for the two classes, $H_B(\alpha, \nu)$ and $H_O(\alpha, \nu)$, the optimal threshold vector $(\bar{\alpha}, \bar{\nu})$ that is selected for performing the partition of the image components has to satisfy the following criteria functions: $H(\bar{\alpha}, \bar{\nu}) = \max\{H_B(\bar{\alpha}, \bar{\nu}) + H_O(\bar{\alpha}, \bar{\nu})\}$ over $\alpha = 0, 1, \cdots, A-1$ and $\nu = 0, 1, \cdots, V-1$. The computational complexity for finding the optimal maximum is $O(A^2V^2)$ [10]. A fast 2D entropic threshold technique can be used to reduce this to $O(2(A+V))$. The blocks can be classified into four groups on the basis of $(\bar{\alpha}, \bar{\nu})$: (1) homogeneous background, (2) homogeneous onjects, (3) textured background/objects, or (4) the edges between them.

Temporal Segmentation. Temporal segmentation is performed only on non-shot cut frames and determines the temporal relationships between frames for the extracted image components. The

frame difference contrast (FCON) of block (m, n) is given by

$$FCON(m,n) = \sum_{x=0}^{3} \sum_{y=0}^{3} |I(x,y,t_n) - I(x,y,t_{n-1})|^2 \tag{7}$$

where $I(x, y, t_n)$ is the gray level of the pixel at (x, y) in frame n. To compute the optimal temporal segmentation threshold $\bar{F}$, we set the probabilities for the temporal *unchanged* and *changed* regions, $P_{uc}(i)$ and $P_c(i)$, respectively, in the same manner as for the shot detection algorithm, and we use the entropy computations to arrive at the criteria function $H(\bar{F}) = \max \{ H_{uc}(F) + H_c(F)\}$ over $F = 0, 1, \cdots N$. The temporal relationships of the coordinate blocks among frames is then classified into two groups: $FCON(m, n) < \bar{F}$ for unchanged blocks, and $FCON(m, n) \geq \bar{F}$ for changed blocks. The efficient search algorithm described for shot detection can also be applied here to reduce the computational complexity to $O(N)$.

Structural Video Components. After the motion estimation procedure has been applied [11], the temporal and spatial segmentation results are integrated to determine the structural video components. This process is based on the following rules: (1) the uncovered background (high motion compensation errors) includes blocks that are detected as changed regions by temporal segmentation but are taken as one portion of the large background region by spatial segmentation, (2) the homogeneous moving objects (low motion compensation errors) consist of blocks that are detected as object regions by spatial segmentation, (3) the newly appearing objects are detected as changed regions by temporal segmentations, but they have high motion compensation errors because no correspondences were found for them in previous blocks, (4) texture background and texture moving objects have high motion compensation errors (as in newly appearing objects), but their motion vectors for the connected blocks are quite similar. Therefore, the changed regions can be further partioned into *uncovered background* (resulting from fast moving objects), *moving objects*, *newly appearing objects* or *newly appearing backgound* (resulting from global camera motion), and *texture stationary background* (resulting from camera jitter and moving edges). The unchanged regions can be partioned into *stationary background* and *still objects*. The activity-based key frame selection is determined automatically on the basis of these structural video components.

3.2 Visual Feature Extraction

Since the video shot is considered the basic database unit for digital video, preprocessing extracts a collection of visual feature values from each frame and associates them with their corresponding shots. The begin and end frames for each shot are identified in the database as shot boundaries, and for each shot we have $(VideoId, ShotId, VisualFeatureVector)$ as the database representation of the visual content of the shot. Users access, query and browse the video database to retrieve shots based on information in the $VisualFeatureVector$. We first give a complete inventory of the low-level frame and shot-based features extracted during VDBMS preprocessing. Following this inventory, we describe the process for extracting camera motion information.

Low-level Visual Features. The low-level features maintained in the $VisualFeatureVector$ are listed below. Features are extracted for each frame, and "per shot" average, minimum, standard deviation and variance values are based on the collection of values for all frames belonging to a shot. In Table 1, we show a portion of the database schema used to store the the video, video shot and frame features, including the GiST SR-tree index creation for the feature vectors. The $GSR\ INDEX$ is discussed in Section 3.5 .

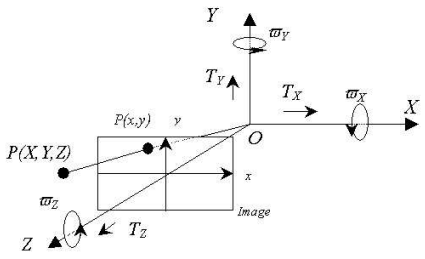


Figure 6: VDBMS camera model.

- Color Histogram HSV format: color content of frame images using 256 color levels.
- Color Histogram YUV format: spatial distribution of colors and content in a vector of dimension 32.
- Color Layout (MPEG7 standard): color content and structure; includes information about the spatial relationships of the pixels and can be used to distinguish between two images with identical color content but different color structure. Frame images are divided into 16 blocks with color histograms computed for each block, resulting in a vector of dimension 16×32.
- Color Moment: mean and variance of the luminance (gray level) component.
- Texture Tamura: each frame image is divided into 16 blocks and the average and standard deviation of the luminance component are computed for each block.
- Texture Edges: detects image pixels which are edge points by computing the direction, $1 \leq d \leq 9$, at each pixel. A directional histogram is created from the percentages of pixels in each direction.
- Motion Histogram: based on 8×8 blocks. The direction, $1 \leq d \leq 9$, of motion for each block is computed. A directional histogram is created from the percentages of blocks in each direction.

New MPEG7 visual features have recently been introduced (see http://www.cselt.it/mpeg) and these have been added to the VDBMS feature extraction process:

- Dominant Color in YUV: for each of 8 dominant colors, percentage and YUV components, in descending order of percentage.
- Scalable Color: color histogram is encoded using the Haar transform; this allows queries with histograms of larger dimension than those stored in the database.
- Homogeneous Texture: characterizes the region texture of an image using the energy and energy deviation in a set of frequency channels.
- Edge Histogram: spatial distribution of five types of edges in local image regions (horizontal, vertical, diagonal 45 degrees, diagonal 135 degrees and non-directional). Edge directions are computed as in Texture Edges. Images are divided into 16 blocks, and the direction of the 4×4 sub-blocks (i.e., dominant pixel direction) is computed for all 16 blocks.

```
CREATE TABLE VideoStream
(
VideoId int,
Movie video("format mpeg"),
Annotation string,
AnnotatorId int,
)
KEY (VideoId);
```

```
CREATE TABLE VideoShot
(
VideoId int,
ShotId int,
Annotation string,
AnnotatorId int,
StartFrame int,
EndFrame int,
KeyFrame image("format jpeg")
);
```

```
CREATE TABLE FrameFeatures
(
VideoId int,
ShotId int,
FrameId int,
CH vector ("1000"), // dim 32
CH2 vector ("1000"), // dim 256
CL vector ("1000"), // dim 512
CMMEAN vector ("1000"),// dim 1
CMVAR vector ("1000"), // dim 1
TTMEAN vector ("1000"), // dim 16
TTVAR vector ("1000"), // dim 16
TE vector ("1000"), // dim 9
MH vector ("1000"), // dim 9
);
```

```
CREATE GSR INDEX FFea_CH ON FrameFeatures(FrameFeatures.CH);
CREATE GSR INDEX FFea_CMMEAN ON FrameFeatures(FrameFeatures.CMMEAN);
CREATE GSR INDEX FFea_TTMEAN ON FrameFeatures(FrameFeatures.TTMEAN);
```

Table 1: Partial VDBMS database schema.

The Camera Motion Detection Algorithms. We use motion vectors extracted directly from MPEG1 streams to identify specific types of camera motion between frames. We first use an efficient method to eliminate the smooth areas of a frame image where motion vectors cannot be used. Then we apply qualitative classification algorithms to distinquish between tracking, panning, tilt, zoom, rotation, and still. The camera model used in our algorithm is shown in Figure 6. Any point $P(X,Y,Z)$ in a 3D scene is mapped to a point $p(i,j)$ in a 2D image. If f represents the focal length of the camera and we apply projection rules, the points P and p are related by $x = f(\frac{X}{Z})$ and $y = f(\frac{Y}{Z})$. According to the camera model, any motion of the point $P(X,Y,Z)$ can be described as $V = T \times \Omega \times P$, where $V = (V_X, V_Y, V_Z)^t$ is the velocity of point P in a 3D scene, $T = (T_X, T_Y, T_Z)^t$ is the parallel component of the velocities, and $\Omega = (\Omega_X, \Omega_Y, \Omega_Z)^t$ is the rotation velocity of point P. For any point p in the image, the optical flow is $OF(x,y) = (u(x,y), v(x,y))^t$, where $u(x,y) = \frac{d_x}{d_t}$ and $v(x,y) = \frac{d_y}{d_t}$. Taking into consideration the zooming motion of the camera, the optical flow of each point p can be represented as a linear combination of different kinds of camera motion:

$$u(x,y) = \frac{xy}{f}\Omega_X + (f + \frac{x^2}{f})\Omega_Y - y\Omega_Z + \frac{f}{Z}T_X - \frac{x}{Z}T_Z - f[arctan(\frac{x}{f})](1 + \frac{x^2}{f^2})r_{zoom} \quad (8)$$

$$v(x,y) = (f + \frac{y^2}{f})\Omega_X + \frac{xy}{f}\Omega_Y + x\Omega_Z - \frac{f}{Z}T_Y - \frac{f}{Z}T_Z - f[arctan(\frac{x}{f})](1 + \frac{y^2}{f^2})r_{zoom} \quad (9)$$

The above equations imply that if there is a dominant camera motion between frames, the optical flow will have different modes for each kind of camera motion. A qualitative camera motion strategy can therefore be used to distinquish between different patterns of optical flow in each frame.

We first extract the motion vectors from the MPEG1 video stream. Since there may be frame areas that are either smooth or contain a very large object (rendering the associated motion vector

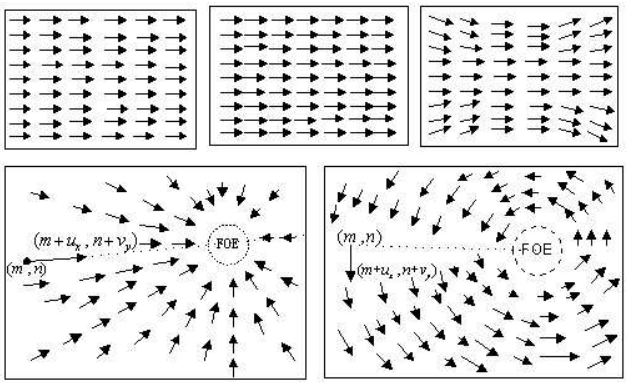


Figure 7: Motion vectors for tracking (top left), small panning (top middle), large panning (top right), zooming (bottom left) and rotation (bottom right).

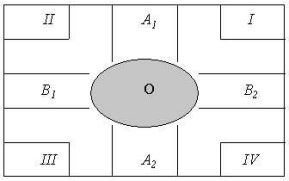


Figure 8: Frame image partition for evaluating motion vectors.

useless) we eliminate these areas. We split each frame into mn blocks (B_{mn}). If $\bar{V}_{ij}$ represents the motion vector of point $p(i,j)$ then the mean of each block is calculated as $E[\bar{V}_{mn}] = \frac{1}{K}\sum_{i,j\in B_{mn}} \hat{V}_{ij}$, where K is the number of motion vectors in block B_{mn}. Since motion vectors in a small area should be similar both in direction and length, we eliminate block B_{mn} if (1) the mean $E[\bar{V}_{mn}]$ is smaller than a given threshold, since either the energy of the block is too small or the directions of the motion vectors in the block differ significantly, or (2) the covariance $E[\bar{V}^2_{mn}]$ is bigger than a given threshold. The remaining motion vectors are those with relatively higher energy and uniform direction in small regions.

Motion vectors can estimate camera motion, but they may vary with the view angle used to capture the video. The tracking operation shown in Figure 7 (top left) is the same for large and small view angles. However, motion vectors vary with respect to view angle for panning operations (top middle and right). In order to classify camera motion accurately, we split the motion vector of each frame image into 9 areas as shown in Figure 8.

Fixed Operation. A fixed operation implies that the camera is still. For any motion vector, $\hat{V}_{ij} = (u_x, v_y)$ for point $p(x,y)$, the energy is computed as $\|\hat{V}_{ij}\| = \sqrt{u_x^2 + v_y^2}$. If $\hat{V}_{ij}$ is smaller than a given threshold, it is a small energy vector. If more than half of the motion vectors in a given frame are small, the motion is treated as fixed.

Pan and Tilt Operations. Panning, tilting, and vertical/horizontal tracking are treated as vertical/horizontal operations. Compared to other motions, these have a dominant motion direction, and thus, most of the vector directions in a frame with one of these operations will be the same. Our process to classify the vertical/horizontal motions is as follows:

1. Calculate the directions of all motion vectors and use a direction histogram to determine the

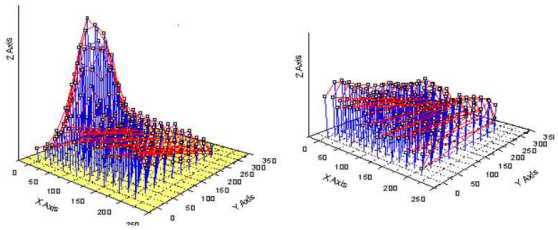


Figure 9: $W(i,j)$ distribution for zooming (left) and for rotation (right).

dominant direction (i.e., the direction representing the greatest number of vectors).

2. If more than half of the vectors are in the dominant direction, the motion of the frame is a vertical or horizontal operation. The degree of this direction will distinquish vertical from horizontal motion. If half or less of the vectors have a dominant direction, a different process will be used to evaluate the motion.

3. Recall that the view angle has an influence on the precision of camera motion extraction. For panning operations, there is some vertical movement for the motion vectors at the corner, but the motion vectors at the corners for horizontal tracking operations are still parallel. Therefore, (referring to Figure 8) if the motion of the frame is horizontal: With respect to v_y, if more than two areas among I, II, II, IV satisfy $\frac{\|v_y\|}{\|\bar{V}_{ij}\|} \geq 0.2$, then the motion is panning. Otherwise the motion is considered horizontal tracking. With respect to u_x, if more than two areas among I, II, II, IV satisfy $\frac{\|u_x\|}{\|\bar{V}_{ij}\|} \geq 0.2$, then the motion is tilting. Otherwise the motion is considered vertical tracking.

Zoom and Rotate Operations. For images where less than half of the vectors have a dominant direction, we look for zooming or rotation. Figure 7 (bottom) shows the distribution of motion vectors for zooming and rotation. For zooming, all motion vectors point to (or radiate from) the focus of expanding (FOE.) For rotation, all the vertical lines of the motion vectors will point to the FOE. Our strategy for distinguishing these two camera motions is based on this distinction. We define an integer array $W(i,j)$, $i \in [0, M]$ and $j \in [0, N]$, where M, N are the width and height of the frame. Initially, we set $W(i,j) = 0$ for all i, j. Then, for any point $p(i,j)$ in the frame, consider its motion vector (u_{xi}, v_{yj}) and the extended line of the motion vector given by $y = \frac{v_y}{u_x}x + \frac{nu_x - mv_y}{u_x}$. For each point in the frame that is on the motion line, we increment the value of $W(i,j)$. After evaluating all points on all extended lines of the motion vectors in the frame, the comparative distribution of $W(i,j)$ for points inside and outside of the image FOE appears as in the plot on the left in Figure 9, that is, for an area surrounding the FOE, $W(i,j)|_{(i,j)\in FOE} >> W(i,j)|_{(i,j)\notin FOE}$. If such an area does not exist, a similar strategy verifies whether a comparative distribution of $W(i,j)$ for points inside and outside of the image FOE appears as in the plot on the right in Figure 9. In this case, the motion will be a rotation operation.

3.3 MPEG7 Compliance

The MPEG7 standards for multimedia content descriptors for audio-visual data are supported by the VDBMS system in two ways:

- VDBMS video preprocessing extracts nearly all low-level features which have been defined by MPEG7 [17] as standard (color, texture, motion, etc), and users can retrieve video shots

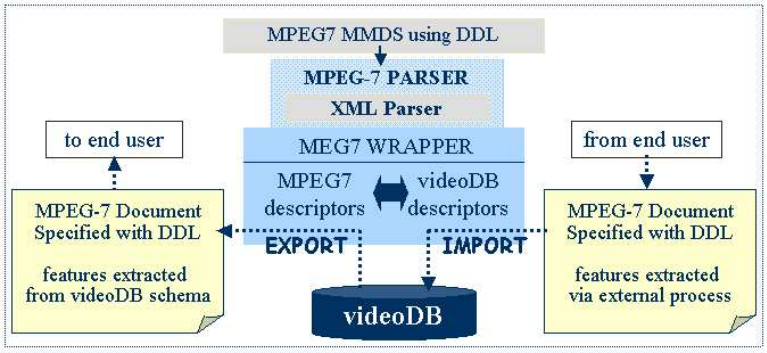


Figure 10: MPEG7 document support for feature import and export.

based on any combination of these features. Queries combining multiple low-level features can be used to approximate high-level content-based searches.

- VDBMS has developed an XML wrapper that can import any MPEG7 document specified with Data Definition Language (DLL) [16] and map its descriptors to the VDBMS object-relational database schema. The wrapper also supports the export of VDBMS extracted features and other meta-data from the database in the format of an MPEG7 document.

The XML wrapper enables the VDBMS system to make use of any available pre-extracted meta-data formatted as MPEG7 documents without preprocessing the video itself. In addition, features which VDBMS video preprocesing does not extract - most importantly event-based and other semantic features – can be integrated as VDBMS meta-data via this mechanism. In Figure 10, the *import* function takes as input a user-supplied MPEG7 document which is generated using Multimedia Description Schemes (MMDS) and contains the high and low-level feature descriptors. This document is passed through VDBMS's MPEG7 wrapper to extract, parse and map the descriptors to the VDBMS database feature schema. The video and its documented MPEG7 features are then stored inside the VDBMS database where they can be used for image and content-based queries. The *export* function extracts existing feature descriptors from the VDBMS database and sends them through the wrapper where they are mapped to the MPEG7 descriptors. The generated document can be used by other video processing tools or databases.

3.4 Hierarchical Video Summarization for Medical Data

We have developed a hierarchical video summarization strategy for medical video content, which provides users with an overview of the content at various levels of abstraction. To generate an overview, the key frames of a video are preprocessed to extract *special frames* (such as black frames, slides, clip art, sketch drawings) and *special regions* (such as faces, skin or blood-red areas). A shot grouping method is applied to merge spatially or temporally related shots into groups. Visual features and a priori content knowledge about the medical videos are integrated to assign the groups into predefined sematic categories. Based on the video groups and their semantic categories, the video summaries for different levels are constructed by group merging, hierarchical group clustering, and semantic category selection. Users choose the level of summary to access: the higher the level, the more concise the summary, the lower the level, the greater the level of detail contained in the summary. A detailed description of this process is given in [30].

3.5 Indexing and Query Processing

The meta-data which represents and indexes video content occupies more disk space than the video itself. The enormous magnitude of this meta-data and its storage in the database as high-dimensional vectors present serious indexing and searching difficulties in the execution and optimization of feature-based queries.

The Generalized Search Tree (GiST) [29] developed at Berkeley is an extensible data stucture that allows programmers to develop indices over any kind of data, supporting any lookup over that data. The GiST packages unifies B-tress, R-trees, R-trees, SR-trees, hB-trees (and many others) in one data structure, providing both data and query extensibility. GiST v1.0 provided low-dimensional indices, with no support for nearest-neighbor queries. GiST v2.0 supports high-dimensional indexing and nearest-neighbor search. GiST v2.0 is not, however, available for the Shore v2.0 upon which VDBMS is based. The VDBMS research group extended the indexing capability of Shore v2.0 by incorporating the GiST v2.0 implementation of the SR-tree as the high-dimensional index, and modified the query processing layer of PREDATOR to access the Shore/GiST index. VDBMS added the *vector* ADT to be used by all feature fields in the Features and FrameFeatures database tables, and implemented

```
CREATE GSR INDEX <table>_<fieldname> <table> (<table>.<fieldname>);
```

to create an instance of the GiST SR-tree for the given field to be used as the access path in feature matching queries. The multi-dimensional indexing structure handles the high-dimensional feature vectors that are produced by visual feature extraction and are used in image similarity searches.

In multi-feature image similarity queries, users present a sample image and query the database for images "most similar" to the example based on some collection of visual features. Although the database images can easily be ranked for each feature separately, results must be presented to the user in a combined similarity order. The query evaluation model used for a similarity search does not generally return a collection of exact matches, but rather a ranked collection of results with a score (or grade) attached to each result. The aggregate grade for a given result is obtained by combining the grades of several atomic similarity rankings, where the atomic ranking is based on a single feature or attribute of the database object. Single-feature similarity queries in multimedia retrieval are quite standard [5, 15]. The challenge for multi-feature similarity search is the ranking of results based on an overall aggregate grade, using several atomic rankings as input.

Many algorithms have been proposed in the literature to address aggregation ranking, notably Fagin's algorithm [8], the TA, CA and NRA algorithms [9], the Quick-Combine algorithm [13], the multi-step aggregation algorithm [20] and the Stream-Combine algorithm [14]. Two alternatives exist for implementing rank-join algorithms for databases: table functions or encapsulation in a physical query operator. Since there is no straightforward method for pushing query predicates into table functions [23], their performance is severely limited. Implementing a rank-join algorithm as a pipelined query operator, however, is very appealing for query optimization and allows for handling nested joins and views efficiently. The operator permits greater flexibility in generating candidate execution plans as opposed to the use of table functions. The encapsulation of the rank-join algorithm in an operator makes it possible to shuffle the evaluation plan operators in seeking the best plan.

For a rank-join algorithm to be implemented as a pipelined query operator, the algorithm must have two key properties. First, the algorithm should be *incremental*. An incremental rank-join algorithm does not depend on specifying the number of required results beforehand, rather it provides the next result whenever called for. Secondly, the operator must support *pipelining*. For a query operator to be part of a pipeline, the output of one operator should be valid input to the

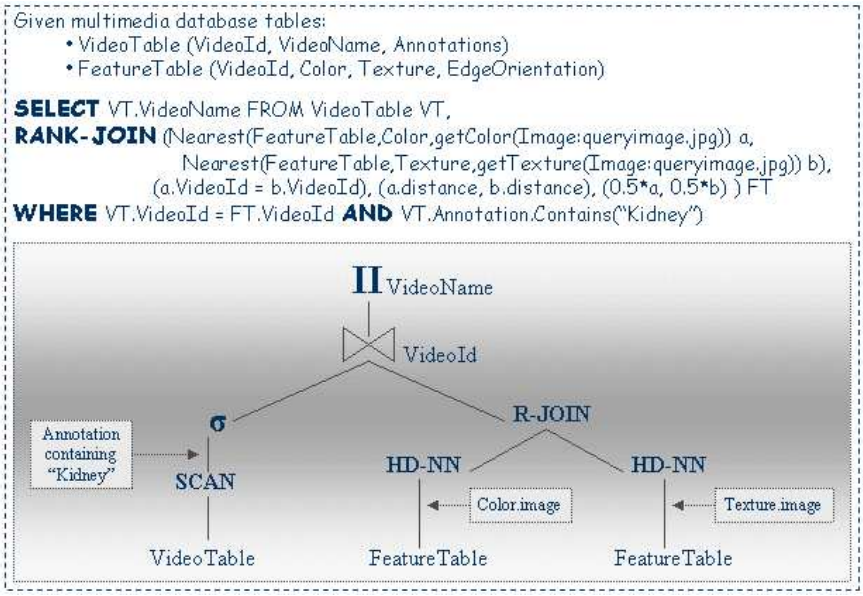


Figure 11: A rank-join query and its execution in the VDBMS query plan.

next operator in the pipeline. The pipelining property allows for the realization of join hierarchies and nested views, and hence a wider range of query evaluation plans. Rank-join algorithms that depend on the availability of a random access to the inputs cannot be realized as pipelined operator; random access is not possible when the input arrives as output from another operator.

The NRA (no-random-access) algorithm in [9] assumes only sorted access is available on the input streams. We have adapted the NRA algorithm so that it can be realized as a pipelined query operator, and we have encapsulated and implemented the algorithm as the No-Random-Access-RankJoin (NRA-RJ) logical query operator for VDBMS. NRA-RJ takes multiple ranked objects as input and produces a global ranked list of the objects as output. The objects in each input list are associated with grades and are sorted in descending order according to these grades. The operator requires three parameters: a join condition to determine the correspondence between objects from different streams with possibly different schema, a grade field name for each input stream, and the weighting expression for computing the combined grade. NRA-RJ supports image similarity matching with respect to multiple features, returning the top k results, where users specify the value for k. Each database frame image is represented as a point in high-dimensional space, thus the similarity problem is transformed to a nearest-neighbor query on a high-dimensional structure.

Using the NRA algorithm directly in the implementation of the NRA-RJ operator was complicated by two problems. First, the algorithm depends on a predefined value for k, the number of top results to be retrieved. We needed an *incremental* version of the algorithm which produces the *next top* object when needed by the caller. The second problem is that the output from the algorithm does not have exact grades associated with the output objects. Instead, each object has a range from worst grade to best grade. This prevents pipelining the operator in the query plans, since the exact ranks (grades) will be available only from the source input streams.

We describe the NRA-RJ operator in terms of the operations *Open*, *GetNext*, and *Close*. *GetNext* is implemented as a binary operator for practical reasons (the algorithm still holds for more than two inputs). Internal state information needed by the operator consists of a *priority queue* holding objects encountered thus far. Objects are sorted on worst grade in descending order, and

ties are broken using best grade. To allow for pipelining, inputs to the algorithm may be source streams or output streams from other algorithm executions. Therefore, each object in the input streams is associated with a range of grades from worst grade w to best grade b (where $w = b$ for exact grades). At depth d (the number of objects retrieved from each input stream), the proposed algorithm maintains the bottom values $\underline{b}_1^{(d)}$ and $\underline{b}_2^{(d)}$. The worst grade of an object R is computed as $t(w_1, w_2)$, where t is the weighting function and w_i is either the worst grade of the object according to input i, or 0 if the object has not yet been encountered in input stream i. Similarly, the best grade of an object R is computed as $t(b_1, b_2)$, where b_i is the best grade of the object according to input stream i, or $\underline{b}_i^{(d)}$ if the object has not yet been encountered in input stream i.

In the *Open* operation, the operator initializes the internal state information and opens the left and right child iterators. The *Close* operation destroys the state information and closes the input iterators. The algorithm for the core operation, *GetNext*, begins by checking the priority queue (buffer) to see if an object can be reported. An object can be reported if its worst grade is still greater than the best grades of all other objects. The maximum best grade for objects encountered thus far is obtained from the buffer. For objects not yet encountered, a threshold value can be used as an upper bound of the maximum possible best grade. The threshold is obtained by applying the weighting function to the best grades of the last encountered left and right objects. The maximum best grade in the buffer is maintained so that a scan of the whole buffer is not necessary for each call. To deal with grade ranges, the algorithm uses the best grades from the input streams to update the bottom values and to update the best grade of objects in the buffer. Optimization of the algorithm is achieved by introducing the *batch* depth technique. In this technique, whenever the depth d increases—by retrieving objects from left and right iterators—a minimum depth step s must be performed. That is, the operator must retrieve at least s objects from each of its input streams. The batch depth technique may cause the retrieval of extra database objects but will reduce the overhead of maintaining the objects in the operator priority queue.

Recently, Apostol et al. introduced a new rank-join algorithm, the J^* algorithm [19]. The basic version of the algorithm does not use any random access, and it support join hierarchies (i.e. pipeline of join operations). No other efforts to introduce the rank-join algorithm as a query operator have appeared in the literature. We have also implemented J^* in VDBMS, and Section 4 presents the results of a study in which we compare the performance of these two operators.

3.6 Video Streaming and Search-based Buffer Management

Continuous-media servers that support content-based search and retrieval use a main memory buffer to store the requested media streams before sending them on to the user. Caching parts of media streams that may be referenced in the near future enhances streaming performance in two ways: (1) it reduces the number of references to disk storage, and (2) it minimizes delay associated with the start of streaming. Precise caching decisions are difficult to make, however, since they depend on future knowledge about expected-streams. We have developed an efficient buffer prefetching and replacement policy based on knowledge collected from the content-based search manager and the streaming manager. By integrating knowledge from the search and streaming components, VDBMS can achieve better caching of media streams, thus minimizing initial latency and reducing disk I/O.

Optimal prefetch and replacement policies prefetch data before its first reference and replace the data block that will not be referenced for the longest time. The obvious difficulty is dependence on knowledge about future tasks, which is generally not available. In the case of video streaming, however, there is an inherent connection between the streaming and the searching processes. Choices

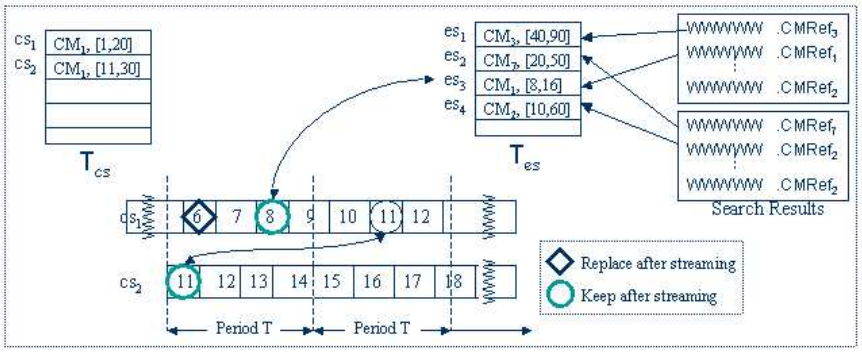


Figure 12: The search-based replacement policy. Two current streams cs_1 and cs_2 are referencing common blocks, and some expected-streams have been collected from the search results. Page 6 of cs_1 is removed after use as it is no longer referenced by current or expected-streams. Page 11 of stream cs_2 will be referenced by cs_1 and is kept in the buffer. Page 8 of cs_1 is expected for reference by expected-stream es_3 and is also cached.

for streaming are usually based on search results, and this relationship can be used by the buffer manager to prefetch and cache pages expected for reference. Many factors must be considered when basing prefetching or replacement decisions on search results. Streaming decisions based on the search context are probabilistic: new streaming requests can be based on any of the search results or even on none of them. Also, since the caching space is now shared by pages for current as well as expected streams, there will be increased overhead in the replacement policy associated with balancing the space assigned to each.

Buffering policies for media streaming have been investigated in several studies. The work in [3] introduces a memory-efficient prefetching schedule based on fixing the time-displacement between prefetching requests. Replacement policies for media streams have been studied in [7, 18, 21]. The target applications are systems designed for streaming purposes only, such as video on demand (VOD) applications. In [21], two replacement policies are introduced, based on caching data that is expected for play by other streams within the shortest time period. The algorithms produce effective buffer hit ratios for long streams (videos lasting more than half an hour) and for streams frequently requested by clients. The Use&Toss strategy is suggested for sequential access patterns in [28]. This strategy does not consider caching the page after using it, and hence the page can be replaced immediately after use. In each of these studies, the focus is on systems dedicated to media streaming. Most of the previous studies consider VOD-like applications where the video data set is small and streaming lasts for a long time. Our application domain is based on requests for streaming video shots, and is quite different. The data set is large and the streaming time is short. Methods that take into account the relationship between search results and streaming requests to determine buffer management policies have not previously been addressed in the literature. This new approach has been developed and implemented in VDBMS. A detailed description of our buffer management policy is given later in this section.

The Stream Manager. The stream manager is responsible for handling the special needs of video streaming. Each request for video data needs to be streamed with a predetermined rate; MPEG1 needs (on average) a 1.5 Mbps display rate. Violating the rate of streaming by either increasing or decreasing the display rate may result in overflow at the client buffer or hiccups at the client side. To hide the latency associated with access to disk storage, the stream manager streams part

of the data (a *segment*) while prefetching the next segment into the memory buffers. Since many stream requests are serviced simultaneously by the manager, resources such as memory buffers and disk bandwidth must be divided among the streams. This is achieved by serving each stream request *periodically*, and serving additional concurrent streaming requests within that period. Due to limited memory and disk bandwidth, the manager can only serve a specific number of requests within a single period. To serve requests in real-time, the segment referenced next should be retrieved into the buffer before the end of the current period.

We have implemented a stream manager layer above the buffer manager layer in VDBMS, and its functionality is as follows: (1) admit a new stream request if the maximum number of concurrent streams has not been reached, otherwise delay the request and re-try when one of the current requests finishes, (2) schedule segment prefetching by sending requests to the buffer manager. Each page of the allocated segment is fixed in the buffer pool until the page is streamed, (3) send the segment to the client according to a predetermined streaming rate. The segments are processed page-wise, and each page is unfixed and returned to the buffer manager after streaming its content, and (4) communicate with the query manager to keep track of search results.

The memory buffer is organized into pages of fixed size, and the segment of a stream that is serviced within a period consists of multiple buffer pages. The main memory buffer is divided into two areas, the *buffer pool* which serves video streams, and the *database buffer pool* which is mainly used by the search manager. Buffer pools are limited in size and are controlled by the *buffer manager*, providing basic functionality such as allocation and replacement of pages.

Search-based Buffer Management. Search-based policies are designed to achieve the following goals:

- Consider both current and expected streams in caching and prefetching decisions,
- Adapt quickly to the probabilistic nature of caching or prefetching expected streams, e.g., a prefetched page may turn out to be unreferenced and therefore should not be kept in the cache,
- Service the maximum number of requests from the cache, and
- Introduce minimal overhead to the time-sensitive streaming process and the core buffer management functionality.

Our technique depends on information collected from different system components about currently serviced streams and expected-stream requests. The stream manager stores the information in its own data structures and uses it to guide the allocation and replacement of the buffer manager as follows. When a new stream request is admitted for streaming, the portion of the stream to be accessed is declared. The manager records the information as a tuple of the form T = *(CM#, start, end)*, where *CM#* is a unique identifier for the stored stream, *start* and *end* indicate the starting and ending block numbers of the streamed data. For current streams, the manager keeps the tuples in the lookup table T_{cs}. The manager modifies the starting block for each serviced stream at the end of each period to track the currently streaming blocks. In addition, as the query manager replies to a user query with a set of candidate results, the stream manager receives a copy of the search results and considers the top-ranked set of results as likely for future streaming. We refer to the top-ranked results as *expected-streams*, and the stream manager records the CM#, starting and ending blocks for the expected-streams in a separate lookup table T_{es}. The information in the

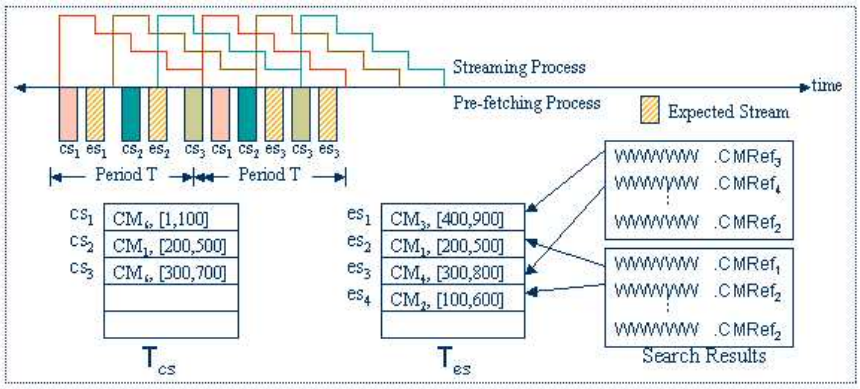


Figure 13: The search-based pre-fetching policy. This scenario shows three current streams cs_1, cs_2, cs_3 and two search results. The period T can serve five concurrent streams, so the stream manager uses the extra resources to prefetch two expected-streams from the T_{es} table. The top-ranked expected-streams from each search result are chosen before lower-ranked ones.

two lookup tables is used with different confidence levels, since the information in T_{cs} describes the actual system status, whereas T_{es} represents only an expectation of future references.

Our core buffer manager uses the Generalized CLOCK [26] (GCLOCK) replacement algorithm for buffer page replacement. We associate the counter "keep_weight" with each page. As a page is unfixed and returned to the buffer pool, the counter is set by the stream manager to a predetermined value. GCLOCK views the buffer pool as a circular list with a pointer identifying the next page to check for replacement. When a page is to be allocated, the pointer scans the unfixed pages searching for a page to allocate, and continues to decrement the counters until reaching a page with a counter equal to zero.

The Replacement Policy. In the search-based replacement policy, pages in the buffer pool that are referenced by either current or expected-streams are considered for caching. To maximize the number of caching pages, we do not cache pages that will be referenced by current streams after a long period of time has passed. Also, we prefer caching pages that will be referenced by current streams to those referenced by expected-streams, assigning higher keep_weight values to pages for current streams than to those for expected-streams.

As a page P is streamed by request R_i, the manager determines whether any concurrent streams will reference P by checking the table T_{cs} for a request R_j with the same CM#, such that P is within the interval given by the R_j's starting and ending blocks. To avoid situations where P is also referenced by R_j in the current period, we offset R_j's starting block by the length of the period. One match in the table is enough to decide to keep the page in the buffer pool. If a match is found, the stream manager assigns a value $w_c \geq 1$ as the keep_weight value of the page to inform the buffer manager that the page should not be replaced for some time. On the other hand, if no matching entry in T_{cs} is found, we check T_{es} for a match. To avoid caching pages expected for reference after a long period of time has passed, we restrict caching to the first segment of expected-streams. If a match is found, manager assigns a value $w_e \geq 1$ as the keep_weight of P. However, we choose w_e to be less than w_c to reflect the fact that we are less confident about expected-streams than actual streams. If no match is found in either T_{cs} or T_{es}, the manager sets the keep_weight to zero, indicating that the page can be replaced immediately by the core buffer manager. Figure 12 presents a scenario illustrating the replacement policy.

The Prefetching Policy. With knowledge collected from the search manager, we can predict with high probability that one of the expected-streams will be requested. We utilize the fraction of the period unused by current streams to prefetch the *first segment* of the top ranked expected-streams into memory. While serving the current stream requests, the manager tracks the utilization of the streaming period. If it detects a number of serviced streams less than the maximum possible, it consults the T_{es} table, chooses one of the expected-streams, and prefetches its first segment into the memory buffer. Afterwards, it immediately unfixes its pages and sets the keep_weight to $w_e \geq 1$. In this way, the segment will be kept in the buffer pool for some time before being replaced. The stream manager does *not* keep information about these requests in T_{cs}, since they do not belong to current streams.

The stream manager loops around the expected-streams, bringing in the first segment of top-ranked streams before serving the lower ranked expected-streams. If the prefetched segment turns out to be unreferenced, it is aged in the buffer pool, and the keep_weight is eventually reduced to zero, forcing the segment to be replaced out of the pool. This process is handled separately from the tracking of actual status of expected-streams. If an expected-stream becomes an actual request, most of the pages in the first segment would already be cached in the buffer pool, so that the number of references to the lower level storage would be significantly reduced. A scenario for the prefetching policy is shown in Figure 13.

Our system follows the client server architecture, where each client request is directed to the appropriate processing unit. Stream requests are directed to the stream manager process and search requests are forwarded to the search manager process. The two processes share the same address space and can easily exchange information. The stream manager is implemented as a multi-threaded module, where different threads are started to handle these tasks:

- Serve new stream requests in first come first served (FCFS) order. If a new request arrives and the number of current streams is less than the maximum number of supported streams, the stream request is admitted. Otherwise, delay it until one of the current requests finishes.
- Switch between concurrent streams to prefetch the next segment for each stream. Update the T_{cs} table with the current stream status (starting block).
- Keep track of already streamed pages and consult the T_{cs} and T_{es} tables to set the keep_weight value of the page before handing them to the core buffer manager (unfixing).
- Communicate with the search engine. Each time a search request is serviced, the stream manager updates the T_{es} table with the top-ranked search results.
- Prefetch expected-streams. The stream manager continuously monitors the period, and if the number of current streams is less than the maximum supported streams, the manager services one of the expected-streams, where top-ranked expected-streams are always serviced first.

4 A Development Environment

A useful development environment must be designed in a flexible, extensible way so that new algorithms, new data types, or entirely new components can be easily added. The VDBMS system was designed from the start to support the easy interfacing of new structures and components.

Our development process required the implementation of modules with well-defined interfaces and encapsulated functionality. This process has not only allowed us to video-enhance the original Shore/PREDATOR system by investigating, developing and implementing the fundamental components to support full video database functionality, it has also given us a platform for integrating and evaluating algorithms and components from other sources. Our system can be used to *test* the correctness and scope of algorithms, *measure* the performance of algorithms in a standardized way, and *compare* the performance of different implementations of a compoment.

To illustrate the effectiveness of VDBMS for new algorithm implementation and algorithm performance analysis, we briefly describe a performance evaluation for two pipelined rank-join operators in executing multi-feature queries. The study compares the NRA-RJ operator developed by the VDBMS research group and the J^* operator introduced by Apostol et al. We implemented both the J^* operator and (for a baseline comparison) the non-pipelined version of the NRA algorithm as a multi-way rank-join operator, MW-RJ. Although most query optimizers are restricted to binary operators, MW-RJ provides a reference line for the best possible performance. A multi-feature query for the k top-ranked results was issued against the VDBMS features:

Q: Retrieve the top k video shots "most similar" to a given image, based on m visual features.

The query evaluation plan has m NN operators on m different visual features, and $m-1$ rank-join binary operators are used, where the results of one operator are pipelined to the next operator in the pipeline. The number of features m varies from 2 to 6, and the number of top-ranked results k varies from 5 to 100. To evaluate the operators, we used the following performance measures: (1) query running time for retrieving the top matching k output results, (2) size of the buffer maintained by the operator, and (3) number of database accesses in disk pages. While the number of database accesses should give a good indication of the time complexity of the operator, the experiments show a significant CPU time complexity difference between the two operators that affects the total running time, especially for small numbers of inputs.

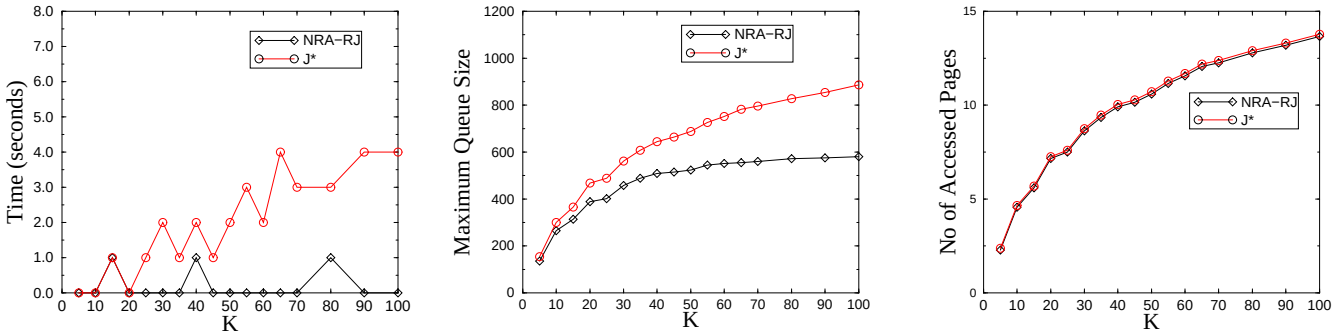


Figure 14: Comparing NRA-RJ and J^* for $m = 2$.

Figures 14 and 16 give performance comparisons for NRA-RJ, J^* and MW-RJ, for $m = 2$ and $m = 3$ respectively, where m is the number of input sources that give a pipeline of length $m - 1$. For $m = 2$, NRA-RJ is identical to MW-RJ since there is no pipeline. Figure 14(a) compares the total running time of the NRA-RJ and J^* operators. The J^* algorithm has a significant CPU

overhead due to the execution of its underlying A^* graph search algorithm, which considers more join combinations. Thus, NRA-RJ shows a faster execution time. Both operators are nearly equal in the database access count depicted in Figure 14(c). NRA-RJ has a smaller maximum queue size than that of J^*, as shown in Figure 14(b), and the difference increases as k increases (i.e., as more results are requested). The difference in the maximum queue size and in the execution time can be explained by the fact that the J^* algorithm has to consider more join combinations than NRA-RJ since it was developed for a general join condition. When used in self-join problem settings, the generality of the J^* algorithm causes expensive unnecessary computations that increase both the queue size and the running time.

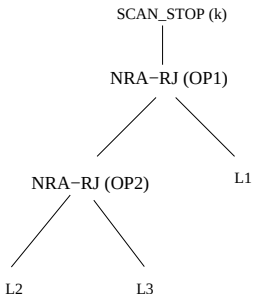


Figure 15: A query pipeline with $m = 3$.

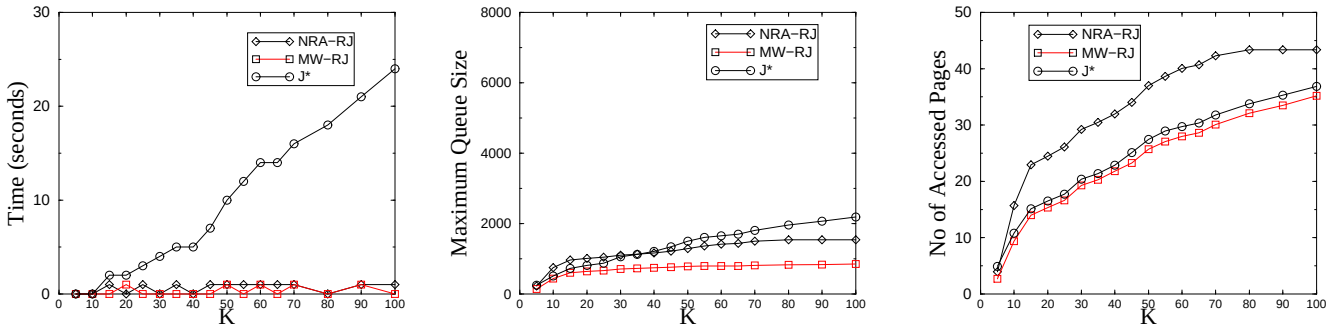
For $m = 3$, Figure 16 compares the NRA-RJ, J^*, and MW-RJ operators. Figure 16(a) shows that NRA-RJ still outperforms J^* in total running time, and the pipeline does not affect the speed of the NRA-RJ operator when compared with MW-RJ. For the maximum queue size given in Figure 16(b) and the number of database accesses given in Figure 16(c), we make the following observations:

- NRA-RJ has a larger maximum queue size and more database accesses than MW-RJ. To understand this difference, we clarify how NRA-RJ operates in a pipeline of 3 inputs. Figure 15 shows NRA-RJ with three input streams, L_1, L_2 and L_3. When the top NRA-RJ operator, OP_1, is called to produce the next top ranked object, several *GetNext* calls for the left and right children are invoked. According to NRA-RJ's GetNext algorithm, OP_1 gets the next tuple from its left and right children at each step. Hence, OP_2 will be required to deliver as many top ranked objects for L_2 and L_3 as for L_1. These calls to the ranking algorithm in OP_2 force L_2 and L_3 to retrieve unecessary objects and result in larger queue sizes with more database accesses. We will refer to this as the *local ranking* problem, that is, the NRA-RJ operator in the early pipeline stages tends to retrieve more database objects in order to deliver as many ranked tuples as required by the next NRA-RJ operator.

- The J^* operator has less database access cost than NRA-RJ and close to that of MW-RJ, despite NRA-RJ's *local ranking* problem. In contrast to NRA-RJ, the J^* algorithm does not retrieve equal numbers of objects from its left and right children.

- For the same reason that J^* has less disk accesses than NRA-RJ, J^* starts with smaller maximum queue size than NRA-RJ. However, as in the case for $m = 2$, J^* begins to save many candidate join combinations in the queue, causing its maximum queue size to become larger than that of NRA-RJ as k increases. This also explains the fact that J^* has a larger

queue size than MW-RJ, even though both are retrieving almost the same number of database objects, as shown in Figure 16(c).



(a) (b) (c)

Figure 16: Comparing NRA-RJ, MW-RJ and J^* for $m = 3$.

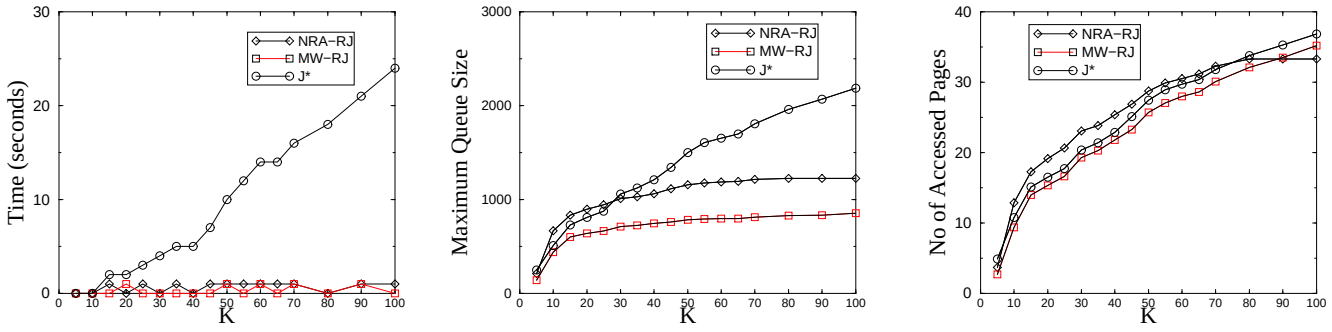
We now evaluate the scalability of the two pipelined operators with respect to the length of the query pipeline m. By fixing $k = 20$, the operators NRA-RJ and J^* are again compared with respect to our three chosen performance metrics. As m increases from 2 to 6, NRA-RJ has a larger queue size because of the increased local ranking overhead in the pipeline. As NRA-RJ encounters greater database access, I/O cost begins to dominate total running time. The overhead finally affects the running time enough to make NRA-RJ performance worse than J^*, demonstrating clearly that J^* is scalable in terms of increased ranked inputs while NRA-JR is not.

Our evaluation of the performance of NRA-RJ led to an important insight: minimize the excessive local ranking calls in earlier stages of the pipeline. Our solution was to unbalance the depth step in the operator children. We changed the NRA-RJ *GetNext* algorithm to reduce the local ranking overhead by changing the way it retrieve tuples from its children, that is, to require less expensive *GetNext* calls to the left child, which is also an NRA-RJ operator. Using different depths in the input streams had a major effect on the performance. Figure 17 gives the a comparison between the modified NRA-RJ, the J^* and the MW-RJ operator. In the optimized version of the NRA-RJ operator, one tuple is retrieved from the left NRA-RJ child for each p tuples retrieved from the right input child (in the figure, $p = 2$.) The optimized NRA-RJ operator showed great performance improvements in both the maximum queue size and in the number of database accesses, due to the reduction of local ranking overhead in the inner pipeline stages.

With this improvement, the optimized NRA-RJ operator is superior to the J^* operator even for large m. The optimized NRA-RJ operator is an order of magnitude faster, has less space requirements, and has a comparable number of disk accesses.

5 Conclusion

We have introduced a video database research initiative that began in June 2000 and has culminated in the successful development of a video-enhanced database system that supports comprehensive and efficient database management capabilities for digital video libraries. The fundamental concept was to provide a full range of functionality for video as an intrinsic, well-defined database



(a) (b) (c)

Figure 17: The optimized NRA-RJ operator.

data type, with its own description, parameters and methods. System components such as video preprocessing, query processing and buffer management were implemented using algorithms and techniques developed by the VDBMS research group. This paper presents detailed descriptions for some of these algorithms and techniques. We define our system as a research platform, as it supports the easy integration of externally developed algorithms and components, and allows the system to be used for measuring and comparing the performance of different implementations of algorithms or components in a standardized way. This concept was illustrated by a performance study of rank-join query operators. An analysis of the resulting performance data led to a modification in the implementation of our multi-feature query operator which significantly improved its performance.

References

[1] Elisa Bertino, Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Access control model for video database systems. *In Proceeding of CIKM, Ninth international conference on information and knowledge management*, pages 336–343, November 2000.

[2] A. Brink. Thresholding of digital image using two-dimensional entropies. pages 803–808, 1992.

[3] Edward Chang and Hector Garcia-Molina. Effective memory use in a media server. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, Athens, Greece*, pages 496–505. Morgan Kaufmann, 1997.

[4] S. Chang, W. Chen, H. Meng, H. Sundaram, and D. Zhong. Videoq: an automated content-based video search system using visual cues. *Proceedings of the fifth ACM interternational conference on Multimedia*, pages 313–324, 1997.

[5] J.Y. Chen, C. Taskiran, A. Albiol, E.J. Delp, and C.A. Bouman. Vibe: A compressed video database structured for active browsing and search. In *In Proc. SPIE: Multimedia Storage and Archiving Systems IV 3846*, pages 148–164, 1999.

[6] M. G. Christel. Visual digests for news video libraries. In *Proc. ACM Multimedia Conf.*, pages 303–311, New York, 1999. ACM.

[7] A. Dan and D. Sitaram. A generalized interval caching policy for mixed interactive an d long video environments. *IS&T SPIE Multimedia Computing and Networking Conference, San Jose, CA*, jan 1996.

[8] Ronald Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences (JCSS)*, 58(1):83–99, Feb 1999.

[9] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middlewar. In *PODS'2001 Santa Barbara, California*, May 2001.

[10] J. Fan, G. Fujita, J. Yu, K. Miyanohana, T. Onoye, N. Ishiura, L. Wu, and I. Hirakawa. Hierarchical object-oriented image and video segmentation algorithm based on 2d entropic thresholding. pages 141–151, 1998.

[11] J. Fan and F. Gan. Motion estimation based on uncompensability analysis. pages 1584–1587, 1997.

[12] J. Fan, R. Wang, D. Xing, and F.Gan. Image sequence segmentation based on 2d temporal entropy. pages 1101–1107, 1996.

[13] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10–14, 2000, Cairo, Egypt*, pages 419–428, 2000.

[14] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous envir onments. In *In: Proceedings of the IEEE International Conference on Inform ation Technology: Coding and Computing (ITCC 2001), Las Vegas, USA, 2001*, 2001.

[15] A. Hamrapur, A. Gupta, B. Horowitz, C. Shu, C. Fuller, J. Bach, M. Gorkani, and R. Jain. Virage video engine. *SPIE Proc. Storage and Retrieval for Image and Video Databases*, pages 188–197, 1997.

[16] Multimedia Content Description Interface. Mpeg7 ddl working draft 4.0 – part 2: Description definition language. 2000.

[17] Multimedia Content Description Interface. Text of iso/iec 15938-3/fcd information technology, part 3-visual. 2000.

[18] Frank Moser, Achim Kraiss, and Wolfgang Klas. L/mrp: A buffer management strategy for interactive continuous data flows in a multimedia dbms. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, Zurich, Switzerland*, pages 275–286. Morgan Kaufmann, 1995.

[19] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *VLDV'01, Rome, Italy*, 2001.

[20] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE'99, Sydney, Austrialia*, pages 22–29. IEEE Computer Society, 1999.

[21] Banu Özden, Rajeev Rastogi, and Abraham Silberschatz. Buffer replacement algorithms for multimedia storage systems. In *Proceedings of IEEE International Conference on Multimedia Com puting and Systems, ICMCS*, pages 172–180, 1996.

[22] A. Pentland, R. Picard, and S. Sclaroff. Photobook: tools for content-based manipulation of image databases. *SPIE Proc. Storage and Retrieval for Image and Video Databases*, pages 34–47, feb 1994.

[23] Berthold Reinwald, Hamid Pirahesh, Ganapathy Krishnamoorthy, George Lapis, Brian T. Tran, and Swati Vora. Heterogeneous query processing through sql table functions. In *ICDE'99, 23-26 March 1999, Sydney, Austrialia*, pages 366–373, 1999.

[24] P. Seshadri and M. Paskin. Predator: An or-dbms with enhanced data types. In *SIGMOD 1997*, Tucson Arizona, May 1997.

[25] Shore. Shore storage manager architecture. June 1999.

[26] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[27] J. Smith and S. Chang. Searching for images and video on the world wide web. *CTR Technical Report*, 1996.

[28] Michael Stonebraker. Operating system support for database management. *CACM*, 24(7):412–418, 1981.

[29] M. Thomas, C. Carson, and H. Hellerstein. Creating a customized access method for blobworld. March 2000.

[30] X. Zhu, W. Aref, and A. Elmagarmid. Hierarchical video summarization for medical data. page to appear, 2001.