LIVE VIDEO DATABASE MANAGEMENT SYSTEMS

by

RUI PENG
M.S., University of Central Florida, 2008
B.S., Zhejiang University, 2001

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2010

Major Professor: Kien A. Hua

ABSTRACT

With the proliferation of inexpensive cameras and the availability of high-speed wired and wireless networks, networks of distributed cameras are becoming an enabling technology for a broad range of interdisciplinary applications in domains such as public safety and security, manufacturing, transportation, and healthcare. Today's live video processing systems on networks of distributed cameras, however, are designed for specific classes of applications. To provide a generic query processing platform for applications of distributed camera networks, we designed and implemented a new class of general purpose database management systems, the live video database management system (LVDBMS). We view networked video cameras as a special class of interconnected storage devices, and allow the user to formulate ad hoc queries over real-time live video feeds.

In the first part of this dissertation, an Internet scale framework for sharing and dissemination of general sensor data is presented. This framework provides a platform for general sensor data to be published, searched, shared, and delivered across the Internet. The second part is the design and development of a Live Video Database Management System. LVDBMS allows users to easily focus on events of interest from a multitude of distributed video cameras by posing continuous queries on the live video streams. In the third part, a distributed in-memory database approach is proposed to enhance the LVDBMS with an important capability of tracking objects across cameras.

ACKNOWLEDGMENTS

Earning a Ph.D is by far the most challenging task for me. Fortunately I did not have to go through everything all by myself. I take this opportunity to thank the people who have been there when I needed them the most and to whom I am deeply grateful.

My academic advisor, Dr. Hua, is definitely the first one I want to thank. His hard work and attention to details is exemplary to all his students and ensures the best quality of his Ph.D graduates. He has always been there for me when I was confused, frustrated, or desperate. He is the utmost reason that I was able to finally march to the end of the tunnel after so many difficult years. I would like to thank my dissertation committee members for providing their valuable and helpful suggestions. My dear friends at the Data Systems Lab are a great treasure to me and have always been of solid support to my research as well as my personal life. Many of them now have a great job which they well deserve after years of extraordinarily hard work at UCF. I wish them the best luck in their career and personal success.

I thank my wife, Yifang, from the bottom of my heart for her understanding, support, and sacrifice. Her company has made me feel that I am the happiest man in the world, even when I was going through my biggest setbacks these years. She stood firmly behind me and unselfishly backed me up on every decision I made, including switching my career path to finance. I owe everything to her and this Ph.D degree belongs to her as well. My parents, Junmin and Qi, also receive my warmest thanks. Their confidence in me and their support, financially and spiritually, is my source of power to fight my way through the rough years. Last but not least, I thank my parents-in-law who always cared for me and my wife even when they were going through difficulties themselves. I am greatly proud of my family. Without them none of my achievements would have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Cameras are a special class of sensors that are widely used in many applications ranging from traffic monitoring, public safety and security to healthcare and environmental sensing. They generate great volumes of data in the form of live video streams. In contrast to entertaining movies, such live video captured by specialized cameras are generally not as interesting. People who constantly monitor these cameras, such as baggage screeners at airports, can quickly become fatigued. Moreover, when tens of thousands of cameras are present, such as those deployed on streets in a city, it is expensive, if physically feasible, to house a huge number of monitors to support the various monitoring operations. In many scenarios, critical events do not happen very often, and it is quite inefficient to have one monitor tracking only a few cameras constantly and intensely. With latest advances in VLSI technology, networks of distributed cameras, consisting of many inexpensive video cameras and distributed processors, will bring far greater monitoring coverage. It will become virtually impossible for human beings to keep track of all the objects or events under each camera. The distributed nature of these networks, coupled with the real-time nature of live videos, greatly complicates the development of techniques, architectures, and software that aim to effectively mine data from a world of live video feeds. In this dissertation, we propose a *Live Video Database Management System* (LVDBMS) as a solution to address the aforementioned problems. LVDBMS is designed to allow users to easily focus on events of interest from a multitude of distributed video cameras by posing continuous queries on the live video streams. With LVDBMS automatically and continuously monitoring live videos feeds and taking care of the intercommunications among distributed processors, a user can easily use a desktop display to manage a very large number of cameras and receive notifications when critical events happen.

The main contributions of this dissertation are the design and implementation of novel techniques proposed to enable LVDBMS and these techniques are presented in three major parts of this dissertation.

The first part is a generic framework to share and disseminate general sensor data across the Internet. The rapid increase of sensor networks has brought a revolution in pervasive computing. However, data from these fragmented and heterogeneous sensor networks are easily shared. Existing sensor computing environments are based on the traditional database approach, in which sensors are tightly coupled with specific applications. Such static configurations are effective only in situations where all the participating sources are precisely known to the application developers, and users are aware of the applications. A pervasive computing environment raises more challenges, due to ad hoc user requests and the vast number of available sources, making static integration less effective. This part of the dissertation presents an Internet framework called *i*SEE (Internet Sensor Exploration Environment) which provides a more complete environment for pervasive sensor computing. *i*SEE enables advertising and sharing of sensors and applications on the Internet with unsolicited users much like how Web pages are publicly shared today.

The second part is the design and development of a LVDBMS. With the proliferation of inexpensive cameras and the availability of high-speed wired and wireless networks, networks of distributed cameras are becoming an enabling technology for a broad range of interdisciplinary applications in domains such as public safety and security, manufacturing, transportation, and healthcare. Today's live video processing systems on networks of distributed cameras, however, are designed for specific classes of applications. To provide a generic query processing platform for applications of distributed camera networks, we designed and implemented a new class of

general purpose database management systems, the live video database management system (LVDBMS). We view networked video cameras as a special class of interconnected storage devices, and allow the user to formulate ad hoc queries expressed over real-time live video feeds. In this part of the dissertation we introduce our system and present the live video data model, the query language, and the query processing and optimization technique.

The third part is a cross camera tracking technique using a distributed in-memory database approach. Cross camera tracking in real-time with non-overlapping views is an enabling functionality for LVDBMS but still remained an unsolved problem due to its computation intensiveness in a time-stringent environment with a large number of live video feeds. The limitation with the form of LVDBMS in the second part is that objects are detected and tracked only within the same scene captured by the same camera. When an object moves out of sight of the camera and enters the view of another camera or even the same camera afterwards, it will be recognized as a new object. The motivation of this work is to fill this void and greatly enhance LVDBMS with the critical capability to track objects from a camera to another when at the same time fulfilling the real-time requirement of a live video database. In this part of the dissertation we propose a distributed in-memory database approach to tackle this problem by pushing down object tracking operations to be as close to the cameras as possible and leveraging an efficient color histogram-based tracking mechanism to track objects across cameras.

The rest of this dissertation is organized as follows. Chapter 2 discusses the Internet Sensor Exploration Environment (*i*SEE). A Live Video Database Management System (LVDBMS) is presented in Chapter 3. In Chapter 4 we propose the distributed in-memory database approach to tracking objects across cameras. Finally the concluding remarks of this dissertation are offered in Chapter 5.

2. INTERNET SENSOR EXPLORATION ENVIRONMENT – iSEE

2.1 Introduction

The emergence of pervasive computing technology has enabled many applications in different environments (e.g., (Diegel, 2004; Symonds, 2007)). With the advent of modern technology that enables massive production of small, inexpensive, and wireless networked sensors, distributed sensor networks have become pervasive nowadays and revolutionized pervasive computing. Instead of personal computers, hundreds of sensor networks are to be deployed everywhere, providing a pervasive environment that enables people to move, work, and communicate without knowing the presence of computers processing sensor data. However, such ideal settings are not easy to realize, as data captured from these systems are constantly buried in the Internet, partly due to the dynamic nature of the data and the lack of a common framework for sharing such information in the public. While emerging sensor networks provide us with a vision of a powerful pervasive computing environment, feasible frameworks to share data across fragmented and heterogeneous sensor networks have not yet taken a concrete shape, due to the lack of data sharing techniques and cooperation among different sensor data providers. As ever-growing sensor-based services become part of our daily life, they call for new technologies that enable publishing, searching, browsing, and integrating sensor data on the Internet.

Consider sensor networks, called Sensor Webs (Delin, 2001), deployed by NASA in different locations which include several greenhouses at Huntington Botanical Gardens in California, wetlands at the Kennedy Space Center in Florida, ice sheets in Antarctica, deserts in the mid-west, and a simulated greenhouse in an Amazonian rainforest. Although real-time streaming outputs from these deployments can be viewed at the "NASA/JPL Sensor Web" web site, the sharing of these data streams must be through a specific application from NASA, i.e. the

visualization software, and there is no easy way to leverage new research tools developed by third parties for these data. It is also inconvenient to deploy data fusion tools to combine NASA sensor data with those of other organizations.

The enablement of these absent functionalities requires a new sensor computing environment that facilitates:

- Sensor data sharing,
- Sensor application sharing, and
- On-the-fly data integration from various sensor sources.

Recently there has been a growing interest in sensor data management with research activities focusing mainly on either:

- dealing with packet routing and power conservation issues as well as secure communication mechanisms in sensor networks (Ganesan, 2003; Intanagonwiwat, 2000), or
- managing the sensor networks as a distributed database (Demers, 2003; Gibbons, 2003; Madden, 2002a; Madden, 2002; Tan, 2007).

The latter line of work is more relevant to our work. Madden (2002a) proposes a technique, called Fjords, for query processing on continuous, never-ending sensor data streams. In this work, the sensor network is modeled as a streaming data source by providing a sensor proxy as the sensor's interface into the query processor. Madden (2002b) develops the TAG service that distributes declarative queries into the sensor network and coordinates sensors on in-network aggregation. This scheme pushes query operators into the network and aggregates partial results at intermediate nodes, resulting in greatly improved query efficiency with decreased power usage.

In the Cougar Sensor Database Project, Demers (2003) presents a database approach to sensor networks, i.e. the client "programs" the sensors through queries in a high-level declarative language similar to SQL. In this project, sensor streams are modeled as virtual relations. These studies present novel architectures for sensor query processing using database technology, and provide effective techniques for sensor databases and query systems. However, their primary limitation in supporting sensor computing over the Internet is the problem scale, being within a single sensor network. Internet sharing of sensor data is therefore not facilitated in this environment as client-server architecture is assumed between the user query interface and the sensor database, as shown in Figure 1.

Figure 1. Client-Server architecture of TinyDB and Cougar

In another project called the IrisNet, Gibbons (2003) presents techniques for building and querying wide area sensor databases. In contrast to TinyDB and Cougar, this work deals with widely deployed, resource-abundant, powered sensors such as webcams and organizes them into a distributed database. This entire database is logically a single XML document that is partitioned among multiple sites. Original and effective techniques are proposed for fragmenting and partitioning a database, routing and processing queries, and caching remote data locally. That work supports sensor data sharing in the sense that a web-accessible query interface is provided for users to access the database. However, Gibbon's approach tightly couples its application with its data, i.e. it prevents third-party applications other than the customized query

interface from accessing the data and therefore provides no support for application sharing, as illustrated in Figure 2. Furthermore, integration of data from IrisNet with those of other organizations, say a TinyDB database, is nearly impossible since an IrisNet application only works with a predefined database.

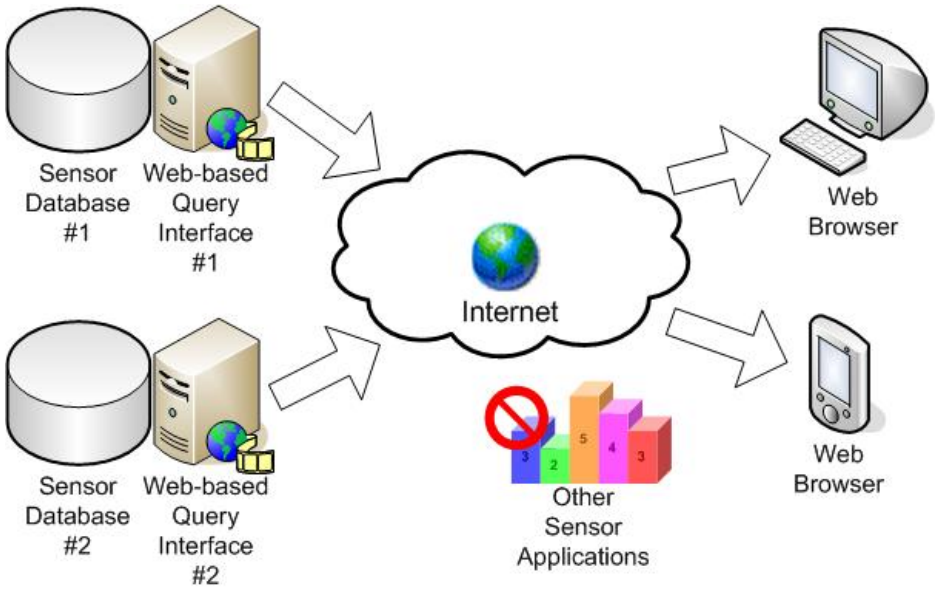


Figure 2. Web-based architecture of IrisNet.

Generally speaking, all three aforementioned projects (i.e. TinyDB, Cougar, IrisNet) take the database approach in the sense that a sensor network is modeled as a distributed database, regardless of the different sensor types and deployment scales. These schemes are too restrictive to allow sharing of sensor data in real time as required in a pervasive computing setting. These environments are also not designed to support application sharing and on-the-fly data integration. All these limitations are addressed in this work.

We have designed and implemented a new pervasive sensor computing framework, called ***i*SEE** (*Internet Sensor Exploration Environment*), to provide an environment for sharing sensor

information and applications over the Internet. In particular, the *i*SEE framework provides the following functionalities:

(1) allowing data owners to conveniently publish information about their sensors, and

(2) assisting unsolicited data users in

- efficiently searching for relevant sensor data,
- logically integrating the information on-the-fly, and
- intelligently browsing them in a wide variety of ways.

By extending the Internet into the physical world, we envision an Internet of sensors, where innumerable sensing systems, deployed by numerous service providers expose different real-time data which can be shared freely among a wide variety of unsolicited users, much like the way web pages are publicly shared today.

The remainder of this chapter is organized as follows. We discuss the design issues in detail in Section 2.2. The *i*SEE architecture and core techniques are introduced in Section 2.3. The *i*SEE prototype is presented in Section 2.4. We present our experimental results in Section 2.5. Finally, we offer our concluding remarks in Section 2.6.

2.2 **Design issues of iSEE**

In this section, we discuss the design issues needing to be addressed in *i*SEE in order to overcome the limitations of today's sensor database techniques.

What components would *i*SEE need to have? First, a publishing environment needs to be in place to facilitate the sharing of sensor data on the Web. Second, a search mechanism is needed to enable unsolicited users to discover relevant sensors. Finally, an intelligent browser provides

the user interface to view, query, analyze, and integrate sensor data. More specifically, the *i*SEE framework focuses on the following issues:

Sensor data publishing. Since publishing is the first step for sharing data with unsolicited users, a publishing mechanism is required to expose streaming sensor data sources available online. Sensors are generally limited in power and memory (Callaway, 2003), and must deliver data continuously at well-defined intervals in the form of streams (Madden, 2002a). While current web servers show good efficiency when publishing static objects such as HTML web pages, they are not capable of publishing dynamic sensor streams. It is not feasible in terms of timeliness to first save the streaming data into a file and then share it as a static object.

In our approach, we have designed a *Sensor Service Description Language* (**SSDL**) for sensor providers to create XML advertisements for their sensor services, where each service consists of one or more data streams. The SSDL uses a set of XML schemas to capture sensor meta-data so that anyone who discovers the advertisements will have sufficient information to access the sensors' real-time data. *i*SEE also provides a *Publication Facility* for publishing SSDL advertisements on the Internet.

Meta-data management. Meta-data are structured descriptions of the actual data collected by sensors or sensor networks. Meta-data provide the semantics of the real data, and play an important role in data sharing. On one hand, sensor owners need to use meta-data to advertise their sensors; on the other, data users rely on them to identify and access the appropriate data. A major challenge for effective meta-data management is the heterogeneity in sensor meta-data. There is currently neither a standard for sensor meta-data creation, nor a shared framework to manage them. Different research groups gather data with different goals in mind, and most probably employ a meta-data schema best for data collection, not for sharing.

In *i*SEE, we have built a meta-data management framework based on an online *Sensor Registry Server*, which registers SSDL advertisements, stores them in its database, and facilitates advertisement discovery. Essentially, *i*SEE addresses the challenge of meta-data heterogeneity by leveraging SSDL as a standard for sensor meta-data representation.

Streaming data retrieval. In order for a user to access sensor data over the Internet, communication protocols need to be defined to regulate data delivery. Current real-time communication protocols, such as RTSP (Schulzrinne, 1998) and RTP (Schulzrinne, 1996), focus on multimedia delivery and address issues like encoding/decoding, synchronization, etc. They cause unnecessary overhead for sensor data delivery that has a more relaxed real-time requirement.

*i*SEE provides a new light-weight protocol called the *Sensor Stream Control Protocol* (**SSCP**) that defines a set of interfaces between the server and browser. Through this protocol, users can remotely control the "playback" of data streams similar to playing back video. SSCP augments another *Sensor Stream Transport Protocol* (**SSTP**), which defines a packet format for data exchange, to transmit real-time streaming data.

Sensor data browsing. Sensor data need to be presented to the user once they are delivered. While HTML web pages contain all the information for a standard web browser to display, streaming sensor data do not. If original sensor data are shown on the screen without interpretation, the user will only see a screen of numbers. Special stream presentation techniques are needed to provide users with more insight into the data and more intuitive browsing experience.

Summarization is a natural way to present data streams with potentially unlimited information. In this work, we consider two summarization approaches, namely data stream visualization and

query processing. We note that other techniques such as data mining and various statistical analysis techniques can also be added later. Our *i*SEE *Sensor Data Browser* is capable of (1) searching for relevant sensors, (2) receiving data streams from sensor data sources, (3) integrating multiple sensor data sources on-the-fly, (4) presenting data streams with an embedded Visualizer module, and (5) selecting registered third-party sensor applications to access the data.

Sensor application sharing. Sensor applications are software programs or tools that interpret, process, and present sensor data. Examples include data visualization modules, analysis tools, and query processors. Sensor data are heterogeneous in semantics and formats. To help users better understand complex sensor data, data providers may design a specific application for their sensors. Furthermore, third parties might want to provide or sell tools for popular sensor servers available online, e.g., analyzing NASA's data to extract useful information or combining NASA's data with other data sources. Thus, it is highly desirable to allow seamless integration of new sensor applications into the browser to enhance its capability.

Our solution to this issue is the *Application Plug-in Mechanism* that allows an *i*SEE user to install sensor applications as software plug-ins to extend the functionality of the basic *i*SEE browser. Through this mechanism, any sensor application developed by the data provider, the end user, or a third party can be conveniently plugged into the *i*SEE browser to access sensor data, provided it implements the required interfaces. Thus, the *i*SEE browser provides flexibility and extensibility in that it not only includes a generic *Visualizer* plug-in for general data presentation, but can also utilize customized application plug-ins to achieve more complex data visualization and/or analysis tasks.

On-the-fly data integration. Currently available approaches for sensor information integration are based on distributed databases consisting of predefined sets of data sources with homogeneous schemas. Such static configurations are effective only in situations where all the participating sources for the given application are precisely known, and users are aware of their application. An open web environment raises more challenges, due to the ad hoc user requests and vast number of available sources, making static integration substantially less effective. The challenge is to significantly reduce the time and skill needed to integrate data sources on-the-fly.

A significant advancement brought forth by this research in *i*SEE is the incorporation of capabilities to facilitate ad hoc integration of sensor sources immediately after they are discovered. In *i*SEE, we allow data providers to define local schemas for their sensor data sources using SSDL, and end users to compose *user-defined schemas* as an integrated schema of multiple local sensor schemas. Such user-defined schemas, providing a specific view into the world of sensors, are similar to the *view* concept in traditional database management systems. User-defined schemas can also be shared with other users as a special type of sensor application plug-in. Thus, in contrast to today's approaches, the *i*SEE browser is not tightly coupled with any specific sensor data source, but rather enables the user to connect to sensor data sources in an ad hoc manner and integrate them on the fly to create new information.

Figure 3 depicts the *i*SEE environment. The fundamental differences between *i*SEE and the traditional database approach are presented in Table 1. We summarize these differences as follows:

- The database approach generally requires new application development to support new sensor data, whereas the same *i*SEE browser can be used to access newly deployed data sources.

- In the database approach, users are aware of their sensor applications. In contrast, sensors are advertised in the *i*SEE environment to share the data with potentially very large numbers of unsolicited users.

- *i*SEE enables the user to compose a user-defined schema to do ad hoc integration of various *i*SEE data sources. In contrast, data integration is static in the database approach as the data sources are tightly coupled with the application.

- *i*SEE facilitates sharing of new applications and tools for a given data source. In the database approach, the database schema is known only to the application developer. This makes application sharing difficult.

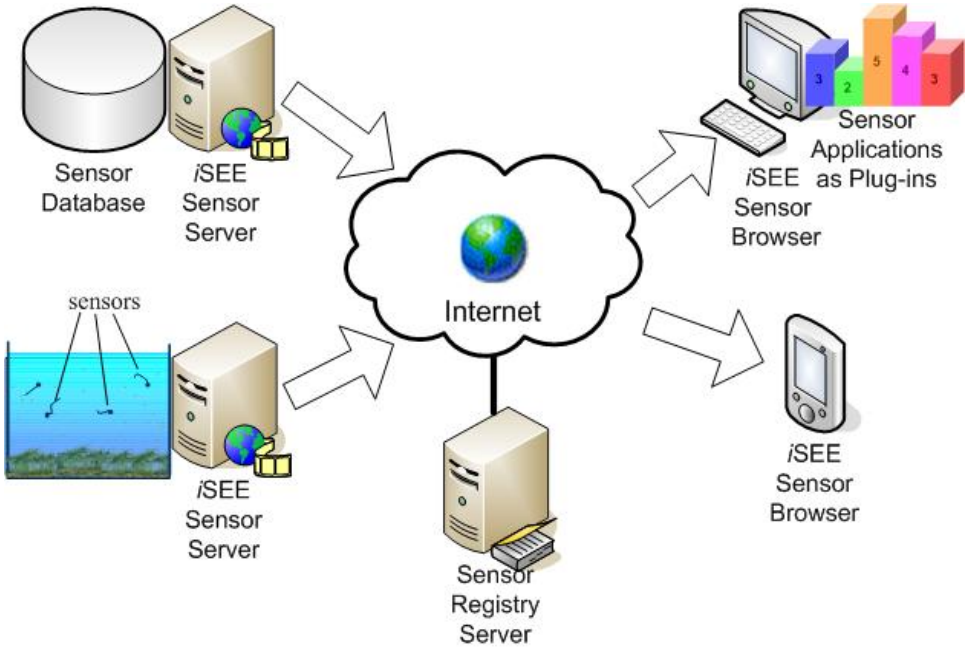


Figure 3. Sensor computing environment of *i*SEE

Table 1. Differences between Sensor Database Management Systems and *i*SEE

	Database Approaches	Proposed *i*SEE Framework
Applications	Customized	Generic Browser
Data Users	Targeted users	Unsolicited users
Data Integration	Predefined	Ad hoc
Application Sharing	Difficult	Shared as software plug-ins

Thus, the contribution of this work is the development of an Internet framework *i*SEE which provides a more complete environment for pervasive sensor computing compared to existing solutions. *i*SEE enables users to share not only sensor data, but also sensor applications. Furthermore, it facilitates ad hoc data integration of various sensor sources.

2.3 iSEE framework

Having discussed the design issues and the *i*SEE approach in the last section, we describe the *i*SEE framework in this section. We begin with an overview before discussing the various components in detail.

2.3.1 Overview

For illustration purposes, consider a stream monitoring sensor network deployed in a brook by a group of geographers. The sensors measure water temperatures, turbidity, and precipitation, and periodically send back the data to an Internet-connected computer. These data arrive as a stream and are published on the Internet by the geographers. Independently, a biologist, studying the life zone in the same brook, searches for the information on the Internet. Within a few seconds, she learns about the data published by the geographers, and starts to browse and analyze the data

stream on her own computer. As the biologist keeps discovering more sensors deployed by other researchers in other parts of the brook, she begins keeping track of the parts of the brook having the highest temperature. *i*SEE enables these data sharing activities with the basic operations illustrated in Figure 4.

To publish the data, the geographers provide relevant meta-data for their sensors, and a *sensor service advertisement* is automatically generated by the *publication facility* to hold the meta-data using the *Sensor Service Description Language* (SSDL). A *Sensor Data Server* publishes the service on the Internet by registering the advertisement with the *Sensor Registry Server* (SRS), which stores and indexes it in a database. When the biologist looking for related sensors enters search criteria into a local *i*SEE *Sensor Browser*, this software sends a search request to the SRS. After SRS completes the search process, it sends back the relevant advertisements to the browser. As the browser displays the descriptions of discovered services on the screen, the biologist identifies the desired service and clicks on it. In response, the browser establishes a session with the sensor server using a *Sensor Stream Control Protocol* (SSCP), gets the real-time stream using a *Sensor Stream Transport Protocol* (SSTP), and directs the data to the embedded Visualizer for display. The biologist also has the option to exploit any of the registered plug-ins within the browser to process and/or present the data in a variety of ways. In this case, the new plug-in needs to be registered with the *Plug-in Manager* to extend the functionality of the *i*SEE browser. The various components of the *i*SEE framework and their functionality are described in the following sub-sections.

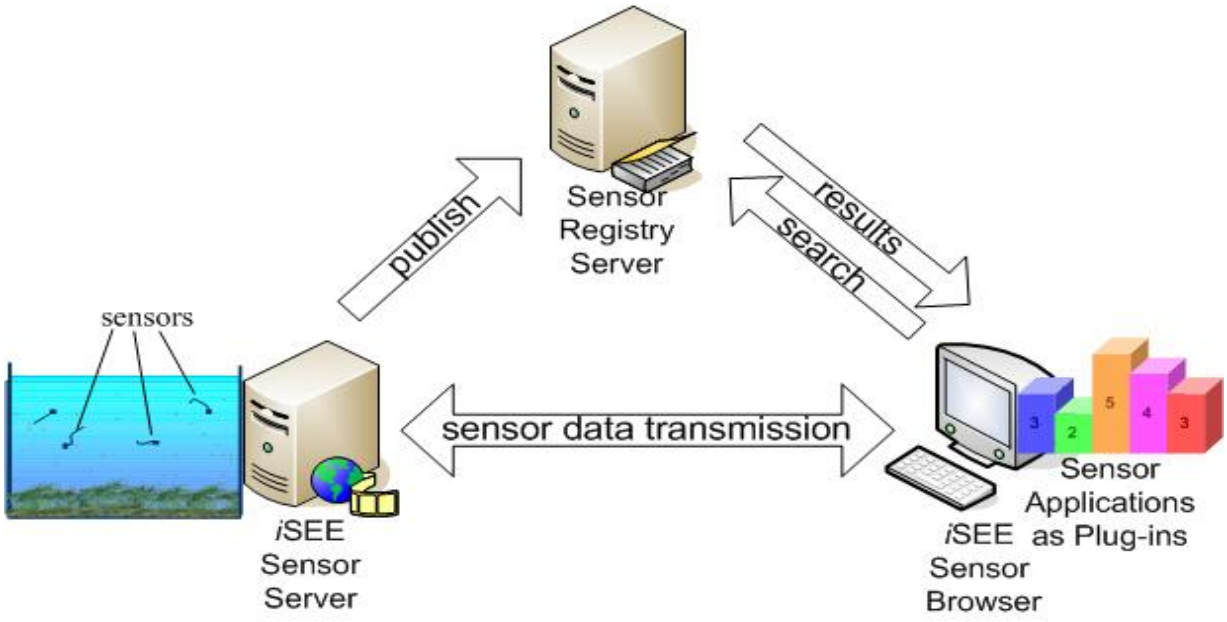


Figure 4. Basic operations in *i*SEE environment

2.3.2 Sensor Service Description Language

The Sensor Service Description Language (SSDL) provides a set of standard formats for defining sensor services. The publication facility at the sensor server allows data providers to enter meta-data and automatically creates sensor service advertisements according to the XML schemas given by SSDL. Each service advertisement advertises a sensor service consisting of a collection of related sensors. For example, we can publish traffic monitoring sensors deployed at busy locations throughout the city to support various transportation engineering applications.

Figure 5 shows two important XML schemas in SSDL and a sample service description is given in Figure 6. As shown in this example, a sensor service has the following important fields in addition to general descriptive fields such as name, description, location, and provider.

- Target: This is entry point for accessing the advertised service.

- Protocol: This field specifies the protocol information used for this service. There are two protocols in *i*SEE environment, namely the SSCP and SSTP.
- Updatetime: It indicates the date when this version of the advertisement was updated.

SSDL allows sensor providers to customize the format of their sensor data streams in the user-defined field DataField. For instance, the sample description in Figure 6 indicates four distinct data fields in the sensor data: Temperature, Turbidity, Precipitation, and Time. In *i*SEE, we define a data frame as the unit for sensor data transmission and a data stream as a sequence of data frames, each consisting of a datum for each of the data fields. A data stream in this sense can also be logically viewed as a streaming relational table, each row of which is a data frame. Therefore, defining a data stream in *i*SEE is similar to creating a table in traditional databases.

By providing standard formats for sensor data, *i*SEE enables data integration and creation of user-defined views, and avoids accessing incompatible data from different sensor sources. In *i*SEE, users can publish not only the sensor data itself but also the various applications which facilitate the different uses of those sensor data. These applications are deployed as software plug-ins in *i*SEE Sensor Browsers. SSDL also supports the publication of such plug-ins. Consequently, a highly popular sensor data service would encourage third-party development of tools and applications as *i*SEE plug-ins.

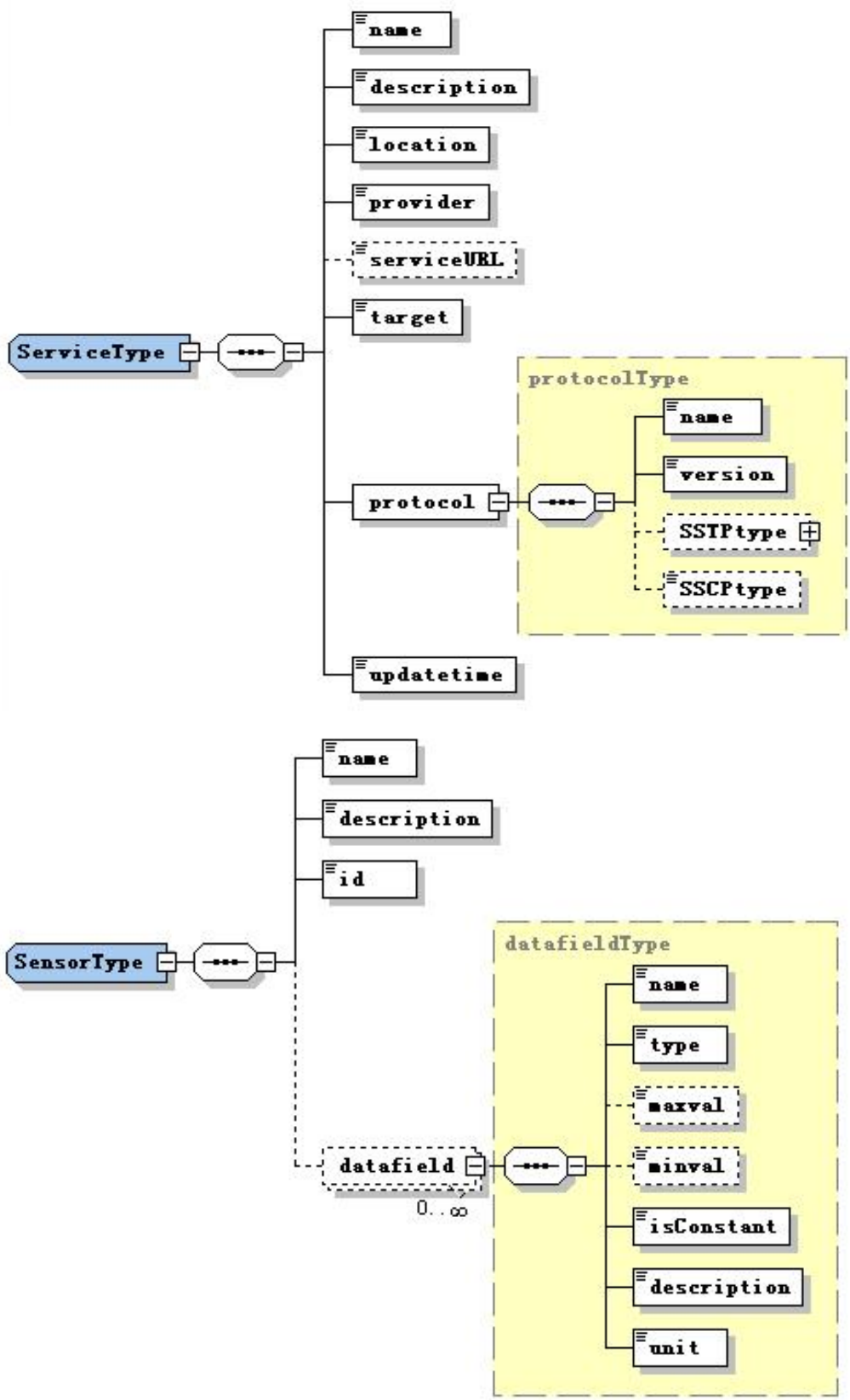


Figure 5. Schemas for defining a service and its data streams

```
<?xml version="1.0" encoding="UTF-8"?>
<SSDL xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="D:\SIGMOD2005\SSDL\SSDL.xsd">
    <Service>
        <name>Stream Monitoring in Great Brook</name>
        <description>Monitors the water temperature, turbidity and precipitation in the Great Brook</description>
        <location>Great Brook</location>
        <provider>Data System Group</provider>
        <target>http://localhost:8080/services/Sensorservice</target>
        <protocol>
            <name>SSCP</name>
            <version>1.0</version>
        </protocol>
        <updatetime>11-1-2004</updatetime>
    </Service>
    <Stream>
        <id>Stream1</id>
        <protocol>
            <name>SSTP</name>
            <version>1.0</version>
            <SSTPtype>
                <type>MulticastUDP</type>
                <MulticastIP>229.6.7.8</MulticastIP>
                <MulticastPort>4123</MulticastPort>
            </SSTPtype>
        </protocol>
        <Sensor>
            <name>Great Brook Sensor #1</name>
            <description>Monitors the water temperature, turbidity and precipitation in the Great Brook</description>
            <id>gbrook</id>
            <datafield>
                <name>Temperature</name>
                <type>byte</type>
                <isConstant>false</isConstant>
                <description>Water temperature</description>
                <unit>Fahrenheit</unit>
            </datafield>
            <datafield>
                <name>Turbidity</name>
                <type>decimal</type>
                <isConstant>false</isConstant>
                <description>Water turbidity</description>
                <unit>NTU</unit>
            </datafield>
            <datafield>
                <name>Precipitation</name>
                <type>decimal</type>
                <isConstant>false</isConstant>
                <description>Precipiation depth</description>
                <unit>inch</unit>
            </datafield>
            <datafield>
                <name>Time</name>
                <type>time</type>
                <isConstant>false</isConstant>
                <description>Time when the datum is sampled at the sensor</description>
                <unit>ms</unit>
            </datafield>
        </Sensor>
    </Stream>
</SSDL>
```

Figure 6. A sample sensor service advertisement

2.3.3 iSEE Sensor Server

The *i*SEE sensor server is responsible for publishing sensor data on the Internet and delivering the data to browsers upon request. As shown in Figure 7, it has three major components, namely the *Publish Manager*, the *Stream Manager*, and the *Session Manager*.

The Publish Manager allows a user to provide sensor meta-data, from which a service description is generated according to SSDL. It then automatically connects to the sensor registry server and registers the sensor data source. The Stream Manager accepts incoming sensor data and directs them to either local applications (e.g. local file system, databases), or remote clients through the Server Session Manager.

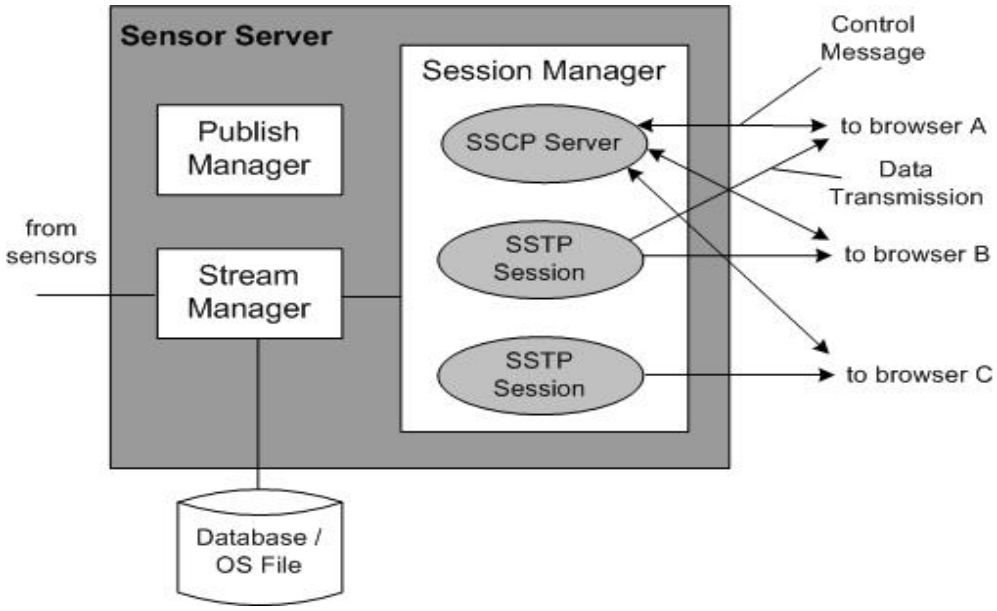


Figure 7. *i*SEE Sensor Server

The Server Session Manager interacts with remote clients and provides two important functionalities. First, it accepts client requests and control messages using a SOAP server. The operations supported by the SOAP server are defined in the SSCP. Second, it delivers the stream to clients using SSTP. The details of SSCP and SSTP are discussed in Section 3.6. For the first

request for each stream, the Server Session Manager spawns a new SSTP session; whereas each subsequent request for the same stream joins the existing session. Essentially, a browser uses two separate channels to interact with the server, one for control messages, and one for data delivery. The control channel is bi-directional, but the data channel only goes in one direction.

2.3.4 Sensor Registry Server

The Sensor Registry Server defines a set of services supporting the publication and discovery of service advertisements. Specifically, our current Sensor Registry Server has the following capabilities, as illustrated in Figure 8.

It accepts sensor service advertisements from data providers, which are stored, classified, and indexed within the internal advertisement database.

It enables unsolicited users, unaware of the deployed sensor networks, to identify their desired data through a sensor browser. At the same time it also provides sufficient information for the browser to access the service.

It allows information seekers to browse the registry data using any kind of web browser.

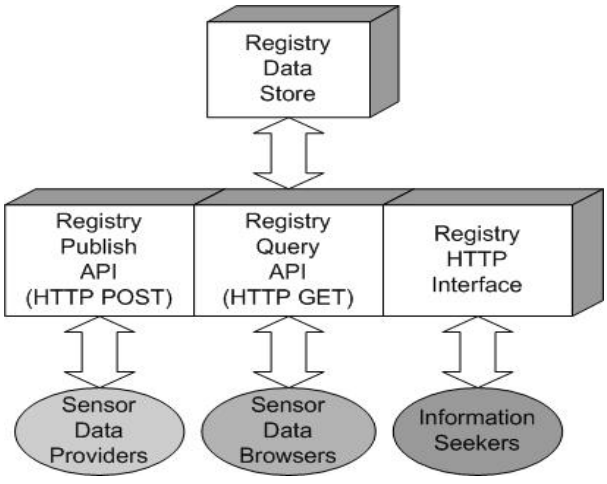


Figure 8. Sensor Registry Server

Universal Description, Discovery, and Integration (UDDI) [20] is a registry protocol that defines a set of services supporting the description and discovery of businesses and their exposed web services. This registry serves well in the e-business environment as an advertisement. However, the current UDDI specification (Version 3) contains a relatively fixed set of fields, such as *Name, Description*, and URI, and it is unable to accommodate dynamic service descriptions such as sensor service advertisements in any of its fields. Therefore it is not suited for the *i*SEE environment. We note that although the Sensor Registry Server is designed primarily for the *i*SEE environment, it can be easily extended to accept any machine readable descriptions of resources and can potentially become a generic registry server on the Internet.

2.3.5 iSEE Sensor Browser

The *i*SEE Sensor Browser consists of three components, the *Discover Manager*, the *Session Manager,* and the *Plug-in Manager*, as depicted in Figure 9.

The Discover Manager is responsible for discovering relevant sensor services on the Internet. It accepts user search criteria from the browser user interface, formulates a query request, and sends the request to the Sensor Registry Server. As the search results containing the service advertisements return, an internal XML parser extracts useful information from the advertisements and presents a summary of every discovered service to the user.

The Session Manager is the counterpart of the Session Manager at the Sensor Server. Upon user's request for specific data streams, it connects to the Sensor Server and establishes a session for each incoming data stream. A session has two channels, control channel and data channel, for transmitting SSCP control messages and SSTP data packets, respectively. The incoming data streams are directed to the Plug-in Manager for processing and presentation to the user. Optionally, it can also dump the data into a local file or database.

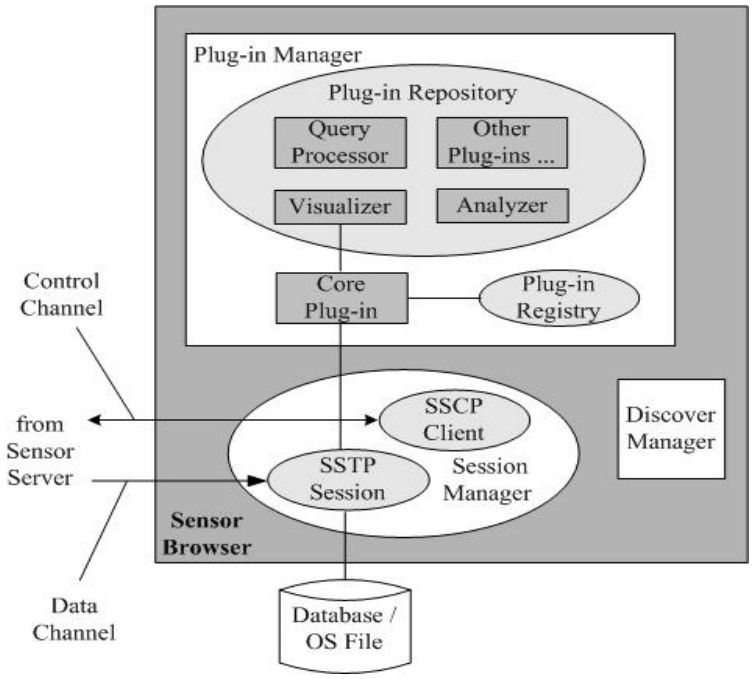


Figure 9. *i*SEE Sensor Browser

The Plug-in Manager accepts registration of plug-in modules, and dynamically invokes plug-ins at the user's choice. The invocation of any plug-in in the Plug-in Repository is through the core plug-in, which looks in the Plug-in Registry for the entry point of the requested plug-in before forwarding data to it. In the current *i*SEE environment, the sensor browser has an embedded plug-in, the *Visualizer*. The browser can easily be enhanced with the addition of more complicated plug-ins such as continuous query processors (e.g. (Abadi, 2003; Arasu, 2004; Olston, 2003)) and data stream mining tools (e.g. (Aggarwal, 2003)). As long as a plug-in implements the necessary interfaces required by the Plug-in Manager and registers with the Plug-in Manager, users will be able to see and select it from the browser interface.

Currently, the embedded Visualizer is responsible for displaying the incoming data streams on the screen by default. While there could be hundreds of ways of presenting data to the end user, visualization is probably the most intuitive for most situations. The design of the current *i*SEE Visualizer implements a simple *Presentation Frame* concept analogous to a frame in a video

stream. A *Presentation Frame* is the minimum unit of data for display. That is, a presentation frame is a collection of data frames that are visualized together on the screen. For example, the Visualizer may display the presentation frame by drawing a time series chart where every point corresponds to a data frame. We can view a continuous stream arriving at the Visualizer as a sliding window over the data stream. A sliding window of size six is shown in Figure 10. By visualizing these consecutive presentation frames at a high display rate, we create the experience of motion similar to video. In other words, the standard *i*SEE Visualizer summarizes and presents sensor data to the end user in the form of data animation.

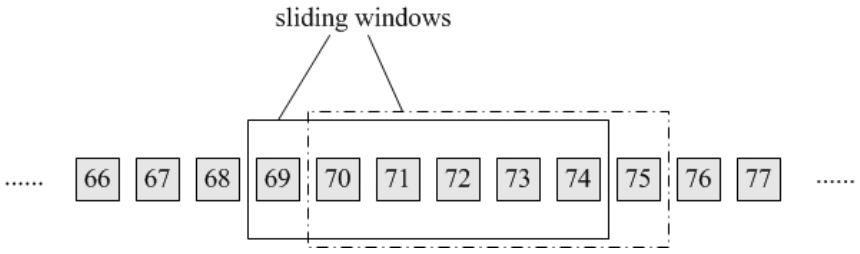


Figure 10. Presentation Frames

2.3.6 iSEE Protocols

The *Sensor Stream Control Protocol* (SSCP) is an application-layer protocol built atop SOAP [21]. It defines the operations a sensor browser can use to interact with a sensor server. As a simple analogy, the SSCP acts as the network remote control for sensor stream playback by providing a set of useful methods. SSCP does not deliver the stream itself, but rather uses *Sensor Stream Transport Protocol* (SSTP) for real-time data transmission. The semantics of the control methods are listed below with an illustration presented in Figure 11.

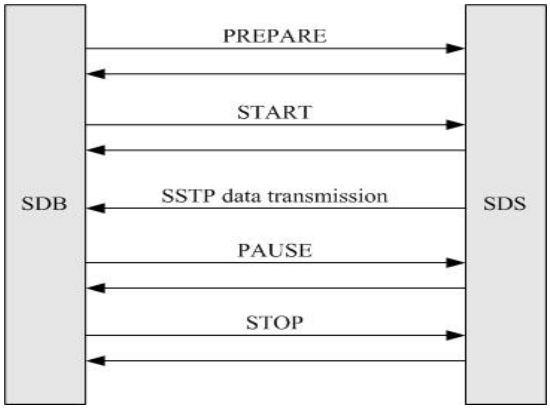


Figure 11. SSCP Operations

PREPARE. Request the sensor server to allocate resources for a sensor stream and prepare for the data delivery. If this is the first request for a particular stream, the sensor server creates a new server session and adds the requesting browser to the session's browser list. Subsequent requests can share this stream by simply adding their browser to this session.

START. Request the sensor server to start the data transmission through the data channel. If this is the first request for a stream, the sensor server starts an SSTP transmission on the data channel to the browser. For subsequent requests for this stream, the sensor server lets their browsers join the existing SSTP transmission. This operation can also be used to resume data transmission after a pause operation.

PAUSE. Temporarily stop the stream transmission without releasing server resources. Any data transmitted during the pause period are lost. If the PAUSE request is from the last active sensor browser on this stream, the sensor server stops the data transmission.

STOP. Notify the sensor server to stop the data transmission, free server resources, and close the corresponding server session if this is the last active browser receiving this stream; otherwise, the sensor server continues the data transmission for other browsers while purging the resources for the stopping browser.

The *Sensor Stream Transport Protocol (SSTP)* is another application-layer protocol for transmitting streaming data. It is based on UDP to leverage its broadcast/multicast capability so that the system scales to a large number of clients. The rationale for using UDP rather than SOAP in sensor data transmission is to reduce overhead. Though SOAP is designed to be a light-weight protocol, it is not necessary to wrap every data packet with a SOAP envelope. In contrast, the current SSTP only adds a sequence number to a data packet. The *sequence number* increments by one for each SSTP packet, and is used by the Sensor Browser to detect packet loss and ensure in-order delivery.

2.4 *i*SEE prototype

As a proof of concept, we have developed a prototype using the techniques discussed in Section 3. The *i*SEE Sensor Browser provides three ways to access and explore a sensor service: (1) use the search mechanism to find relevant services, and open them; (2) select one of the bookmarks, and select the corresponding service; and (3) enter the URL to access the service directly if the information is available. To use the first option to search for a sensor service, the user enters a set of keywords as the search criteria. To make the search more specific, the user can enter additional keywords for the provided metadata fields. The keywords entered for *Datafields* are compared with the names of the sensor data fields declared by the sensor providers.

After the "Search" button is clicked, the browser connects to the Sensor Registry Server and retrieves matching entries from the server. The list of found services is shown on the screen together with the meta-data of the selected service as in Figure 12. The *i*SEE sensor browser displays standard descriptive information such as the name of the service, its description, and the locations of the sensors to help the user identify the desired sensor service. The user can now select a service by clicking on its name and dragging it to the bookmark panel on the right of the

browser. To view the selected sensor data, one can select the "Plug-in Visualization Tool" tab, activating the Visualizer which prompts the user to choose from a number of different visualization methods. Figure 13 shows a screen shot when the user views the variation of the water temperature on a time series chart.

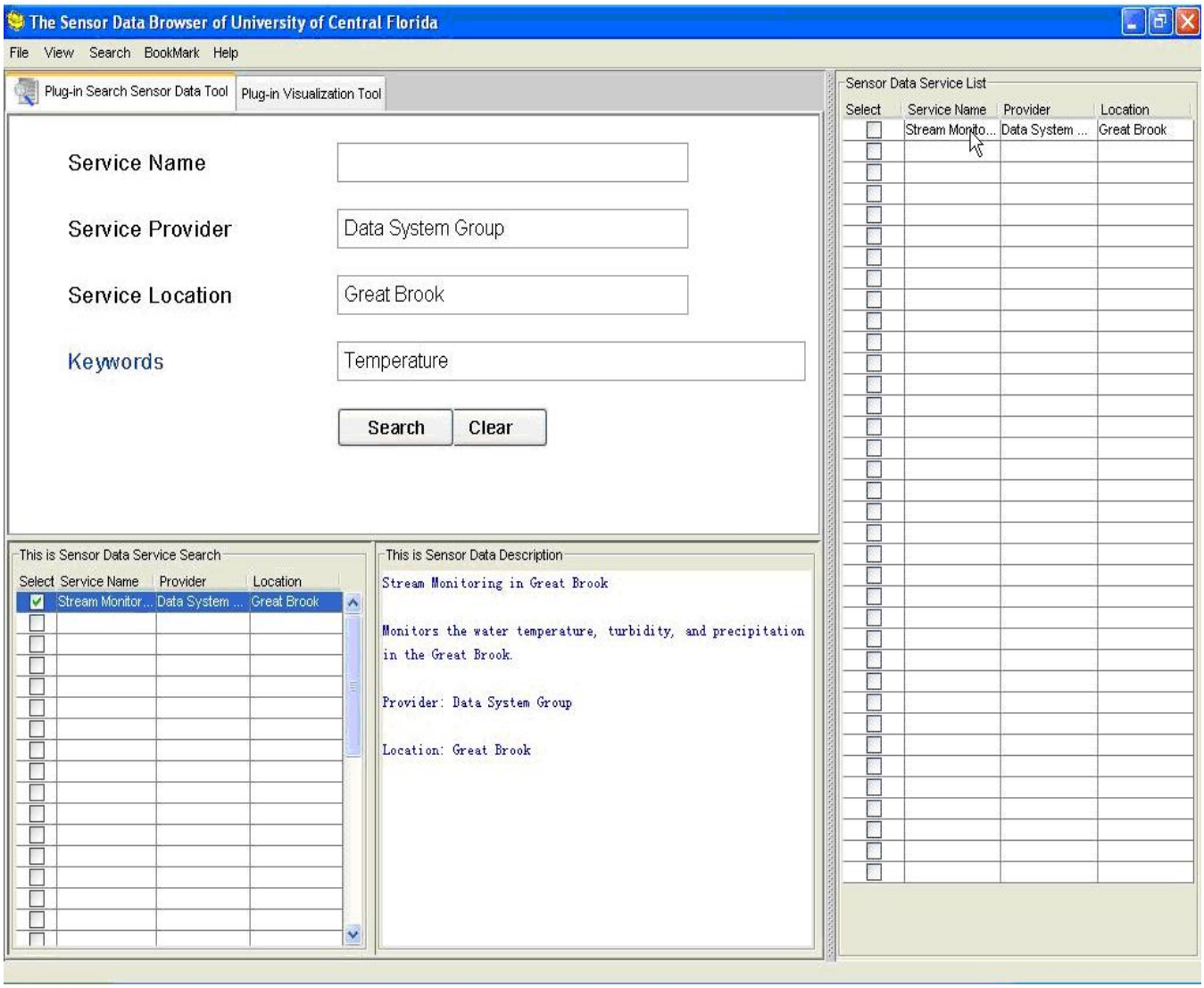


Figure 12. *i*SEE Sensor Browser Interface - Search

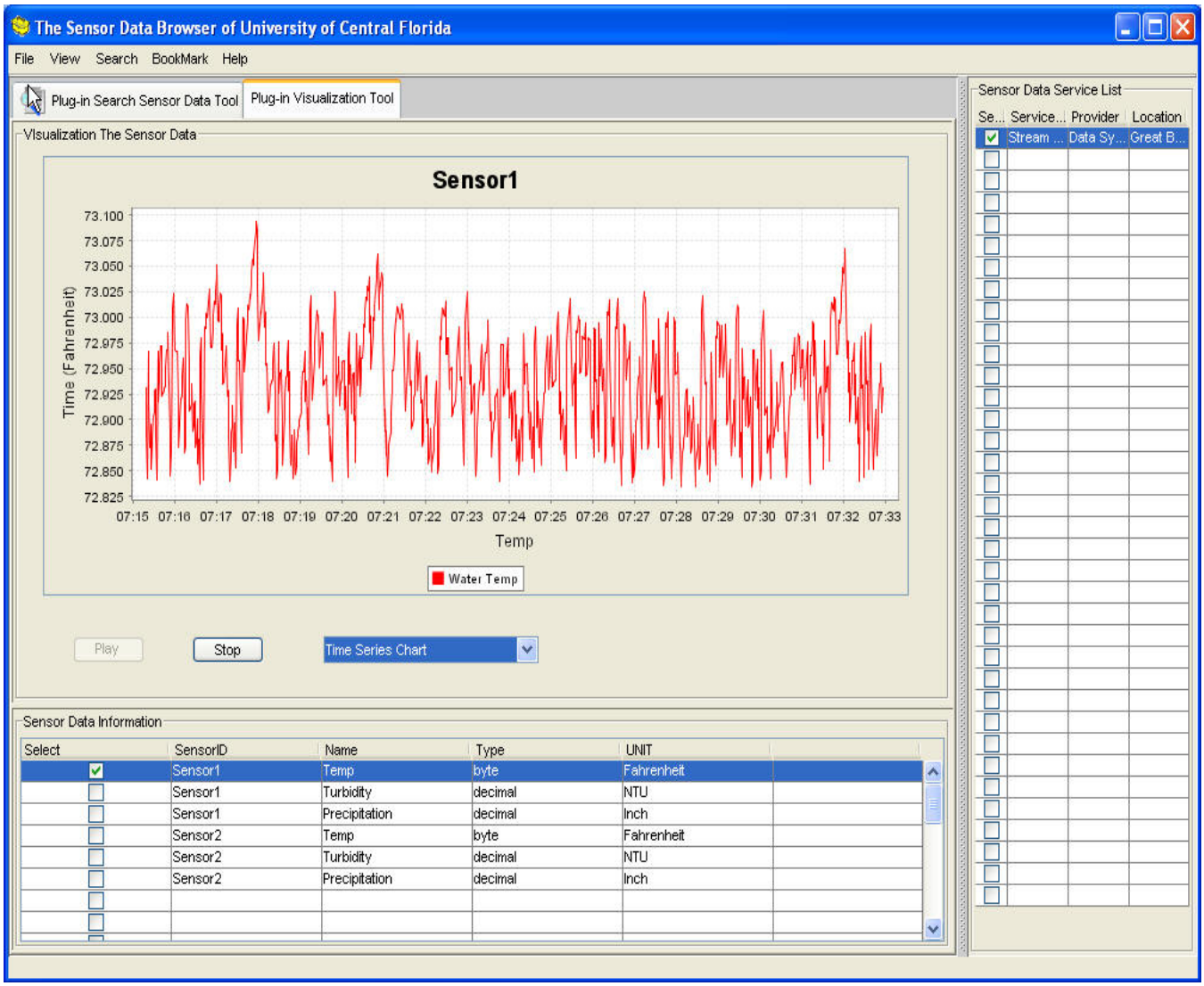


Figure 13. *i*SEE Sensor Browser Interface - Visualizer

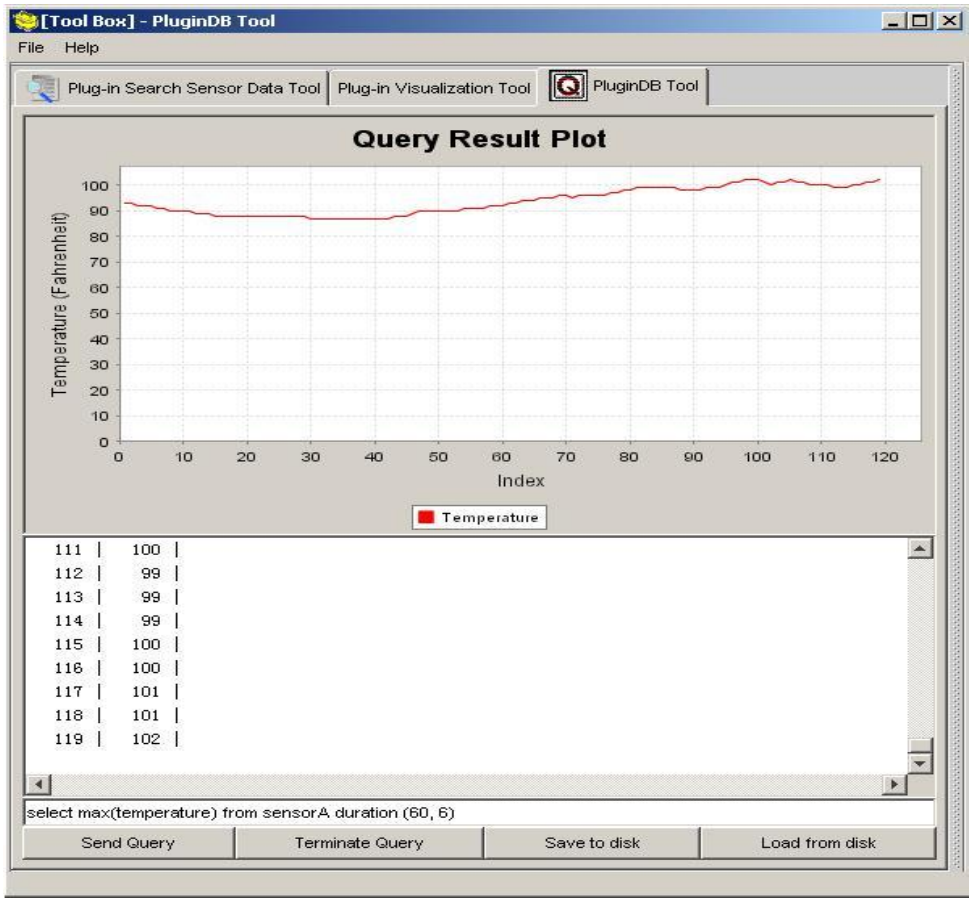


Figure 14. Plug-inDB: the fist plug-in to the *i*SEE browser

In our current *i*SEE prototype, we have developed a query processor called the Plug-inDB for sensor data. It is the first plug-in for the *i*SEE browser to demonstrate the extensibility of the *i*SEE approach. The user can use Plug-inDB to retrieve and extract information from various sensor data sources. A screen shot of the Plug-inDB is shown in Figure 14.

Plug-inDB supports continuous queries on data streams from sensor sources. That is, it continues to refresh the query results in real-time until the user explicitly terminates the query. We have defined the following grammar for continuous querying:

SELECT *expression-list*

FROM *sensor*

WHERE *qualification*

GROUP BY *grouping-list*

HAVING *group-qualification*

DURATION (*sample_rate*, *aggregation_rate*);

Each expression in the *expression-list* could be the name of the original data field of the sensor, or a simple arithmetic aggregation expression on the various data fields of the data stream. Because the purpose of this study is to demonstrate the feasibility of the *i*SEE framework, the first version of Plug-inDB will only support queries on one sensor stream. We plan to extend Plug-inDB to support union and join operations on multiple sensor streams in a future enhancement. The predicate in the WHERE clause is processed using reverse polish notation. It is used to build a data filter to eliminate data items not satisfying the predicate. Plug-inDB supports standard predicate operators such as the arithmetic operators (+, -, ×, /, MOD), logic operators (and, or, not), and arithmetic comparison operators (>, <, =, <>, <=, >=), as well as parentheses to make queries easier to formulate. The GROUP BY statement is used to aggregate data items into different groups as specified in the *grouping-list*. The aggregation expressions in the SELECT clause are computed according to these groups. The HAVING statement is also used in the aggregation stage, preventing those groups that do not satisfy the *group-qualification* predicate from participating in the aggregate computation for the next refresh cycle of the query result.

The most significant difference between our query language and the traditional query languages is that we support a DURATION clause. This clause is specially designed for continuous queries and allows the user to specify a slower sampling rate and aggregation rate to conserve resources. For example, "DURATION (2, 4)" indicates the desire to sample data at a rate twice as slow as the basic rate while the aggregation computation is performed after every four data items are received

2.5 **Performance study**

To evaluate the proposed framework for pervasive sensor sharing, we perform experiments on the prototype system presented in Section 4. The experiments are designed to evaluate the efficiency and effectiveness of two key scenarios in *i*SEE: sensor discovery and data delivery.

In order to evaluate the performance of the first scenario, we focus on the most common functions in the Sensor Registry Server (SRS): *Publish* and *Query*. A *Publish* is the action of an *i*SEE Sensor Server sending meta-data of its sensors to the SRS whilst a *Query* refers to the process of an *i*SEE Sensor Browser searching for a specific sensor resource and retrieving its meta-data from the SRS. The goal of the experimentation is to determine how the SRS performs under regular traffic conditions and under conditions with heavy concurrent requests. We conducted two set of experiments on an HP SRS workstation with a 1.8 Gigahertz Pentium 4 CPU and 2 Gigabytes of memory. The first set of experiments was set up to establish the baseline performance of the SRS by measuring the efficiency of completing common registry tasks, namely the *Publish* and *Query* functions, in an environment where there is no concurrent request to SRS. The second set of experiments was to evaluate the same benchmark of with concurrent requests to SRS.

In the first experimental setting, SRS was loaded with around 1000 registry entries. We then collected the response times for *Publish* and *Query* functions, respectively. Figures 15 and 16 show the baseline performance for executing each function 10 times.

The second set of experiments measures the performance of SRS with an increasing concurrency of processes. These tests intended to determine the impact of concurrent traffic on the performance of the registry functions, by evaluating the *Publish* and *Query* functions with other concurrent *Publish* and *Query* traffic. Data were gathered for four scenarios, 1 *Query* per second, 1 *Publish* per second, 1 Query & 1 *Publish* per second, and 2 Query & 2 *Publish* per second. The results are also illustrated in Figures 15 and 16.

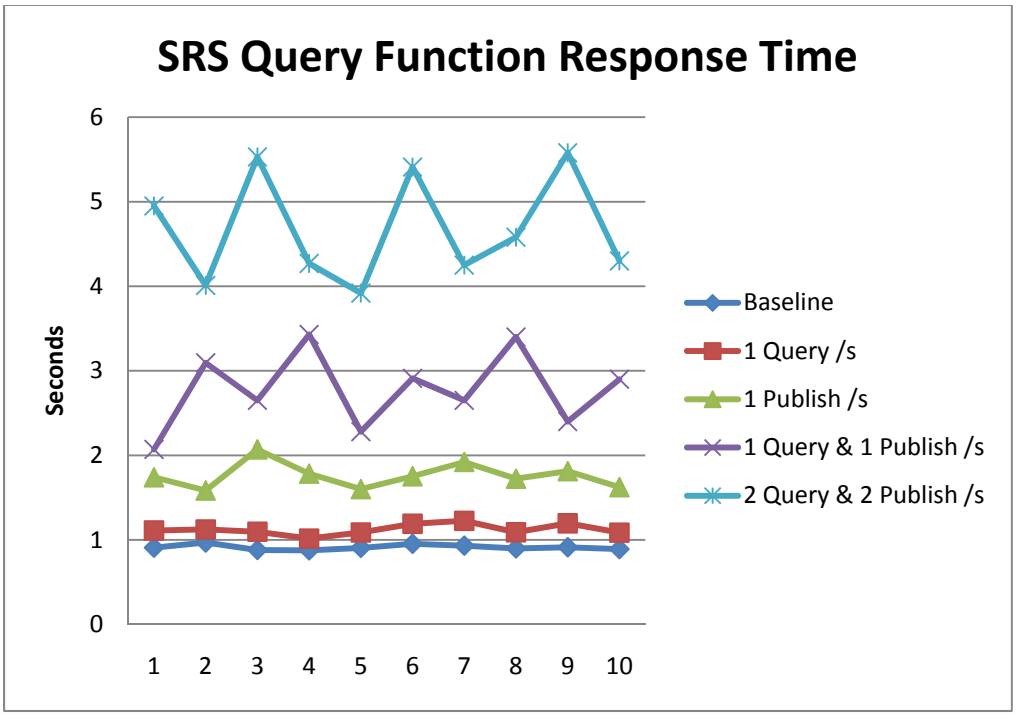


Figure 15. SRS Query Response Time

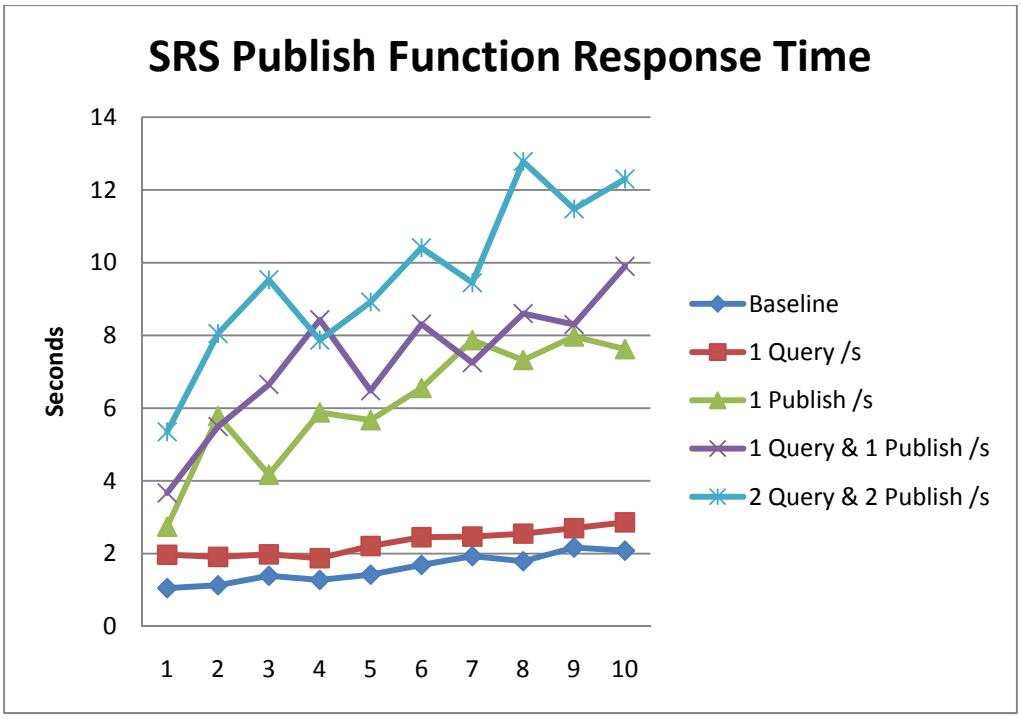


Figure 16. SRS Publish Response Time

From the results, we made the following observations:

- *Publish* is about twice as costly as *Query*.
- The response time of both functions rises sharply as concurrency increases.
- The performance of SRS shows greater swings with the increase of concurrency.

These observations are in line with our anticipation, as inserting an entry to the registry also triggers an update to the index structure of the underlying database to facilitate faster queries. In addition, concurrency incurs considerable overhead for SRS to maintain data integrity and consistency.

To evaluate the performance of the second scenario, we conducted experiments to collect statistics for Sensor Stream Control Protocol (SSCP). These experiments are designed to gather

empirical data under different data loads and provide useful insights that would help design performance optimization techniques.

We used seven test cases in our experiments. For each of the cases, a Sensor Browser initiated 100 SSCP calls. Under *i*SEE framework, each request is first de-serialized, which is then passed to the service as a function argument, and finally the response forms through serialization. Our test cases are different by the data structure that the response returns, as listed below in Table 2:

Table 2. Test cases

Test Case #	Return type in each call
1	The same integer
2	Random integer
3	Constant integer array
4	Random integer array
5	Constant complex object
6	Random complex object
7	The same complex object as the request

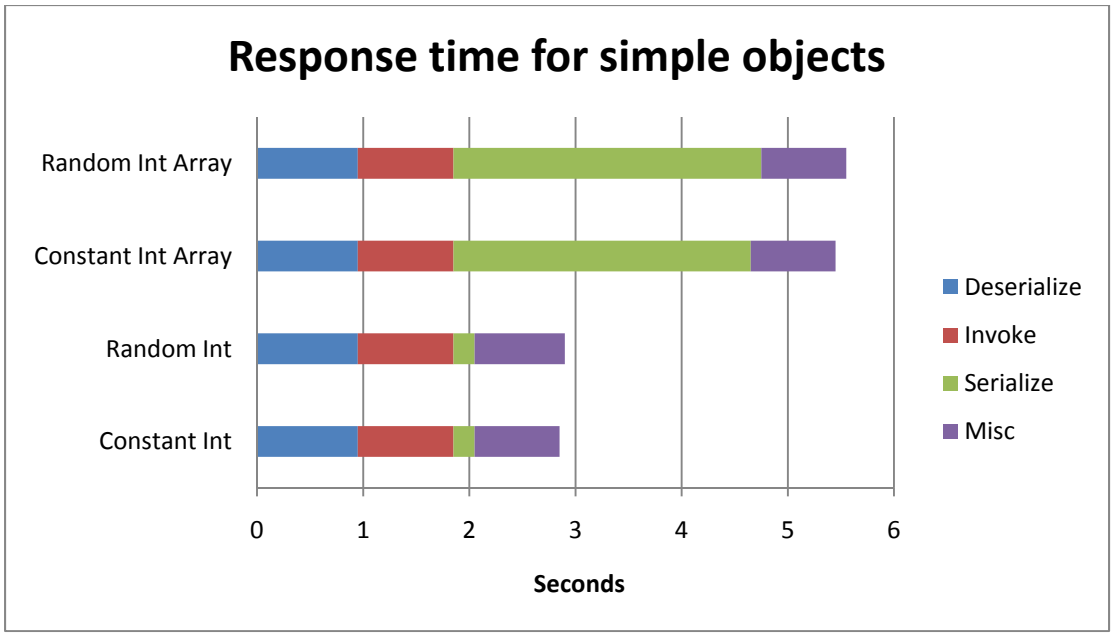


Figure 17. Simple objects

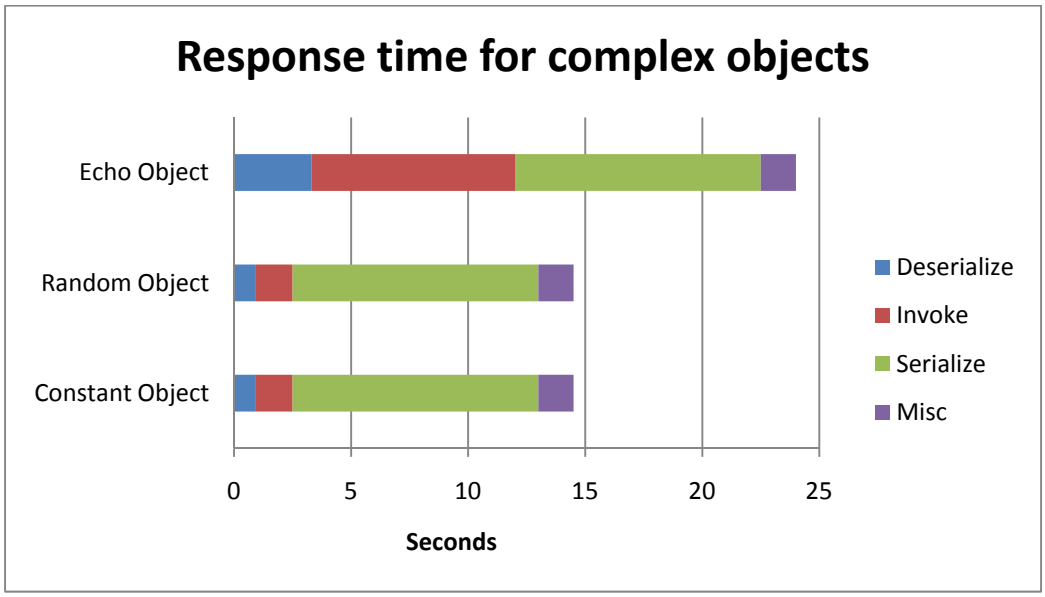


Figure 18. Complex objects

The cumulative timing information for 100 runs in different test cases is illustrated in Figures 17 and 18. Figure 17 compares the response times for test cases 1 – 4 where response objects are

simple objects whilst Figure 18 depicts the results for test cases 5 – 7 where response object are complex objects. Our experimental results indicate the following characteristics of SSCP:

- SSCP handles simple objects better than complex objects. This is naturally expected as de-serialization cost of complex objects is much higher than that of simple objects.
- SSCP does not differentiate constant data from varying data streams. The implication is that SSCP will not optimize for sensors with mostly constant readings. Therefore one of our planned future works is investigating the frequency of constant sensor readings and developing optimization techniques, such as caching or compression, for mostly constant sensor readings.
- Serialization is the most dominant component in response time of arrays and other complex objects. This indicates that future improvements on serialization cost reduction will have a great impact on the overall response time. Optimization techniques such as caching and compression templates are within the scope of our future investigation.

2.6 **Conclusion**

The major contribution of this work is the development of *i*SEE, the first pervasive sensor computing framework that facilitates sensor data sharing, sensor application sharing, and on-the-fly data integration. We envision a new Internet, an Internet of sensors that globally interconnects smart devices and sensor networks. We identified the key challenges in supporting such an environment and proposed solutions accordingly. Our techniques include: (1) the Sensor Service Definition Language for defining services, (2) the Publication Facility for publishing sensor services on the Internet, (3) the Sensor Registry Server for sensor meta-data registration and discovery, (4) two communications protocols for sensor data delivery, (5) the *i*SEE Sensor Browser for accessing and integrating streaming sensor data, (6) the Plug-in mechanism enabling

sensor application sharing, and (7) two plug-ins, namely the Visualizer and Plug-inDB, for visualizing and querying sensor data respectively. We have built a prototype as a test bed to evaluate these techniques.

Future research can focus on developing more sophisticated general-purpose plug-ins for browsing, querying, and analyzing sensor data in the *i*SEE environment. Privacy issues of data content as well as reliability issues about allowing multi-level access to data uploaded by different types of users are also worth further investigation. Extending the current framework with specialized toolkits that effectively integrate data over current sensor database systems such as TinyDB and Cougar are also highly desirable. It is also worthwhile to leverage the proposed framework to develop and deploy new interesting sensor-based applications in a pervasive computing environment.

3. LIVE VIDEO DATABASE MANAGEMENT SYSTEMS – LVDBMS

3.1 Introduction

Cameras are a special class of sensors that are widely used in many applications ranging from traffic monitoring, public safety and security to healthcare and environmental sensing. They generate great volumes of data in the form of live video streams. In contrast to entertaining movies, such live video captured by specialized cameras are generally not as interesting. People who constantly monitor these cameras, such as baggage screeners at airports, can quickly become fatigued. Moreover, when tens of thousands of cameras are present, such as those deployed on streets in a city, it is expensive, if physically feasible, to house a huge number of monitors to support the various monitoring operations. In many scenarios, critical events do not happen very often, and it is quite inefficient to have one monitor tracking only a few cameras constantly and intensely. With the recent advances in VLSI technology, networks of distributed cameras, consisting of many inexpensive video cameras and distributed processors, will bring far greater monitoring coverage. It will become virtually impossible for human beings to keep track of all the objects or events under each camera. The distributed nature of these networks, coupled with the real-time nature of live videos, greatly complicates the development of techniques, architectures, and software that aim to effectively mine data from a sea of live video feeds. In this work, we propose a *Live Video Database Management System* (LVDBMS) as a solution to address the aforementioned problems. LVDBMS is designed to allow users to easily focus on events of interest from a multitude of distributed video cameras by posing continuous queries on the live video streams. With LVDBMS automatically and continuously monitoring live videos feeds and taking care of the intercommunications among distributed processors, a user can easily use a desktop display to manage a very large number of cameras and receive notifications when critical events happen.

There have been extensive research activities tackling multimedia databases, especially video databases. *Content-based image retrieval* (CBIR) systems (Flickner, 1991; Hua, 2006) mainly deal with low level features such as color, texture, shape, and preliminary semantic features such as keywords. Additional CBIR research with enhanced query capabilities on spatial relationships between objects in images has also been proposed in (Jiang, 1997; Kuo, 2000; Marcus, 1996; Mehrotra, 1997). There are a number of content-based video retrieval systems (Chang, 1997; Flickner, 1991; Koh, 1999; Oh, 2000; Oomoto, 1993) that adopt the techniques for image retrieval by treating a video as a sequence of images. Techniques for building semantic video databases can also be found in (Adali, 1996; Jiang, 1997). More recently, video databases with spatiotemporal query capability have been proposed in (Donderler, 2003; Donderler, 2002). However, there is no other work targeting a general-purpose live video database management system that provides real-time spatiotemporal query support over distributed camera networks.

We introduce a novel concept called *Live Video Databases* as a new class of databases built upon a multitude of real-time live video streams. The fundamental difference between a video database and a live video database is as follows. While the former deals with stored video files, the latter deals with real-time video data, streaming from cameras treated as a special class of "storage" devices. In this work, we focus on the continuous query model and distributed query processing techniques for spatiotemporal live video databases. There are three primary stages to process a query: *query registration*, *query decomposition*, and *query execution*. At the first stage, a user poses a continuous query by registering a predicate and an action with the database management system, which will evaluate the condition continuously in real time. An event-based query language is designed for users to specify spatiotemporal queries on live video

streams. The language offers sufficient expressiveness for defining events as spatiotemporal relations between objects across multiple live videos and allows users to specify certain actions upon detecting such an event. At the second stage, a complex query, possibly spanning multiple live video sources, is decomposed into single-video sub-queries. In the last stage of query processing, sub-queries are evaluated on individual live videos in real time. The results from the sub-queries are synthesized to form the final answer, which indicates if the user-specified event has occurred. Once the predicate is satisfied, it triggers the LVDBMS to execute the user specified actions (e.g., notify the user and archive relevant video clips).

We note that there are many existing video surveillance systems over networked cameras (Ahmedali, 2006; Hampapur, 2005). These systems, however, are designed for a specific class of applications. As an example, one cannot simply use a system intended for monitoring patients in a hospital for incident detection on highways. Our LVDBMS approach manages live video streams coming from hundreds of video cameras, much like a traditional database management system managing standard data sets stored on multiple disk drives. The LVDBMS provides an application-independent SQL-like query language to facilitate ad hoc queries, and potentially a programming environment to simplify the development of a variety of applications for distributed camera networks. In other words, software developers would be able to develop advanced systems for different purposes much like the way they would develop database applications today.

The main contributions of this work are:

- a spatiotemporal data model for live video streams and events of interest from distributed cameras,
- an event-based query language for composing spatiotemporal queries, and

- query processing techniques for computing query results in a network of cameras.

The remainder of this chapter is organized as follows. In Section 3.2, we discuss some related work including traditional video DBMS and continuous query processing techniques. We introduce the proposed techniques for live video database management systems in Section 3.3. The design and implementation of the LVDBMS prototype is presented in Section 3.4 followed by the experimental study in Section 3.5. Finally, we offer our conclusions and discuss future work in Section 3.6.

3.2 **Related work**

There has been extensive research for *video database management systems* (VDBMS) in recent years and many prototype systems have been designed and implemented (Adali, 1996; Chang, 1997; Donderler, 2003; Flickner, 1995; Guting, 2000; Jiang, 1997; Oomoto, 1993). In the *Advanced Video Information System* (AVIS), Adali (1996) introduce a data model to represent objects and their association based on a frame segment tree. However, only temporal relations of objects are captured in this model and hence the query language has no support for spatial queries or spatiotemporal queries. Chang (1997) presents VideoQ, a VDBMS that enables the user to base their video search on visual features and spatial-temporal relations. Although these features are generated from an automatic analysis of the captured video, this is done off-line. Li (1997) extends the TIGUKAT *Query Language* (TQL) and the *Object Query Language* (OQL) to two general-purpose multimedia query languages called *Multimedia OQL* (MOQL) and *Multimedia TQL* (MTQL), respectively, which support temporal and spatial queries. In (Oomoto, 1993), a video data abstraction and an SQL-like query language named VideoSQL are proposed. In this model, the hierarchical structure is extracted through the introduction of interval inheritance based on inclusion. BilVideo (Donderler, 2003) is a more recent rule-based VDBMS

in which an SQL-like language provides support for a broad range of spatiotemporal queries. The facts for spatiotemporal queries are extracted and stored in a knowledge base; to answer a spatiotemporal query, the query processor only needs to communicate with the knowledge base to obtain the results.

All the video database systems presented in the literature thus far address only static videos in the sense that they extract and index all the objects in the entire video collection a priori. In the context of live video databases, nevertheless, the video data are generated online, and thus no or very little prior knowledge exists about the objects in video data. Furthermore, video analysis is considered a "pre-processing" operation in conventional VDBMSs in that the feature extraction phase is done off-line with less performance concern, while an LVDBMS needs to segment and track salient object in real time with high efficiency.

3.3 **Live Video Database Management System**

In this section, we propose LVDBMS, an integrated environment for querying distributed cameras. We describe a generic event model over live videos and introduce a spatiotemporal query language based on that data model, followed by a presentation of continuous query processing techniques.

3.3.1 Architecture

As a traditional video DBMS must be able to manage a large number of videos, an LVDBMS needs to accommodate a potentially large number of cameras as well. A naïve approach to managing a network of cameras is to use a central server that accepts video streams from all the cameras in the network. However, this is not a scalable solution as the server will eventually run out of bandwidth and processing capability with increasing numbers of cameras in the network. In this work we propose a more scalable approach as illustrated in Figure 19.

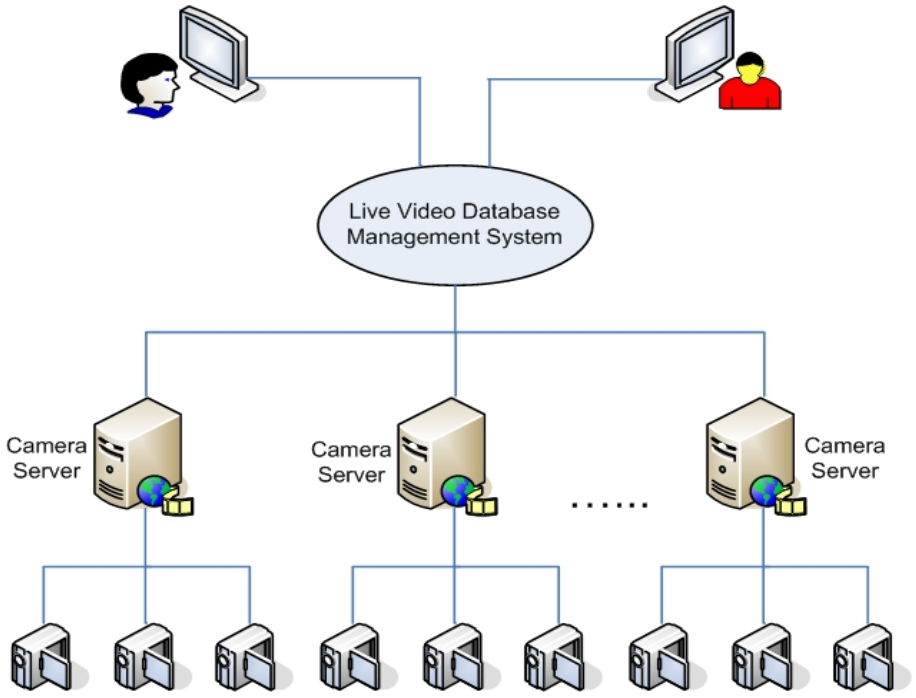


Figure 19. A three-tier architecture for camera management

We add an additional tier, the camera servers, in between the LVDBMS and cameras. Camera servers are processors that manage nearby cameras and receive video streams directly from them. When a complex query is entered, the LVDBMS decomposes it into a tree of sub-queries and sends relevant sub-queries to appropriate camera servers. Camera servers, accepting live videos from associated cameras, are responsible for executing sub-queries on the video streams produced by individual cameras and returning answers. By leveraging the processing capability of the distributed camera servers, the LVDBMS no longer needs the bandwidth and computational resource to process all the live videos. The communications between the LVDBMS and camera servers now involve only textual commands rather than video data, making the entire camera network scalable to a large number of cameras. Alternatively, a hierarchical or peer-to-peer architecture could also be considered to build camera networks for increased scalability at the price of increased design complexity.

3.3.2 Event Data Model

Queries for events of interest provide an intuitive way for users to extract meaningful information out of live video streams. However, the mapping and translation from the lower-level video signals to higher-level semantic event descriptions would be a far more daunting process without a generic and unified event data model. In this work, we present a data model that formally defines important concepts of the LVDBMS including streams, objects, operators, and events.

DEFINITION 1: A **live video** V is a stream of tuples <vf, t>, each consisting of a video frame (vf) and a timestamp (t). The stream is sequenced by t ascending order of the t value.

DEFINITION 2: An **object snapshot** O_s is a portion of a video frame, vf, and semantically represents an object in the real world. The function that defines O_s is:

$O^{vf}_s(x, y)$ = True, if the pixel at (x, y) is in the object in the frame vf, where (x, y) is the relative coordinate within the video frame vf.

DEFINITION 3: A **moving object** O of a live video stream, V, is a sequence of object snapshots that correspond to the same real object. The function that defines O is:

$O^v_s(x, y, t)$ = True, if $O^{vf}_s(x, y)$ is True for video frame vf at timestamp t, for some <vf, t>$\in$ V.

DEFINITION 4: A **collection of objects** C for a live video V is a set of some moving objects $O^V_1, O^V_2, \ldots O^V_n$ that reside in V:

$C(V) = \{O^V_1, O^V_2, \ldots O^V_n\}$, where O^v_i is a moving object of V.

Objects are a vital part of the model, as they represent the smallest semantic unit in the LVDBMS. Based upon the above definitions, the LVDBMS transforms a live video *V* consisting of lower-level signals into a collection of objects C(V) with more meaningful information. The

techniques for detection and tracking of objects in a live video stream are discussed in Section 3.6.

Once moving objects are detected and interacting with one another, there is a need to specify events of interest in terms of certain relations between objects. An event in the database research community commonly refers to a change of state in the database, such as an update to a table. Analogously, an *event* in the LVDBMS is defined as a change of spatiotemporal relations between objects in live videos. Our approach to specifying spatiotemporal relations is by defining operators that can be applied to objects and streams.

DEFINITION 5: A **spatial operator** S_o evaluates the spatial relations between objects in a live video V. The possible values of S_o are:

$S_o \in$ {equal, overlap, meet, covers, coveredby, inside, contains, disjoint, west, east, south, north, southeast, southwest, northeast, northwest, appears}

The spatial relations are categorized into three groups: appearance properties that indicate if an object appears, topological relations describing incidence and neighborhood, and directional relations representing the relative positioning of objects. This comprehensive set of topological relations was proposed in (Egenhofer, 1991), in which two objects may "coincide (equal), intersect (overlap), touch externally (meet), touch internally (covers and the reverse coveredby), be inside (and the reverse, contains), or be disjoint". In addition, eight directional operators (i.e. west, east, south, north, southeast, southwest, northeast, and northwest) are defined to provide further information on the relative positioning between two objects.

The spatial operators defined in this model are chosen to capture the most common and basic spatial relations that occur between two objects. There are more advanced spatial relations such

as a convolution of two ropes that are not covered by this model. The problem of automatic recognition of such a relation from videos can be a significant research topic on its own.

The operators of the first two categories are binary while *appears* is unary. For convenience of notation, we treat S_o as a binary operator in this section. When S_o is applied over two object snapshots, the output value is Boolean: true when the two object snapshots satisfy the spatial relation specified by S_o in a video frame vf, and false if otherwise.

DEFINITION 6: A **spatial event snapshot** E_s is obtained by applying a spatial operator S_o over two object snapshots O_{s1} and O_{s2} in the same video frame vf:

$$E_s = S_o(O^{vf}_{s1}(x, y), O^{vf}_{s2}(x, y))$$

As indicated by the above definition, a spatial event snapshot is essentially a Boolean value, that indicates if a certain spatial relation holds between two object snapshos or not.

DEFINITION 7: A **spatial event stream** $E_s(t)$ is a sequence of spatial event snapshots obtained by applying a spatial operator S_o over two moving objects O_1 and O_2 in the same collection of objects C(V):

$$E_s(t) = S_o(O^V_1(x, y, t), O^V_2(x, y, t))$$

Spatial events are primitive events defined to describe spatial relationships between two objects within the same video. By introducing spatial operators, the LVDBMS is able to relate objects within the same video based upon their relative positions and extract another layer of semantics. Once spatial event streams are defined, the LVDBMS needs only to deal with only Boolean data streams rather than much larger video streams.

DEFINITION 8: A **logical operator** L_o evaluates the logical relations between objects in a live video V. The possible values of L_o are:

$L_O \in$ {and, or, not}

DEFINITION 9: A **temporal operator** T_o evaluates the temporal relations between objects in a live video V. The possible values of T_o are:

$T_o \in$ {ends, starts, overlaps, during, meets, before}

As we continue to build up our event hierarchy, logical and temporal operators are added to give the model more expressiveness. Hamblin (1972) and Allen (1983) addressed the issue of temporal interval relations. The temporal relations used in this work are illustrated in Figure 20. Logical operators, similar to spatial operators, can be applied to both snapshots and streams. However, temporal operators can only be applied to streams as historical data need to be accessed in addition to the current snapshot to evaluate any temporal operator.

DEFINITION 10: A **composite event snapshot** E_c is either a spatial event snapshot, or obtained by applying temporal and logical operators over spatial event streams.

DEFINITION 11: A **composite event stream** $E_c(t)$ is a sequence of composite event snapshots ordered by timestamp t. A composite event stream is either a spatial event stream or obtained by applying temporal and logical operators over spatial event streams. The grammar is therefore the following:

$E_c(t) \rightarrow E_s(t) \mid T_o(E_{c1}(t), E_{c2}(t)) \mid L_o(E_{c1}(t), E_{c2}(t))$

While each primitive spatial event is defined within the scope of a single live video, composite events can span multiple videos. With different combinations of logical and temporal operators, a composite event can be built with significant complexity and therefore allows LVDBMS to offer a query language with sufficient expressiveness. The above definitions provide a flexible way for users to specify spatiotemporal relations between objects.

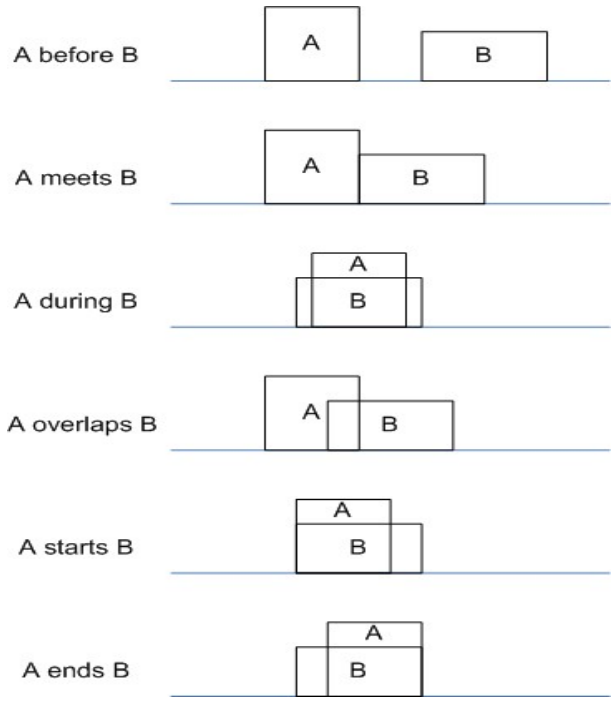


Figure 20. Temporal relations

Through the event data model we presented above, the LVDBMS bridges the semantic gap by mapping lower-level video signals to higher-level semantics, in terms of spatiotemporal relations between objects. More specifically, the LVDBMS is able to extract events of interest as streams consisting of event frames ordered by their timestamps, represented as <*ef, t*> where *t* is the timestamp and *ef* is an event frame that takes only two values: TRUE when the event is detected within the current window and FALSE when it is not.

3.3.3 Query Language

Real-world applications of distributed camera networks can be of high complexity in terms of the number of objects and the relationships among them. A live video database is a collection of objects captured continuously in real time and interrelated by temporal and spatial relationships. In a complex application where the number of videos is large, it can be difficult to track all possible objects in every live video in real time. Using the event data model we proposed in the

previous subsection, a rich set of events can be defined over only a subset of the available videos, which allows the LVDBMS to process queries on-demand while performing computations only on the subset of relevant videos.

The basic form of an event-based query is the following:

ACTION <action>

ON EVENT <composite event> <window>

Some examples of the action field can be: "notify the administrator via email", "play the last 10 seconds of video", or "archive all relevant video clips." The discussion on designing a complete set of actions is beyond the scope of this work as we focus on the event detection phase. Our event detection engine provides higher-level applications with application programming interfaces (API) and therefore an application is free to customize their own set of actions based on user needs. The ON EVENT clause actually specifies the predicate for a continuous query of the LVDBMS.

DEFINITION 12: A **continuous query** Q is obtained by evaluating a composite event stream $E_c(t)$ given a processing window $w \in [0, +\infty)$.

The processing window is defined as an interval that starts with the time when query is executed and lasts for duration as specified with *w*. A starting time other than the query execution time can be easily added to the model by modifying slightly on the language. Nonetheless, we use just a single value *w* to represent the processing window for simplicity of presentation. The size of the processing window *w* can be specified in the form of the amount of time or the number of frames, but both ways are similar in nature.

Three types of queries could be defined by varying the value of *w*.

DEFINITION 13: A **snapshot query** is a continuous query with processing window w = 0. A snapshot is only fired at the present time and evaluates just the current state.

DEFINITION 14: A **sliding-window query** is a continuous query with a positive, finite processing window w. Only frames within the sliding window will participate in the query.

DEFINITION 15: A **landscape query** is a continuous query with an infinite processing window w = +∞. All frames of the participating videos will be used to evaluate the query result.

The LVDBMS supports all three types of query, but sliding-window queries are the focus in this work as the other two are special cases of sliding window queries. To query live video data, a user specifies sliding-window queries that create a moving window over video streams and calculate query results from the frames within the window. Every query in LVDBMS is a continuous query running indefinitely until explicitly stopped.

The composite event field is defined by an event composition expression following the Backus–Naur Form syntax in Figure 21.

In the LVDBMS, each live video stream has a unique ID and each object within a video stream has a unique object ID. Therefore we can reference an object by its unique global ID: <stream ID>.<object ID>. There are two types of objects in LVDBMS: *static objects* and *dynamic objects*. Static objects refer to those that are user-defined by drawing a rectangle on the video screen and appear in a video at the time a query is specified. Thus at query time a user can reference static objects by their IDs. Dynamic objects are those not viewed by the camera at the query time but are detected automatically by the system.

<composite event>	::=	<logicspatial event> (<temporal op> <logicspatial event>)? <window>
<logicspatial event>	::=	<and event> (“or” <and event>)*
<and event>	::=	<spatial event> (“and” <spatial event>)*
<spatial event>	::=	“(” <composite event> “)” | <appear event> | <binary event> | “not” <spatial event>
<appear event>	::=	<appearance op> “(” <video> “.” <object> “)”
<binary event>	::=	<spatial operator> “(” <video> “.” <object> <video> “.” <object> “)”
<spatial operator>	::=	<topological op> | <directional op>
<appearance op>	::=	“appear”
<topological op>	::=	“equal” | “overlap” | “meet” | “cover” | “coveredby” | “inside” | “contains” | “disjoint”
<directional op>	::=	“southeast” | “southwest” | “northeast” | “northwest” | “west” | “east” | “south” | “north”
<temporal op>	::=	“before” | “meet” | “during” | “overlaps” | “starts” | “ends”
<video>	::=	“V” [“1”-“9”] ([“0”-“9”])*
<object>	::=	“O” [“1”-“9”] ([“0”-“9”])*
<window>	::=	[“1”-“9”] ([“0”-“9”])*

Figure 21. LVDBMS event schema syntax

3.3.4 Query Processing

As previously mentioned, a query is processed in three stages, namely *query registration*, *query decomposition*, and *query execution*. The registration and decomposition are executed once for each query, but the query is continuously executed until explicitly stopped. Figure 22 and Figure 23 illustrate the three stages.

Query registration. In this stage, a user (or an application) specifies a composite event and an action through a graphical user interface (or other APIs). The query parser accepts the query, parses it into tokens, and generates a query tree. This query tree structure is then registered with the LVDBMS and serves as the starting point for query decomposition.

Query decomposition. A query containing a composite event may involve objects in multiple live videos. The task for the query decomposer is to decompose a complex query into sub-queries, each of which can be evaluated on one video. The input of the query decomposer is the parse tree, and the output is a query decomposition tree showing the query plan. In this stage, the query tree on a composite event is decomposed into individual leaf nodes corresponding to spatial events, and the remaining tree consisting of the hierarchy of temporal and logical operators. All the sub-queries are assigned unique IDs. Each sub-query on a spatial event involves only one specific video stream and can thus be forwarded to the relevant camera server for evaluation. The LVDBMS then traverses the remaining tree structure, instantiates a query processor, and waits for event streams from participating camera servers. The query processor, in turn, instantiates logical and temporal operators and allocates a dynamic queue for each operator. Figure 24 illustrates the process of query decomposition and query processor instantiation. Upon receiving a sub-query from the LVDBMS, each camera server also instantiates a query processor with the spatial operator and corresponding queue. In this work we name the query processor in the LVDBMS as the *parent query processor*, and the spatial query processor at each camera server as the *child query processor*.

Query execution. During this stage, the participating camera servers, after instantiating a child processor, invokes a real-time video segmentation and tracking module to extract objects from the live video. The child query processor in the camera server is responsible for calculating the

spatial relationships between the objects and evaluating if a spatial event is detected. The output of the child query processor is an event stream stored in the queue and being sent to the corresponding parent query processor. As the parent query processor receives results of sub-queries from child query processors at participating camera servers, it evaluates the composite event by calculating each of the operators, starting from the leaf nodes. The output stream from each operator is stored in the associated queue and serves as the input for the parent operator. The queue periodically checks if any event frames have an expired timestamp (i.e. less than the current time minus the processing window *w*) and removes those. The implementation details of operators are presented in Section 3.4. Once a composite event is detected, the LVDBMS triggers the action associated with the query. The query and all sub-queries continue running until explicitly stopped by the user.

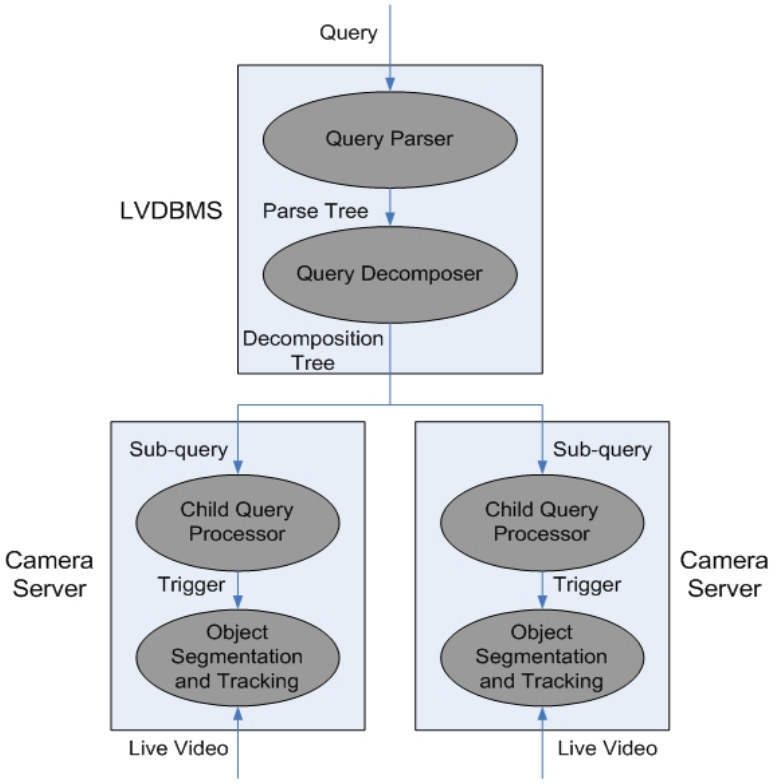


Figure 22. Query registration and decomposition

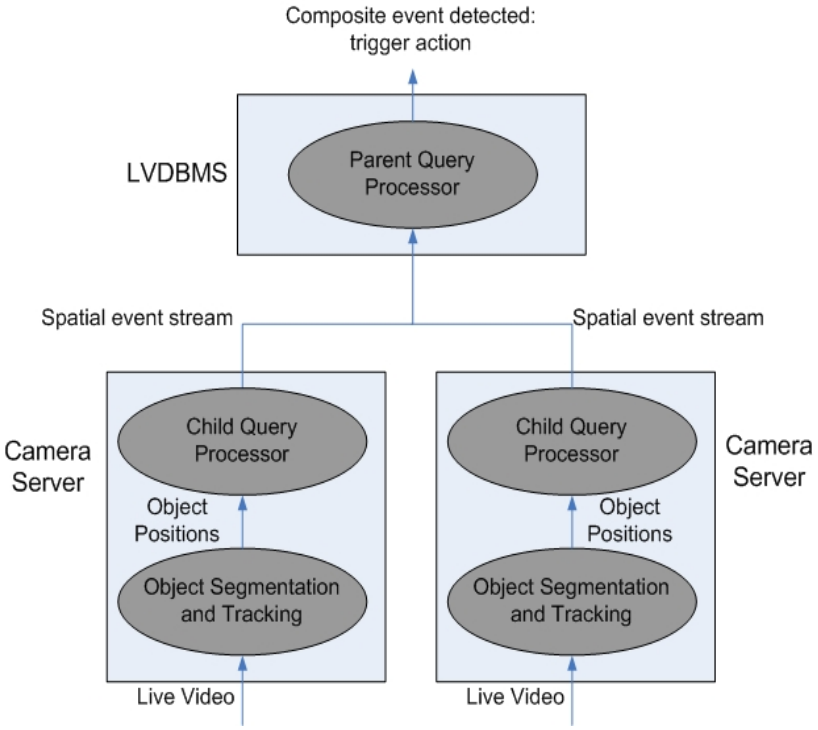
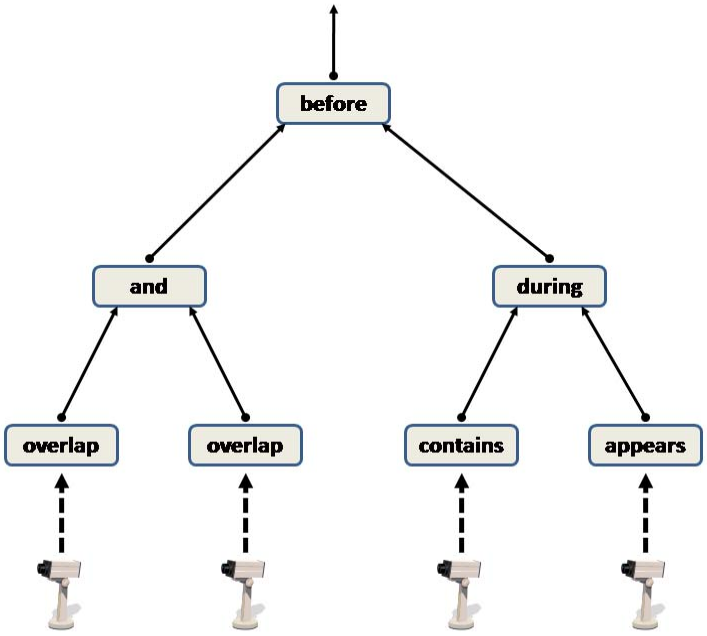


Figure 23. Query execution



Query tree

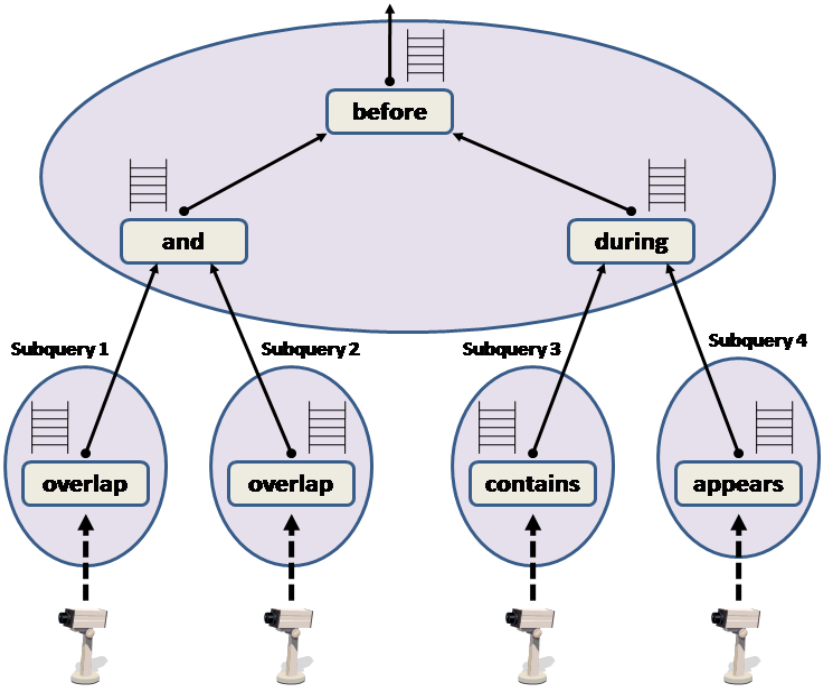


Figure 24. Query decomposition and execution plan

3.3.5 Query Optimizations

There are two major query optimization strategies used in the LVDBMS.

The first strategy is delta coding and aims to reduce the data rate of events streams by observing the following facts:

The value of each event frame is boolean, either True or False;

The frequency of events is less than the frame rate;

When an event happens, it usually lasts for some time.

Based on these observations, we design optimized event streams to replace the original events streams that camera servers send to the LVDBMS. An optimized stream captures the change of value in the original stream and every event frame in the optimized stream corresponds to such a change.

Given an event stream:

$E = \{<ef_0, t_0>, <ef_1, t_1>, \ldots <ef_n, t_n>\}$,

The optimized event stream is obtained by applying the following formula:

$E' = \{<ef_0, t_0>\} \cup \{<ef_k, t_k> | ef_k \neq ef_{k-1}, 1 \leq k \leq n\}$

The following is an example of this optimization strategy, where *T* denotes True and *F* denotes False.

Original: F T T T T F F F T T T F F F F T T T

Optimized: F T F T F T

The advantages of this strategy are:

- It is a lossless compression. The optimized event stream is semantically equivalent to its original stream without any information loss.
- The optimized event stream reduces the frame rate on the average. In the worst case, it would have the same frame rate if the original stream, were it to change values at every frame.
- The communication and computational costs of sending, receiving, processing, and buffering event streams are greatly reduced.

When applying this optimization strategy on camera servers, only minor changes need to be made to the implementation of a camera server. Once the objects in the video exhibit the specified spatial relationship, the spatial event is fired and the camera server informs the LVDBMS of the detection of the event and its start time. When the camera server detects that

the objects no longer maintain the spatial relationship, it notifies the LVDBMS of the end time of the event.

The second strategy optimizes the query decomposer and leverages the distributed processing capability of camera servers. This strategy is based on the following observations:

A camera server typically manages multiple cameras that are physically close. A query involving one camera will likely involve others attached to the same camera server. This can be thought of as the rule of *spatial locality* in the LVDBMS.

Based on this observation, our second optimization strategy is to push any logical and temporal operator down to a camera server whenever the operator only involves video streams from a same camera server. When the query decomposer sees a new query tree, it traverses the tree and marks the camera server ID at each node starting from the leaf up. If the child nodes of a parent node have different camera server IDs, the parent node will not be marked. Then the query decomposer examines the tree from the root down and chops any sub-tree with the root marked with a camera server ID. The chopped sub-trees will be assigned unique IDs and conveyed to corresponding camera servers. As previously mentioned, the LVDBMS will instantiate a parent query processor around the remaining tree structure whereas the child query processors will be instantiated by camera servers around the received sub-trees. Figure 25 illustrates this query optimization strategy using the same example in Figure 24(a) and assuming the left two and right two leaf operators share the same camera server.

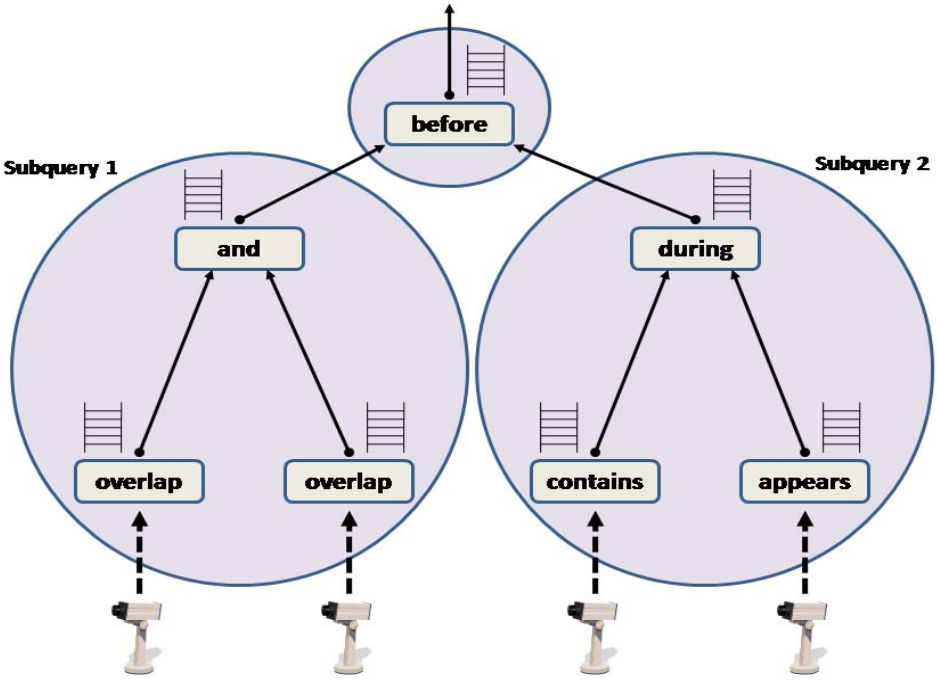


Figure 25. Execution plan after optimization

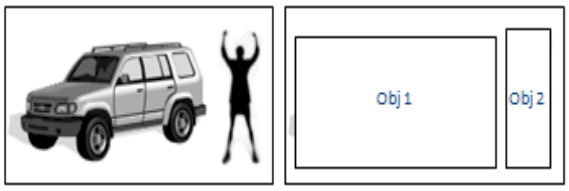
3.3.6 Object Segmentation and Tracking

In the query execution stage, carried out on the camera servers, the underlying video processing module tracks salient objects (if any), in real time, in preparation for higher-level processing. This section describes how object tracking methods are employed in order to determine the various directional and spatial relationships that exist among objects.

Object tracking is carried out in three stages: first salient objects are identified, then tracked from frame to frame, and finally their paths are analyzed. We then derive bounding rectangles containing the objects and calculate their centroids (their center of mass). Our multiple object tracking algorithm, derived from work presented in (Lieb, 2004) consists of the following basic steps:

- Background subtraction is performed, by calculating median pixel values over a sliding window of 100 frames.
- Optical flow is calculated between consecutive frames (we use the Lucas-Kanade approach (Lucas, 1981)).
- A particle filter supplies multiple hypotheses for the locations of objects (Thrun, 2002).
- Outlier removal and a grouping operating are performed to detect the moving objects.

Once objects have been segmented and identified for tracking purposes (trajectories generated), minimum bounding rectangles are computed around the objects and the objects' centroids are calculated (see Figure 26). There are different structures to represent objects, such as blobs (Kauth, 1997), bounding boxes, eclipses, active contours, or representative points (e.g., the centroid or corners). In particular, the Minimum Bounding Rectangle (MBR) is one of the most commonly used approximations in the database research domain (and also eases higher-level processing computations, such as calculating object overlap, etc.). The centroids and bounding boxes are then utilized in higher level object processing.



(a) (b)

Figure 26. Object tracking and representation

We note that the algorithms for object segmentation and tracking are not a contribution of this work. Although the current implementation of the algorithm is not perfect, the LVDBMS framework does not limit itself with this single algorithm. More sophisticated algorithms, such

as (Ahmedali, 2006; Hampapur, 2005; Kim, 2006), when properly adopted, can be seamlessly integrated into the framework.

3.4 **System prototype**

Figure 27 depicts the software architecture of the LVDBMS. The system is implemented using Java 5.0 with Java Media Framework 2.1.1 for live video capturing. Our implementation of LVDBMS consists of three major components: the user interface, the LVDBMS server, and the camera server; all the components interact with each other through TCP/IP.

Figures 28 and 29 display the main window and video window of LVDBMS user interface. The main window allows a user to choose different views and combinations of cameras and monitor query status. The video window provides functionality to create user-defined objects, static or dynamic.

Figure 30 illustrates how a user can specify a query to detect scenarios where someone comes and sits on a chair:

appear(V1.DO1) before overlap(V1.DO1 V1.SO1) before covers(V1.DO1 V1.SO1) 10

This query is evaluated to be true in the last frame once the person's minimum bounding rectangle covers that of the chair.

One of the major tasks of the LVDBMS server and camera servers is to evaluate operators. Logical operators over streams are evaluated by performing the conventional logical operations over every snapshot of the participating streams. Temporal operators are also evaluated on the LVDBMS and/or camera servers. The inputs for a temporal operator are two event streams $E_1(t)$ and $E_2(t)$, the current time T, and a processing window w. Formulas have been developed for each of the operator. For example, the formula for *before* is:

before($E_1(t)$, $E_2(t)$, T, w) =

True, if $\exists$ T-w $\leq t_1 < t_2 < t_3 < t_4 \leq$ T, $[t_1, t_2]$ is the latest period where $E_1(t)$ is true and $[t_3, t_4]$ is the latest period where $E_2(t)$ is true.

False, otherwise.

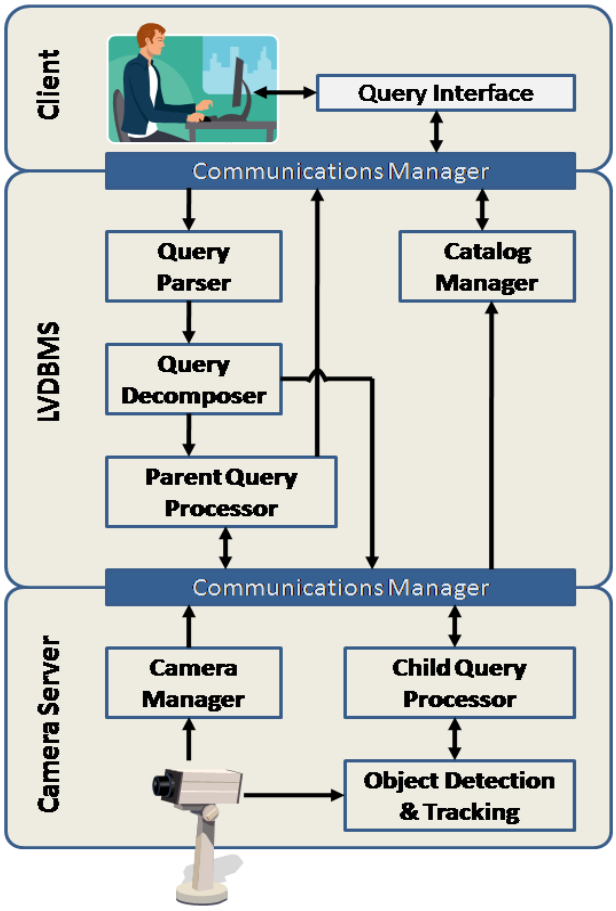


Figure 27. System architecture

Spatial operators are evaluated on camera servers by expanding their operands into sets of qualifying objects (one set for each operand). A set could be empty (no objects satisfy the operand's selection criteria, or in the case of dynamic objects, none currently exist in the video stream), may contain a single object (in which case the query specified a particular video stream

and object id at its inception into the LVDBMS), or can contain many objects (i.e., all dynamic currently detected in a video frame).

Operator evaluation then proceeds by performing a nested loops join between the two sets of objects. Note that, depending upon the operator, if certain conditions are met that would permit the operator to return *true*, the nested loops join can exit early, before performing all n^2 computations assuming the size of each set is n.

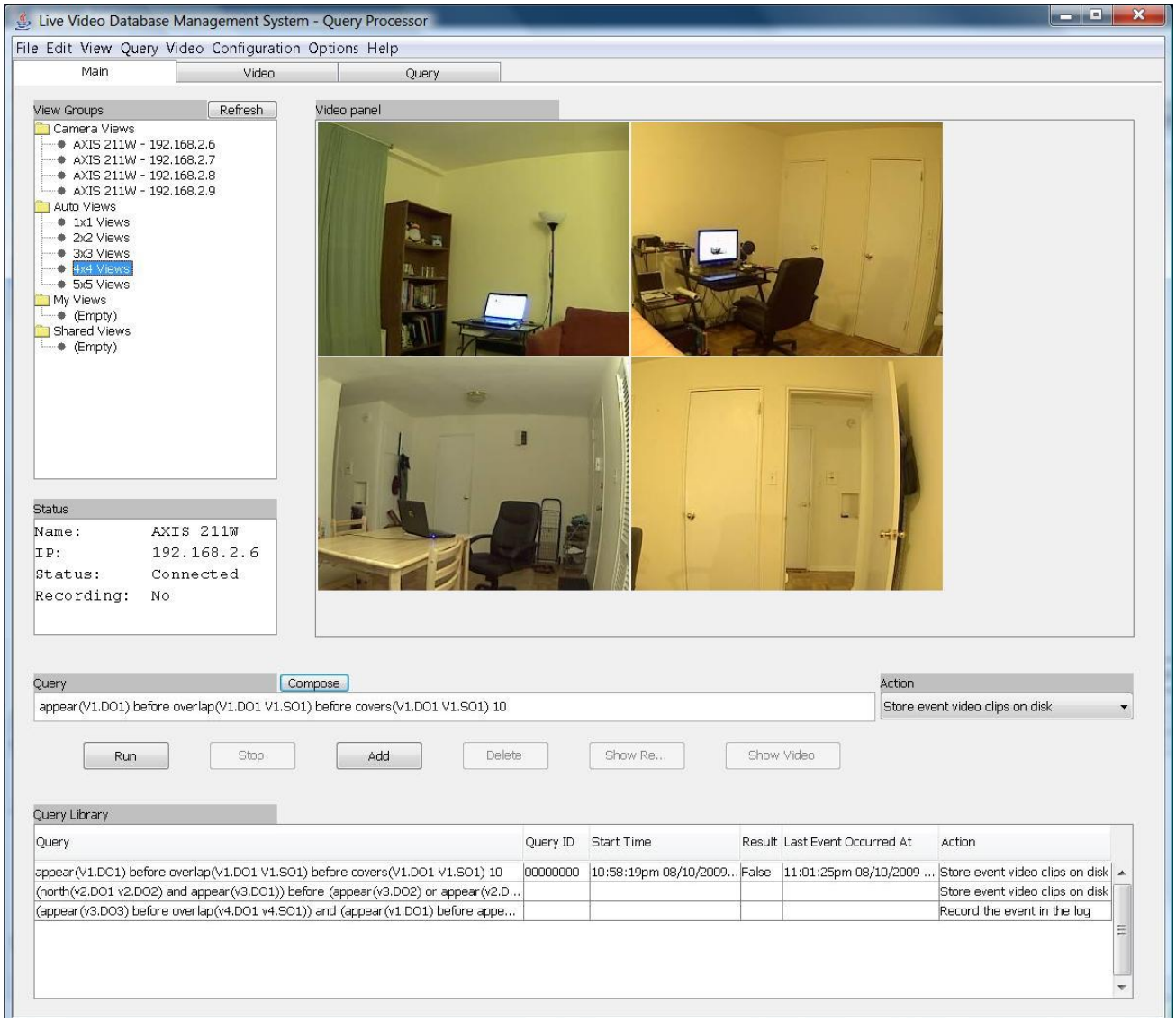


Figure 28. User interface – main window

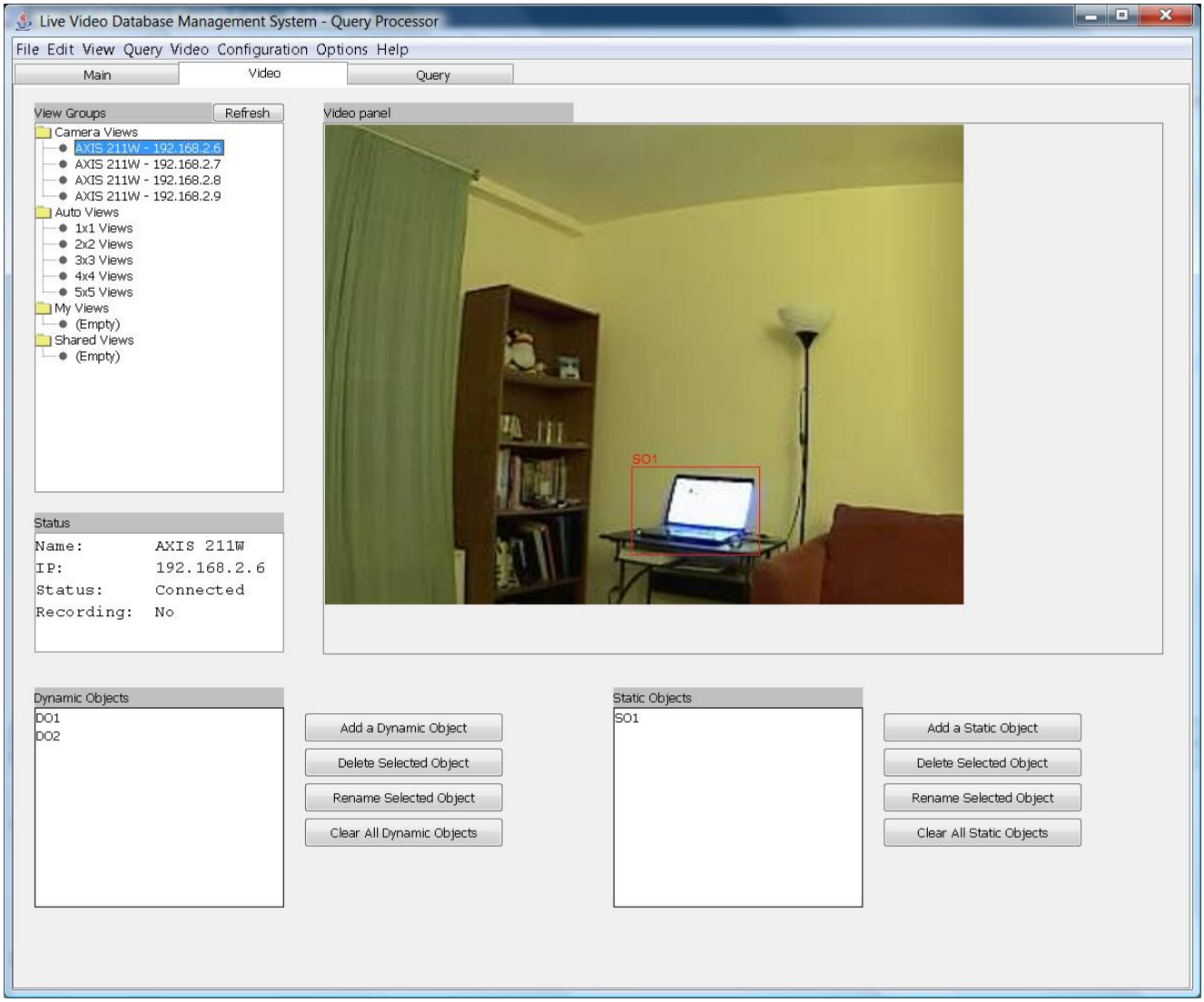


Figure 29. User Interface – video window

In addition to the aforementioned set and nested loops join operations, a queuing model is also implemented as a part of the operator evaluation logic. Operators pertaining to a particular camera server are conveyed to that camera server in a tree-structured hierarchy within the child processor. Each leaf node in this tree is associated with it a queue, to enqueue the results of each individual operator as it is evaluated on each frame. The LVDBMS only needs to be informed when the state of an operator changes, e.g. when the value of an *overlap* between two objects

changes from True to False. This tree-structured queuing model permits some of the query evaluation computation to be offloaded to execution threads on the video servers, permitting the LVDBMS to be notified only when object relations specified by the low-level query operators change, rather than having to evaluate an entire complex query each time a new video frame is received from each camera on each video server associated with a complex query.

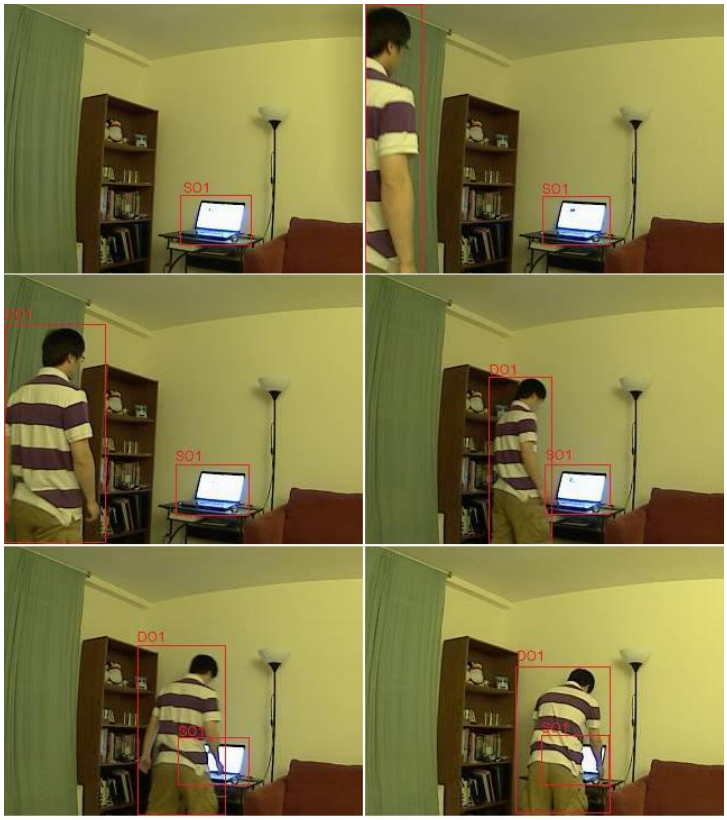


Figure 30. Event Detection

3.5 **Experimental study**

We study the performance of LVDBMS in providing real-time event detection services on top of live videos within the system prototype presented. We focus on the efficiency of query processing and the scalability of the framework.

In our experiments, we have a LVDBMS server running on a HP laptop with 2.0GHz CPU and 4GB of RAM and a camera server running on a HP desktop with 2.5GHz CPU and 4GB of RAM. Our four live video cameras are AXIS 211W wireless cameras with a resolution at 320 x 240 and a frame rate at 15 frames per second. For evaluation purposes, we have also created software simulators for AXIS cameras and camera servers so that we can simulate arbitrary number of cameras and camera servers in our experiments. All communications are wireless through an 802.11g access point. Our experimental setting is illustrated in Figure 31.

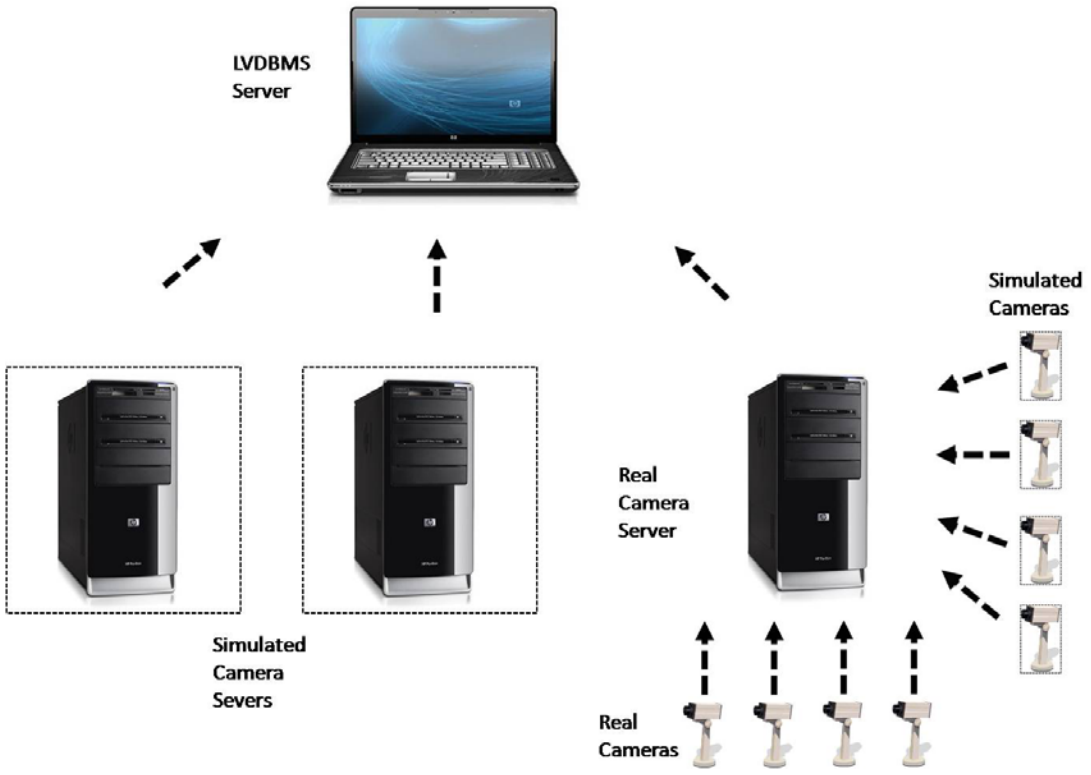


Figure 31. Experimental setting

In our first set of experiments, we measure the query processing overhead and evaluate the scalability of camera servers with increasing number of cameras. Specifically, we vary the number of cameras connecting to the camera server and collects data on the actual frame rates at

each camera. Then we calculate the average frame rate as the performance benchmark as it reflects the overhead of query processing. Two scenarios are tested:

- No query is running and all cameras are in view-only mode. This serves as the reference performance where there is no query overhead.
- Each camera is involved in at least one query that is running at the camera server. This is to make sure event detection is running on every video.

The results are shown in Figure 32. When the number of cameras increases from 1 to 11, the average frame rate decreases slowly and by less than 7%. This shows that very little overhead was incurred by query processing activities at the camera server when there are no more than 11 active cameras. However, when more cameras are connected, the frame rate at each camera slows down faster and reaches close to 50% degradation. This indicates that the camera server start to become saturated once the number of cameras passes 12. As the CPU utilization does not exceed 60% at all tests, we conclude that the reason of this saturation is due to bandwidth limitations with the wireless network and can be addressed with a better network connection.

In our second set of experiments, we focus on the performance impact brought by increasing number of queries. We conduct comprehensive experiments in 18 different scenarios corresponding to 18 different numbers of cameras. For each of the 18 scenarios, we vary the number of active queries from 1 to 10 and record the frame rates data to calculate average frame rate.

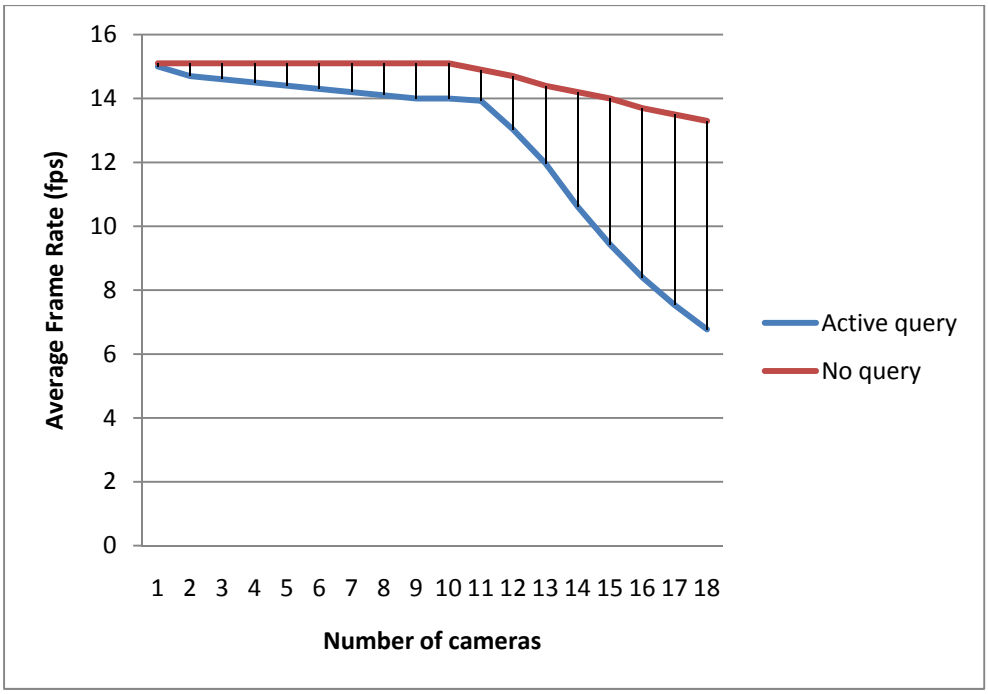


Figure 32. Impact of number of cameras on performance

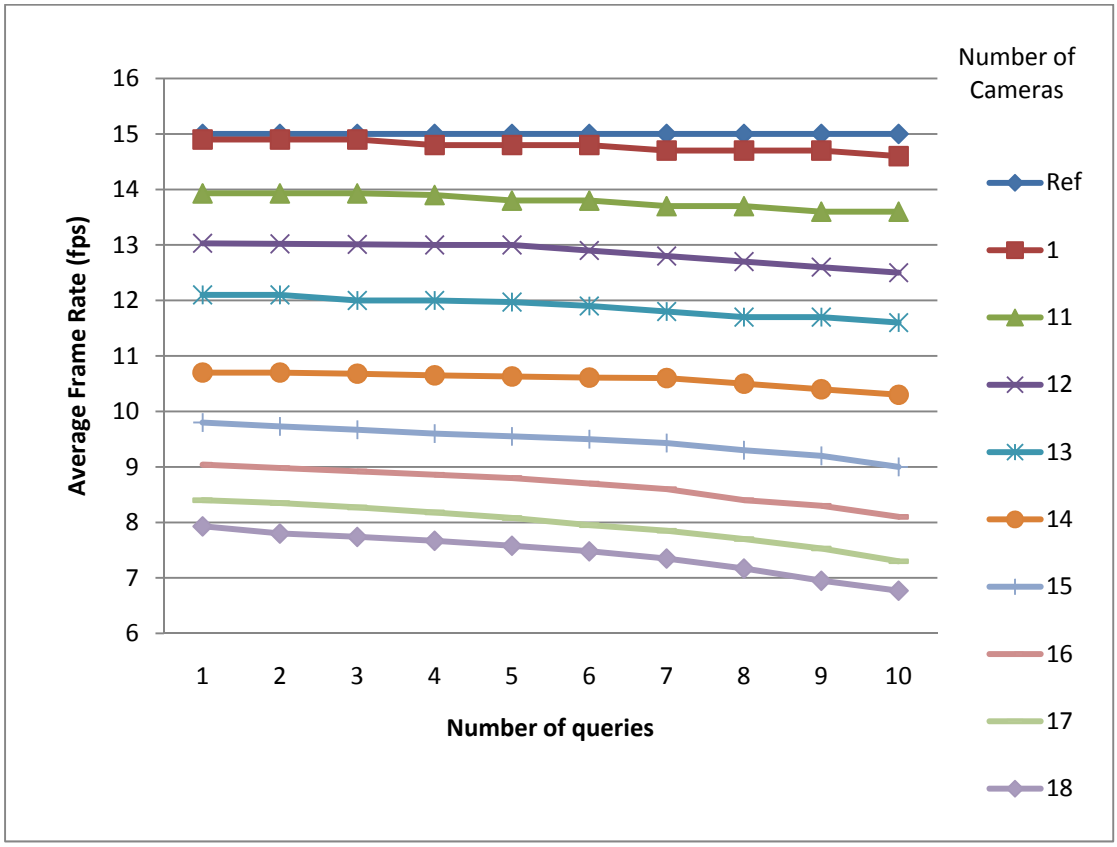


Figure 33. Impact of number queries on performance

Figure 33 depicts the detailed results from our experiments. The performance under un-saturated situations (i.e. with no more than 11 cameras) is similar and the data between 1 and 11 are omitted in the diagram. As indicated in the diagram, performance of a camera server degrades only mildly as the number of queries increases and the maximum degradation is 14%. The exhibited scalability of LVDBMS is expected because more queries only increases the computational complexity at LVDBMS server and has little impact on camera servers. Specifically, the object detection operations from one camera at a given camera server can be shared by all queries concerning that particular camera. In other words, the camera server only needs to process each frame from the camera once before all objects are detected. This nice characteristic gives the camera server good scalability with increasing number of queries and therefore helps the whole LVDBMS framework to accommodate a large number of queries.

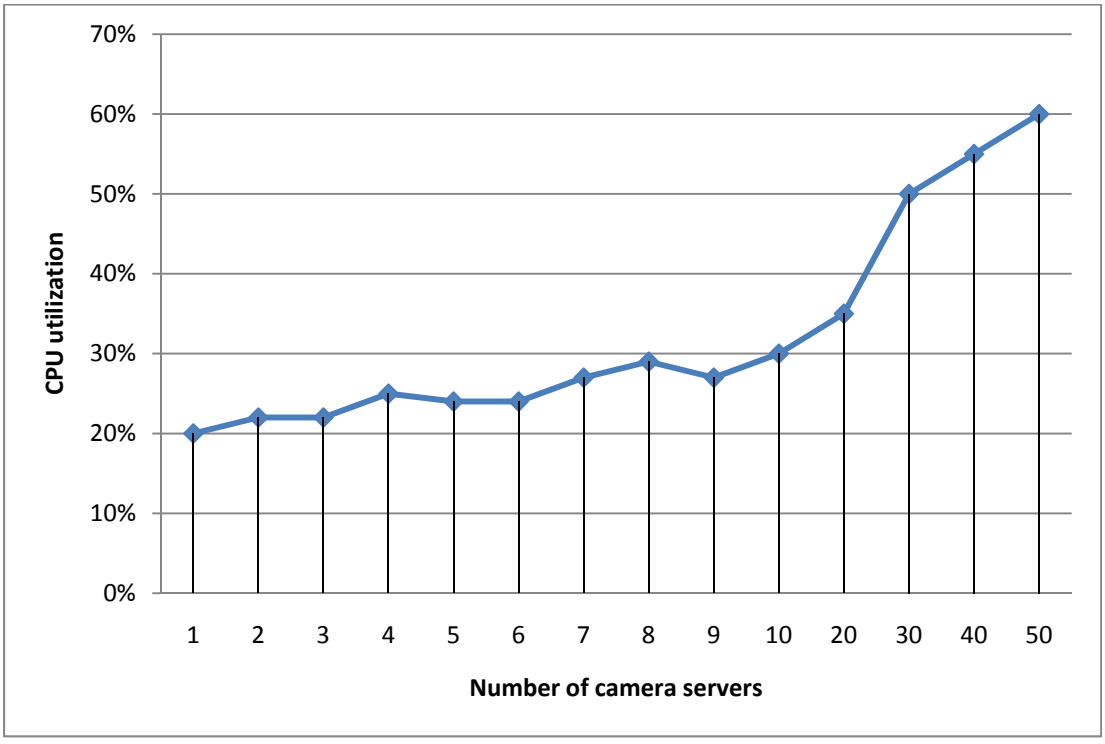


Figure 34. LVDBMS server CPU utilization

We also study the scalability of LVDBMS server in our third set of experiments. In these experiments, we simulated a number of camera servers along with one real server and measure the CPU utilizations at LVDBMS server. Each simulated camera server has a random number of queries and a random number of cameras. As illustrated in Figure 34, the results indicate that LVDBMS server scales well with a large number of camera servers and the CPU is never saturated even with 50 camera servers. This is due to the fact that most of the intensive video-related computations are pushed to camera servers and typical communications between LVDBMS server and camera servers are only binary streams which require little bandwidth and computational resources from LVDBMS server.

Our query optimization techniques also contribute to this good scalability. In our experimental study, we also evaluate the impact of our query optimization strategy on computational and communication costs by comparing the two execution plans in Figure 24(b) and Figure 25. The results are as follows:

- With optimization, the LVDBMS achieves a saving of 70% in terms of computations on evaluating operators.
- Each participating camera server evaluates 40% more operators on average with optimization.
- Number of data messages exchanged between the LVDBMS and camera servers decreases by over 50%.

In summary, this optimization strategy has two major advantages:

- By pushing some logical and temporal operators down to the camera servers, it allows those operators be processed locally, greatly reducing the data communications between the LVDBMS and camera servers.
- By shifting a portion of the computation to camera servers, it leverages the computing capability of distributed camera servers to reduce the load on the LVDBMS.

3.6 **Conclusion**

In this work we present a new class of database management systems termed *live video database management systems* (LVDBMS). Unlike traditional video databases, which are comprised of video files previously captured, stored and indexed, a live video database operates on real-time, low-latency live video streams obtained from networks of distributed cameras. In this work we introduced the following concepts:

- a query language suitable for expressing ad hoc spatiotemporal and event-based queries over live video data,
- distributed processing techniques for the real-time evaluation of continuous queries posed over live video data, and
- query optimization techniques for efficient stream computations.

The proposed LVDBMS approach enables rapid development of applications for distributed camera networks, similar to how general-purpose relational database management systems are used in database application development today.

Our future work includes investigation of more advanced query optimization techniques. We intend to extend the query model, currently based on high-level semantics such as spatiotemporal relationships between objects, to also include low-level visual features such as color and textures,

as well as motion trajectory. It is also desirable to develop a software development environment (i.e., an application programming interface, or API) to support application development using a host programming language, such as Java or C#. Finally, the LVDBMS approach is best suited in supporting camera networks with infrastructures owned by the same organization. In a separate research effort, we are investigating a peer-to-peer environment to allow different organizations to share and query various camera networks deployed on the Internet. A specialized browser is used as the user interface with various capabilities supported as software plug-ins.

4. CROSS CAMERA OBJECT TRACKING IN LVDBMS

4.1 Introduction

Multiple live cameras surveillance is a labor intensive task whose effectiveness is largely determined by the number and vigilance of the operators. However, human vigilance deteriorates quickly as the operators who constantly monitor these cameras, such as baggage screeners at airports, quickly become fatigued. Moreover, when tens of thousands of cameras are present, such as those deployed on streets in a city, it is expensive, if physically feasible, to either employ a large number of operators or house a huge array of monitors to support the various monitoring operations. As a result, video analytics as a recent technique for automated visual monitoring and surveillance have been developed to help solve the aforementioned challenges.

In this work, we address a different aspect of video surveillance challenge, namely a framework to support a variety of surveillance applications over a general-purposed camera network. In other words, many independent user groups are able to share the same camera network for their different surveillance needs much like people are sharing their computer servers and network infrastructure today. To enable this new capability in camera networks, a technique called *Live Video Database Management System* (LVDBMS) (Peng, 2010) has been recently proposed. This camera network approach views the distributed video cameras as a special class of storage, with their live video streams as database content. This database model allows different users to query the video streams in an ad hoc manner to satisfy various surveillance needs. As a database approach, LVDBMS also enables rapid development of surveillance applications for a general-purposed camera network much like database applications are developed for general-purposed computers today.

Unlike video database systems that deal with stored video files, LVDBMS is built atop live videos and consists of the following core components: a spatiotemporal data model for live video streams and events of interest from distributed cameras, an event-based query language for composing spatiotemporal queries, and query processing techniques for computing queries on live video sources. LVDBMS, in its current form, does not have the capability to track objects across cameras that share no overlapping views as required in a fully automated surveillance system. In many scenarios where the camera has a limited resolution and the observable areas are constrained by the camera setup, the whole surveillance area cannot be possibly covered by a single camera. Thus the capability to recognize objects while they appear in different cameras is a requirement as to enable monitoring of wide areas. Furthermore, installing cameras with shared views to observe a wide area is not often a feasible option because of computational and economic concerns. Therefore in most scenarios, the systems are required to be compatible with multiple cameras that do not share common areas of view.

The problem of cross camera tracking without overlapping views presents a number of challengs in a live video database for a number of reasons. First, a live video database such as LVDBMS needs to segment and track salient objects across cameras in real time with high efficiency. However, existing tracking algorithms as such are typically computation-intense and often require scanning video files several times, making them prohibitive in a time-aware live environment. Second, cross camera tracking with non-overlapping views normally requires a features database for objects that have appeared in each camera, as the appearance of an object may occur in different time and space and in different cameras. It is foreseeable that after a certain amount of surveillance time, the database will have a size that is unmanageable and no longer provide reasonable query response time. In general, video analytics, often in times

relying on computationally sophisticated vision techniques to evaluate stored videos, are inadequate to adopt in a live surveillance environment which imposes strict constraints on the time to process one frame.

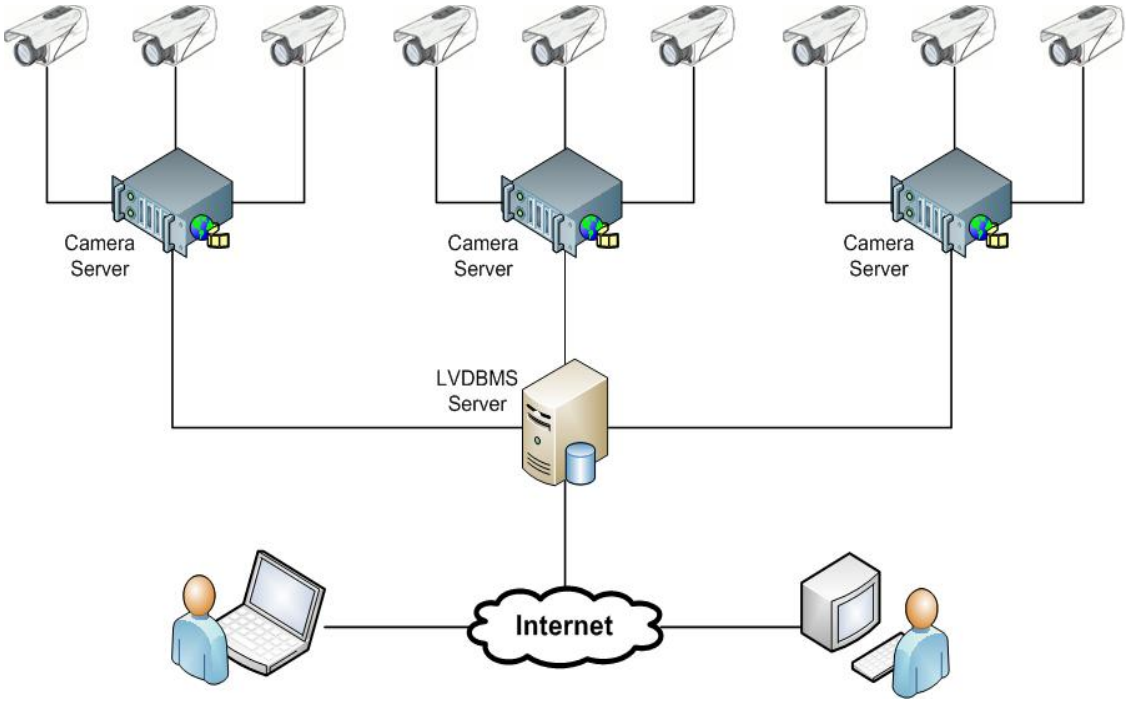
In this work we present a novel technique to address the cross-camera tracking problem within the LVDBMS framework by employing a distributed in-memory database and utilizing unique characteristics of LVDBMS to keep the size of this database minimal. A color histogram-based technique is used to match objects by calculating their similarity scores. Our experimental results show that the proposed technique achieves high matching accuracy while keeping the processing time low.

The remainder of this chapter is organized as follows. In Section 4.2, we discuss related work including the LVDBMS and existing multi-camera tracking techniques. We introduce the proposed techniques for cross-camera tracking in Section 4.3. The experimental study is presented in Section 4.4. Finally, we offer our conclusions and discuss future work in Section 4.5.

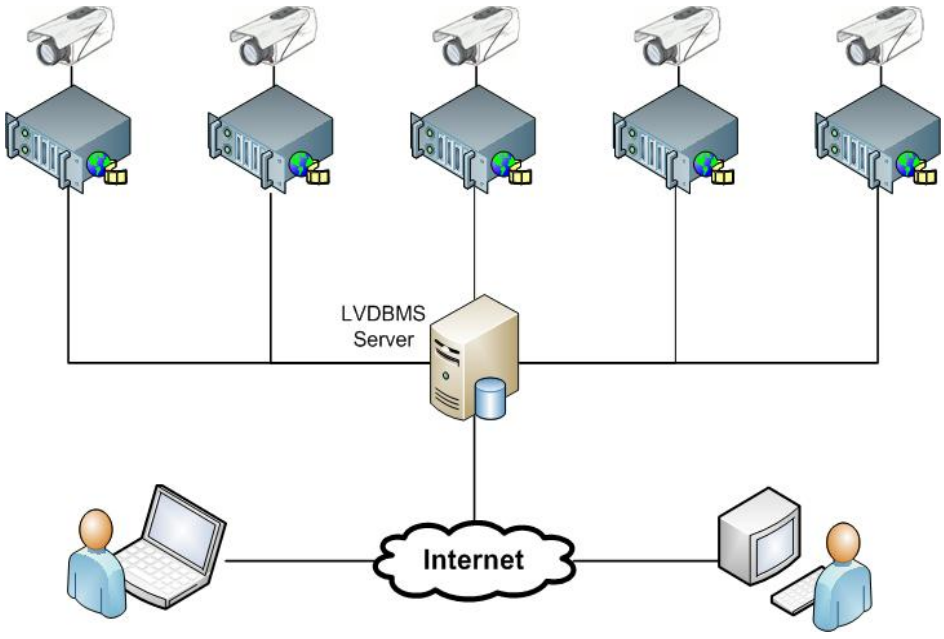
4.2 **Related work**

4.2.1 Live Video Database Management System over a network of smart cameras

LVDBMS (Peng, 2010), in contrast to conventional VDBMS's, is a general purpose live video database management system that provides real-time spatiotemporal query support over distributed camera networks. A user poses a continuous event query using a spatiotemporal query language with LVDBMS, which then evaluates the query predicate continuously in real time. LVDBMS assumes a three-tier distributed network architecture, as illustrated in Figure 35(a), where the middle tier is camera servers which are computers strategically placed to manage nearby cameras and receive video streams directly from them.



(a) Multiple cameras share a common processor



(b) Each camera has its own processor

Figure 35. Three-tier architecture for camera management

Smart cameras distinguish themselves from conventional cameras, purely video signal capture devices, with built-in processors to provide additional video processing functionality (Hengstler, 2006; Kleihorst, 2007; Winkler, 2009). With respect to their computational capabilities, smart cameras can be categorized into two types: (1) the built-in processors have limited computational power and resources, and thus are pre-programmed to perform a number of pre-set video processing operations; and (2) the built-in processors have sufficient computational power that allows them execute sophisticated video processing tasks with customized logic. The architecture of LVDBMS applies to both types (1) and (2) smart cameras. Additionally, for type (2) cameras, LVDBMS has the flexibility to take advantage of the powerful built-in processors and therefore deploys less camera servers. In the extreme case that all smart cameras in the network are type (2) cameras, the architecture of LVDBMS can essentially be reduced to a two-tiered as displayed in Figure 35(b).

When a complex query is entered, the LVDBMS server decomposes it into a tree of sub-queries, and assigns the single-camera sub-queries to appropriate camera servers for distributed processing. Camera servers, accepting live videos from associated cameras, are responsible for executing sub-queries on individual cameras and returning continuous bit streams as answers. By leveraging the processing capability of the distributed camera servers, the LVDBMS server no longer needs the bandwidth and computational resource to process all the live videos as the communications between the LVDBMS server and camera servers involve only textual commands rather than video data, making the entire camera network scalable to a large number of cameras.

A live video database is a collection of objects captured continuously in real time and interrelated by temporal and spatial relationships. LVDBMS provides an application-independent

SQL-like query language to facilitate spatiotemporal queries spanning over multiple cameras. As its syntax indicates, a user can use 20 primitive operators including spatial, temporal and logical operators as building blocks to compose a potentially rather sophisticated query that takes into account multiple objects appearing and interacting in different live videos.

The limitation with the current form of LVDBMS is that objects are detected and tracked only within the same scene captured by the same camera. When an object moves out of sight of the camera and enters the view of another camera or even the same camera afterwards, it will be recognized as a new object. The motivation of this work is to fill this void and greatly enhance LVDBMS with the critical capability to track objects from a camera to another when at the same time fulfilling the real-time requirement of a live video database.

4.2.2 Cross camera tracking techniques requiring overlapping views

There has been extensive work in the research community of cross camera surveillance that requires overlapping views. Calibrated cameras were used by Jain (1995) along with a model to retrieve the 3D location of an object. Correspondence was established by linking multiple views of a single object with a single 3D location. Cai (1999) utilized intensity and geometric features along with multiple calibrated cameras to match objects for tracking. Multi-variant Gaussians are used to model the features and matching is done through the use of Mahalanobis distance. Mittal (2003) created a region-based technique by calculating the depth of foreground objects points. The points were then mapped and clustered on the ground plane to locate objects. Chang (2001) establishes correspondence by computing epipolar lines from the top most point of an object. The distance between the epipolar lines along with height and color was combined to be used in tracking. Bayesian networks are also used by Dockstader (2001) in which features include estimation and appearance of sparse motion. In (Kang 2003) an algorithm is proposed by

registering the ground planes in the stationary and pan-tilt-zoom cameras and stabilizing camera sequences through affine transformation. Tracking is done by projecting object locations onto a global coordinate system and object regions are partitioned into their polar forms to model object appearances where color variation is modeled as a Gaussian distribution.

In (Lee, 2000) camera calibration is not required but to recover the information of camera calibration motion trajectories extracted from multiple views were matched against one another and the matches with highest hits were used to compute plane homographs.

All aforementioned techniques assume shared views of the cameras. This condition is not usually feasible due to high economic and computational costs to cover a wide area. Moreover, this assumption cannot be made when designing for a general purpose video database. With this consideration, we do not assume overlapping views in our techniques.

4.2.3 Cross camera tracking techniques without overlapping views

Huang (1997) discussed a probability-based method for tracking cars along a highway in calibrated cameras, assuming that vehicles must be traveling in one direction and they are in one of three lanes. The feature was the mean value of the object color and therefore it is naturally not suited for tracking objects with non-uniform colors such as a person. Gaussian distributions were used to model transition times. Assuming the probabilities of initial transition were obtained, the problem became one to find correspondence by assigning weights. Kettnaker (1999) reconstructed object paths from different cameras by employing a Bayesian formulation. The allowed topology of movement routes and the transitional probabilities must be manually input into the system. Histograms were used to represent object appearances. To track people in a context without shared camera views, Collins (2001) presented a method involving a 3D site

model in which tracking is achieved through cross correlating and normalizing sensed objects with their locations in calibrated cameras onto the model.

Porikli (2003) used a brightness transfer function (BTF) in a tracking technique for cameras without shared views. Cameras are paired up and for each pair a BTF is computed to link color values observed between corresponding observations in the two cameras. As soon as such mappings are known, the problem of establishing correspondence become transformed appearance models or histograms matching. This mapping, however, may change across frames determined by an array of camera properties including aperture size, focal length, exposure time, scene geometry, and illumination. Thus, the constant BTF that was computed beforehand is often not suitable for use in matching objects for sequences that are reasonably long. Shan (2005) matched appearance between camera views that are not overlapping by learning edge measures in an unsupervised approach. To match two camera observations a different or the same object, Gaussian probability density functions are computed from the two observations. This presented technique required the registration of the vehicles edge images. For non-rigid objects such as people, however, this requirement of object images registration may not be achievable. Additionally, the observations of the objects have to be considerably similar in different cameras.

Makris (2004) used a two-stage method to calculate the network topology of the cameras. In the first stage, the method determined for each camera the zones of exit and entry. In the next stage the co-occurrence events of exit and entry are used in the calculation between the zones to determine links across cameras. The assumption with this technique is that incorrect correspondences would likely result in scattered points in the features space whereas correct correspondences would appear clustered across the features space. The proposed technique

assumes that when an exit and entry in a given time period is more probable than randomness then they have a higher probability to be linked.

Rahimi (2004) presented an algorithm to reconstruct camera calibrations of the ground plane and as the object moved across cameras without overlapping views it reestablishes the complete object track. The Markovian process was used to model the dynamics of moving objects. The algorithm used a non-linear minimization scheme to estimate the trajectory that the object dynamics are most compatible with, taking into account the velocity and location of the object in different cameras. The assumptions made in this work are that object trajectory data were all available and that every object must be moving on the ground planes.

Stauffer (2005) discussed a linking algorithm with the improvement on testing the hypothesis that the expected correlation with no valid transitions of an object is relatively close to the correlation of entry-exit events containing valid transitions. The improvement enabled the technique (unlike (Makris, 2004)) to take care of the case in which the correlations are not because of valid object transitions but exit-entry events are correlated.

These techniques, while highly effective in their respective problem domains, usually incorporate sophisticated vision operations that cannot be completed in time in a real time environment. Moreover, they were not designed to work with a generic live video database without making any assumptions on camera locations or characteristics of objects.

4.3 **Cross-camera tracking with non-overlapping views**

In this section, we discuss the design of the proposed technique. We begin by presenting a high-level overview and follow up with detailed descriptions on the object identification algorithm and database designs.

4.3.1 Overview

We design our technique on the three-tier network architecture within the LVDBMS framework and take advantage of strategically placed camera servers to provide distributed processing power to alleviate the computational load at the LVDBMS server. The LVDBMS server maintains an in-memory database of the current active objects that need to be tracked across cameras with each object's identifier, visual features, and a trajectory of all the cameras that have observed this object at different times. A camera server keeps a fragment of this in-memory database that is pertinent to this particular camera server. That is, a camera server only keeps a local copy of the current active objects that need to be tracked on this camera server, each object represented by its object ID and visual feature vector.

Queries in LVDBMS are processed in three steps, query registration, query decomposition, and query execution. Query registration happens when a user submits a composite query through a graphical user interface. A LVDBMS query parser accepts the query, parses it into tokens, and generates a query tree. By traversing this query tree, the LVDBMS determines if this query involves any *global objects*, defined as those to be tracked across multiple cameras, as opposed to *local objects* which are tracked within the same scene in the same camera. When the LVDBMS finds any object, declared in the query, subject to tracking across cameras, it records the information of this global object in the in-memory database.

The query decomposition stage is where the original query gets decomposed by the query decomposer into individual single-camera sub-queries. The input of the query decomposer is the parse tree, and the output is a query decomposition tree showing the query plan. At this stage, the query tree on a composite event (Figure 21) is decomposed into individual leaf nodes corresponding to spatial events, and the remaining tree consisting of a hierarchy of temporal and

logical operators. All the sub-queries are assigned unique IDs and forwarded to the relevant camera server for evaluation. Any global objects that are newly introduced by the query will be forwarded to pertinent camera servers as well. The LVDBMS Server then instantiates a query processor for the execution of the remaining tree structure with logical and temporal operators. Some of these operators wait for the event streams (i.e., bit streams) from the participating camera servers. For each operator, the query processor allocates a dynamic queue to buffer the output bit stream. The next operator in the tree can then consume data from this FIFO buffer. Figure 36 illustrates the process of query decomposition and query processor instantiation. Upon receiving a sub-query and possibly global objects information from the LVDBMS, each camera server also instantiates a query processor with the spatial operator and corresponding queue. In this work we name the query processor in the LVDBMS as the *parent query processor*, and the spatial query processor at each camera server as the *child query processor*. The global objects are inserted to the camera server's local in-memory database.

During the query execution stage, the participating camera servers after instantiating a child processor invokes a real-time video segmentation and tracking algorithm to extract local objects from live videos. For each detected object, its visual features are calculated and used as a query against the in-memory database of global objects. The features of the object are compared to those of the objects in the database and a similarity score is computed for each pair of comparison. If a match is found in the database, which indicates that a global object is detected in a certain camera, the camera server sends a notification to the LVDBMS server to update its in-memory database.

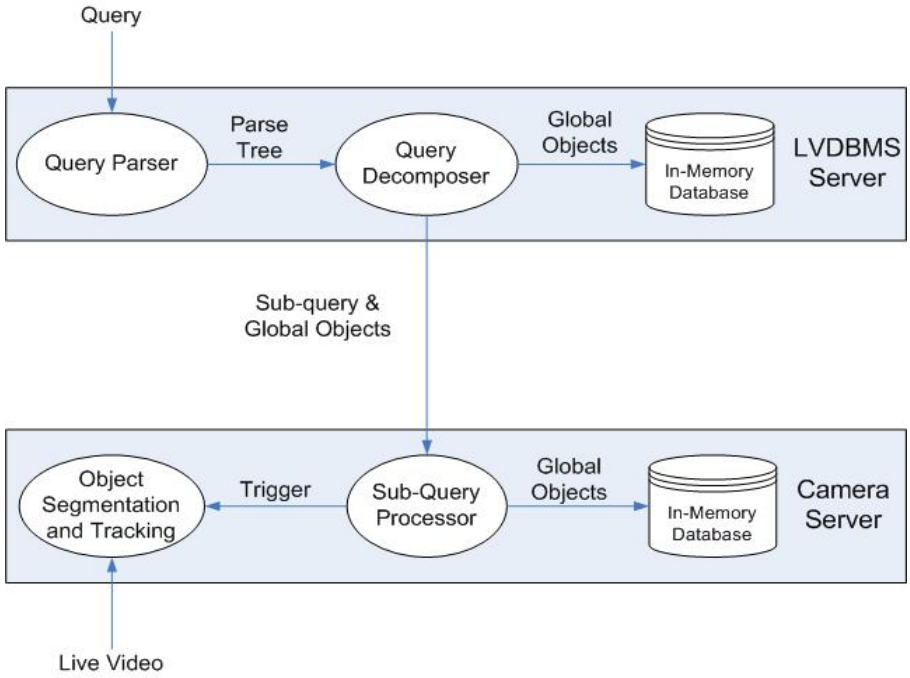


Figure 36. Query registration and decomposition

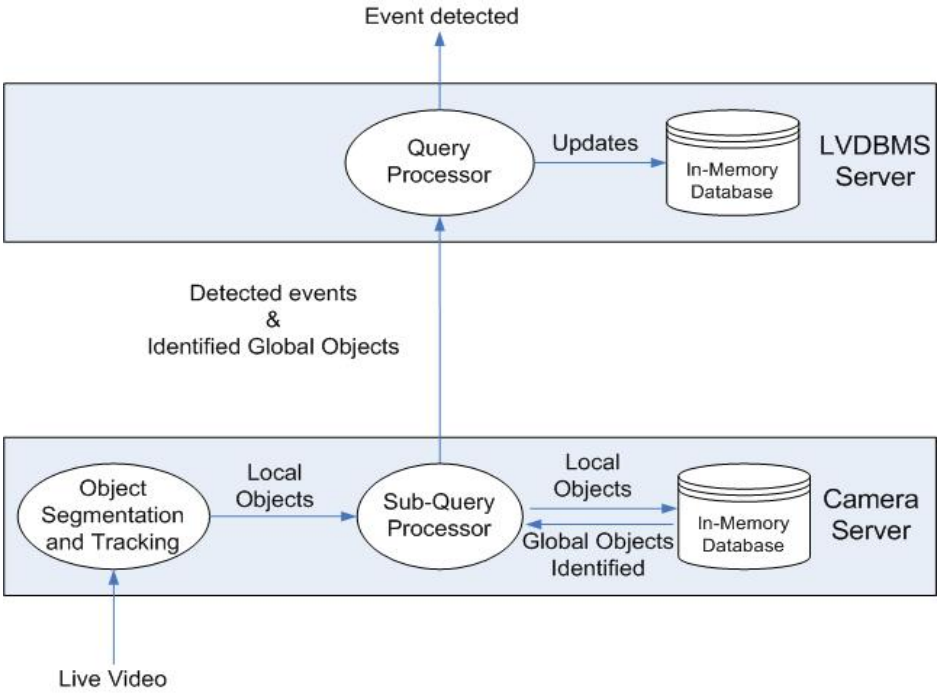


Figure 37. Query execution

As Figure 37 shows, the child query processor in the camera server is responsible for calculating the spatial relationships between the objects and evaluating if a spatial event is detected. The

output of the child query processor is an event stream stored in the queue and delivered to the corresponding parent query processor. As the parent query processor receives results of sub-queries from child query processors at participating camera servers, it assembles and evaluates the composite event by calculating each of the operators, starting from the leaf nodes. The query and all sub-queries continue running until explicitly stopped by the user.

4.3.2 Query language

Following the grammar specified in Figure 21, we use the existing query language in LVDBMS to support existing live video queries. However additional support needs to be made to the query language to allow new queries that require cross camera tracking functionality. In the LVDBMS, each live video stream has a unique ID and each object within a video stream has a unique object ID. Therefore we can reference an object by its unique global ID: <camera ID>.<object ID>. There are two types of objects in LVDBMS: *existing objects* and *dynamic objects*. Existing objects refer to those that appear in a video at the time a query is specified. These objects are either detected by the system automatically using object tracking algorithms or specified by the user drawing a rectangle on the video screen. At query initiation time a user can reference existing objects by their IDs. Dynamic objects are those not viewed by the camera at the query time but are detected automatically by the system. An issue with using dynamic objects in queries is that they cannot be referenced by ID because they do not have one when the query is instantiated. We address this issue by adding two wildcards (‘?’, ‘*’) and one keyword (AS) to the LVDBMS query language. The following examples illustrate the usage of this feature:

‘c1.?’ : matches any dynamic object from camera c1.

c1.*’ : matches any object from camera c1.

'c1.? AS o1' : matches any dynamic object from camera c1 and assigns ID o1 to this object.

'c1.* AS o1' : matches any object from camera c1 and assigns ID o1 to this object.

By utilizing wildcards and the 'AS' keyword, we are able to allow users to define and use global objects in their queries using the query language.

4.3.3 Object tracking algorithm

Suppose there is a network of n smart cameras $S_1, S_2 \ldots S_n$ without assuming shared views and they observe m objects $O^V_1, O^V_2 \ldots O^V_m$ appearing in their surveillance views. These objects are observed at different time periods by different cameras. Let A be the set of all appearances. Moreover, let $A_j = \{A_{j,1}, A_{j,2}, \ldots, A_{j,lj}\}$ be the observation set of camera C_j. Every observation $A_{j,a}$ is extracted from an object in the observation area of camera C_j. The cross camera tracking algorithm is to label the observations from multiple cameras belonging to a given object. Another way to formulate the problem is to think the observation set of each object to be an observation chain with later observations following earlier ones. As an object moves around in a network of cameras, the tracking algorithm aims to take the observations of the object entering a camera and match to the observations of the same object exiting some other camera.

With the design of the distributed in-memory database for global objects, tracking across cameras is essentially achieved through visual features matching of a local object, the "query", against a set of global objects, the "data". In this section we present a color histogram based algorithm to carry out the task of features matching.

Within a discretized color space with axes such as RGB, the color histogram can be computed by making the image colors discrete and calculating the number of occurrences of every color. Thus the color information is represented by three 1-dimensional histograms (one for each of the three

color components). One feature of histograms is the invariance against rotation and translation, and also insensitive against occlusion, scaling, and change of viewing angle. At the same time, different objects typically exhibit considerably different histograms. These nice traits make histograms an excellent representation as a distinguishing visual feature to identify objects in a video surveillance environment.

Due to the time constraints imposed by the live videos, we can only afford a tightly limited amount of computational time in between frames, while we must address the challenges to accurate object tracking, more specifically:

- Background distractions of the object which is represented by a minimum bounding rectangle rather than a closer contour,
- The variety of angles and viewpoints that the object can be observed from,
- Varying image resolutions, and
- Occlusion.

The matching algorithm discussed here addresses all four problems with a reasonably fast matching time.

Let H(i) be a histogram of an object, where the index i denotes a histogram bin. The normalized histogram N(i) is defined as:

$$N(i) = \frac{H(i)}{\sum_i H(i)}$$

Let A_R, A_G, and A_B be the normalized color histograms of the query object, and B_R, B_G, and B_B be the normalized color histograms of a global object in the database. Also let |A| denote the area

of A in terms of number of pixels. The matching score of the two is defined as the intersection of the histograms:

$$\text{Score}(A, B) = \frac{\sum_r \min(A_R(r), B_R(r)) + \sum_g \min(A_G(g), B_G(g)) + \sum_b \min(A_B(b), B_B(b))}{\min(|A|, |B|) * 3}$$

Note that a matching score lies in the interval of [0, 1] and the score is 1 when the two histograms are identical.

4.4 **Experimental study**

Figure 38 illustrates how a user can specify a query to detect scenarios where someone comes through a door observed in one camera and sits on a chair in the view of another camera:

appear(V1.DO1 AS GO) before overlap(GO V2.SO1) before covers(GO V2.SO1) 10

In Figure 38, camera #1 captures a scene where a person is entering through a door and camera #2 captures a person approaching a chair. The tracking algorithm is the underlying logic that recognized the two persons as the same one. This query is evaluated to be true in the last frame once the person's minimum bounding rectangle covers that of the chair.

Figure 39 illustrates a scenario where multiple objects are moving in and out of a camera view. The tracking algorithm is able to detect all objects and correctly track them.

Camera 1:

Camera 2:

Figure 38. Event Detection

Figure 39. Multiple objects tracking

Experiments have been performed to test the speed and the effectiveness of the tracking technique. The three opponent color axes are defined for the histograms in (Ballard 1982) as:

" $rg = r - g$

$by = 2 * b - r - g$

$wb = r + g + b$ "

The notations *r*, *b*, and *g* respectively denote red, blue, and green attributes. *wb* represents the intensity axis. As the *wb* axis is more sensitive to the distance from the source of light and variation of lighting from shadows, these color axes are formulated in a way to sample the intensity in a larger granularity than *rg* and *by*. We divide *wb* axis into 4 sections, while dividing the other two into 8 sections each. Thus the total number of bins is 256 and the threshold of the similarity score is 0.4.

The experiments have a setup of 66 videos that are used to simulate live feeds from an array of cameras. The database is built with objects that appeared in at least two scenes. To evaluate the capability of the algorithm to handle different orientation, scaling, and occlusion conditions, we carefully chose video scenes that contain a variety of views for a given query object. As an example, Table 3 shows the different views of a query object.

Experiments were conducted where our tracking algorithm detects objects from the simulated feeds of the 66 scenes and uses the detected objects as queries to the database. The results are summarized in Table 4. These results demonstrate that the tracking algorithm matches local objects with global objects with a high accuracy. Moreover, the color histogram-based algorithm is fairly insensitive to rotation, scaling, and occlusion.

Table 3. Different views of the same object

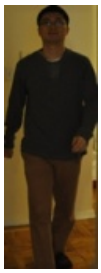
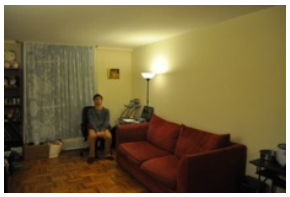
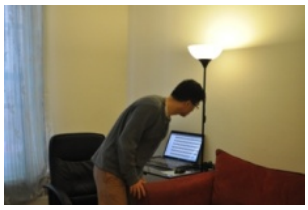
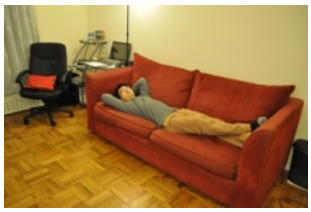
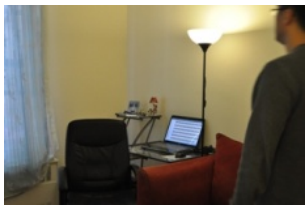
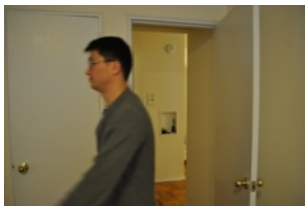
Query 	Scaling 
Rotation  	Occlusion  

Table 4. Match accuracy for different types of views

View	Correctly matched/Total	Percentage
Upright	248 / 252	98.4%
Rotated	91 / 95	95.8%
Scaled	89 / 92	96.7%
Occluded	168 / 178	94.4%
Total	596 / 617	96.6%

The impact of different histogram sizes is shown in Figure 40 where we vary the number of bins from 64 to 8192 bins over two orders of magnitudes. As the match effectiveness does not vary much over the whole span of histogram sizes, the outcomes of histogram intersection are rather not sensitive to the bin numbers in the histograms used in the query and database objects. Note that higher number of bins does not necessarily lead to more accurate matches, but rather greatly increases the computations.

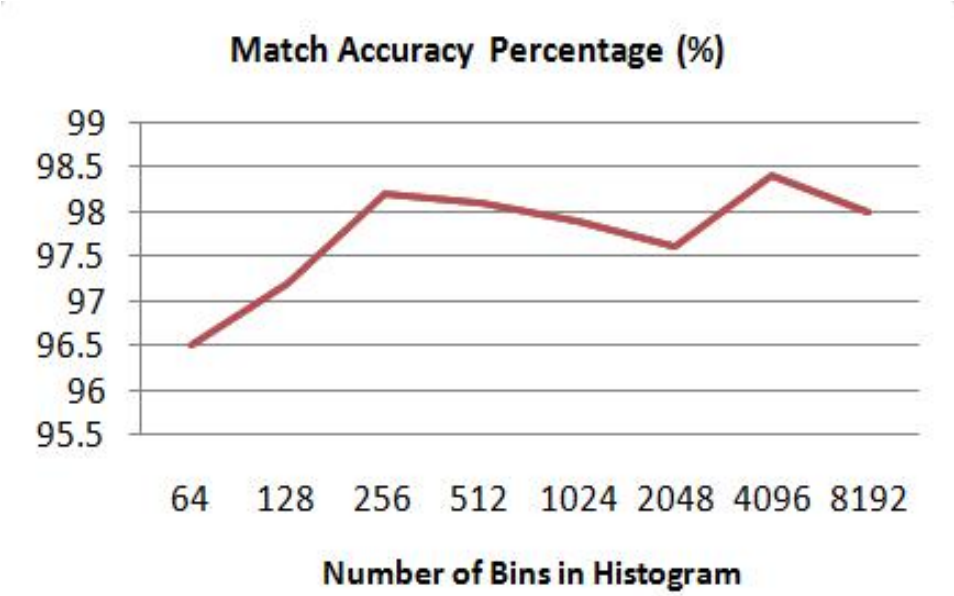


Figure 40. Impact of varying histogram resolution

We also measured the time it takes to complete a query. On an HP workstation with 2 cores at 2.53GHz each, computing three 8*8*4 histograms from a 320*240 image takes about 5 milliseconds and performing a histogram intersection takes about 0.9 milliseconds. Assuming that each camera generates a live video feed at 15 frames per second and that the in-memory database holds 20 objects on average, a camera server is able to process live video feeds from 3 cameras without skipping a frame. As the number of cameras per camera server increases or the size of the database increases, the camera server will have to start to skip frames and the frame

rate it can support will be decreasing linearly. However, this decrease of frame rate is not going to seriously impact the accuracy of query results due to the fact that an object, once it appears, usually stays in the view of a camera for more than a few frames. So even if the tracking algorithm has to skip a couple of frames, it can still get a chance to see what the object looks like. Moreover, as long as the object is identified as a global object before it enters the view of another camera, it will not affect the accuracy of query evaluation.

In summary, the results presented in this section have shown that the proposed technique is able to accurately match query objects with database objects at a high speed.

4.5 **Conclusion**

In this work we address techniques for a general-purposed camera networks. In particular, we presented a cross camera tracking technique as an integrated component in a Live Video Database Management System (LVDBMS). We discussed the design of an in-memory distributed database built with the assistance of camera servers which provide video processing capability close to cameras. A color histogram-based similarity measure was proposed to match a query object against the objects in the database in an accurate yet efficient manner. Our experimental results indicated the proposed approach performs at a level appropriate for real time live video processing.

Future work will add more advanced object trajectory query capability to LVDBMS. Although the current language can support trajectory queries, it is desirable to add new operator, e.g., TRAJECTORY(O1)=TRAJECTORY(O2), to make the formulation of such queries more natural. Trajectory queries can be useful in many scenarios such as anomaly detection, and thus add another layer of semantics to existing surveillance systems.

5. CONCLUSION

With the proliferation of sensor networks and smart camera networks, they enable a whole range of new applications in many different areas. Moreover, the amount of data generated by these sensing devices grows with lightening speed and calls for scalable techniques to manage these data, communicate them in a streaming fashion, process them to mine the semantics underlying the signals, and present them to users in an understandable form. In this dissertation, we have proposed three techniques to address these requirements.

In Chapter 2 we presented the first technique, *i*SEE, the first pervasive sensor computing framework that facilitates sensor data sharing, sensor application sharing, and on-the-fly data integration. We envision a new Internet, an Internet of sensors that globally interconnects smart devices and sensor networks. We identified the key challenges in supporting such an environment and proposed solutions accordingly. Our techniques include: (1) the Sensor Service Definition Language for defining services, (2) the Publication Facility for publishing sensor services on the Internet, (3) the Sensor Registry Server for sensor meta-data registration and discovery, (4) two communications protocols for sensor data delivery, (5) the *i*SEE Sensor Browser for accessing and integrating streaming sensor data, (6) the Plug-in mechanism enabling sensor application sharing, and (7) two plug-ins, namely the Visualizer and Plug-inDB, for visualizing and querying sensor data respectively. We have built a prototype as a test bed to evaluate these techniques.

In Chapter 3 we proposed the second technique, a new class of database management systems termed *live video database management systems* (LVDBMS). Unlike traditional video databases, which are comprised of video files previously captured, stored and indexed, a live

video database operates on real-time, low-latency live video streams obtained from networks of distributed cameras. In this work we introduced the following concepts:

- a query language suitable for expressing ad hoc spatiotemporal and event-based queries over live video data,
- distributed processing techniques for the real-time evaluation of continuous queries posed over live video data, and
- query optimization techniques for efficient stream computations.

The proposed LVDBMS approach enables rapid development of applications for distributed camera networks, similar to how general-purpose relational database management systems are used in database application development today.

In Chapter 4 we described the last technique, a cross camera tracking technique as an integrated component in LVDBMS. We discussed the design of an in-memory distributed database built with the assistance of camera servers which provide video processing capability close to cameras. A color histogram-based similarity measure was proposed to match a query object against the objects in the database in an accurate yet efficient manner. Our experimental results indicated the proposed approach performs at a level appropriate for real time live video processing.

To conclude this dissertation, the three techniques proposed in this dissertation detail the design and development of a complete database management system for live videos. *i*SEE establishes the basic platform to manage general sensor data over the Internet. LVDBMS builds another layer on top to facilitate query processing over a special yet powerful type of sensors, the smart cameras, and is then significantly enhanced with the addition of cross camera tracking capabilities. Beyond this dissertation, there are a myriad of new applications that can potentially

benefit from an LVDBMS, much like the way today's applications utilize general-purpose database management systems.

6. REFERENCES

[1] Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., & Zdonik, S. (2003). Aurora: A New Model and Architecture for Data Stream Management. In VLDB Journal, August.

[2] Adali, S., Candan, K.S., Chen, S., Erol, K., & Subrahmanian, V.S. (1996). Advanced video information systems: data structures and query processing. ACM Multimedia Systems, 4:172–186.

[3] Aggarwal, C. C., Han, J., Wang, J., & Yu, P. S. (2003). A Framework for Clustering Evolving Data Streams. In Proc. of the 29th International Conference on Very Large Data Bases (VLDB).

[4] Ahmedali, T. & Clark, J. J. (2006). Collaborative multi-camera surveillance with automated person detection. Canadian Conference on Computer and Robot Vision.

[5] Allen, J.F. (1983). Maintaining knowledge about temporal intervals. Communications of the ACM, 26(11):832-843.

[6] Arasu, A., Babcock, B., Babu, S., McAlister, J., and Widom, J. (2004). Characterizing Memory Requirements for Queries over Continuous Data Streams. In ACM Transactions on Database Systems, March.

[7] Ballard, D.H., & Brown, C.M. (1982). Computer Vision. Prentice Hall: New York.

[8] Cai, Q., & Aggarwal, J.K. (1999). Tracking human motion in structured environments using a distributed camera system, IEEE Trans. Pattern Anal. Mach. Intell. 2 (11) 1241 – 1247.

[9] Callaway, E. H. (2003). Wireless Sensor Networks: Architecture and Protocols. Auerbach Publications.

[10] Chang, S., Chen, W., Meng, H.J., Sundaram, H., & Zhong, D. (1997). VideoQ: An automated content-based video search system using visual cues. Proc. of ACM Multimedia, Seattle, Washington, USA, pp. 313–324.

[11] Chang, T-H., & Gong, S. (2001). Tracking multiple people with amulti-camera system, in: IEEE Workshop on Multi-Object Tracking.

[12] Chu, X., and Buyya, R. (2007). Service Oriented Sensor Web. Sensor Networks and Configuration, 51 - 74.

[13] Collins, R.T., Lipton, A.J., Fujiyoshi, & H., Kanade, T. (2001). Algorithms for cooperative multisensor surveillance, Proc. IEEE 89 (10) 1456 – 1477.

[14] Delin, K. A. & Jackson, S. P. (2001). The Sensor Web: A New Instrument Concept. In SPIE Symposium on Integrated Optics, San Jose, CA, January.

[15] Demers, A., Gehrke, J. E., Rajaraman, R., Trigoni, N., & Yao, Y. (2003). The Cougar Project: A Work-In-Progress Report. In SIGMOD Record, Volume 34, Number 4, December.

[16] Diegel, O., Bright, G., & Potgieter, J. (2004). Bluetooth Ubiquitous Networks: seamlessly integrating humans and machines. Assembly Automation, ISSN: 0144-5154, Volume 24, issue 2, pp. 168~176.

[17] Dockstader, S.L., & Tekalp, A.M. (2001). Multiple camera fusion for multiobject tracking, in :IEEE Workshop on Multi-Object Tracking.

[18] Donderler, M.E., Saykol, E., Ulusoy, O., & Gudukbay, U. (2003). BilVideo: A video database management system" IEEE Multimedia, 1(10):66–70.

[19] Donderler, M.E., Ulusoy, O., & Gudukbay, U. (2002). A rule-based video database system architecture. Information Sciences, 143(1–4):13–45.

[20] Egenhofer, M. & Franzosa, R. (1991). Point-set spatial relations. International Journal of Geographical Information Systems, 5(2):161–174.

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

[22] Ganesan, D., Estrin, D., & Heidemann, J. (2002). Dimensions: Why do we need a new data handling architecture for sensor networks? HotNets-1.

[23] Gaynor, M., Moulton, S. L., Welsh, M., LaCombe, E., Rowan, A., Wynne, J. (2004). Integrating Wireless Sensor Networks with the Grid. IEEE Internet Computing, vol. 8, no. 4, pp. 32-39, July/Aug., doi:10.1109/MIC.2004.18.

[24] Gibbons, P. B., Karp, B., Ke, Y., Nath, S., & Seshan, S. (2003). IrisNet: An Architecture for a World-Wide Sensor Web. In IEEE Pervasive Computing, Volume 2, Number 4.

[25] Guting, R.H., Bohlen, M.H., Erwig, M., Jensen, C.S., Lorentzos, N.A., Schneider, M., & Vazirgiannis, M. (2000). A foundation for representing and querying moving objects. ACM Transactions on Database Systems, 25(1):1–42.

[26] Intanagonwiwat, C., Govindan, R., & Estrin, D. (2000). Directed diffusion: A scalable and robust communication paradigm for sensor networks. In Proc. of 6th ACM/IEEE Mobicom Conference.

[27] Hamblin, C.L. (1972). Instants and intervals. Proc. of the 1st Conf. of the Intl. Society for the Study of Time, JT Fraser, Springer, Berlin Heidelberg, pp. 325-331.

[28] Hampapur, A., Brown, L., Connell, J., Ekin, A., Haas, N., Lu, M., Merkl, H., Pankanti, S., Senior, A., Shu, C.-F., & Tian, Y.L. (2005). Smart Video Surveillance, Exploring the concept of multi-scale spatiotemporal tracking. IEEE Signal Processing Magazine, March.

[29] Hengstler, S. & Aghajan, H. (2006). A Smart Camera Mote Architecture for Distributed Intelligent Surveillance ACM SenSys Workshop on Distributed Smart Cameras (DSC), Oct.

[30] Hua, K. A., Yu, N., & Liu, D. (2006). Query Decomposition: A Multiple Neighborhood Approach to Relevance Feedback Processing in Content-based Image Retrieval. Proceedings of the 22nd International Conference on Data Engineering.

[31] Huang, T., & Russell, S. (1997). Object identification in a bayesian context, in: Proceedings of IJCAI.

[32] Jain, R., & Wakimoto, K. (1995). Multiple perspective interactive video, in: IEEE International Conference on Multimedia Computing and Systems.

[33] Jiang, H., Montesi, D., & Elmagarmid, A.K. (1997). VideoText database systems. In Proc. of IEEE Multimedia Computing and Systems, pp. 344–351.

[34] Kang, J., Cohen, I., & Medioni, G. (2003) Continuous tracking within and across camera streams, in: IEEE Conf. Comput. Vision Pattern Recognition.

[35] Kauth, R.J., Pentland, A.P., & Thomas, G.S. (1997). Blob: an unsupervised clustering approach to spatial preprocessing of mass imagery. 11th Int'l Symposium on Remote Sensing of the Environment, Ann Arbor, MI, 22(8): 831-843.

[36] Kettnaker, V., & Zabih, R. (1999). Bayesian multi-camera surveillance, in: IEEE Conf. Comput. Vision Pattern Recognition, pp. 117 – 123.

[37] Kim, K. & Davis, L.S. (2006). Multi-camera Tracking and Segmentation of Occluded People on Ground Plane Using Search-Guided Particle Filtering. Proceedings of the 8th European Conference on Computer Vision.

[38] Kleihorst, R., Schueler, B., & Danilin, A. (2007). "Architecture and Applications of wireless Smart Cameras (Networks)," Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on , vol.4, no., pp.IV-1373-IV-1376, 15-20 April.

[39] Koh, J., Lee, C., & Chen, A.L.P. (1999). Semantic video model for content-based retrieval. Proc. of IEEE Multimedia Computing and Systems, vol. 1, pp. 472–478.

[40] Kuo, T.C.T. & Chen, A.L.P. (2000). Content-based query processing for video databases. IEEE Transactions on Multimedia, 2(1):1–13.

[41] Lee, L., Romano, R., & Stein, G. (2000). Monitoring activities from multiple video streams: establishing a common coordinate frame, IEEE Trans. Pattern Recogn. Mach. Intell. 22 (8) 758 – 768.

[42] Li, J.Z., Ozsu, M.T., Szafron, D., & Oria, V. (1997). MOQL: A multimedia object query language. Proc. of the 3rd Int. Workshop on Multimedia Information Systems, Como, Italy, pp. 19–28.

[43] Lieb, D., Lookingbill, A., Stavens, D., & Thrun, S. (2004). Tracking Multiple Moving Objects from an Autonomous Helicopter. http://robots.stanford.edu/cs223b04/project_reports/P2.pdf

[44] Lucas, B. D. & Kanade, T. (1981). An iterative image registration technique with an application to stereo vision. Proceedings of Imaging understanding workshop, pp 121-130.

[45] Marcus, S. & Subrahmanian, V.S. (1996). Foundations of multimedia information systems. Journal of ACM, 43(3):474–523.

[46] Madden, S. & Franklin, M. J. (2002a). Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In International Conference on Data Engineering (ICDE).

[47] Madden, S., Franklin, M. J., Hellerstein, J. M., & Hong, W. (2002b). TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. OSDI.

[48] Makris, D., Ellis, T.J., & Black, J.K. (2004). Bridging the gaps between cameras, in: IEEE Conf. on Computer Vision and Pattern Recognition.

[49] Mehrotra, S., Chakrabarti, K., Ortega, M., Rui, Y., & Huang, T.S. (1997). Multimedia analysis and retrieval system (MARS project). Proc. of the 3rd Int. Workshop on Information Retrieval Systems, Como, Italy, pp. 39–45.

[50] Mittal, A., & Davis, L.S. (2003). M2 tracker: a multi-view approach to segmenting and tracking people in a cluttered scene, Int. J. Comput. Vis. 51 (3) 189 – 203.

[51] Oh, J. & Hua, K. A. (2000). Efficient and Cost-effective Techniques for Browsing and Indexing Large Video Databases. SIGMOD Conference: 415-426.

[52] Olston, C., Jiang, J., & Widom, J. (2003). Adaptive Filters for Continuous Queries over Distributed Data Streams. In Proc. of the ACM Intl Conf. on Management of Data (SIGMOD).

[53] Oomoto, E. & Tanaka, K. (1993). OVID: Design and implementation of a video object database system. IEEE Transactions on Knowledge and Data Engineering, 5:629–643.

[54] Peng, R., Aved, A. J., & Hua, K. A. (2010). Real-Time Query Processing on Live Videos in Networks of Distributed Cameras. International Journal of Interdisciplinary Telecommunications and Networking, 2(1): 27-48.

[55] Porikli, F. (2003). Inter-camera color calibration using cross-correlation model function, in: IEEE Int. Conf. on Image Processing, 2003.

[56] Schulzrinne, H., Rao, A., & Lanphier, R. (1998). Real Time Streaming Protocol (RTSP). Network Working Group RFC 2326.

[57] Schulzrinne, H., Casner, S., Frederick, R., & Jacobson, V. (1996). RTP: A Transport Protocol for Real-Time Applications. Network Working Group RFC 1889.

[58] Symonds, J., Parry, D., & Briggs, J. (2007). An RFID-based system for Assisted Living: Challenges and Solutions. The International Council on Medical and Care Compunetics Event, June 8-10, Novotel Amsterdam, The Netherlands.

[59] Tan, H. O., Korpeoglu, I., Stojmenovic, I. (2007). A Distributed and Dynamic Data Gathering Protocol for Sensor Networks. IEEE 21st International Conference on Advanced Information Networking and Applications (AINA-07), Niagara Falls, Canada, May 21-23.

[60] Thrun, S. (2002). Particle Filters in Robotics. In Proceedings of the 17th Annual Conference on Uncertainty in AI (UAI).

[61] Universal Description, Discovery and Integration of Web Services, http://www.uddi.org.

[62] W3C Simple Object Access Protocol (SOAP), http://www.w3.org/tr/SOAP.