Using Stream Semantics for Continuous Queries in Media Stream Processors

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

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

Pilho Kim
Electrical and Computer Engineering
Georgia Institute of Technology
phkim@ece.gatech.edu

Ramesh Jain
Electrical and Computer Engineering
Georgia Institute of Technology
jain@ece.gatech.edu

Abstract

In this demonstration paper we present a stream query processor capable of handling media (audio, video, motion ...) and feature streams. We show that due to their inherent semantics, a number of constraints can be specified on the streams and the dependencies among streams. These are expressed using a Media Stream Definition Language (MSDL). We also show how MSDL constructs are utilized by the query planner and executor, for example, to reduce redundant stream elements. The system is demonstrated using an immersive environment application called IMMERSI-MEET *that enables a user to launch continuous queries against a live meeting.*

1 Introduction

Over the past few years, processing continuous queries over one or more 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 (see [4] for a repository of application domains and example queries). Our motivation arises from an immersive system application where we need to continually evaluate queries over *media* and *feature streams*.

A *media stream* is usually the output of a sensor device such as video, audio or motion sensor that produces a continuous or discrete signal, but typically cannot be directly used by a data stream processor. Instead, it needs to be post-processed by one or more transformers to produce *feature streams* which are data streams correlated to the media stream temporally and in terms of content. Very often, we would want to evaluate a query over one or more feature streams, but would want to output a part of the media stream. For example, assume that there is a video stream V coming from a camera and a motion feature detector M that outputs a stream of characters "Y" or "N" such that "Y" is output if it detects a motion. We pose the query:

```
select S format realVideo
from video V, motion_feature M
where I is s_regexp(M, 'Y+') and S is clip(V,I).
```

where the function s_regexp(stream, regular-expression) returns all time intervals of where a character stream matches the regular expression, and the function clip(stream, time-interval) selects the portion of the stream corresponding with the time interval. In this query, we want to output the video from the camera whenever the motion sensor detects a movement.

In this demonstration paper, we describe our on-going research on a general-purpose media stream management system, which permits a designer to describe the properties of media and feature streams using a media stream description language. These properties are then used by the stream query planner to schedule operators in the media stream processor. In the demonstration, we present IMMERSI-MEET, an application built around an immersive environment for an instrumented meeting room being developed by the Experiential Systems Group at Georgia Institute of Technology.

2 Our Approach

The basic premise of our approach is that not all streams are created equal – they differ in content, streaming properties, operations permitted on them, and how they would be

used in the application. Furthermore, in the case of media and feature streams, explicit *inter-stream constraints* exist and can be exploited in the evaluation of continuous queries in the spirit of semantic query optimization. We express these properties using a media stream declaration language MSDL.

Media Stream Description Language: The IMMERSIMEET system distinguishes between *continuous streams*, where values of different types come at a specified data rate, and *discrete streams* where sources push values intermittently. For example, a video is declared as a data type:

```
create type frame as media_record {
        integer frame_number,
        image content
}.
create type video as continuous stream of frame
        at 30 per second,
        timestamp as frame_number.
```

Here media_record denotes the object as a record containing at least one media object. A stream containing a media object is a media stream. Note that the timestamp (an intrinsic property of any stream) is specified to be frame number than the default unit of absolute time from the start of the stream. The stream instance is declared as:

```
create media video V1 {
        format mpeg,
        skip 1 in every 4 in input.
}
```

The optional keyword skip provides the directive that the media has sufficient content redundancy that dropping 1 in every 4 elements (an element is a time tick, i.e., every frame here) from the input will not hurt the queries. The default is to skip nothing, thus the output video cannot drop any further frames. In contrast, consider a stream that outputs a PowerPoint presentation in the form of an image sequence. This is simply declared as:

```
create type ppt_slide as {
        image slide_content
}.
create type ppt_stream as continuous stream of ppt_slide.
```

Now assume there is a software that tracks users' I/O events like mouse-clicks and keystrokes. This software, in conjunction with the PowerPoint stream, serves as a discrete feature stream that intermittently reports whether the user goes forward or reverse during a presentation – we call this the *punctuation stream* of the ppt_stream. While in principle, the idea of punctuation here is similar to that in [5], namely, to treat a single unbounded stream as a combination of streams that terminate at punctuation boundaries, in our case, the punctuation itself comes from a different stream. Further, there is yet a content extraction software, which, at every tick of the punctuation stream, processes the current PowerPoint presentation and outputs a *slide content stream* containing the textual content corresponding to each slide with every mouse click of the user. Clearly, the feature streams allow us to specify some inter-dependency among these three streams. To capture this dependency in MSDL, we first have to declare the base types and the stream types:

```
create type ppt_marker as
        textToken ('FWD'|'REV').
create type marker_stream as discrete stream of ppt_marker.
create type ppt_text as record {
        integer slide_number primary key,
        string(30) slide_title,
        string(1000) text_content
}.
create type ppt_content as discrete stream of ppt_text.
```

Now, the dependencies can be declared while creating the concrete stream instances.

```
create media ppt_stream S1.
create feature marker_stream M1 on S1
        with avg_delay 1 second and
        max_delay 2 second where
            punctuates(M1, S1).
create feature ppt_content C1 on S1, M1
        with avg_delay 2 second with S1 where
            start_synchronized(M1, C1) and
            identifier(S1,C1.slide_number) and
            repeats(C1.fields).
```

Note the usage of the primary key for the record, and the usage of the keyword on in specifying the dependence relation between streams. In MSDL, any dependence declaration must have at least one dependency specifying predicate in the body. The punctuates(discrete_stream, continuous_stream) predicate denotes a first kind of inter-stream dependency. In addition, one or more with delay specifier serves as a directive to the stream processor to set some guidelines on the temporal asynchrony between streams, thus accounting for the computation time for feature extraction. In the second case, we illustrate the dependency predicate start_synchronized(discrete_stream, discrete_stream) that denotes that the start of the stream elements of M1 and C1 are synchronized, but not their ends. This is expected because M1 only outputs a token stream while C1 outputs a variable length record. The assumption made here is that two successive signals from M1 does not occur before C1 is completed. If this possibility exists, the start_synchronized predicate must be replaced by the predicate order_synchronized to mean that the i-th elements of the two streams denote *corresponding objects*. Finally, the predicate identifier(stream1, stream2_var) is a directive that ascribes a variable from a punctuating stream as the key to another, possibly continuous stream. The implication of

this *identification constraint* is discussed later. The repeats constraint asserts that all fields of the streaming record may be identical across punctuations, which is likely to happen if the PowerPoint presenter goes back and forth between slides.

In some cases, a feature stream needs to be considered as independent. Consider the example query – typically, the media stream corresponding to a motion sensor is declared, but not used by any query. Instead, all queries use a feature stream produced from it. As in the other cases, the motion sensor m would typically declare a data rate ρ_m. But now, the corresponding feature stream would have to declare a data rate ρ_f as $\alpha \times \phi(\rho_m) + \delta$, where α is the drop rate, δ is a delay and $\phi()$ is an experimentally determined function. The feature itself will be declared as create independent feature ... to ensure that the definition is compiled to compute the stream properties of the feature, and then used for query planning and operator scheduling.

Stream Constraints in Query Evaluation: As stream declarations are registered, the stream constraints are interpreted to construct a set of *evaluation directives*. The skip keyword for the stream S produces a directive S.dropElement(<fraction>). We call this a directive because the stream query engine may choose not to exercise the directive. In the example case, if the input and output formats dictate that the clip procedure needs the input video stream to be uncompressed into raw frames, the frames will be dropped – if it is more expensive to uncompress and drop frames than to convert directly into the output format without dropping frames, the directive will be ignored. This illustrates a primary distinction between media streams and data streams – while the clip operator can be used for any media stream, its implementation will depend significantly on the format of the data and hence will be a primary determinant in any cost-based planning operation exercised by the stream query processor.

The effect of the dropElement directive is much more pronounced in the case of a punctuating feature. By our declaration, the punctuating feature may arrive with a maximum possible delay of δ_{max} compared to the video stream. The video stream is therefore queued in chunks of $\delta_{max} \times frame\_rate$, and then upon the arrival of the next feature element from the punctuating stream, the frame $\delta_{avg} + \epsilon$ is selected (i.e., marked for selection), where ϵ is a uniformly distributed random variable between 1 and $\frac{\delta_{avg}}{2}$. The rest of the frames are dropped, with minimal chances of information loss.

As a consequence of the start_synchrnonized predicate, the stream query planner/scheduler ensures that the queue for the punctuating signal also supplies the selection operator for the slide-content stream, from which the content record is picked at the arrival of the punctuating token. It is possible to define a "handler" for the repeats constraint. Note that while the text content of the slide might repeat, the timestamp for the $(i+1)$-th occurrence of the same slide is always increasing. One possible way to "handle" this case, is to project only the primary key (i.e., the slide number) as a way to compress the output, but at this time we are still evaluating if such special treatments give us any benefit.

In IMMERSI-MEET all text terms for a window around the current slide are indexed, with the slide number (the primary key) being used as the record pointer. The identification constraint is used to *induce* a join between the "unidentified" stream and the "identifying variable" from a feature stream, by supplying it a primary key. Simply, it constructs an intermediate bookkeeping table in the query processor by associating every slide number to the timestamp intervals of the frame stream. This table serves as an index for any query that needs to access specific portions of the frames based on slide content. This is used, for instance, in a query that requests a stream upon the occurrence of an event is expressed by extending the SQL syntax as:

```
show ppt_stream on event (
        select frames F
        from ppt_stream, features(ppt_stream)
        where F->text_content like "%View Unfolding%"
).
```

The function features(<media_stream>) is a convenient macro that hides the exact feature to be used from the user. When we refer to *some* feature related to the frame, we use the notation F-><attribute_name>) as if this were a property of the frame itself. This indirection is compiled away before execution. The plan uses the text index to spot the keyword, and the bookkeeping table to acquire the timestamp on the ppt_stream. The temporary table makes the frame searching more efficient. When the frame is located, the system opens a channel to the user to stream the presentation from that point on.

System Architecture: The logical architecture of the system is strongly influenced by the STREAM project [3]. Our current focus is to demonstrate the dependency utilization fragment without focusing on load shedding and operator schedulin [1], or the communication optimization considered in [2].

3 The Demonstration

In the previous section, we described the operations of the query processor primarily in terms of the relatively simple PowerPoint stream and its features. The immersive environment of the IMMERSI-MEET environment is much richer in the variety of media streams. It is built around a meeting room with a number of cameras and microphones.

There is a designated camera and microphone for the primary speaker. Every other participant has a dedicated microphone and shares some cameras. In addition, the doorway to the meeting room is equipped with a camera and a motion sensor.

The computer of the presenter is instrumented with customized software written around a clickstream and keystream tracker software. This software parses the text content of the focus window of the audio-stream processors from the microphones assume only speech signal is present and is routed through a speech-to-text converter that provides about 80% correct transcription. The database will also have static relations that keep the details of participants, presenters, and the topics of their presentations. The video streams from the cameras will be processed through some elementary motion-processing and feature extraction routines. These detectors will compute a set of simple features like positions and bounding boxes of clusters of motion. They will be further processed by a set of "elemental event detectors" to determine events like "standing-up", "sitting-down", "entering-room", "raising hand", etc. under the simplifying assumption that only single events occur in the meeting room at any time. Each event detector then sends a stream of events (previously declared as dependent feature streams), each described using a set of parameters like an error estimate of event detection.

During demonstration, the meeting room will be "simulated" by first pre-recording the different channels of the meeting on a number of different computers. These computers will then send a set of media and feature streams to a processing computer that runs the media stream processor. We will show an interface to register streams and queries. Further, we will present a "debugger's console" which would show the operators used for a query and the states of the queues as the execution occurs. The queries we would show would include both the notification case, where the user gets connected to the meeting when a specified event occurs, and the "log upon event occurrence" case, where the output is a media record composed by projecting portions of the media that satisfy the query and other attributed computed from feature streams or from static relations.

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, August (to appear) 2003.

[2] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1), March 2003.

[3] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Conference on Innovative Data Systems Research (CIDR)*. IEEE, January 2003.

[4] The Stanford STREAM Group. Stream Query Repository. web site at http://www-db.stanford.edu/stream/sqr/, January 2003.

[5] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, May 2003.