Using Temporal Semantics for Live Media Stream Queries

Bin Liu[1], Amarnath Gupta[2] and Ramesh Jain[3]

[1] School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta GA 30332, USA
[2] San Diego Supercomputer Center, University of California San Diego, La Jolla, CA 92093, USA
[3] Department of Computer Science, University of California Irvine, Irvine, CA 92697, USA

Abstract. Querying live media streams is a challenging problem that becomes an essential requirement in a growing number of applications. We address the problem of evaluating continuous queries on media streams produced by media sources such as webcams and microphones. The temporal attributes and the order of stream tuples play essential roles in live stream generation and query execution. Furthermore, the temporal constraints and query semantics of related streams provide additional query optimization opportunities. We investigate the modeling issues and introduce the query processing techniques of a live media stream management system (MedSMan), including media capturing, automatic feature generating, declaration and query languages, temporal stream operators and querying algorithms. A prototype is implemented and we present experimental results to show the performance of our prototype using various real-time media experiments.

1 Introduction

In recent times, processing continuous queries over live and unbounded streams has become a major research area in data management. A number of research groups are developing data stream processing systems for a wide variety of problem domains including network data management, traffic monitoring, business data analysis, environmental sensor networks and immersive environments. At the same time, the proliferation of various sensor devices (e.g., webcam, microphone, RFid, etc) has fuelled many applications utilizing massive media streams in a unifying way. We consider the problem of continuous querying on multiple live media streams, by taking advantages of automatic and real-time multimedia information processing techniques. The basic premise behind delivering multimedia information is that while each individual media channel contains some information, it is the synchronous combination of the channels that captures the intended semantics of the content. 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 by different sensors and streamed to a central processor. This makes live multimedia stream query systems distinct. First, their primary problem is to effectively combine multiple media streams as well as auxiliary non-media information to answer standing queries about the situations observed by the media sensors. For example, consider a professional conference room equipped with multiple cameras and microphones capturing the activities of both the speaker and audience. A remote user wants to connect to the speaker's video only when he or she talks about "multimedia database". This is different from research issues in the standard image, video and audio database systems which centers around similarity queries and problems like scene detection and shot segmentation using different features. Second, unlike alphanumeric symbolic streams, media streams often cannot be directly queried. Instead, queries on them are evaluated by computing feature streams from them. The data model for media streams need to capture this media-feature dependency. Third, real-life applications often specify their queries in terms of events occurring over intervals of time (i.e., an interval event with a start and a end time). The events need to be expressed in terms of the underlying media streams as well as the derived feature streams.

1.1 Prior Work

There has been considerable prior work on data stream management systems (DSMSs), such as OpenCQ [1], NiagaraCQ [2], Aurora [3], Telegraph [4], COUGAR [5], and STREAM system [6]. A number of research issues of DSMSs have been received significant attentions, including data models, continuous query semantics, query languages, blocking operators, memory requirements, cost metrics and statistics, approximations, adaptivity and query optimizations, and scalability issues. Much of this earlier work has focused on alphanumeric symbolic streams, while live media streams have received less attention - due to the heterogeneity of multimedia and tremendous on-line processing costs (in terms of time, memory and CPU). However, advances in multimedia information systems and digital signal processing techniques are decreasing the media processing costs and making the queries of live media streams viable. As a result, there is a need for a unifying data stream management system effectively combining extensible digital processing techniques and the general DSMS research.

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. We previously studied the media stream generation, as well as feature function implementations (especially for feature streams derived from single media or feature stream) in [7]. Considering the *interval* nature of media tuples, we do not use window based approaches [8], but a per-tuple triggering approach. Temporal sequence research has developed sequence models [9] and access modes (e.g., stream-probe) [10] for accessing sequences. But our problem is different since for any tuple in one stream, there are no fixed corresponding tuple(s) in another stream. Therefore, we treat all joining streams equally in accessing and triggering operations. We cannot directly use many efficient algorithms developed for temporal join, such as RI-tree-based [11], partition-based [12, 13], and index-based [14, 15] algorithms, because media streams have very high arrival rates compared to typical relation updates. The construction overhead of these algorithms triggered by each tuple arrival may not be offset by the benefit of using them. Further, we studied event modelling and the related environment modelling issues in [16]. In this paper, we focus on stream processing techniques by utilizing temporal constraints and query semantics between related media and feature streams, particularly for the purpose of query optimizations.

1.2 Example

We use a typical live media query example for illustration throughout this paper. Consider a surveillance application in which both live video and audio are used to automatically detect potential intrusions. A video stream ($video1$) is captured by a camera and an audio stream ($audio2$) is produced by a microphone. If abnormal movements occur in $video1$ and abnormal sounds occur in $audio2$ at the same time, a possible intrusion is identified and the corresponding video frames should be displayed.

1.3 Outline

The rest of the paper is organized as follows. Section 2 discusses the modelling issues of media and feature streams. Section 3 presents the query techniques of our system, including query languages, stream operators, query execution, cost metrics and optimizations. A brief introduction of system implementation is introduced in Section 4. We run a number of real-time media stream queries and analyze the results in Section 5. Finally, we conclude our work and future research in Section 6.

2 Media and Feature Stream Model

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_s, t_e] (for a time-point attribute $t_s = t_e$, a single time point is represented as t_d), and any other media or feature attribute. We give a series of conventions and definitions that make more precise the notions of media and feature streams.

Convention 1 *A sequence T, is said to be* ***continuously well-ordered*** *iff (1) T is well-ordered, and (2) for each time-unit $(t_{si}, t_{ei}]$, there must be one and only one directly following time-unit $(t_{si+1}, t_{ei+1}]$ in T, where $t_{ei} = t_{si+1}$. 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 sequence T is said to be* ***discretely well-ordered*** *if and only if T is well-ordered, i.e., the continuity clause does not apply to 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 sequence of tuples, each consists of a sequence number (m_{sqno}) uniquely identifying its position in stream, a pair of start and end timestamps $(t_s, t_e]$ whose domain is a continuous time set, and a media valued attribute v_m valid only during $(t_s, t_e]$.*

Definition 2. *A* ***discrete stream*** *is a sequence 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 sequence 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.*

For example, a pixel movement detection is derived from two consecutive video frames. Table 1 shows a feature tuple is derived from two media tuples of same media stream.

Table 1. A feature can be derived from two media tuples

$f_{sqno}(k)$	$v_f(k)$	$t_f(k)$	$m_{sqno}(i)$	$v_m(i)$	t_{bi}	t_{ei}
			$m_{sqno}(j)$	$v_m(j)$	t_{bj}	t_{ej}

Note the start and end time of a tuple are the *valid* time defined in temporal database. In particular, 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. In the above example, the representing interval of $f_{sqno}(k)$ derived from $m_{sqno}(i)$ and $m_{sqno}(j)$ is $(t_{bi},\ t_{ej}]$.

2.2 Special Querying Issues

A number of unique issues should be considered in designing a general data model for time-based media and feature streams.

Time Attributes and Order Our definitions in previous section require explicit time attributes in both media and feature streams. Stream tuples are generated and queried not in isolation but in synchronization. Further, we assume all tuples in a stream are either continually well-ordered by intervals or discretely well-ordered by time-points. Note tuples from different streams may not be totally ordered, but partially ordered. For the presence of explicit time attributes, tuples in a relation produced from a stream by sliding window operators are also ordered in time, rather than a bag of unordered tuples. As we will present in later sections, the order of tuples is necessary to guarantee tuple continuity for queries over intervals.

Stream Uncertainties Uncertainties exist in media stream capturing, feature stream generating and stream querying. Figure 1 shows a timing diagram of synchronization among related media and feature streams. The notations in the figure are defined as follows:

$t^R_{s(k)}$: The start timestamp of the k-th tuple in stream R.
$t^R_{e(k)}$: The end timestamp of the k-th tuple in stream R.
$T^R_{(k)}$: The interval of the k-th tuple in stream R.
$v^R_{(k)}$: The timestamp indicating when the k-th tuple in stream R is generated.
$D^{R,S}_{(k)}$: The generation delay of the k-th tuple in stream R which is derived from stream S.

In this paper, we assume a feature stream is derived from a single media stream and a tuple in a derived stream has the same sqno as its source tuple in a deriving stream, thus setting up the mapping between two streams. One the one hand, the intervals ($T^R_{(k)}$) of the deriving streams (e.g., media streams M and N) are variable. On the other hand, the generation delays ($D^{R,S}_{(k)}$) of tuples in the derived streams (e.g., feature streams X and Y) are not constant, which depend on the transmission delays and computation delays. In the extreme case, if a deriving tuple takes too long to generate its derived tuple, it might affect the following tuple(s). For example, the $(i+1)$-th tuple in stream N has to be skipped since it arrives before its previous tuple finishes the i-th feature tuple generation for stream Y. As a result, the derived tuples in feature stream Y are not continual. This non-determinism nature of media and feature streams must be addressed by query synchronization.

Query Synchronization In most multimedia applications, especially for audio-visual applications, the synchronization between different media streams needs to be precise to satisfy perceptual continuity, when viewed by the human user. Synchronization is required not only in media composition but also in many other phases throughout the entire media stream query processing.

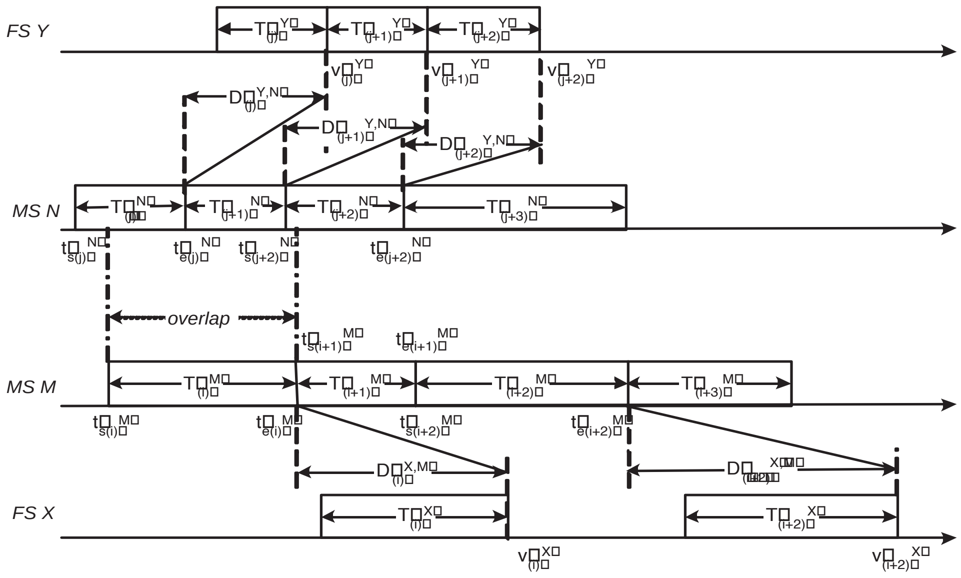

Fig. 1. A timing diagram of synchronization among media and feature streams.

Live media query processing contains multiple sub-processings (threads), such as media capturing, feature generating and stream querying, thus forming a "producer - consumer" relation. Each consumer thread depends on the results of its producer thread(s). Because of the uncertainties of streams, specific operations performed on some tuples in a thread may cost longer than the expected times of both its producer and consumer threads. In order to preserve the continuity in stream tuple order, we can implement these threads through a *blocking approach*, i.e., a producer thread does not send new input to its consumer until the consumer finishes processing the previous input. By this way, we may slow down the data rate of the producer. An alternative is a *non-blocking approach*, i.e., to skip some inputs that cannot be processed by its consumer (e.g., the $(i + 1)$-th tuple in stream N is skipped in Figure 1).

Due to the unbounded nature of streams, continuous queries over streams are often defined in terms of sliding windows, either tuple-based or time-based [8]. Nevertheless, such windows do not take the tuple intervals into account. In those applications, a tuple (e.g., temperature reading) is instant-based, rather than interval-based. This is not true for typical media stream tuples, and may have problems (e.g., false join) in joining tuples with non-overlapping intervals. A loose window including multiple non-overlapping tuples cannot satisfy the strict temporal join constraint for media or feature tuples from different streams. Instead, we require a more precise and strict metric to join them. We apply an overlap join (*O_Join*) TSJ_1 defined in [17]: *All participating tuples that satisfy the join condition share a common time point.* Tuples whose temporal attributes overlap in time have the *highest temporal relevance*, thus can be joined. Those non-temporal attributes of tuples can only be joined after satisfying the premise that the corresponding temporal attributes are overlapping in time. As shown in Figure 1, the i-th tuple in stream M can be overlap joined with three tuples in stream N, i.e., the j-th, $(j+1)$-th and $(j+2)$-th.

In a query plan involving more than one unsynchronized input streams, it is essential to decide which one(s) act as the trigger stream to execute the query plan. In general, we can make every input stream as the trigger such that each of their arrival tuple will trigger the query execution. By this way, the query delay may be reduced. However, as we show in later section, this is not absolutely true in all cases, due to the complex constraints and synchronization among related streams and threads. Instead, there are scenarios where a “master-slave” approach is preferred to reduce the overall query cost (including query delay, CPU, memory, etc), i.e., to select one input stream as the master stream triggering the query execution and treat all the other input streams as salve streams tuning to the master stream. In particular, this approach allows to reduce unnecessary high-cost feature tuple generations and the overall performance improvement is significant.

Stream Operator Complexities The stream based operators are quite different from the traditional record based operators. In many cases, it is the responsibility of a stream operator to determine tuple(s) from deriving stream(s) as its input to generate the corresponding output tuple(s) at a specific time. In addition, the uncertainties of tuple arrival and tuple generation cost bring even more complexities to stream operators.

A feature stream is produced by one or more stream operators (called *transformers*) operating on one or multiple related media or other feature streams. Feature streams are complex in terms of tuple values, deriving media or feature streams, feature tuple interval semantics, and generation costs. Moreover, fragments from multiple media streams can be composed to form a media stream of a new format. In general, a stream operator should take N ($N \geq 1$) input (media or feature) stream(s) and generate a new stream of a format defined by users. Our definition of stream operator also applies to the standard query operators, such as selection, projection, join, etc.

The stream operator design and implementation depend on users' specific requirement. Due to the heterogeneity of media streams and different application needs, it is desirable for a querying system to be extendable to user-defined operators, i.e., the system allows users to design and implement their own transformers and plug codes into the querying system [18].

3 Query Processing

3.1 Media and Feature Stream Description Languages

We have designed a Media Stream Description Language (MSDL) and a a Feature Stream Description Language (FSDL) for media stream capturing and feature stream generating [7]. In our example, there are two media streams. First, *video*1 is captured from a webcam connected to a local port (vfw://0) with a data rate of 10 frames per second (FPS):

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

Second, *audio*2 is captured from a local port (dsound://) with audio clip buffer size of 40ms:

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

Different media streams may have different sensor-dependent initialization parameters. For example, *video*1 is defined with a data rate, while *audio*2 is defined with a clip buffer size.

Further, we can generate one movement detection feature stream from *video*1 and a sound detection feature stream from *audio*2:

```
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
with   mv_sn:=getFrameNum(frame_num)
       mv_bt:=getFrameTime(frame_bt)
       mv_et:=getFrameTime(frame_et)
       mv_pixel:=getMovementNum(content);

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);
```

Note any feature tuple in both feature streams is derived from a single media tuple of the deriving media stream.

3.2 Query Expression

We also designed a query language, MF-CQL [16], extended from CQL [19]. Our query example can be issued as:

Query 1 Select content From video1, mvFStream1, audio2, sdFStream3 Where mv_pixel > 5000 And sd_energy > 32.0;

The logical query plan is shown in Figure 2, where:

mv: Movement feature transformer for video;
sd: Sound feature transformer for audio;
FSel: Selection operator for feature stream;
Proj$_X$: Projection operator filtering out attribute(s) X;
MFetch: Map operator fetching media tuples via corresponding feature tuples;
Fbi-Join: Binary overlap join operator for two feature streams.

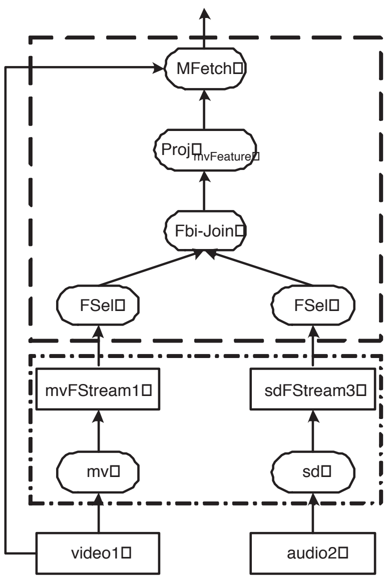

Fig. 2. Query plan example.

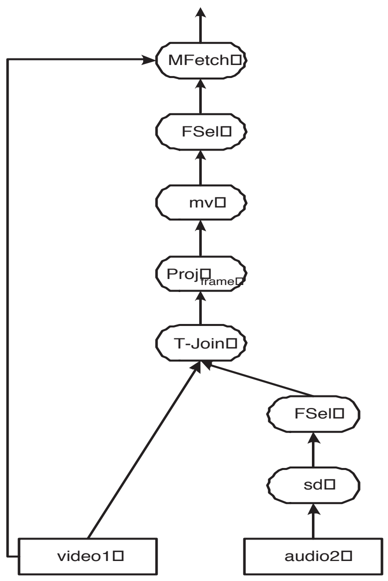

Fig. 3. Optimized query plan.

3.3 Cost Model for Operators

Table 2. Notation for cost

Notation	Description
N_{comp}	Comparing number for pairing two tuples
C_{comp}	Unit cost of one comparison
$\lambda_X(t)$	Arrival rate of stream X at time t
$N_{X,Y}(t)$	Number of tuples in stream Y to be paired by a tuple in X at time t

Queue length for MFetch A MFetch operator is used to inactively pull the buffered tuple(s) in its queue at a time, triggered by the corresponding feature tuple(s). The queue length for a stream R at time t is:

$$L_Q(t) = \int_{t-D(t)}^{t} \lambda_R(\tau)\, d\tau, \tag{1}$$

where t is the time when a target tuple is retrieved, and $D(t)$ denotes the delay from the arrival time of the target tuple to t. Note our implementation maintains a common media/feature tuple queue for each media/feature stream, and sets a queue length to maximum satisfying all MFetch operators associated with a given media stream.

O_Join Cost A traditional, cardinality-based cost model is incapable of producing cost estimates of join over unbounded streams, and a unit-time basis cost model as a new metric should be proposed [20].

The selection of temporal attributes used in O_Join (including T-Join) depends on the types of joining streams in a join query:

Case(1): If the O_Join operates on two feature streams derived from a same media stream, the comparing attribute is *sqno*, thus $N_{comp} = 1$;
Case(2): Otherwise, the comparing attributes are both the *start* and *end* timestamps of interval, thus $N_{comp} = 2$.

A unit-time based cost formula of O_join between streams R and S during period $[t_1, t_2]$ is:

$$C_{R\bowtie S}(t1,t2) = \frac{C_{comp} \times N_{comp}}{t2 - t1} \times \int_{t1}^{t2} [\lambda_R(t) N_{R,S}(t) + \lambda_S(t) N_{S,R}(t)]\, dt, (t2 > t1). \tag{2}$$

The cost of O_m*-Join* operator joining m $(m > 2)$ streams is:

$$C_{\bowtie m}(t1,t2) = \frac{2C_{comp}}{t2 - t1} \times \int_{t1}^{t2} \sum_{i=1}^{m} \sum_{\substack{j=1 \\ j \neq i}}^{m} \lambda_{S_i}(t) N_{S_i,S_j}(t)\, dt. \tag{3}$$

Query Delay For one qualified 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 delay is both media stream dependent and individual tuple dependent, because (1) different media tuples have different tuple extents and *feature computation delays* (FCDs); (2) a particular media tuple may be joined with multiple tuples from other streams; thus (3) different tuples in one stream joined with a common tuple in another stream may have different delays.

3.4 Query Optimization

By utilizing the temporal constraints and query semantics among media and the derived stream streams, we can apply a number of optimizations.

Reverse Order in O_Join

According to equations (2) and (3), we should minimize the number (NX,Y(t)) of tuples to be paired to reduce the join cost. The ordering nature of sqno and interval timestamps provides optimizations for join operator. The O_Join is a merge scan join by using the ordered sqnos and interval timestamps. This is similar to Temporal Equijoin (TEJ-1) algorithm [21]. Rather than using ascending order in TEJ-1, our implementation takes the descending order, because each new arrival tuple (i.e., trigger tuple) has the latest sqno and timestamp. We pair it with every tuple in the other stream queue from rear to front, i.e., in a reverse order, and terminate pairing at the first tuple that can not be overlap joined in time; thus we minimize the comparison number.

Push Down T-Join The query plan shown in Figure 2 has two sub-processes – the feature generating and the stream querying. One problem of this approach is that the feature generating thread always runs and is independent of the following querying thread, i.e., whether a feature tuple is really used or not in the querying subprocess, it is always computed. According to our experiments, the feature computation cost (in terms of delay, CPU and memory) is the most significant factor in overall query cost. Nevertheless, the temporal constraints between related streams, together with application semantics, provide optimization opportunities in many scenarios. In our example, The sound feature generation costs much lower than the movement feature generation. The movement feature is only necessary when the predicate of sound feature is qualified (normally only in a small portion of time the abnormal sound is detected). Upon these observations, we develop a optimization rule by using T-Join (a special form of O_Join only joining on time attributes) as follows:

Rule 1 *Push down temporal join* (T-Join)*:*
Proj_f (Sel(FeatTran(R)) Fbi-Join S) =Sel(FeatTran(Proj_m(R T-Join S))),
where FeatTran *is the feature transformer from media stream R of type m to feature type f, and S is another join (media or feature) stream.*

The rule implies that other operators are processed after applying temporal join operator T-Join first to reduce the number of unnecessary operations, especially the high-cost feature generations. This rule is especially useful in the cases where one feature transformer is expensive while another one is much cheaper. We apply this rule to our example and have an optimized logical query plan shown as Figure 3.

3.5 Query Execution

Figure 3 shows there is no explicit feature stream generating thread. Insteand, the feature generation operators are integrated into the stream query processing thread. This integration not only simplifies issuing queries (i.e., reducing the declaration of feature stream generation), more important, it allows performing optimizations covering every phase of media stream querying in a unifying way.

Note the node of *video*1 can be split into two node instances (i.e., creating two pointers to same memory address); thus we get a tree structured query plan, called *transformation tree* (TT). By taking a post-order serialization, we can get a *serialized transformation tree* (STT) consisting of tree types of nodes:

Data node: It is an input stream and a leaf node of TT. It has a queue buffering tuples at this node at a time.

Transformer node: It is one of the various operators inside TT. It takes one or multiple data nodes as input and produces a new data node of any format.

Number node: It is the number N ($N \geq 1$) of data nodes that a transformer node takes as input.

For example, the STT of Figure 3 is serialized as:
Ex1:

@video1, @video1, @audio2, \$sd, #1, \$Sel$_{sd}$, #1, \$T-Join, #2, \$Proj$_{mv}$, #1, \$mv, #1, \$Sel$_{mv}$, #1, \$MFetch, #2;

where $@, \$$ and # are the token symbols used in our query parser indicating the data node, transformer node and number node, respectively.

The algorithm to execute the STT is:

```
List SST = makeCopy(SST0);  // make a new instance for each run
while (SST is not empty)
    Node node = STT.getFirstNode(); //will remove node from SST
    if (node is a DataNode)
        DataNode dataNode = (DataNode) node
        stack.push(dataNode);
     else  //must be a Transformer Node
        TranNode tranNode = (TranNode) node;
        Node node1 = STT.getFirstNode(); //must be a Number Node
        int num = ((NumNode) node1).intValue();
        DataNode dataNode_im = new DataNode(); //intermediate result
        for (int = 1 to num)   // get num Data Nodes
            Node node2 = stack.pop();
            inputDataList.add((DataNode)node2);
        dataNode_im = TranExec(tranNode, inputDataList); //transform
        stack.push(dataNode_im);
DataNode result = (DataNode) stack.pop();
```

By default, every new arriving tuple in *video*1 or *audio*2 can trigger the query plan. We notice *video*1 is a direct child of T-Join. A fact is that if there is no tuple available in the queue of *audio*2 at a time, it is unnecessary for a new arriving *video*1 tuple to trigger the query plan, since the T-Join will not produce any output at this time. Therefore, a "master-slave" approach is preferred to reduce the number of triggering streams. Only *audio*2's new arriving tuples need to trigger the query plan, and *video*1's tuples just wait in queue for being processed by T-Join when triggered by *audio*2.

4 Implementation

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 the Java based query engine via Java Native Interface (JNI)) for real-time audio/video capturing and feature generating. The stream description and query languages are implemented using Java Compiler Compiler (JavaCC).

Our implementation consists of two major components, including media/feature stream generation [7] and stream querying execution. The former permits a designer to directly capture various live data streams from different sensor devices, and form media streams consisting of logical media tuples. Then, more meaningful feature streams can be automatically derived from media streams for the query purpose. Stream queries are parsed and generate physical query plans. MedSMan runs physical plans using an individual tuple triggering approach for media and feature stream query execution, thus reduces query delays. The query execution of the two sub-processes approach (see Figure 2) is presented in [16].

We implement the SST algorithm and provide an open system which enables users to design, implement and upload special operator codes fulfilling their application requirements. These codes are dynamically loaded at run-time [18]. This goal is achieved by using Java's dynamic class loader capabilities. In addition, we also support a language structure allowing users to specify the master stream in the join query for query optimizations. More important, by integrating a set of related individual operators and using our SST algorithm, users can design "super" operators that can take multiple media streams of any formats as input and perform the special transformation or query functionalities.

5 Experiments and Analysis

We run a number of query examples on multiple live media and feature streams, and evaluate performances for two different implementing approaches. We investigate FCDs, query delays, tuple queue and optimization effects. Our experiments run on a XP machine with dual $2.4GHz$ CPUs and $2GB$ RAM.

5.1 Blocking Approach

We begin with the blocking approach which creates a strict "producer-consumer" relationship among media capturing thread, feature generating thread and stream querying thread. We examine the FCDs and their impacts on the deriving media streams. With the media and feature stream defined in Section 3.1, we issue two examples performing single feature queries as follows:

Query 2 Select content From video1, mvFStream1 Where mv_pixel > 5000;

Query 3 Select clip From audio2, sdFStream3 Where sd_energy > 32;

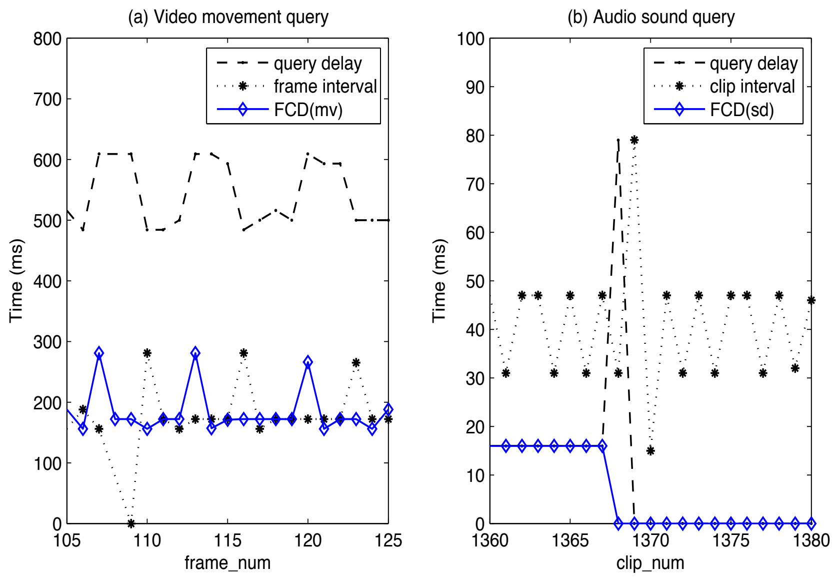

Fig. 4. Performances of single media single feature queries.

The average frame interval, FCD(mv) and query delay for Query 2 are 204.2 ms, 199.7 ms and 463.3 ms, respectively. Figure 4 (a) shows they vary with the video frame number. The average interval is much greater than the expected value, i.e., 100 ms, since we use the blocking approach and the average FCD(mv) is greater than the expected data rate of $video1$. The video thread only produce a new frame after the previous one is computed by the mv transformer. As a result, the total query delay is accumulated and significant. In Query 3, the average clip interval, FCD(sd) and query delay are 39.99 ms, 5.89 ms and 9.11 ms, respectively. These metrics are much better than those of Query 2, because the average FCD(sd) is quite small compared to its audio clip interval (i.e., 40 ms), although there are a few big jitters as shown in Figure 4 (b).

Then, we perform Query 1 which joins two feature streams derived from two different media streams. Figure 5 (a) shows the query delays for each video frame and each audio clip. Note one video frame overlaps with multiple sequential audio clips (similarly, one audio clip may overlaps with one or two video frames). The average delay of the video frames is 393.1 ms, and the average max delay of the audio clips is 622.5 ms for the oldest one waiting in queue, since an optimization is implemented by making a slower feature (i.e., mvFestream1) tuple trigger the faster feature (i.e., sdFStream3) tuples waiting in queue, in order to remove an unnecessary trigger stream. Figure 5 (b) shows the varying length of sdFStream3's queue with an average of 13.9, which indicates the necessary size of buffering memory of $audio2$ for this query.

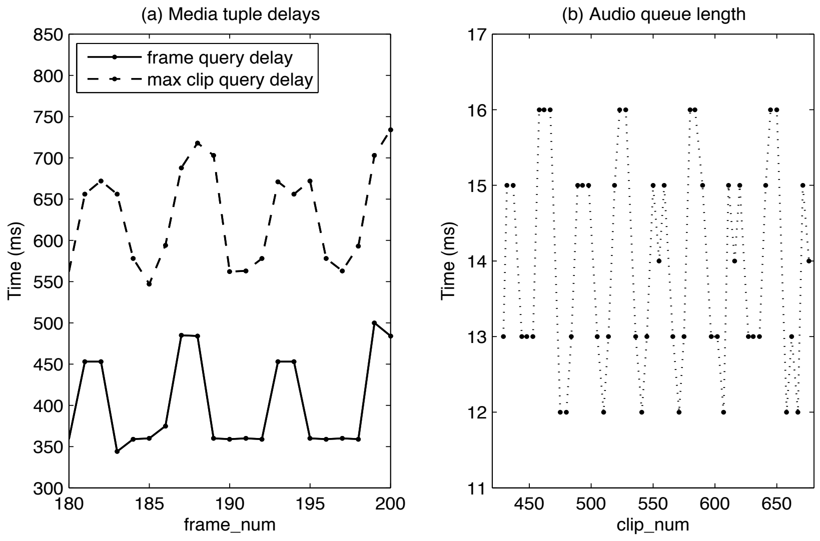

Fig. 5. Performances of video and audio join query.

The above experiments and results show FCDs play a significant role in determining dynamic tuple intervals and total query delays. However, we can utilize the temporal constraints and query semantics between streams to reduce the number of unnecessary feature generations, thus improve the overall performance.

5.2 Non-blocking Approach

We perform Query 1 by executing the serialized tree expressed in Ex1 through the non-blocking approach. In addition, we indicate $audio2$ as the master stream for the purpose of optimization. We are interested in the performance of T-Join operator which plays a key role in the query execution and optimization. In order to avoid the outside performance affects from other operators, we assume the selectivities of both Sel_{mv} and Sel_{sd} are 1. An important metric discussed here is the $selectivity(video1)$, i.e., M/N, where M and N are the number of frames entering and leaving the query plan tree, respectively. T-Join determines this metric value when assuming the selectivities of related selection operators as 1. Besides, we define an output window W_{T-Join}, which is the maximum number of tuples that T-Join can output at a time. This metric implies that if the T-Join produces more than W_{T-Join} tuples, only the latest W_{T-Join} tuples are output and the other earlier tuples are skipped. The necessity of W_{T-Join} is explained as follows.

From Figure 3, the qualified M ($M \geq 1$) tuples out of T-Join are projected and then sent to feature transformer mv, which performs the feature generation for every tuple. If FCD(mv) is greater than the expected frame interval or M is large, mv will be a blocking operator which blocks the entire query plan execution for a period of $SUM_{FCD} = \sum_{i=1}^{M} FCD_i$. During a blocking period, $video1$'s new arriving tuples are buffered in queue but not processed. However, if SUM_{FCD} is too large, the new tuples will replace the old ones in $video1$'s queue, since the size of the queue is limited. As a result, the following M_Fetch operator cannot retrieve the deriving media tuple which is required by a qualified but seriously delayed feature tuple.

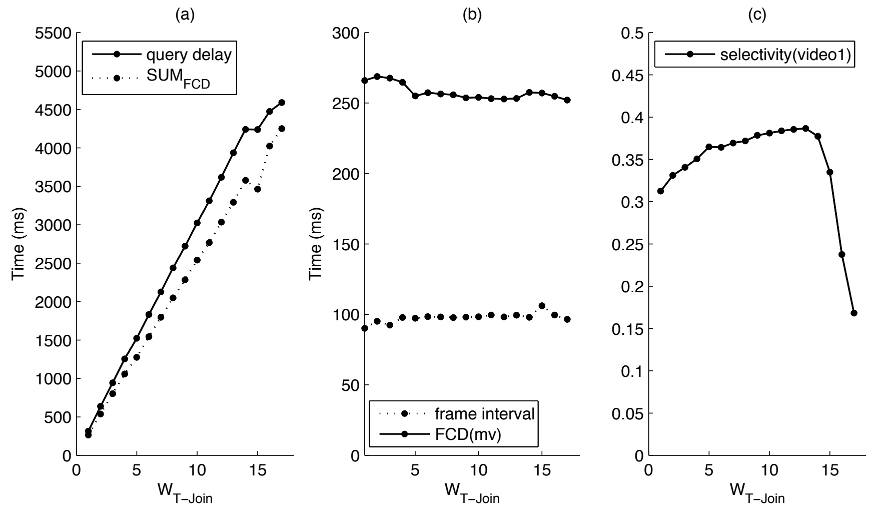

Fig. 6. Performance affects of W_{T-Join}.

As shown in Figure 6 (a), SUM_{FCD} increases linearly as W_{T-Join} increases, so does the query delay. Figure 6 (b) shows the frame interval and FCD(mv) do not change much with W_{T-Join}. The selectivity of $video1$ reaches the climax about 0.38 with $W_{T-Join} = 13$ as shown in (c). After that, it drops dramatically and jumps to 0 after W_{T-Join} is greater 17, because we set the size of $video1$'s queue as 50 and the average frame interval is about 97 ms. This implies that a query delay greater than 4850 ms will make the qualified feature tuple fail to fetch its deriving media tuple, which has been dequeued. Note all the metrics in Figure 6 are average values.

Figure 7 shows which portions of the original video frames sequences are queried out by choosing different W_{T-Join}. Obviously, in case of $W_{T-Join} = 1$, the output frames are the most evenly distributed and the query delay is minimal (314.8 ms). As W_{T-Join} increases, the selectivity may not reduce and may even increase a little (before reaching the climax), but the output tuples are not evenly distributed, which is not preferred in an application where allocating system

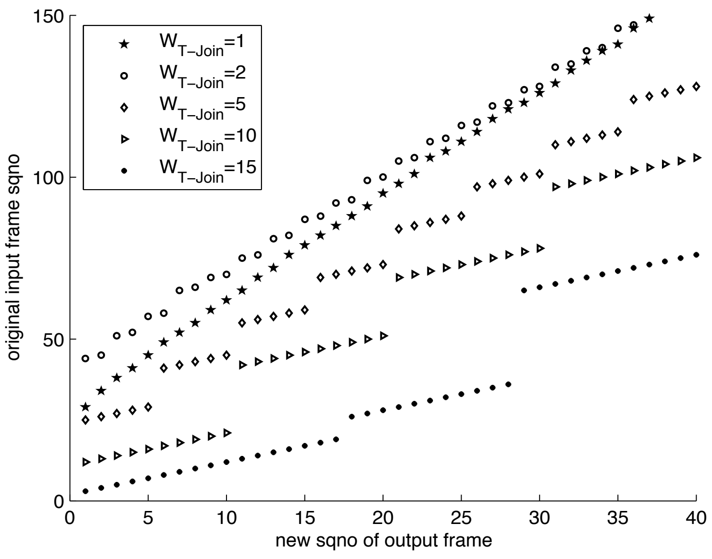

Fig. 7. Temporal distributions of output tuples.

resource to monitor evenly across time is important. Moreover, the query delays also increase linearly.

The average number of media tuples getting queried in a unit period by using non-blocking approach may not be greater than that of using the blocking approach (e.g., the ratio of the former to the latter is 70% when W_{T-Join}=1). However, this only holds when assuming the selectivities of related selection as 1, which is not true in typical queries and forces the mv transformer to work in the worst blocking status. In actual cases, SUM_{FCD} is much less; thus the non-blocking approach works more efficiently. Further, the non-blocking approach does not delay the media stream capturing thread. This implies the media signal sampling resolution is not degraded; thus more media tuples have chances to be captured and then queried. This approach also makes the capturing thread independent of querying thread. This is essential when multiple queries share a common media stream. In addition, the average query delay is smaller when setting W_{T-Join} as 1. The optimal value of W_{T-Join} depends on the actual selectivities of related selection operators, the tuple intervals of joining media streams, and the FCDs of feature streams.

6 Conclusion and Future Work

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. A number of distinct issues in modelling media stream are investigated. By utilizing the coherent temporal constraints, as well as query semantics, of media streams and the derived feature streams, we can design efficient stream operators and query execution algorithm for live media and feature stream querying. We analyze the cost metrics of media stream querying and introduce several optimizations. One additional advantage of our system implementation is its openness for users to design, implement and plug their own operators into our system. A number of experiments are run over live media stream queries by using both blocking and unblocking implementations. We also discuss the metrics and performances of our system.

In the near future, we will investigate scalability issues, sharing of multiple queries, and more complex features derived from multiple media or existing feature streams out of a large number of distributed sensors. We also plan to investigate other factors, such as media transmission delays and compression/decompression, that may affect system performance.

References

1. Liu, L., Pu, C., Tang, W.: Continual queries for internet scale event-driven information delivery. IEEE Transactions on Knowledge and Data Engineering **11**(4) (1999) 610–628
2. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagaracq: a scalable continuous query system for internet databases. In: International Conference on Management of Data, Dallas, Texas (2000) 379 – 390
3. Abadi, D., Carney, D., Cetintemel, U., et al.: Aurora: A new model and architecture for data stream management. VLDB Journal **12**(2) (2003) 120–139
4. Madden, S., Shah, M., Hellerstein, J.M., Raman, V.: Continously adaptive continous queries over streams. In: ACM SIGMOD International Conference on Management of Data, Madison,Wisconsin (2002)
5. Bonnet, P., Gehrke, J., Seshadri, P.: Towards sensor database systems. In: Proc. 2nd Int. Conf. on Mobile Data Management. (2001) 3–14
6. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Symposium on Principles of Database Systems, Madison,Wisconsin (2002) 1–16
7. Liu, B., Gupta, A., Jain, R.: A live multimedia stream querying system. In: Proceedings of the 2nd international workshop on Computer Vision Meets Databases. (2005) 35–42
8. Golab, L., Ozsu, M.T.: Issues in data stream management. ACM SIGMOD Record **32**(2) (2003) 5–14
9. Seshadri, P., Livny, M., Ramakrishnan, R.: Seq: A model for sequence databases. ICDE (1995) 232–239
10. Seshadri, P., Livny, M., Ramakrishnan, R.: Sequence query processing. ACM SIGMOD (1994) 430–441
11. Enderle, J., Hampel, M., Seidl, T.: Joining interval data in relational databases. In: Proc. ACM SIGMOD. (2004)
12. Soo, M.D., Snodgrass, R.T., Jensen, C.S.: Efficient evaluation of the valid-time natural join. ICDE (1994) 282–292

13. Lu, H., Ooi, B.C., Tan, K.L.: On spatially partitioned temporal join. VLDB (1994) 546–557
14. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-Tree: An efficient and robust access method for points and rectangles. In: SIGMOD Conference. (1990) 322–331
15. Elmasri, R., Wuu, G.T.J., Kim, Y.J.: The time index: An access structure for temporal data. VLDB (1990) 1–12
16. Liu, B., Gupta, A., Jain, R.: Medsman: a streaming data management system over live multimedia. In: Proceedings of the 13th annual ACM international conference on Multimedia. (2005) 171–180
17. Tansel, A.U., Clifford, J., Gadia, S.K., Segev, A., Snodgrass, R.T.: Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings (1993)
18. Ramachandran, U., Lillethun, D., Liu, B., Nakazawa, J., Hilley, D., Horrigan, S., Cooper, B.: Mediabroker++: A platform for applications in a dynamic pervasive environment. Submitted to International Conference on Distributed Computing Systems (2005)
19. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: Semantic foundations and query execution. Technical report, Stanford University (2003)
20. Kang, J., Naughton, J.F., Viglas, S.D.: Evaluating window joins over unbounded streams. Proc. of the 2003 Intl. Conf. on Data Engineering (2003)
21. Gunadhi, H., Segev, A.: Query processing algorithms for temporal intersection joins. ICDE (1991) 336–344