The Image Stack Stream Model, Querying, and Architecture

Alfonso F. Cárdenas Raymond K. Pon Bassam S. Islam
Computer Science Department, University of California at Los Angeles
3731 Boelter Hall, UCLA
Los Angeles, California
{cardenas, rpon, bassam}@cs.ucla.edu

Abstract

Rising volumes of multimedia are being gathered with the increasing deployment of sensors. We present the Image Stack stream model/view of data for querying and visualizing the streaming data. This view is independent of the presence of a DBMS, since more sensors will capture data on a real-time basis. We outline requirements for modeling and visualizing streaming multimedia data with motivating queries. We present the Image Stack data model, a high-level query language for the model, and a system architecture and design to support these requirements. We provide highlights of a prototype implementation in Java and Java Data Objects, bypassing the use of a DBMS as permanent storage.

1. Introduction

Growing types and volumes of multimedia data (alphanumeric, image, sound, and video) are being captured by increasing deployment of sensors (environmental, geophysical, medical, etc). We address the challenge of querying and visualizing of information from multiple streams of different but related types of data, focusing on the access and presentation of data through time. Recent work by others has been reported on video indexing and accessing by content and visual languages [1]-[2]. However, it has generally focused on viewing individual video streams and not on the multiple heterogeneous streams that we are addressing. Furthermore, because much of the expected data streams are multidimensional, using existing streaming data management technology to answer spatio-temporal queries over multidimensional real-time and archived data is difficult. An example of such a query is:

> *"Display the locations of intersections of the UCLA boundaries with Westwood Blvd and Sunset Blvd where the poison fume level exceeds value Y now."*

A stream is an ordered sequence of frames or values. A frame could be an image, a photograph, a frame in a video stream, a text report, or an alphanumeric record changing through time. Current DBMS's (relational or object DBMS's) deal well with alphanumeric record type structures once they are stored and loaded into a database. Unfortunately, it is impractical to store in a DBMS the voluminous data that is arriving rapidly from many sensors.

There has been research regarding alphanumeric data

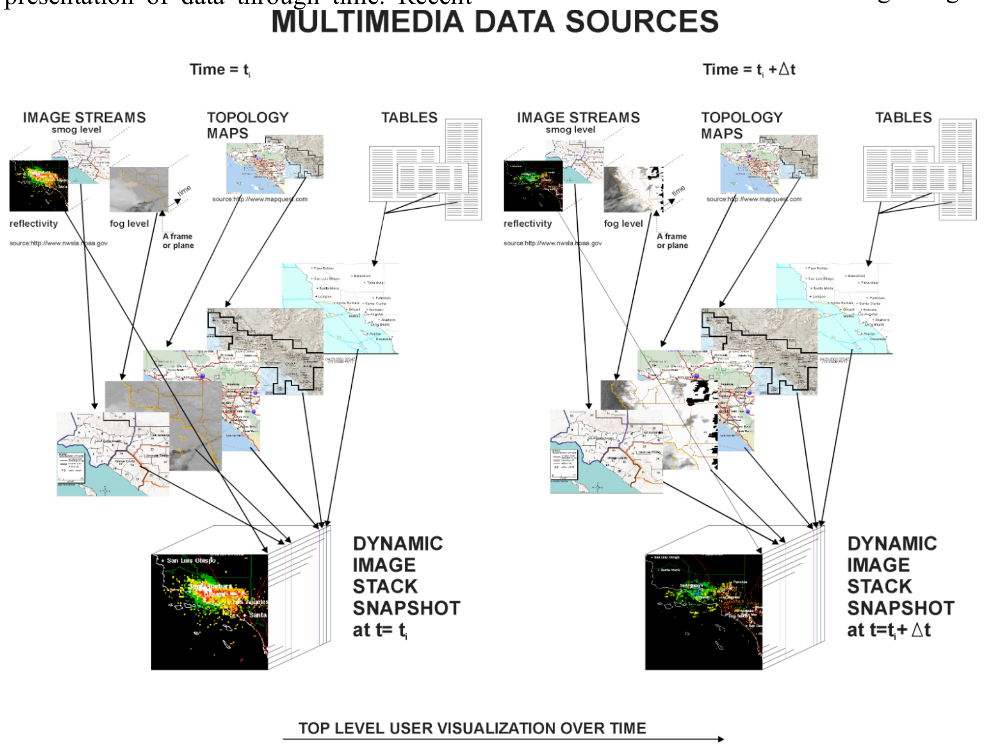
MULTIMEDIA DATA SOURCES
Time = t_i
Time = $t_i + \Delta t$
IMAGE STREAMS
smog level
TOPOLOGY MAPS
TABLES
reflectivity
fog level
time
A frame or plane
source:http://www.nwsla.noaa.gov
source:http://www.mapquest.com
DYNAMIC IMAGE STACK SNAPSHOT at t= t_i
DYNAMIC IMAGE STACK SNAPSHOT at t=$t_i + \Delta t$
TOP LEVEL USER VISUALIZATION OVER TIME

Figure 1: Stream of two Image Stacks.

stream processing. The Cougar project [3] manages data from sensor database systems by using abstract data type functions to represent sensor devices. The NiagraCQ project [4] has focused on retrieving XML data, querying, and monitoring them for some interesting change, the optimization of queries by grouping similar queries together, and on optimization based on the rate of arrival of data items. The STREAM project [5] has developed a prototype of a data stream management system with support for typical relational DBMS's. The TinyDB project [6] has developed continuously adaptive continuous query techniques, in-sensor-network aggregation of data, and visualizations for sensor data. Like the STREAM project, the TelegraphCQ project [7] has also developed a general system to process continuous queries over data streams. The Aurora Project [8] has focused on optimizations for real-time data stream processing.

In contrast to other streaming data management projects and the advances introduced in [9]-[10], we make the following advances with the Image Stack model:

- A high-level query language enabling *visualizing and querying* historical data with real-time data.
- The ability to query *massive multi-dimensional* data that arrive from a data stream.
- Language semantics allowing querying of absolute and relative points in time of a data stream, encapsulating historical and real-time data as a *single continuum*.

Like ours, prior projects have a query language for querying streaming data. NiagaraCQ's language is based on XML; whereas, the other projects are based on SQL. We present a query language, isOQL, based on ODMG OQL [11], intended for use upon data streams. Borrowing notions such as time windows from otherdata stream management projects, isOQL also provides for visualizing and querying historical data.

Other systems have focused on querying simple tuples from continuous streams while the Image Stack focuses on querying massive multidimensional streaming data. We provide an architecture for processing, querying, and visualizing multidimensional multimedia data. Being a data model for high-level querying and visualizing streaming and archived data, the Image Stack has not focused on optimizations for streaming alphanumeric data, but can leverage existing stream processing technologies to take advantage of such optimizations.

Many of the mentioned projects have focused on managing streaming real-time data along with some stored relational data, but have treated historical data as a separate entity. The semantics of CQL [12] and similar stream-based languages are based on querying data from a real-time stream and treat historical data as a different entity (i.e., a separate table). However, we argue that this approach is not intuitive. For example, from a user's perspective, current smog data and smog data from 10 years ago are both instances of smog data, and so the two should not be differentiated. Sliding window semantics of existing stream-based languages may provide some historical data within the same stream being queried in real-time, but data that is available before the query has been issued or falls outside the sliding window of the query is not available for querying. Only the Aurora project has addressed this issue by introducing connection points to store and process historical along with streaming data. In [7], CACQ encapsulates historical and real-time data as a single entity to support disconnected operation, allowing users to register queries and return intermittently to retrieve the latest answers, by applying old data to new queries and then new data to old queries, when new data becomes available. isOQL takes a similar approach, but unlike CACQ, isOQL provides for the explicit specification of an absolute historical point in time in a data stream, in addition to the current time and points relative to the current time.

We have proposed the *Image Stack model/view* as an attractive way to visualize multiple types of data and set up as a local or user view database to support major types of multimedia queries. Figure 1 shows an example of the Image Stack which consists of several planes. Each plane contains a different type of two-dimensionally encoded data that are co-registered to the same coordinate system. This stack is be composed of elements at a point in time from different data streams. The example also shows how several data sources from which data at one point in time could be gathered logically and viewed as planes in the Image Stack. In many multimedia applications, such as environmental analysis, the main interest is visualizing the changes and trends. This leads us to introduce the notion of a *stream of Image Stacks*, also illustrated in Figure 1 for a stream with two stacks showing change through time.

Figure 2 illustrates the basic data model of the Image Stack. Conceptually, a Stack object contains many Frame objects, which are simply represented as a two dimensional grid of Cell objects. For example, each Cell object may contain a smog value at different locations and be collectively grouped together with a Frame object. A Stack object may contain a Frame object that contains smog level data, a Frame object that contains population density data, a Frame object that contains temperature data, and other Frame objects containing other types of data over the same region. Each of these Frame objects are co-registered so a Stack object may be represented as a three-dimensional cube. We later extend this basic data model to support explicit relationships for all levels of the data model (e.g.,

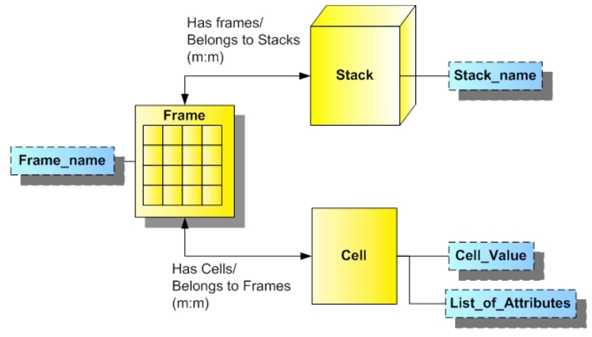

Figure 2: High-level Image Stack model

stack, frame, or cell) and to support temporal querying..

2. High-Level Queries for Multidimension Visualization

We have developed a high-level query language, isOQL, based on ODMG 3.0 OQL for its native object orientation and encompassing of SQL's features, including extensions for operations provided by the Image Stack model.

2.1 Querying Language Grammar

We have used the ODMG OQL as the basis for isOQL, with the adition of several operators and semantic additions to support the Image Stack model. We have created a grammar that is disjoint from the OQL BNF. Our grammar branches off the OQL BNF at the highest level with the following rule:

Query := selectExp | newQuery | expr

This rule is the same as the OQL BNF, with the addition of the *newQuery* expression:

*newQuery :=displayExp {***OVERLAY** *displayExp}*
*displayExp :=***DISPLAY** *displayList newFromClause [whereClause]*

All of the queries that make use of the visualization features of the query language start with the **DISPLAY** keyword. This allows us to keep queries that use visualization operations separate from traditional OQL SELECT statements. The **OVERLAY** keyword allows two or more visualization queries to be superimposed on one another in a single display screen.

displayList :=displayAttribute {, displayAttribute}
displayAttribute := **(** *displayFunction* **)**

After the DISPLAY statement, the display function follows enclosed in parenthesis. We currently have three classes of display functions: plot functions, cine functions, and contour functions.

displayFunction := plotFunctions | cineFunctions | contourFunction

The plot functions, defined below, simply plot the cells of the image in a user-selectable color.

plotFunctions := **PLOT_VALUE** *expr [***IN** color*]*
| **PLOT_POINT** *plotLocation [***IN** color*]*
| **PLOT_CHANGE** *expr, expr chageColorExpr*

If the user does not select a color, the system will select a color from a pool of default colors. The **PLOT_VALUE** function plots the value of the current cell value in a gradient color that corresponds to the value of the cell. The **PLOT_POINT** function uses the **LOCATION** keyword to plot the current cell that is being examined. The **PLOT_CHANGE** function takes two cell values as inputs (two cells from the same image at two different time periods), and plots the increases in one color gradient, the decreases in another color gradient, and optionally plots no change in yet another color.

cineFunctions := **CINE** *expr {, expr}*
| **CINE_CHANGE** *expr {, expr} changeColorExpr*
| **CINE_PERCENT_CHANGE** *expr {, expr} changeColorExpr*
changeColorExpr := **INCREASES IN** color, **DECREASES IN** color *[,* **NOCHANGE IN** color*]*

All cine functions output a number of images in an animation. These images are generally the same images at different points of time. These animations help visualize the change of a certain parameter over the same area at different points of time. The **CINE** function takes a number of images as input, and simply displays a slideshow of the images. The **CINE_CHANGE** function takes as input a number of images (of the same area), and shows an animation of the images, displaying the increases between images in one color gradient, and decreases in another color gradient. The **CINE_PERCENT_CHANGE** is identical to the CINE_CHANGE function, except that the change between images is normalized to a percentage.

contourFuction := **CONTOUR_VALUE** *expr* **EVERY** *expr* **IN** color

The **COUNTOUR_VALUE** function plots contour lines on the image, at a user-definable numeric interval, with a user-definable color.

newFromClause := **FROM** *newInteratorDef {,newIiteratorDef}*
newIteratorDef := iteratorDef | streamDef

Our high-level language queries also include extensions in the traditional FROM clause to better handle stream and image data. The new addition is the *streamDef:*

streamed **:=** *expr[* **[***windowExpr***]** *]* **AS** Alias

The *streamDef* rule allows us to select multiple images in the FROM clause, whether they contain different kinds of data, or the same data at different intervals of time. For the latter case, we have several options for selecting the time periods for the desired images:

windowExpr := windowInstance | windowInterval | windowIntervalFreq

To select a single snapshot of image data at a certain point in time, we would simply use the *windowInstance* rule:

windowInstance := timeExpr

To select all of the images in a time interval, we would use the *windowInterval* rule:

windowInterval := timeExpr **TO** *timeExpr*

If we wish to select images in a time interval at certain periodic intervals (e.g., images between 2000 and 2002, taken every two months), we would use the *windowIntervalFreq* rule:

windowIntervalFreq :=timeExpr **TO** *timeExpr* **EVERY** *unitExpr*

The following rule defines the allowable expressions of time (e.g., January 2002, **NOW**, or **NOW** – 10 years):

timeExpr := **NOW** | **NOW** – *unitExpr* | **NOW** + *unitExpr* | *timeLiteral*

We use the keyword **NOW** to indicate the most current snapshot of the image data. We also allow mathematical expression to specify a time period relative to NOW. This mathematical expression can reference events in the past and can reference events in the future that may be extrapolated from current data through processes such as regression. Furthermore, we allow the specification of a *timeLiteral* (e.g., "January 1, 2001") to indicate an absolute point in time, as opposed to a relative point in time. Further research is being conducted into defining bounds on how stale the NOW data is allowed to be, and how often the NOW data needs to be refreshed relative to the frequency of change of the data for it to remain "fresh."

2.2 Example Queries

We present several motivating queries, which are typical of the type of major decision-making queries that are not automated today by any generalized DBMS, Geographical Information systems, or visualization system. We show the isOQL incarnations for four queries. The results of the queries would be sent to an imaging or visualization system for display, which are beyond the scope of this paper.

Example 1: *Display the locations of intersections of UCLA boundaries with Westwood Blvd and Sunset Blvd where the poison fume level exceeds value Y now.*

```
DISPLAY (PLOT_POINT LOCATION)
FROM Road AS R, School_Boundaries AS SB,
    Poison_Levels_Plane_Stream[NOW] AS PL
WHERE SB.Name LIKE "UCLA" AND
    (R.Name LIKE "Westwood Blvd" OR
    R.Name LIKE "Sunset Blvd") AND
    (SB INTERSECTS LOCATION) AND
    (R INTERSECTS LOCATION) AND
    PL.Cell_Value > Y
```

In the FROM clause, we are examining Road and School_Boundaries which are extents containing objects representing information about roads and school boundaries. Poison_Levels_Plane_Stream is a stream of planes, or a plane stream, as defined in the previous section, indicating time-varying poison levels over a geographic area. This is simply an array of Planes, indexed by time and where each instance of a Plane is a geographical map indicating poison levels. The **LOCATION** keyword indicates the current location of the query execution. The **LOCATION** construct is analogous to the current tuple being processed in relational DBMS's. The NOW keyword can be used as an index in a stream, retrieving the most current plane from the stream. In the preprocessing phase of the query, the NOW symbol is replaced by a numerical index of the most recent plane.

Example 2: *For locations with smog levels over X, show elevation with contour lines every 25 feet for places where there are school districts, showing smog level in purple.*

```
DISPLAY (PLOT_VALUE S.Cell_Value IN
    PURPLE), (CONTOUR E.Cell_Value EVERY
    25 IN BLACK)
FROM Smog_Plane_Stream[NOW] AS S,
    School_Districts_Plane AS SD,
    Elevation_Plane AS E
WHERE SD.Cell_Value <> NULL AND
    S.Cell_Value > X
```

The DISPLAY command merely displays the result of the query on the viewing device. The PLOT_VALUE function paints the current location of the query execution with a shade of the specified color related to the Cell_Value of the input parameter. The CONTOUR function draws contour lines given the color of the line, the value to contour over, and the distance between contour lines.

Example 3: *Compare the current smog level to a year earlier showing clearly the differences in red for higher smog levels and in blue for lower smog levels.*

We can answer the above query by OVERLAYing the results of two queries, utilizing the OVERLAY operation, but it is expected that the above query form would be common and to rewrite such a complex query for different parameters and different streams would be tedious; thus, we use the display function, PLOT_CHANGE as the following isOQL query illustrates:

```
DISPLAY (PLOT_CHANGE S1.Cell_Value,
    S2.Cell_Value INCREASES IN RED,
    DECREASES IN BLUE)
FROM Smog_Plane_Stream[NOW] AS S1,
    Smog_Plane_Stream[NOW–1 Year] AS S2
```

Example 4: *Compare smog level to two, four, six, eight and ten years earlier and show a stream of images clearly indicating for each image the differences in red for higher smog levels and in blue for lower smog levels compared to the prior period; provide also the percentage change in population density and ethnic mix versus the prior period.*

```
DISPLAY (CINE_CHANGE S INCREASES IN
    RED, DECREASES IN BLUE), (CINE E)
    (CINE_PERCENT_CHANGE P INCREASES
    IN GREEN, DECREASES IN PURPLE),
FROM City_Areas AS CA
    Smog_Plane_Stream[NOW-10 Year TO NOW
    EVERY 2 Year] AS S,
    …
WHERE CA.name = 'East Los Angeles' AND CA
    CONTAINS LOCATION
```

We could have explicitly written conditions for displaying increases and decreases; however, we chose instead to simplify the queries by using **CINE_CHANGE**, **CINE_PERCENT_CHANGE**, and **CINE**. In this query,

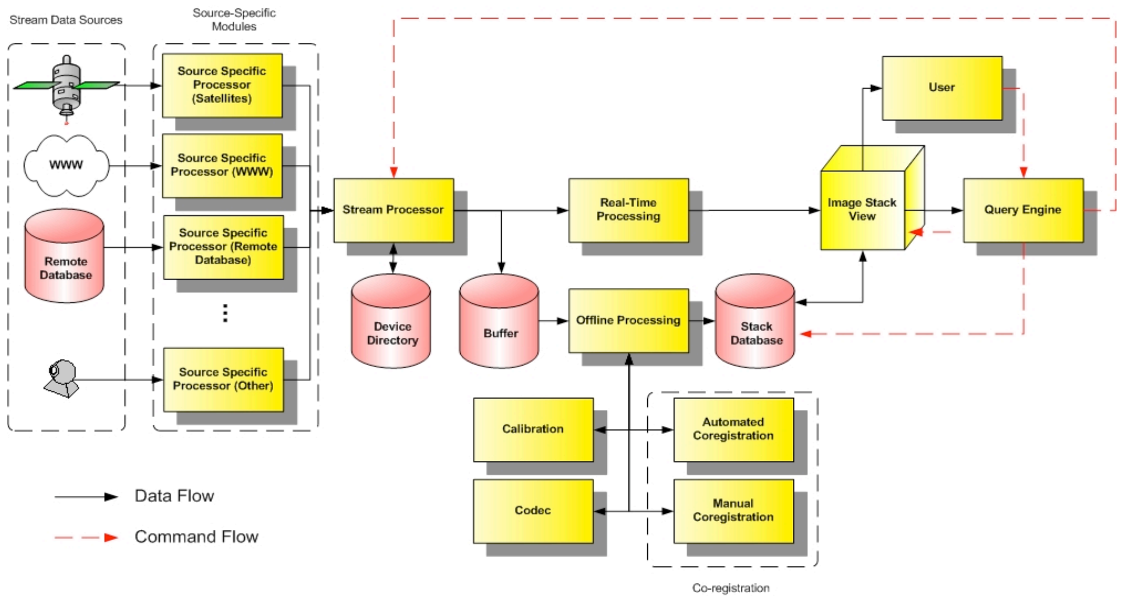

Figure 4: System architecture.

we see that from the base streams, we can select a desired substream for any desired time interval and frequency.

3. System Architecture and Design

3.1 Heterogeneous Data Sources

Figure 4 shows the overall system architecture that will incorporate heterogeneous data sources, real-time and offline processing of data, and the Stack view and querying. The Stream Processor serves as the entry point for stream data into the system. Heterogeneous data sources are handled by special sub-processors that are spawned from the stream processor and are able to handle the extraction of data from each type of source. The drivers for extracting each of the possible input data stream type is stored in a device database. The heterogeneous tuples are then forwarded from the sub-processors to the main Stream Processor, which would then forward the data for further processing. This procedure homogenizes the heterogeneous data into a single framework, so it can be easily manipulated by the rest of the system.

The Stream Processor, the real-time, and offline processing modules are independent modules. This architecture is very much similar to the mediator architecture originally introduced in [24]. This allows for extensibility and flexibility. If a new data stream source is made available, a new source-specific module can be added to make the data source available to users with little modification to the rest of the system. Furthermore, the source-specific modules hide much of the source-specific interfaces from the rest of the system, making system implementation easier and much more modularized. Also, if new processing techniques are developed, they can be added to one of the processing modules as a sub-module.

3.2 Offline versus Online Processing

Note that data streams will not necessarily reside in a conventional DBMS since a majority of sensor data will be too voluminous and will use the Internet as the primary means of providing such data. The Stream Processor may retrieve data continuously real-time, on an ad-hoc basis or on a predefined schedule.

If a DBMS is available that can support the Image Stack model, then its role is shown as the path on the bottom in Figure 4. Data is captured by the Stream Processor and stored temporarily in a buffer. Once the data is stored in the buffer, the data may be compressed, decompressed, calibrated, or co-registered in offline processes via the Offline Processor. The results of the offline processing are stored into the Stack Database so that when the stack view

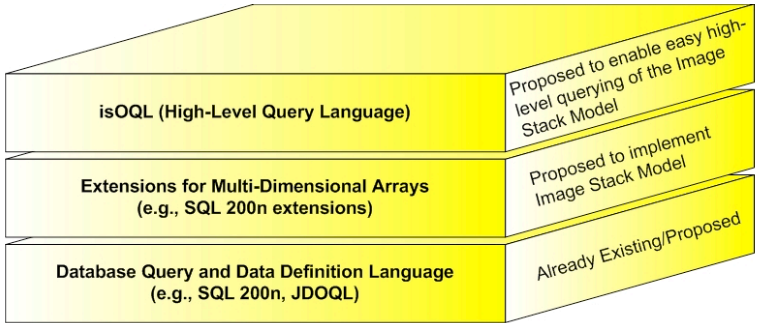
isOQL (High-Level Query Language)
Proposed to enable easy high-level querying of the Image Stack Model
Extensions for Multi-Dimensional Arrays (e.g., SQL 200n extensions)
Proposed to implement Image Stack Model
Database Query and Data Definition Language (e.g., SQL 200n, JDOQL)
Already Existing/Proposed

Figure 3: Possible layering of languages.

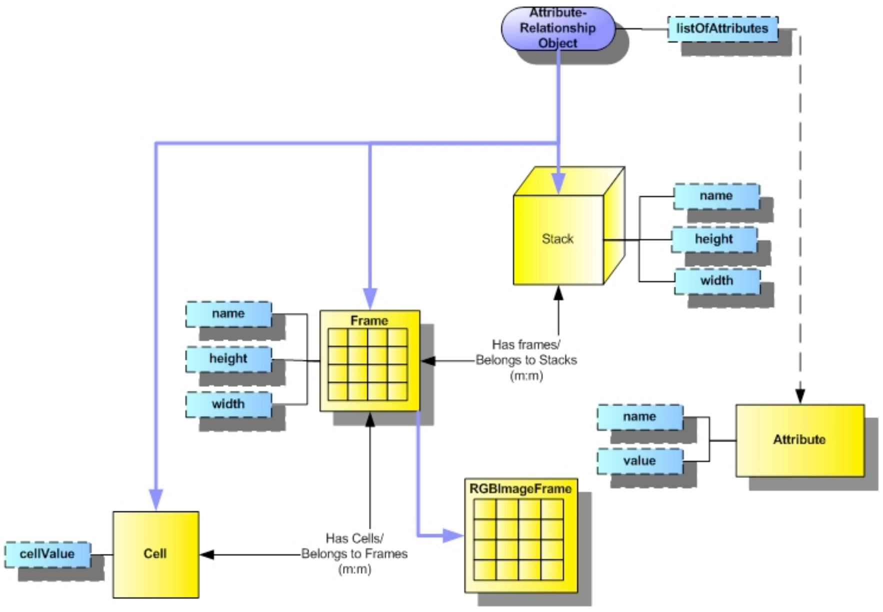

Figure 5: Class architecture for JDO implementation.

is generated later, such processing need not to be done later.

The stream processor, when used in the offline path, will selectively determine which data items should be stored in the DBMS since the data source may provide an infinite stream of data, which cannot all be stored. This selection of data may be based on frequency-based sampling or criteria-based sampling defined by the user or on an on-demand basis, which is an approach taken in the TinyDB project [6]. The selection of data may also be based on identifying which streams have particular characteristics that are of interest, employing methods such as the use of the Hamming Norm [15] or clustering techniques [16]. Other sampling and caching techniques and criteria that may be adapted for the system are discussed in [17] to [20]. An approximation of the data stream or summary data structure may be also employed, such as wavelets [21][22], histograms [23], or sliding windows to reduce the amount of data stored.

If no DBMS is involved, then we can take the streams directly through a real-time stream processing phase via the Real-Time Processor and then provide the Image Stack stream view. Unlike in offline processing, in the real-time processing, the streams are co-registered with possibly less accuracy, depending on the availability of fast algorithms and hardware, which may employ heuristics or approximations and is beyond the scope of our work.

By separating the processing of data into two paths: real-time and offline, data can be processed and viewed on a real-time basis or may be viewed later. Thee data flow path is split into these two paths because we may have different algorithms in processing real-time data and data to be stored in the DBMS in regards to performance, speed, and accuracy.

3.3 Stack View Management and Querying

The Image Stack View is the main interface for the user to view and access the stack data. Queries and data manipulation in the Image Stack View takes place via the Query Engine module. A user issues an isOQL query on the data in the Image Stack View, which would send the query to the Query Engine. Figure 3 shows the layering of the database languages made use by the system. At the top layer is isOQL, which is used to interact with the Image Stack model. Below isOQL are extensions needed to the existing DBMS to handle multi-dimensional data. Beneath the extension layer are the existing database languages, such as JDOQL. The language layering allows isOQL and the Image Stack view to be used in a variety of environments. The high level query language, isOQL, is used so that it would be easier for the user of the system to query it, hiding much of how the Image Stack model is implemented in the DBMS. isOQL incorporates the real-time and DBMS aspects of the system.

The Query Engine determines whether the data necessary to answer the query is present in the Image Stack View, or if more data would need to be loaded from the Stack Database, or if real-time data needs to be requested from the Stream Processor. If necessary, it sends commands (shown in dashed arrows in Figure 4) to the Stack Database to load the data into the Image Stack View and executes the query there. If the query requires real-time data, the Query Engine sends a request for data to the Stream Processor and sends the data to the Image Stack View via the Real-Time Processor. It is possible that the user would wish to use the output of a query as one of the inputs to another. Since the results of all queries are initially stored in the Image Stack View, the user may store the query results in the Stack Database.

4. System and Data Model Implementation

Figure 5 shows the class structure diagram used in Java to implement the above functionality. A Stack, Frame, and a

Cell are subclasses of special super-class, called the Attribute-Relationship Object. An Attribute-Relationship Object has a dynamic list of Attribute objects. Since the Stack, Frame, and Cell classes are Attribute-Relationship Objects, they have a list of Attributes, so any number of user-defined attributes can be added to the whole Stack, Frame or single Pixel, providing storage of data at all levels.

We have implemented the data model in Java Data Objects (JDO) [26], making use of the OO-DBMS FastObjects t7 [27]. We have a preliminary implementation of isOQL built on top of JDOQL, which is demonstrated at [13], using land surface temperature data from the MODIS instrument [14] aboard the Terra satellite, which is already co-registered. Future work will adapt advanced co-registration methods such as [25] if co-registration data is not available.

5. Conclusion and Future Work

We are pursuing further development of the Image Stack view along with streams of stacks over a multitude of multimedia streams. The intent is to provide such a view over data whether or not a DBMS is used at all. We highlight an isOQL engine and Image Stack system without use of a DBMS since many of the data streams broadcasted via the Internet will not reside in a DBMS.

Acknowledgement

This work is partially supported by National Science Foundation Grant # IIS 0140384, “Multimedia Stream Modeling, Relationships and Querying.”

References

[1] Flickner M (1995) Query by Image and Video Content: The QBIC System. Computer 28(9)

[2] Chang S, Chen W (1997) VideoQ: An Automated Content Based Video Search System Using Visual Cues. ACM Multimedia

[3] Bonnet P, et al (2001) Towards Sensor Database Systems. 2nd Int’l Conference on Mobile Data Management, Hong Kong.

[4] Chen J, et al (2000) NiagaraCQ: A Scalable Continuous Query System for Internet Databases. SIGMOD 2000: 379-390

[5] Motwani R, et al (2003) Query Processing, Resource Management, and Approximation and in a Data Stream Management System. CIDR 2003, Pacific Grove, California: 245-256

[6] Madden S R, et al (2003) The Design of an Acquisitional Query Processor for Sensor Networks. SIGMOD, San Diego, CA

[7] Chandrasekaran S, et al (2003) TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. CIDR 2003

[8] Carney D, et al (2002) Monitoring Streams - A New Class of Data Management Applications. VLDB Conference, Hong Kong

[9] Cárdenas A F and Michael P A (2002) Image Stack Model of Multimedia Data. DMS 2002, San Francisco, CA.

[10] Cárdenas, A F, et al (2003) Image Stack Viewing and Access. Journal of Visual Language and Computing 14: 421-441

[11] Cattell R G (2000) The Object Database Standard: ODMG Release 3.0, Morgan Kaufmann, San Francisco, CA

[12] Arasu A, Babu S, Widom J (2003) An Abstract Semantics and Concrete Language for Continuous Queries over Streams Relations. International Conference on Data Base Programming Languages

[13] MMSS Group (2003) Geophysical TimeLine [Online] Available: http://www.mmss.cs.ucla.edu/GeoHome.htm

[14] NASA (2003) MODIS Web [Online] Available: http://modis.gsfc.nasa.gov/

[15] Cormode G, et al (2003) Comparing Data Streams Using Hamming Norms (How to Zero In). Transactions on Knowledge and Data Engineering 15(3): 529-540

[16] Guha S, et al (2003) Clustering Data Streams: Theory Practice. Transactions on Knowledge and Data Engineering 15(3): 515-528

[17] Cho J, Garcia-Molina H (2000) Synchronizing a database to Improve Freshness. SIGMOD

[18] Cho J, Ntoulas A (2002) Effective Change Detection using Sampling. VLDB Conference, Hong Kong, China

[19] Olston C, Widom J (2002) Best-Effort Cache Synchronization with Source Cooperation. ACM SIGMOD, Madison, Wisconsin

[20] Olston C, Jiang J, Widom J (2003) Adaptive Filters for Continuous Queries over Distributed Data Streams. SIGMOD, San Diego, California

[21] Gilbert A, et al (2001) Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. VLDB Conference: 79-88

[22] Gilbert A, et al (2003) One-Pass Wavelet Decompositions of Data Streams. Transactions on Knowledge and Data Engineering 15(3): 541-554

[23] Guha S, et al (2001) Data Streams Histograms. Symp Theory of Computing: 471-475

[24] Wiederhold G (1992) Mediators in the Architecture of Future Information Systems. IEEE Computer: 38-49

[25] Chen Y, et al (2002) Efficient Global Optimization for Image Registration. IEEE Transactions Knowledge and Data Engineering 14(1): 79-92

[26] Sun Microsystems (2003) Java Data Objects [Online] Available: http://access1.sun.com/jdo/

[27] FastObjects (2003) FastObjects t7 [Online] Available: http://www.fastobjects.com/us/FastObjects_PandS_t7.asp