A Live Multimedia Stream Querying System

Bin Liu
Electrical and Computer Engineering
Georgia Institute of Technology
bliu@ece.gatech.edu

Amarnath Gupta
San Diego Supercomputer Center
University of California San Diego
gupta@sdsc.edu

Ramesh Jain
Department of Computer Science
University of California Irvine
jain@ics.uci.edu

ABSTRACT

Querying live media streams captured by various sensors is becoming a challenging problem, due to the data heterogeneity and the lack of a unifying data model capable of accessing various multimedia data and providing reasonable abstractions for the query purpose. In this paper we propose a system that enables directly capturing media streams from sensors and automatically generating more meaningful feature streams that can be queried by a data stream processor. The system provides an effective combination between extendible digital processing techniques and general data stream management research.

1. INTRODUCTION

Live sensors ranging from simple RFIDs to webcams are becoming common in many applications, from homeland security to businesses, and are producing enormous volumes of continuous sensor data that we call media streams. Expectedly, the large scale deployment of various sensors gives rise to more complex applications where multiple live streams of heterogeneous media are jointly monitored for querying interesting events [7]. A *media stream* is usually the output of a sensor device such as a video, audio or motion sensor that produces a continuous or discrete signal, but typically cannot be directly used by a data stream processor. To evaluate queries on media streams, one needs to continuously extract content-based descriptors, that we call *features*, from them and identify the qualifying media portions by evaluating queries on the generated *feature streams*, which are post-processed by one or more transformers and correlated to the media streams temporally and in terms of content.

While the progress in computer vision and other digital processing areas has enabled generating many automatic features in real time, synchronously combining individual media channels and derived feature streams remains a challenge. Moreover, users prefer a query system that allows efficiently performing complex queries over media and feature streams by exploiting the semantic constraints among them. Consider an application where a seminar is broadcast from a meeting room equipped with cameras and microphones for the presenter and the audience, along with a camera capturing presentation slides. A remote participant decides to "tune in" when some specific events occur, such as "the presentation begins" – which can be detected by the changes in the view of the slide camera and the voice of the presenter. While individual media channel or a derived feature stream captures some aspects of such an event, it is the synchronous combination of all the streams that captures the entire intended semantics of the content and makes the event detection easier or more effective than only using one media or one aspect of that media.

The MPEG-7 standard [9] provides a framework of standardized tools that can be used to describe and efficiently manage multimedia content. There are two categories of audiovisual features that can be extracted from each type of media streams. Low level features, like color, texture or face recognition, audio energy and speech, are those that can be extracted automatically. High level features need more human annotations and cannot be processed in real time. We use the term *live multimedia* to refer to the scenario where the multimedia information is not "produced" though manual editing, but is captured in a real-life setting – where the different sensors are observing different aspects of the same real-life situation, and streaming the captured data to a central processor. The primary problem for the *live multimedia management system* (LMMS) is how to effectively combine multiple media streams as well as auxiliary non-media stream information to answer standing queries about the situations observed by various media sensors. The MPEG-7 standard only attempts to insert additional descriptive features computed from a media object inside the object itself, but does not handle any issue related to multiple and concurrent media streams over which live multimedia queries need to be evaluated. In this context, a unifying data model that can access heterogeneous media streams and provide effective interpretations as well as reasonable abstractions for them is necessary.

Processing *continuous queries*, which are persistent queries that are issued once and then logically run continuously over live and unbounded streams, has become a major research area in data management. Many projects, such as OpenCQ, NiagaraCQ, Aurora, Telegraph, COUGAR and STREAM system, have addressed broad research issues including data models, query languages, query optimizations, etc [4]. However, to our knowledge, no stream related project in recent literature addresses the media stream continuous querying problem. The true challenge lies in the effective intersection of data stream management research and the research in various digital signal processing techniques, including but not limited to the image, video and audio processing.

Most of the video based event recognition approaches (such as [6]) to activity recognitions assume models for specific activity types (such as human activities), and depend on video-based procedural recognition methods designed for particular domains. Our research goal is to develop a declarative approach to general event queries by using various media streams. Unlike research issues in standard image, video and audio database systems (e.g., QBIC [3]) centering around queries like object detections and similarity retrievals using different features, we contend the LMMS should be able to make use of the additional contextual information and semantic constraints to answer more complex queries over live media streams. Different to the time-instant based VDBMS system [2], which stores both videos and extracted features in tables before querying them, we aim at processing *interval-based* media and feature streams in real time.

We approach the live multimedia query problem by proposing a general-purpose media stream management system, called *MedSMan*. It permits a designer to directly capture various live data streams from different sensor devices, and forms media streams consisting of logical media tuples. Then, more meaningful feature streams can be automatically derived from media streams for the query purpose. Our system is open to any user-defined *feature generation functions* (FGFs), via which various features are computed from media streams. We design description languages for capturing and representing various media streams from sensor types such as webcam and microphone, and generating automatic feature streams through particular FGFs. To issue continuous queries over live media streams, a *Media and Feature Stream Continuous Query Language* (MF-CQL) is designed and implemented by exploiting the semantics of media and derived feature streams, as well as the inter-stream constraints among them. Although MedSMan is designed to work independently of how features are produced, in this paper we consider only automatically extractable features.

The rest of this paper is organized as follows. In Section 2, we introduce the data model for MF streams. Both media capturing and feature generating, together with the stream description languages, are presented in Section 3 through concrete examples, followed by a brief introduction to the query processing. The implementation is presented in Section 4. We run a number of query experiments in Section 5, before concluding our work in Section 6.

2. DATA MODEL FOR MF STREAMS

2.1 Formal Definitions

Because of their continuous nature and the stream dependency, both media and feature elements require explicit and exact timestamps. The time attributes provide valuable information for the stream generating and query processing. In our framework, the element of a media or feature stream is defined as a *tuple*, which consists of a logical sequence number (sqno) indicating its position in a stream, a temporal extent defined by a pair of *start* and *end* timestamps (t_b, t_e] (for a time-point attribute $t_b = t_e$, we represent a single time point as t_d), and any other media attribute.

CONVENTION 1. *A set T, is said to be* ***continuously well-ordered*** *iff (1) T is well-ordered, and (2) for each time-unit $(t_{bi}, t_{ei}]$, there must be one and only one directly following time-unit $(t_{bi+1}, t_{ei+1}]$ in T, where $t_{ei} = t_{bi+1}$. We refer to a continuously well-ordered set of time-units as a* ***continuous time set****. A corresponding tuple value holds in each time-unit.*

CONVENTION 2. *A set T is said to be* ***discretely well-ordered*** *if and only if T is well-ordered, i.e., the continuity clause does not apply for two consecutive units. We refer to a discretely well-ordered set of time-points as a* ***discrete time set****. At each time-point t_d, a corresponding tuple value holds.*

DEFINITION 1. *A* ***continuous media stream*** *is a set of tuples, each consists of a sequence number (m_{sqno}) uniquely identifying its position in stream, a pair of start and end timestamps $(t_b, t_e]$ whose domain is a continuous time set, and a media valued attribute v_m valid only during $(t_b, t_e]$.*

DEFINITION 2. *A* ***discrete stream*** *is a set of tuples, each consists of a sequence number (m_{sqno}) uniquely identifying its position in stream, a media or non-media valued attribute v_m, and a time-point t_d, defined on a discrete time set domain, indicating when v_m arrives intermittently.*

DEFINITION 3. *A* ***feature stream*** *is defined as a set of tuples, each consists of a sequence number (f_{sqno}) uniquely identifying its position in stream, a feature value attribute v_f, a time-point attribute t_f indicating when v_f is computed, and a set $\bar{m}_{sqno}$ identifying the media tuples or a set $\bar{f}_{sqno}$ identifying other feature tuples, from which a feature tuple is derived.*

2.2 Stream Semantics and Operator Classes

While being absent in the general purpose Data Stream Management System (DSMS) [1], direct stream-to-stream operators are necessary for media stream systems, since media and feature streams are (1) quite different in operations; and (2) dependent. Besides, we need compose media stream(s) for presenting query results. The four subclasses of stream-to-stream operators, shown in Figure 1, are:

- MS-to-FS operators that generate feature streams from media streams;
- FS-to-MS operators that map back to the deriving media stream fragments from feature tuples;
- FS-to-FS operators that generate feature streams from existing feature streams;
- MS-to-MS operators that reformat a media stream or compose a composite media stream from multiple media streams.

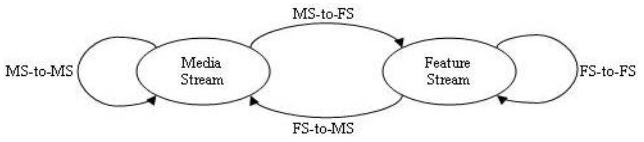

Figure 1: Stream operator classes and semantics.

Algebraic Operators In order to manipulate both media and feature streams, we design and implement a number of algebraic operators. The *Selection* and *Projection* operators are similar to those in traditional DBMSs. The *Scan* operator is used to get the next tuple of a same stream in a push mode. Working in a pull mode, the *Fetch* operator maps and retrieves the deriving media or feature tuples that generate the given feature tuple(s). An *O_Join* operator overlap-joins multiple tuples with different intervals but sharing a common time point. A *Compose* operator reformats a single media stream, or multiplexes multiple media streams as output. We also design and implement a set of period operators, such as *P_Scan*, *P_Detect*, and *P_Aggregation*, which are out of the scope of this paper.

2.3 Query Delay

A number of cost metrics are investigated, such as the queue lengths and operator costs. In this paper, we focus on the *query delay*, which is significantly determined by the *feature computation delay* (FCD). For one satisfying tuple in a media stream, its query delay is defined as the time difference between it enters the system and it leaves the topmost operator. Query delays are both media stream dependent and individual tuple dependent, since (1) different media tuples have different tuple extents and FCDs; (2) a particular media tuple may be joined by multiple tuples from other streams; thus (3) different tuples in one stream joined by a common tuple in another stream may have different delays.

3. MF STREAM PROCESSING

3.1 Media Stream Generation

3.1.1 Media Generation Model and Classifications

The raw streams captured from sensors are bit streams wihthout explicit media semantics. To correctly retrieve and interpret them, the details of *registering data* (such as media format, coding and size) must be known [10]. Particularly, de-multiplexing, decompression, or decryption procedures may be applied to retrieve individual media tracks from a raw stream. The heterogeneous nature (such as continuity, media type and data rate) of media streams greatly increases the processing complexity. However, there are some general principles on how to create various media streams from sensors and separate them into sequences of media tuples.

The *media stream generation model* $\mathcal{M}$ consists of a sensor type S_T, a sensor source S_S, a media tuple definition T_M, a set of sensor-dependent initialization parameters $\bar{P}$, and a media stream type M_T. S_T plays a significant role in determining other parameters. Basically, there are three categories of M_T according to the differences in stream continuity and data rate:

1. **Continuous stream**, whose values are continuous over time, such as an audio. It can be split into a sequence of media tuples with any appropriate size specified in $\bar{P}$. For example, an audio is cut into clips of $40ms$.
2. **Pseudo-continuous stream**, whose tuples are discrete, but we prefer to consider it as a continuous stream–each tuple spans an entire interval defined by two sequential tuple arrival timepoints. This type of stream typically consists of tuples arriving at a specified data rate defined in $\bar{P}$, e.g., 30 frames per second for a video.
3. **Intermittent stream**, whose tuples are intermittent, such as a sequence of Powerpoint slides. It can be split into intervals terminating at punctuation boundaries [11], or by tokens from a different stream [5].

Both the continuous stream and the pseudo-continuous stream are defined on continuous time sets, while the intermittent stream is defined on a discrete time set.

3.1.2 The MSDL Language

A *Media Stream Description Language* (MSDL) is designed for defining various media tuples. Basing on a specific T_M, a media stream instance conforming to particular S_T and $\bar{P}$ can be declared and initialized. For example, a video tuple definition and a video stream instance can be declared as:

```
create type frame { integer frame_num primary key,
            time frame_bt, time frame_et, image content };
create media stream video1 of frame from
        sensortype cam sensorsource vfw://0 datarate 10.0;
```

Since *sqno* (i.e., frame_num in this example) works as a unique identifer for each tuple in a stream, it is declared as a primary key. Futher, most query operators require the explicit timestamp attributes t_b (i.e., frame_bt) and t_e (i.e., frame_et). The sensorsource (S_S) defines a local or remote URL connecting the capturing sensor. The sensortype (S_T) cam indicates that the output media stream is a pseudo-continuous video stream, and requires a datarate parameter denoting the data rate of a video, which means each video frame spans an extent of $100ms$ on average. In contrast, an audio tuple definition and an audio instance can be declared as:

```
create type audioclip { integer clip_num primary key,
            time clip_bt, time clip_et, audiobuffer clip };
create media stream audio2 of audioclip from
  sensortype mic sensorsource dsound:// capturebuffersize 40;
```

Different to cam, the mic (S_T) indicates a continuous audio stream, and requires a capture buffer size (e.g., $40ms$) to cut the stream into a sequence of audioclips.

3.2 Feature Stream Generation

3.2.1 Characterizing Features

A feature stream is produced by one or more transformer functions operating on media streams or other feature streams. A feature tuple value is generated from its deriving media or feature tuple(s) by one or multiple specific FGFs, with different generation costs. A feature stream has tuples with complex-valued attributes, and a reference to the media/feature streams from which it is derived. Properties of feature values include:

1. The feature value v_f can be either a single or a complex value (e.g., vector, set or any other value that can be processed directly by a query processor).
2. Its temporal attribute t_f is defined on a discrete time set. We assume v_f is computed and available only by t_f, regardless of when the computation begins and how long it takes.
3. A single feature tuple may be derived from multiple media tuples, or multiple feature tuples derived from the same set of media tuple(s). For example, a movement detection is derived from two consecutive video frames.

Feature Tuple Atomicity A feature tuple is an entity dependent on its deriving media tuple(s), and its value is *atomic* in that its semantics represents one aspect of all media tuples from which it is derived. Therefore, a feature tuple has a *representing interval* equal to the time-unit or set of time-units of its deriving media tuples.

3.2.2 Feature Generation Model

A *feature stream generation model* $\mathcal{F}$ consists of a feature tuple definition T_F, a set of deriving media or feature streams $\bar{D_S}$, a set of feature generation functions $\bar{F_G}$, and a set of optional parameters $\bar{P_F}$ controlling the data rate, delay, etc. We have designed a *Feature Stream Description Language* (FSDL) to create feature tuple definitions. Basing on them, feature stream instances can be declared from deriving $\bar{D_S}$ by using particular $\bar{P_F}$.

3.2.3 FGF Implementations and The FSDL Language

The feature generation functions are implemented as interface between the FSDL and the underlying digital signal processing techniques. Thus, the FSDL is open for users to implement their own FGFs. We introduce the syntax of FSDL through instances of different FGF implementations.

Color-based Object Detection Due to the low computational cost of the algorithm, the color histogram based object detection is a desirable feature for simple object detections when appropriate [12], such as detecting "a red book". The similarity between each video frame and the reference image is evaluated by using a normalization threshold. In our implementation, the histogram consists of 32^3 bins (32 bins each for R, G, B) and the threshold can be adjusted for different objects. Basing on video1, an object detection feature and its feature stream instance can be expressed as:

```
create type odFeature { integer od_sn primary key,
            time od_bt, time od_et, integer od_pixel };
create feature stream odFStream2 of odFeature on video1
with   od_sn:=getFrameNum(frame_num)
       od_bt:=getFrameTime(frame_bt)
       od_et:=getFrameTime(frame_et)
       od_pixel:=getObjdetectNum(content);
```

Each feature attribute must be explicitly declared from its deriving media stream attribute(s). Taking each video frame as input, the FGF getObjdetectNum computes the total number of pixels – whose color falls into the corresponding reference histogram bin with value greater than the threshold – in each frame. The getFrameNum returns the frame number for each frame, while the getFrameTime computes both start and end time for each frame.

Haar-like Face Detection We implement a face detection feature using OpenCV object detectors initially proposed in [13]. The trained classifier (namely a cascade of boosted classifiers working with haar-like features) is also provided with OpenCV. The key FGF getFaceDetectNum computes the number of detected faces in each frame. This feature and a feature stream instance are expressed as:

```
create type fdFeature { integer fd_sn primary key,
            time fd_bt, time fd_et, integer fd_num };
create feature stream fdFStream3 of fdFeature on video1
with   fd_sn:=getFrameNum(frame_num)
       fd_bt:=getFrameTime(frame_bt)
       fd_et:=getFrameTime(frame_et)
       fd_num:=getFaceDetectNum(content);
```

Movement detection Different from the features mentioned above, a movement feature is computed by every two sequential video frames. The FGF getMovementNum computes the total number of pixels, with the difference between the average RGB value of each pixel and that of the pixel at same location in previous frame being greater than a specified threshold. A 3×3 low-pass filter is applied to both frames to reduce the background noise in advance. Such a feature and its feature stream instance can be expressed as:

```
create type mvFeature { integer mv_sn primary key,
            time mv_bt, time mv_et, integer mv_pixel };
create feature stream mvFStream1 of mvFeature
on     video1 [skip 1 in every 2]
with   mv_sn:=getFrameNum(frame_num)
       mv_bt:=getFrameTime(frame_bt)
       mv_et:=getFrameTime(frame_et)
       mv_pixel:=getMovementNum(content);
```

The feature computation costs vary significantly depending on different media types, generation algorithms and available system resources. In the extreme case, a feature generation may last longer than the extent of deriving media tuple, with potentially adverse effects on the entire system performance when for high data-rate media streams. For example, the getMovementNum for frame movement costs longer than $100ms$, thus will delay a video stream with data rate greater than 10 frames per second. However, typical media streams are redundant in content so that skipping some tuples may not affect the query accuracy much. We may skip (e.g., mvFeature) one media tuple in every two, if the media stream data rate is overwhelming. One may wonder why we do not define all feature attributes (e.g. both mv_pixel and od_pixel) in a single feature tuple. The reason is that both getMovementNum and getObjDetectNum functions are time consuming. Placing them as a single tuple might seriously delay, or even block, feature tuple generation for fast incoming video frames. Rather, it is highly recommended to separate the time consuming feature generations into different feature tuples and use parallel feature generation threads. A possible optimization would be to consider if features can be generated lazily, i.e., only when required by a query. We do not explore this solution in this paper.

Sound Energy Detection We also implement features from audio. For instance, an audio feature from audio2 computing the average sound energy of each clip is expressed as:

```
create type sdFeature { integer sd_sn primary key,
            time sd_bt, time sd_et, double sd_energy };
create feature stream sdFStream3 of sdFeature on audio2
with   sd_sn:=getFrameNum(clip_num)
       sd_bt:=getFrameTime(clip_bt)
       sd_et:=getFrameTime(clip_et)
       sd_energy:=getSoundEnergy(clip);
```

Complex Feature Generation A feature can be derived from other existing feature(s). Moreover, we may not want to compute a feature from every deriving tuple, but only in some interesting periods. For example, we compute a speech length feature (slFeature) from an audio for each period when someone is speaking more than a 5-clip duration:

```
create type slFeature {time sl_bt, time sl_et, integer sl };
create feature stream slFStream4 of slFeature on sdFStream3
with   sl_bt:=getPeriodStart()
       sl_et:=getPeriodEnd()
       sl:=getPeriodCount();
when   CONTINUE(sd_energy > 25.0) > 5
```

A slFeature is computed during each period (P) only if a speech lasts more than 5 audioclips. Each P is computed from the when subclause by an *outer period predicate*, which uses a period detection operator CONTINUE to count the number of feature tuples continually qualifying the *inner predicate*. The FGFs (e.g., getPeriodCount) then use the P detected at run-time to compute each feature attribute value.

3.3 Querying MF Streams

We have designed a query language, MF-CQL, which is CQL [1] extended with additional syntax and shortcuts to express the extended semantics beyond DSMS. MF-CQL allows flexible expressions for period detections and period aggregations, which deliver higher levels of expressive power for media query applications. Considering the real-time requirement of most media stream applications, MedSMan's operations are triggered by each arriving (media or feature) tuple per time. For most media streams queries, query predicates are based on their derived feature streams. Then the qualifying fragments of the feature streams are mapped back to their deriving media stream fragments by using *sqno* or interval attributes. We design syntax for efficiently expressing these queries. The query language details will be introduced through various query examples in Section 5.

3.4 System Architecture

Figure 2 shows the overall system architecture. A user creates definitions of media streams and declares their instances via MSDL. The raw streams captured from sensors are processed and transformed to user-defined media streams, which are transmitted to both the query engine and the feature generation component, where various feature streams are automatically generated via FGFs defined in FSDL. The media streams and derived feature streams are queried by the query engine, which outputs the qualifying media portions to the composition component. Finally, the composed media results are returned to the user.

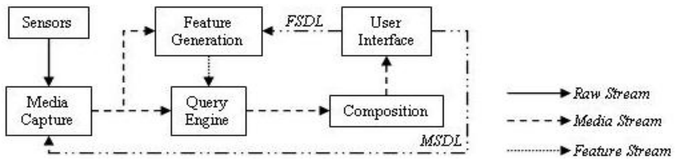

Figure 2: MedSMan system architecture.

4. IMPLEMENTATION

We have designed and prototyped the architecture of media-feature stream declaration and generation, whose dataflow is shown in Figure 3. A media tuple definition T_M is parsed, validated and generated as a *TupleDesc*, which is inserted into a global *Tuple Desc Manager*. Basing on a *TupleDesc*, a media stream instance is declared via *CreateMediaStreamStmt*, generating a *MediaStreamDesc*, which is added into a global *stream manager* and associated with a *media queue manager* controlling a *media tuple queue* (MTQ). Tuples of a defined media stream are generated via the following steps:

1. Capture raw stream from a sensor by a capture thread. Sensor specific parameters $\{S_T, S_S, \bar{P}\}$ defined in MSDL are used during each sensor capture thread setup.
2. Particular *PreAccessCodec* dependent on M_T is applied to the raw stream to get an individual media track.
3. A customized *cutIntoMediaTuple* function working on a media track populates a tuple's attributes defined in T_M.
4. Each generated media tuple is sent to a *media queue manager*, which then inserts it into a corresponding MTQ.

A feature stream instance based on a feature tuple definition T_F is created in similar procedures. Each *createFeatureStreamQuery* contains a *featureStreamID*, a deriving *mediaStreamID* or existing *featureStreamID*, and a sequence of triples of <featureAttr, mediaAttr, function> parsed from $\bar{F_G}$ defined in FSDL. Then, a *FeatureStreamDesc* is created and associated with a *feature queue manager* controlling a *feature tuple queue* (FTQ). It registers to the deriving mediaStreamDesc, thus setting up the mapping between media and feature. Triggered by a registering *media queue manager* with every new media tuple, the *Feature Generator* generates new feature tuples and inserts them into corresponding FTQ. All concurrent threads are implemented in a "producer-consumer" fashion. Each thread triggers its follower(s) as it is triggered by its parent. When a media

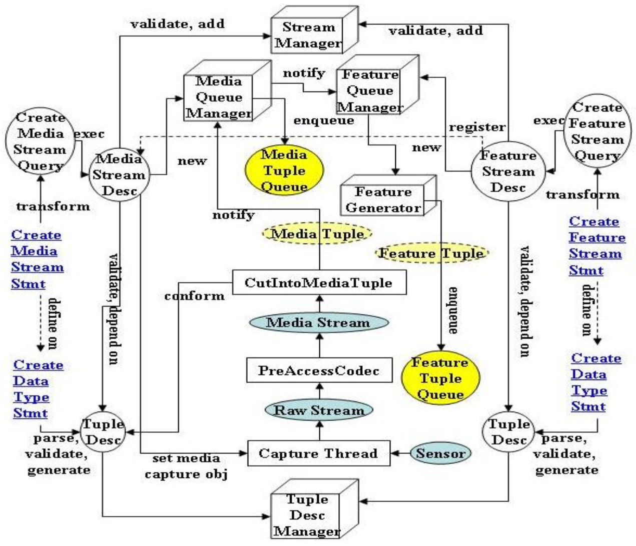

Figure 3: The dataflow of MF-Stream generation.

queue manager gets a media tuple, it will poll all its registered featureStreamDescs, each of which can join and leave its registering mediaStreamDesc's media queue manager at any time. For experiments in this paper, MF streams are captured and generated on a same machine as the central query processor. A more efficient implementation is to use a distributed system (*MediaBroker* [8]) for MF stream generation, and transfer streams to a central query machine via Sockets. The prototype of MedSMan is implemented using Java (JDK 1.5.0). We use APIs provided by Java Media Framework (JMF2.1.1) and OpenCV (integrated with Java based query engine via Java Native Interface (JNI)) for real-time audio/video capturing and feature generating.

5. EXPERIMENTS AND DISCUSSION

Experimental Set-up We run a number of query examples varying in media streams and feature streams, thus evaluate performances for different query types, in terms of query delays. Our experiments run on a XP machine with dual $2.4GHz$ CPUs and $2GB$ RAM. Basing on the instances of video (with the format of RGB, 400x320, 10fps, Length:230400, 24-bit), audio (with the format of LINEAR, 44100Hz, 16-bit,Stereo, 2Channels) and the derived feature streams defined in Section 3, we evaluate the following queries:

Q1: select content from video1, odFStream2 where od_pixel>500;
Q2: select content from video1, fdFStream3 where fd_num>0;
Q3: select content from video1, mvFStream1 where mv_pixel>1000;
Q4: select clip from audio2, sdFStream3 where sd_energy>32.0;
Q5: select content from video1, fdFStream3, odFStream2 where fd_num=1 and od_pixel>1000;
Q6: Select content from video1, mvFStream1, audio2, sdFStream3 where mv_pixel>5000 and sd_energy>32.0;

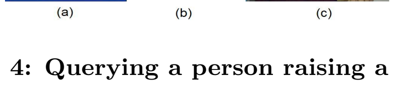

Figure 4: Querying a person raising a book.

Q1 to Q4 are single selection queries against four features, respectively. Q5 is a join query on two features from one media – a face detection feature shown in Figure 4(a) and an object detection feature with a reference image shown in 4(b). One qualifying output video frame is shown in 4(c). Q6 is a join query using two features over two media streams.

Delay Sensitivity The media tuple intervals, FCDs and query delays for Q1 to Q5 are shown from (a) to (e) in Figure 5, respectively, while their averages are listed in Table 1. These values vary with media tuple sqno, sometimes radically. Obviously, FCDs play a significant role in determining dynamic tuple intervals and total query delays. In Q6 (shown in Figure 5(f)), each video frame overlaps with multiple sequential audio clips. The average delay of the video frames is $393.1ms$, and the average *max* delay of the audio clips is $622.5ms$ for the oldest one waiting in queue, since an optimization is implemented by making a slower feature (mv) tuple trigger the faster feature (sd) tuples waiting in queue, to reduce an extra *Scan* operator and queue buffers.

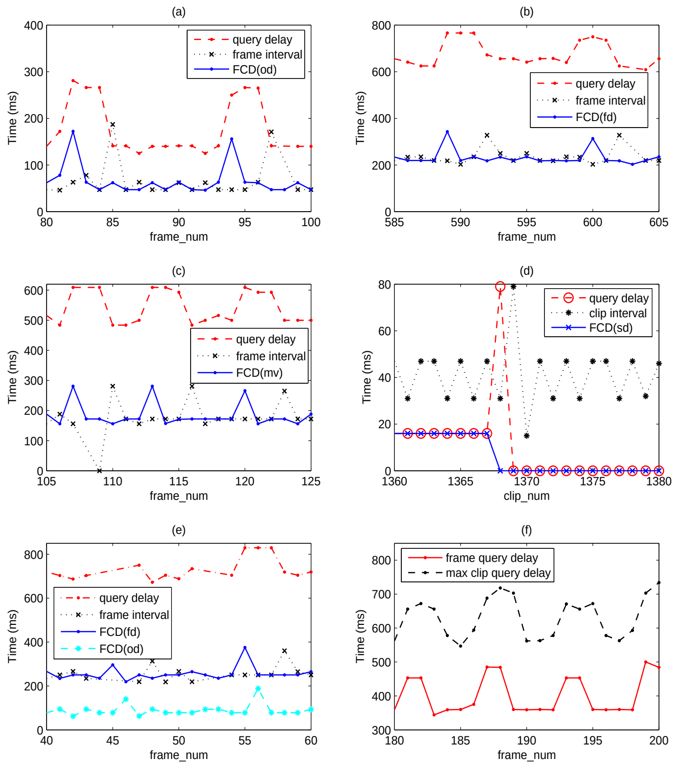

Figure 5: Query delays.

6. CONCLUSIONS

This paper presents our approach to dealing with continuous querying over live heterogeneous media streams by effectively combining extendible digital processing techniques with a general media stream management system. To bridge the two different areas, a unifying data model for media and feature streams is designed, along with description languages managing media and feature streams and a query language manipulating complex queries. A prototype of our system has been implemented with a number of audiovisual features. A key advantage of our system is that it is open to any user-defined feature generation; thus its querying power can evolve as the DSP techniques advance. Due to the increasing use of media data in many emerging applications, we believe this is an area of significant interest to researchers in stream data as well as multimedia data. In the near future, we will investigate scalability issues, considering more complex features derived from multiple media or existing feature streams out of a large number of distributed sensors. We also plan to work on composition operation over multiple heterogenous medias, and investigate more factors, such as media transmission delays and compression/decompression, that may affect system performance.

Table 1: Delay sensitivities (ms)

Query/Feature	AVG interval	AVG FCD	AVG query delay
Q1/od	72.2	66.4	165.6
Q2/fd	239.1	237.3	685.9
Q3/mv	186.1	181.8	543.1
Q4/sd	39.99	5.89	9.11
Q5/od Q5/fd	261.5	89.9 259.3	740.6

7. REFERENCES

[1] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. Technical report, Stanford University, Oct. 2003.

[2] W. G. Aref, A. C. Catlin, and et al. Vdbms: A testbed facility for research in video database benchmarking. *ACM Multimedia Systems Journal, Special Issue on Multimedia Document Management Systems*, 9(6):575–585, June 2004.

[3] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: The qbic system. *IEEE Computer*, 28(9):23–32, Sept. 1995.

[4] L. Golab and M. T. Ozsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, June 2003.

[5] A. Gupta, B. Liu, P. Kim, and R. Jain. Using stream semantics for continuous queries in media stream processors. *ICDE Demo*, April 2004.

[6] S. Hongeng, R. Nevatia, and F. Bremond. Video-based event recognition: activity representation and probabilistic recognition methods. *Computer Vision and Image Understanding (96)*, 2:129 – 162, November 2004.

[7] R. Jain. Experiential computing. *Communications of the ACM*, 46(7):48 – 55, July 2003.

[8] U. Ramachandran, M. Modahl, I. Bagrak, M. Wolenetz, D. Lillethun, B. Liu, J. Kim, P. Hutto, and R. Jain. Media broker: A pervasive computing infrastructure for adaptive transformation and sharing of stream data. *To appear in Pervasive and Mobile Computing (PMC) Journal*, 2, 2005.

[9] P. Salemier and J. R. Smith. *Introduction to MEPG-7:Multimedia Content Description Interface*. Wiley Europe, 2002.

[10] R. Steinmetz and K. Nahrstedt. *Multimedia computing, communications and applications*. Prentice Hall, 1995.

[11] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 15(3):555–568, May-June 2003.

[12] V. V. Vinod and H. Murase. Video shot analysis using efficient multiple object tracking. In *ICMCS '97*, Ottawa, Ontario, CANADA, June 1997.

[13] P. Viola and M. J. Jones. Rapid object detection using a boosted cascade of simple features. In *IEEE CVPR*, volume 1, pages 511–518, 2001.