MavVStream: Expressing and Processing Situations on Videos Using the Stream Processing Paradigm

by

MAYUR ARORA

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2018

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr. Sharma Chakravarthy for providing me an opportunity to work on this project and constantly motivating me and guiding me throughout this thesis work. Without his constant support and guidance this research work would not have been successful. I would like to thank Dr. Vassilis Athitsos for his guidance throughout this research work and taking time to serve in my thesis committee. I would also like to thank Dr. Ramez Elmasri for taking time to serve in my thesis committee.

I would like to extend my gratitude to all the people involved in this project, IT - Lab present members and alumni Abhishek Santra, Manish Kumar Annappa, Nandan Prakash, Soumyava Das and Damini Singh for their constant support and guidance. Also, I would like to thank my family and friends for their constant support and encouragement throughout my academic career.

April 13, 2018

ABSTRACT

MavVStream: Expressing and Processing Situations on Videos Using the Stream Processing Paradigm

MAYUR ARORA, M.S.

The University of Texas at Arlington, 2018

Supervising Professor: Dr. Sharma Chakravarthy

Image and Video Analysis (IVA) has been ongoing for several decades and has come up with impressive techniques for object identification, re-identification, activity detection etc. A large number of techniques have been developed and used for processing video frames to detect objects and situations from videos. Camera angles, lighting effect, color differences, and attire make it difficult to analyze videos. Several approaches for searching, and querying videos and images have been developed using indexing and other techniques. This thesis takes a novel approach by converting a video (through extraction of its contents) into a representation over which queries can be specified. This is similar to querying a relational database but on contents extracted from a video and represented using a richer data model. This thesis focuses on new operators needed and their relevance to express real-world situations, such as Alert if the same person came through the same door n times within an hour. model that is needed for representing extracted video contents. The long-term goal is to significantly augment the image and video analysis capabilities for querying.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

There are independent bodies of research in **Image and Video Analysis(IVA)** and **stream processing (SP)**,mainly applied to sensor data streams. Sensors have allowed us to capture numerical (and structured) data on a 24/7 basis. Now, thanks to the availability of Dash/personal cams, mobile phones, unmanned aerial vehicles (or UAVs), and Google glasses, one can generate continuous, large amounts of video for surveillance, security or personal interest. This has brought about a requirement which is different from traditional image and video analysis [1] which dealt with small amounts of videos, developed customized algorithms for identifying various features, objects, events, and actions, and were primarily analyzed for forensics or postmortem analysis. Analysis of video stream contents *for situation analysis* is critical for understanding/checking events in a video stream (e.g., person enters/leaves a building), spatial and temporal activities that can be inferred from a video (e.g., patterns of car movement in a parking lot, duration of stay by an individual in a facility), and complex activities such as exchange of objects by people crossing each other, pickup of a person by a vehicle etc. Typically, video surveillance is done manually by watching the video frames continuously for identifying the above indicated kinds of events and anomalies. The manual approach is resource-intensive (especially human resource) and is also prone to errors. Some applications, such as security surveillance for home or commercial places like malls, parking lots etc. may tolerate and can accept some degree of errors or approximations but mission critical applications (e.g., target identification) expect higher accuracy.

Research on Image and Video Analysis [1] has given us a large number of techniques – pattern recognition, machine/deep learning, and artificial neural network-based – that have been developed for processing image or video to characterize objects captured with different camera angles, lighting effects, and color differences. Techniques have been developed for object identification and re-identification. Feature extraction, segmentation, and separation of background from foreground use numerous algorithms and machine learning techniques to deal with specific application domains/contexts. Based on our understanding of this, we suggest that stream processing(SP) can be effectively adapted for analyzing videos in a different way than it has been done currently. To the best of our knowledge, stream processing technologies have not been leveraged or exploited for video analysis. Complex stream and event processing techniques and techniques developed for satisfying QoS requirements of real-time applications are required for analysis in this domain.

Video analysis can be viewed as an analysis of relevant contents which are extracted from processing of individual video frames of a stream. To achieve this, there is a need to model the extracted content in a manner such that expressive queries can be posed and evaluated. New data types such as categorical and complex ones such as multi-dimensional vectors are needed to represent histograms, feature vectors, bounding boxes etc. which are extracted from video contents. Also new operators are required, which use these new data types to express meaningful queries specific to the video domain. Query expressiveness depends both on the amount of information that can be extracted as well as the richness of the data model used for representing extracted contents. The ability of query processing to provide accurate results depends on the accuracy of information extracted from the video streams (i.e., on preprocessing). Also, extraction of relevant video (or frame) contents are

resource-intensive and requires computations that are very different from those used in traditional systems (e.g., creation of a feature vector).

1.1 Focus and Contribution of this Thesis

Video can be viewed as a stream of video frames. It can be viewed as unbounded if it is an application that generates video continuously and requires real time monitoring. Otherwise, it can be viewed as a bounded (albeit large at 30/60 frames per second). Our focus in this thesis is to be able to express and process a large class of queries on such video streams. As mentioned above, there is a requirement of new data types as well as new spatial and temporal operators that work specifically on video stream data for expressing situations and querying video streams as needed. The work in this thesis proposes the design and implementation of few data types and several operators and how they can be used to express situations as queries, and further evaluate those queries on extracted video stream data using stream data processing approach. As we already have a stream processing system (MavEStream) developed at the IT Laboratory, UT Arlington, we plan on extending it to include new data types and operators.

As an outcome of this thesis, we can query situations like "*How many people entered the building every hour ?*" or "*Find out if a person entered also exited the building.*" and process them on the video streams and compute results.

The main contributions of this thesis are -

1. A non-traditional but well-defined approach for Video analysis
2. Extension of existing data model for stream processing to represent video stream contents
3. Identifying the limitations of current window support to process video streams and needed extensions

4. Design and implementation of new operators and their semantics for expressing a wide range of queries
5. Implementation of extensions proposed in a prototype
6. Extraction of non-primitive content using event composition

1.2 Thesis Organization

Rest of the thesis is organized as follows -

- Chapter 2 discusses the work done related to this field of thesis
- Chapter 3 discusses about how every type of situation in a video stream can be classified into various query classes
- Chapter 4 elaborates the preliminary work done in this field
- Chapter 5 describes the detailed design of the data model extensions and new operators with sample examples.
- Chapter 6 elaborates the algorithms used and detailed implementation of the data model changes and new operators
- Chapter 7 contains all the experiments that are performed as part of this thesis and results obtained for each
- Chapter 8 includes Conclusion and future work for this thesis

CHAPTER 2

RELATED WORK

The amount of work done in the literature on Querying/Processing Video Stream data is overwhelming as it spans image processing, pattern recognition, and storage, management, and processing of extracted video contents. The NSF report [1] of the workshop held in 2014 provides a scientific assessment of the current state-of-the-art in Image and Video Analysis including what are considered as solved problems, what are considered as problems that require near term and those that require long term investments.

2.1 VIRAT (Video and Image Retrieval and Analysis Tool)

VIRAT is a video surveillance project funded by the Information Processing Technology Office (IPTO) of the Defense Advanced Research Projects Agency (DARPA) [2]. The goal of this project is to develop a software that goes through huge video datasets and alerts if a particular event of interest occurs thus reducing the burden on military operators who either manually have to go through the video contents using fast-forward approach or search using already assigned meta-data tags (through off line processing). An example of such interesting event is "find all of the footage where three or more people are standing together in a group" [3].

Following are the major categories of the events detected by VIRAT [2]:

1. *Single Person:* Digging, loitering, picking up, throwing, exploding/burning, carrying, shooting, launching, walking, limping, running, kicking, smoking, gesturing

2. *Person-to-Person:* Following, meeting, gathering, moving as a group, dispersing, shaking hands, kissing, exchanging objects, kicking, carrying an object together
3. *Person-to-Vehicle:* Driving, getting-in (out), loading (unloading), opening (closing) trunk, crawling under car, breaking window, shooting/launching, exploding/burning, dropping off, picking up
4. *Person-to-Facility:* Entering (exiting), standing, waiting at checkpoint, evading checkpoint, climbing atop, passing through gate, dropping off
5. *Vehicle:* Accelerating (decelerating), turning, stopping, overtaking/passing, exploding/burning, discharging, shooting, moving together, forming into convoys, maintaining distance
6. *Other:* convoy, parade, receiving line, troop formation, speaking to crowds, riding/leading animal, bicycling

VIRAT architecture is mainly based on indexed Data Store of video clips containing the actions to be detected that are similar to the once indicated above. VIRAT trains action detection modules with these archived video data and tries to detect actions listed above in the target video stream. In summary, VIRAT has focused on customized software to detect a large set of predetermined set of interest to the project. It is not a general-purpose approach where the indexed information can be used to process arbitrary queries. Finally, it is an off-line approach to understand the contents of a video.

The work in this thesis is different than that of VIRAT. The focus here is to express and process some classes of situational queries on videos in a generic way rather than develop customized algorithms for each event detection. Also the work in this thesis can work for diverse objects (as long as they can be extracted

during preprocessing) while VIRAT is mainly limited to customized approaches for identification of specific (e.g., human) activity in the videos.

2.2 Live Video Database Management System (LVDBMS)

The components of LVDBMS can logically be grouped into four tiers, the lowest consisting of physical hardware (camera layer). Next is the spatial processing layer, then the stream processing layer and finally the client layer. Queries originate in the client layer and are pushed down to the stream processing layer and then to the spatial processing layer. Data originates in the camera layer and flows upwards. As it moves upwards it is transformed; from a stream of imagery in the lower layer to streams of sub-query evaluations to a stream of Boolean query evaluations [4, 5].

The main intent of LVDMBS is to match objects in one video stream with objects in another stream using bipartite graph comparison. LVDBMS uses a query language called Live Video Query Language (LVQL) to express complex events formed by simple events connected by spatial or temporal operators. For example, a simple event could be a person (or more generally some object) appearing in a scene or moving in front of a desk (where the term scene refers to some portion of the real world that is observed by a camera and rendered into a sequence of frames in a video stream). For example, a complex event could be defined as a person first appearing in a scene and then, within some threshold of time, moving in front of a desk [4].

LVQL supports spatial operators like "Appear()", "North()", "Inside()", "Meet()" etc., temporal operators like "Before()", "Meets()" etc. and traditional Boolean operators like "and", "nor" etc. Additionally, LVDBMS supports the following three types of objects:

1. Static Objects: These objects are not automatically detected, but marked in the video streams by users using the LVDBMS GUI while forming LVQL queries.

They are typically denoted as Appear(s1.12, 50), where s1.12 is the ID assigned to the marked static object and the Appear() operator will return true only if the marked static object s1.12 has bounding box more than 200 pixels.

2. Dynamic Objects: These type of objects are programmed to be automatically detected in LVDBMS and are denoted by an asterisk (* e.g. Appear(s1.*, 200)). Appear() operator will return true if there are any dynamic objects in the video stream that have bounding box more than 200 pixels.
3. Cross-Camera Dynamic Objects: These are the dynamic objects in one stream that are matched with another stream. They are generally denoted by #. For example, Before(Appear(c0.#),Appear(c1.#),60) returns true if an object in stream1 matches an object in stream2 after a time gap of at least 60 seconds.

LVDBMS differs from the work in this thesis, in the type of queries targeted, operators, type of data extracted etc. Following comparison highlights the differences between LVDBMS and the work in this thesis.

1. LVDBMS uses low-level query language LVQL and expects the users to build applications using LVQL to express situations in high-level query languages like CQL,SQL etc. Whereas, the work in this thesis aims to use an extracted representation from video streams using which users are able to express situations using common non-procedural languages like CQL,SQL, without needing to build any application. Appropriate operators for this domain to CQL have been added as part of this thesis which provide clear semantics and can also be used in conjunction with other operators.
2. Most of the operators in LVQL like Appear(), North() etc return only the Boolean values limiting to queries that result in Boolean values using such operators. The work in this thesis aims at a query language that is closed with

respect to the representation allowing composition of any operator with any other available operator and includes aggregation and ordering as well.

3. LVDBMS does not facilitate queries that require tracking of unique dynamic objects within a single stream. Currently proposed operators (Appear()) in LVQL treat all the dynamic objects as same. Appear() translates to if any dynamic object has satisfied the given threshold or not. If unique objects have to be tracked using Appear(), the object to be tracked has to be marked manually using LVQL GUI before the query is executed. Whereas the work in this thesis aims to answer queries related to tracking unique objects within or across streams dynamically.
4. Finally, usage of custom operators introduces a significant learning curve for users to learn the new semantics. Whereas, the work in this thesis uses the standard semantics in CQL and/or SQL.

2.3 MedSMan: A Streaming Data Management System over Live Multimedia

MedSMan [6] presents a system that enables direct capture of media streams from various sensors and automatically generates meaningful feature streams that can be queried by a data stream processor.

MedSMan aims to answer queries like "Display the speakers video when he is answering the 4th question about multimedia" in a broadcast seminar event. MedSMan makes use of the multimedia streams from various devices such as presentation projector, speaker's microphone, video device capturing the podium, movement sensors etc. MedSMan uses the information from these Media Streams to inform users (who have subscribed to this broadcast event) when an event such as the above query occurs.

MedSMan lets the users create definitions of Media Streams at the Stream Processing layer via Media Stream Description Language (MSDL). A Media Stream is usually the output of a sensor device such as a video, audio, or motion sensor that produces a continuous or discrete signal, but typically cannot be directly used by a data stream processor. MedsMan continuously extracts the content-based data descriptors called features using Feature Generation Functions (FGFs) (defined in Feature Stream Description Language or FSDL from the Media Streams to create Feature Streams. These Feature Streams are used by the stream processors to query and return the corresponding segment from the Media Stream as the result. MedSMan uses custom query language called Media and Feature Stream Continuous Query Language (MF-CQL) to continuously query over live media. MF-CQL is CQL [7] extended with additional syntax and shortcuts to express the extended semantics beyond DSMS.

Although MedSMan is focused on answering situational queries with the aid of various sensor data, it can be used to answer queries on live video streams using MF-CQL query language. Following comparison highlights the difference between MedSMan and the work in this thesis.

1. MedSMan expects the user to define (schema) media streams by specifying various attributes that need to be extracted from sensors (including video capturing device). The work in this thesis uses the automatically generated data representation or schema through preprocessing that is appropriate to express simple situational queries on live video streams.
2. MedSMan expects the users to use FSDL to process the media streams to generate Feature Streams. If a particular method to accomplish a task (for example, track unique moving objects in a video stream) that processes Media Stream is not available in MedSMan, then it expects the user to define a new Feature

Generation Function (FGF) to achieve this task pragmatically. Whereas, the work in thesis hides the processing of queries on the video streams from users and users do not need to develop any new methods to express simple situations.

2.4 **Aurora**

Aurora [8–11] is a system for managing data streams for monitoring applications . Continuous queries in Aurora can be specified in terms of a data flow diagram consisting of boxes and arrows (using a GUI). It uses SQuAL (Stream Query ALgebra) consisting of several primitive operators for expressing stream processing requirements. Aurora supports continuous queries, views, and *ad hoc* queries using the same conceptual building blocks.

However, Aurora is only limited for sensor streams and does not focuses on using stream processing for video analysis. The work in this thesis focuses on using stream processing on preprocessed video streams to express situational queries.

In summary, the proposed work, of which this thesis is a part of , is different from the relevant work in the literature. Our goal is to make querying of videos as general-purpose as possible and remove the learning curve present in most other approaches. This approach not only makes it easy to automate, but also has the potential of adding real-time functionality to the framework (as has been done for sensor applications using stream processing). Furthermore, the use of CQL-like language affords optimizations by the system that are typically not possible or has to be done by the developed which is not ideal.

CHAPTER 3

SITUATIONS AS QUERY CLASSES

Queries are typically expressed over numeric and categorical data to filter, sort, group, and aggregate in various ways. SQL and CQL uses attributes and tuples to store information and querying uses this representation for various operators. CQL also uses windows and other mechanisms (e.g., partitioning within a window) for expressing and processing continuous queries. Hence, if one wants to query a video stream, it is important to represent it in a model that is amenable to query processing. Further, queries can match (using join), use attributes values and ranges to filter tuples, and perform aggregations. It turns out that these types of queries are needed for monitoring and surveillance where one is interested in counting objects entered/exited and computing the duration an object (or related objects) stayed in a building/structure. Queries on situations in a video can be classified based on the type of computation used in a query and whether they are directly available in the extracted model or need to be computed as part of query processing. Below, we elaborate on query classes. Query Classes correspond to the type of queries and the information used from the data model. They also indicate the complexity of computation associated with the query. Refer to Section 6. for details on how to express these queries with our current approach. These query classes may even indicate the limitations of the data model currently used as well as the expressiveness of the query languages.

Here we have classified situations into three different types of query classes :

1. **Situations - Set 1** - These situation queries are of aggregation types where information extracted from the video can be directly aggregated based on temporal or other dimensions. Few examples of such situation query sets are -
   (a) How many people/cars enter the building/parking lot every hour on the hour or between a specified interval or on a particular day?
   (b) Identify the slowest/busiest hour of the day (in terms of number of people entering).
   (c) List people/cars who stayed for less than an hour in the building/parking lot.
   (d) List the average duration a person/car stayed at the property on Mondays/weekends.
   (e) Indicate when 2 different people (images given) enter **and** leave the building/parking lot within n minutes of each other.
   (f) Indicate when a specific person (whose image is given) entered the building.
   (g) Did two individuals (images given) enter **or** exit the building within 5 minutes of each other?
2. **Situations - Set 2** - This situation set includes queries where an event or an action need to be computed using the data extracted and then use computations used in Set 1. A Few examples of such queries are -
   (a) Did a person go up to the check post/entrance but turn around (i.e. did not cross/enter)?
   (b) Did two people bump into each other?
   (c) Did a person carry an object for a duration?
   (d) Was a person picked up by a vehicle?
   (e) Did a specific person cross the bridge?

Such types of queries require identification and extraction of event/action such as walking, turning around, carrying an object, getting rid of an object etc.

3. **Situations - Set 3** - These type of queries are more complex in nature and cannot be expressed on the current data model. These queries involve recognizing complex events such as low tide, specific activities in a specific location or certain object/background detection. Few examples of such situation query sets are -
   (a) Identify a number of known people in the protest march video.
   (b) Was the same person picked up from two different places in two videos?
   (c) Identify video segments where two or more people are running out of a house.

   It is evident from the above mentioned situation classes and query examples that these types of queries are not only complex to formulate but achieving a good accuracy from preprocessing on the same is also very difficult.

Traditional image processing has not tried to address the class of queries in the first set. The second set of queries have been addressed in a custom manner VIRAT (`https://en.wikipedia.org/wiki/VIRAT`) being a good example along with many other research prototypes) using techniques that are difficult to generalize to other situations. More recently, third set of queries have been addressed again in a customized manner.

In this thesis, Section 6. addresses processing of queries under Situations - Set 1. It is also indicated how we can proceed towards processing Situations - Set 2 using complex event processing for detection of some of the actions needed. We believe that queries in Situations - Set 3 cannot be currently addressed using our proposed approach and we contend that it cannot be addressed using the current approaches in IVA research as well!

CHAPTER 4

PRELIMINARY WORK

There is preliminary work both for preprocessing videos and for stream data processing. We are building upon these to extend our system for video situation analysis. In this chapter we briefly summarize the previous work done and used for this thesis.

Video querying involves identification of objects of interest, extracting important features from each of the video frames so that this information can be converted into a queryable format. In order to achieve the same, we need a video preprocessing system to extract the content from the video stream and pass this content to a stream processing system (*MavVStream in our case*) to process and get meaningful results. Figure 4.1 shows the overall work flow . In this section, we discuss the two preliminary steps needed for design and implementation of the focused portions of this thesis.

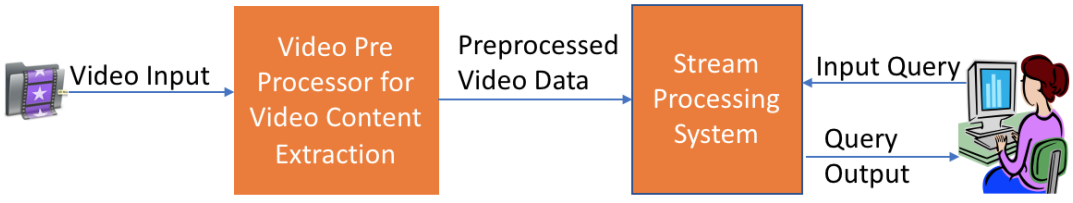


Figure 4.1: Work flow for Analyzing queries on Video Streams

4.1 Video Content Extraction

A typical video contains multiple frames per second. These frames contains objects of interests and these objects can occur across multiple frames. So content extraction in videos is done by processing frames. As our current focus is on video streams for tracking objects that are moving, we have concentrated mainly on the identification of moving objects from a video stream and using that preprocessed data as our input for stream processing. Background information is not always extracted as it is not needed but can be extracted if required for preprocessing. Information extraction for background is difficult to achieve as the object detection is based on movement. Hence, it is very important to determine what content needs to be extracted based on the query classes to be processed.

Video Content Extraction can be divided into two parts :

4.1.1 Primitive Content Detection

Primitive Content describes the content that is needed to be extracted from a video frame such as objects, object types, bounding boxes, feature vectors etc. All these can be extracted with the current IVA techniques. . Our preliminary work has focused on moving objects as a) we are dealing with video rather than an image and b) identification of objects and eliminating the background is easier than identifying static objects. Object identification and re-identification can use used for identifying a car entering a parking lot, or a person entering a building and exiting from a di erent location. Each object from each frame is represented as a tuple in our data model (similar to tuples in a relational model.)

Once individual objects are extracted from a frame, the next step is object labeling or object re-identification where we want to re-identify the *same object* across frames. Typically, a moving object (e.g., person, car) is present in several frames

depending on the field of view, camera angle, and the pace of the moving object. Frame differencing works reasonably well for moving objects. Re-identification of moving objects can pose some problems depending on lighting conditions, posture, overlap with other objects, and occlusion. Feature vector choices and many techniques are available in IVA for moving object re-identification. If the number of unique objects in a video is not very large (tens and not thousands), it is possible to re-identify an object even if it appears again after a long time. If the number of unique objects in a video are very large, some temporal/physical limits may be imposed on object re-identification.

To summarize, we will assume that during the *preprocessing phase*, the following content are extracted as a minimum from a video:

a. Individual moving objects and their identification in each frame assuming a small set of object types (e.g., people, vehicle.)

b. Object bounding boxes and feature vectors (parameterized as needed).

c. Re-identification of objects in the video. If the video is very long, some limits may have to be imposed on the re-identification duration.

d. Any static object (e.g., background objects) that can be extracted and identified.

As part of preliminary work, preprocessing of videos [12] have been achieved either using Matlab. Different approaches have been used for identification of different objects. For example, for the identification and matching of people in a video, RGB Histogram feature vectors have been used as they have proved to be more accurate, and for identification and matching of cars or vehicles in a video, Scale Invariant Feature Transform (*SIFT*) feature vectors have been used as they achieve better accuracy. Traditional video preprocessing does not output a relational representation that is needed for this work. It is not difficult to use any of those preprocessing approaches and generate the tuple representations that is needed for this work.

4.2 MavVStream

MavEStream (**Mav**erick **E**vent **Stream** Processing System) is a data stream processing system developed at IT Lab - University of Texas at Arlington and is implemented in Java. It is developed for processing continuous queries over streams. MavEStream is modeled as a client-server architecture in which client accepts input from the user, transforms it into a form understood by a server (XML format is currently used) and sends the processed input to the server based on predefined protocols. MavEStream is a complete system wherein a query, submitted by the user, is processed at the server and the output is returned back to the application or written to a file. This data model of this system as well as the operators needed and their processing for querying videos will be extended. This extended system is termed MavStream (or Maverick Video stream Processing System.) The various components of MavVStream are shown in Figure 4.2. Feeder is used to read the streaming data from a file or as objects. In this thesis, preprocessed video stream data (described in Section 4) is input to the server using the feeder. The MavVStream server, upon receiving the query from the client, constructs the query plan object for that query. Query plan object is a tree of objects that contain information about all the operators in a query. The instantiator module is used to instantiate all the operators, paths and segments to execute the query. The root operator of the query returns the outputs of the query to the application or generates an event if specified.

4.2.1 Operators

Query processing in a traditional DBMS is not designed to produce real-time response to queries over high volume, continuous, and time varying data streams. The processing requirements of real time data streams are different from traditional applications and demand a re-examination of the design of conventional operators

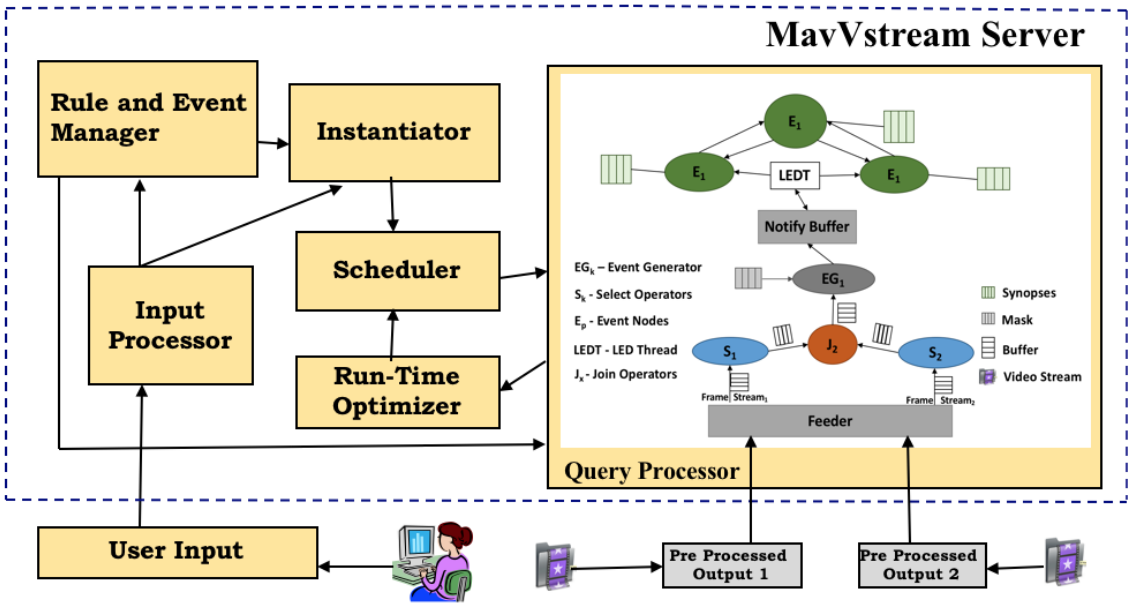


Figure 4.2: MavVStream Architecture

for handling long running queries to produce results continuously and incrementally. Blocking operators (an operator is said to be blocking if it cannot produce output unless all the inputs are available) such as aggregates, sorting may block forever on their input as streams are potentially unbounded. Hence, MavVStream uses a window concept to convert the computation of blocking operator into a non-blocking one. Tuples are processed by these blocking operators until a window is elapsed and output is produced. Each operator is associated with input queues based on the number of operands needed and outputs the results to a queue. The operators are implemented as individual threads which are scheduled by the scheduler using one of the available scheduling algorithms (e.g., round robin, weighted round robin). The operators are suspended by the scheduler thread when the time quantum is elapsed or when they do not have any more tuples to process.

There are various operators that have been implemented in MavEStream system as part of Preliminary work. They are Select, Project, Join, Group By and

Aggregate (sum, count, average, maximum and minimum.) Both physical and logical window types are supported for the whole query. These operators are described below briefly. This thesis extends those operators to process video stream input data and also proposes new ones (blocking as well as non blocking) such as Arrable, Compress Consecutive Tuples (CCT) , and Consecutive Join (cJoin). It also adds a similarity match (sMatch) as an operator to be used in where and join conditions. Further, it proposes a spatial aggregate function (direction) to compute the direction of movement of an object. This can be used to generate an event which can be further composed to infer (or detect) activities such as left turn, u turn, or walking straight. These extensions are specifically designed for video stream input data, which may contain data types and computations that are different from the data types and computations used on traditional sensor stream data. Detailed explanation of these additions is described in Section 5.2.

4.2.1.1 **Select Operator**

SELECT operator is similar to the FILTER (SELECT) operator of RDBMS. The primary function of the select operator is to filter a given input stream based on the specified condition. It has one input queue, and can have more than one output queue. Before starting the operation, it assumes that the condition to be evaluated has been set by the query Instantiator based on user input. SELECT takes in a tuple from the input queue and checks whether it satisfies the given condition. If the condition evaluation is successful then the tuple is output to all the output queues associated with SELECT. If condition evaluation fails then the tuple is discarded and the next tuple in the queue is processed. SELECT checks for the "endQuery" condition and stops its execution if the current tuple has passed the end query condition.

4.2.1.2 **Project Operator**

PROJECT is analogous to the PROJECT operation of a traditional DBMS. It takes in a tuple and discards the specified fields in the form a new tuple. It has one output queue. One of the inputs to project is the position of list of attributes (*passed as data stream definition object for each abstract buffer class*) that needs to be projected out from the tuple.

4.2.1.3 **Join Operator**

Join operates on two streams with multiple tuples on each of its input queues. Join takes a simple or complex join condition in conjunctive normal form (CNF.) With the extension proposed, sMatch can be used as part of the join condition.

There are two types of Join currently supported by MavVStream :

- Hash Join - Mainly used for Equality conditions
- Nested Join - Used for all other cases

In this thesis, Nested Join has been extended to include similarity matching functions to match feature vectors which cannot be joined based on equality.

4.2.1.4 **Group By**

Group By operator groups tuples (with or without a window) based on specified attribute values. This grouping is done for the type of window specified one window at a time. We have two versions - one with aggregate specification and without. All the aggregate operations such as count, max, min, sum, and avg have been implemented using a group by operator in MavEStream.

4.2.2 Instantiator

Instantiator has the responsibility of initializing and instantiating streaming operators and their associated buffers using the query plan object that captures the operator sequence of a query. Clients query is converted into a plan object, which is a sequence of operator nodes represented as a operator tree, where every node describes an operator completely. The query plan object is used as input by the instantiator. Instantiator traverses the plan object in a bottom-up fashion and instantiates all the operators and associates buffers between them. Every operator is an independently executed operator and expects its parameters in a predefined form. Instantiator populates the operator instances with the parameters (e.g., conditions or attributes) defined in the query plan object. It also associates a scheduler with the operator to facilitate communication for scheduling. Instantiator does not start the operator and it only does the necessary initialization.

4.2.3 MavEStream Server

MavEStream server is a TCP Server which listens on a chosen port. It is responsible for converting a plan object into a query instantiation and executing it on the defined data streams to give the desired output. It consists of various modules such as Instantiator, operators, buffer manager and scheduler for executing continuous queries. It also allows new streams to be registered with the system. It also stops a query, which in turn stops all operators associated with the query on receiving command for query termination.

4.2.4 Feeder

Feeder essentially simulates arrival of input streams at a specified rates for processing by the MavEStream. Feeder is responsible for feeding tuples (given out

by the stream sources - preprocessed video stream in our case) to the buffers of leaf operators. Each stream is fed using a separate thread. Feeder thread reads the tuples from the secondary storage (a file) and feeds the tuples to buffers associated with leaf operators. Hence we use as input files which are generated by preprocessing a video stream or using a synthetic video data generator.

4.3 SQL and Limitations of OVER Clause

Stream processing introduced the notion of windows (both physical and logical [7, 13]) mainly to circumvent the blocking operators that would have posed problems for computations on unbounded data streams. Windows also made sense semantically for sensor and other stream data types due to their limited temporal validity. In addition to physical and logical windows, a number of other window types have been proposed (e.g., predicate-based windows [14].) The OVER clause was introduced into SQL 2003 for supporting queries with window- and order-based computation. Note that this is still on traditional Relational data representation.

Briefly, the `OVER` clause can be used in a SELECT clause to aggregate using a window specification. A window function in SQL performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row - the rows retain their separate identities.

A window function call always contains an OVER clause directly following the window function's name and argument(s). This is what syntactically distinguishes it from a regular or aggregate function. The OVER clause determines exactly how the rows of the query are split up for processing by the window function. The PARTITION BY list within OVER specifies dividing the rows into groups, or partitions,

that share the same values of the PARTITION BY expression(s). For each row, the window function is computed across the rows that fall into the same partition as the current row.

You can also control the order in which rows are processed by window functions using ORDER BY within OVER.

There is a notion of a frame which is defined with respect to the CURRENT ROW *as each row of the table is considered*. Several keywords can be used such as ROWS, RANGE, PRECEDING n, FOLLOWING n, PRECEDING UNBOUNDED etc. to formulate a frame. The best way to understand is that a *table* can be reduced into a *result set* (using the WHERE clause); each *result set* can be further *partitioned* using the PARTITION component within the OVER clause; further, *frames* can be defined over each *partition* using ROWS and RANGE expressions to perform aggregates.

The key word CURRENT ROW refers to the current row with respect to which a frame is defined. A restriction of the OVER clause is that it cannot be used with a GROUP BY clause. Also, joins are not possible over frames. Multiple tables can be used in the FROM clause as the OVER clause is applied to the result of the WHERE clause (i.e., theoretically, after the Cartesian product and restriction by the WHERE clause.)

$$VS2 \quad (Camera\_id, \underline{Frame\#}, \underline{Obj\_id}, Feature\_Vector\ as\ [FV], \\ Bounding\_box\ as\ [BB], Timestamp\ as\ ts) \tag{4.1}$$

Below we use the SQL (with OVER clause) in Oracle (version 10.2.0.4.0) to express some of the queries in **set 1** in Section 3 using the Table 5.1 for the entry

Camera_id	**Frame#**	**obj_id**	**[FV]**	**[BB]**	**ts**
1	1	1	[f1]	[0,0,10,12]	1
1	1	2	[f1a]	[bb1b]	1
1	1	3	[f1b]	[bb1c]	1
1	...	...	...	...	...
1	31	1	[f31a]	[bb31a]	21
1	31	2	[f31b]	[bb31b]	21
1	31	4	[f31c]	[bb31c]	21
1	...	...	...	...	...
1	61	2	[f61a]	[bb81a]	31
1	61	4	[f61b]	[bb61b]	31
1	61	5	[f61c]	[bb61c]	31
1	...	...	...	...	...
1	91	2	[f91a]	[bb91a]	41
1	91	4	[f91b]	[bb91b]	41
1	91	5	[f91c]	[bb91c]	41

Table 4.1: **VS2: Relation for the Exit Video Stream**

camera. Table 4.1 is assumed for an exit camera. These tables mimic the extracted contents of entry and exit videos (with bounding boxes and feature vectors.)

Importantly, it contains frame numbers (not all of them), time stamp (as integers), and object numbers in both streams.

The query "How many people entered the building every hour on the hour?" cannot be expressed as there is no time-based window supported in the `OVER` clause. It is not possible to convert this query into a tuple-based window query as the number of occurrences of objects in a frame is not known. However, the other query "How many people entered the building between a given interval (say 21 and 41 inclusive) can be expressed without using the `OVER` clause as:

```
SELECT MIN(ts), MAX(ts), COUNT(DISTINCT oid)
FROM   VS1 WHERE  ts >= 21 AND ts <= 41;
```

The above query is not suitable for video situation analysis as multiple, temporally disjoint appearances of the *same* object need to be identified as separate

occurrences. Currently, this is not possible in SQL or CQL. For example, object 1 is counted once although enters twice in that interval. Similarly, the query "Identify the slowest/busiest hour of the day (with respect to the number of people entering)" cannot be expressed as it needs the query mentioned above and choose the hour with the minimum or maximum number of people. The query "List people who stayed for less than an hour in the building/parking lot" entry and exit relations[1]. Although it is possible to join two computed relations each using window operators in SQL, to the best of our knowledge, it is not possible to perform joins on window frames generated for each relation (as only one can be used along with the OVER clause. This query, ideally, needs to check (match or join) each object entering (with a timestamp ts1) with all objects exiting with timestamp ts to ts+1hr as a window frame. This window frame cannot be specified using another relation window frame. An alternative is to cast this query in a different way. One can do a regular join and *then* check whether the time intervals were within the range expected. The following query does that.

```
SELECT VSE.ts as e_time, VSX.ts as x_time,
          VSE.Obj_id
FROM VS_entry VSE, VS_exit as VSX
WHERE sMatch(VSE.FV, VSX.FV, threshold) AND
          VSX.ts <= VSE.ts + 1 hr ;
```

In the above, the sMatch function (defined in 5.2.3 is used for comparing feature vectors to determine whether they match based on the similarity value between the two feature vectors being greater than equal to the threshold specified. As it is easy to notice, this has two problems: i) entire video is needed as input and ii) the above

[1]If a single camera is recording both entry and exit, it is necessary, as part of preprocessing, to identify objects as entering or exiting using an enumerated-type of attribute. In that case, a single relation can be used and by partitioning the relation on that attribute, this query can be expressed.

will perform join on *all pairs of tuples* (actually a Cartesian product) which is an over kill. This will not be meaningful for large videos or when the monitoring is continuous. Queries in Set 2 cannot be expressed at all as they require extraction and representation of actions (e.g., turning around) which is currently lacking.

Based on our attempt to express meaningful video situations, we have identified the following limitations of `SQL` even with the availability of the `OVER` clause:

- `OVER` does not support time-based windows.
- `Over` Frame specifications cannot move the upper and lower bounds arbitrarily .
- Although multiple OVER clauses can be specified on *different relations*, windowing clause is applied to the result of the FROM and WHERE clause, not for each relation*before* joining and applying the WHERE clause.
- Window-based joins are not allowed under the current specification of the OVER clause.
- Window offsets between joins cannot be specified currently.

4.4 **Summary**

In this chapter, we discussed about all the preliminary work on which this thesis relies on. It includes Video content extraction, which is referred as preprocessing of video streams and converting it into tuples . We also discussed about the architecture of MavEStream and its components. In the end we discussed the limitations of current SQL for specifying windows.

CHAPTER 5

DESIGN

In this chapter, we discuss the extension of the MavEStream into the MavVStream system in terms of the data model and operator extensions. But, before we proceed with that, first we need to understand the data model needed for video contents and why traditional relational representation is not sufficient or rich enough for video processing. Finally, we will discuss the new operators in detail and how we can use them for expressing queries in Situation-Sets 1 and 2 described in section 3.

5.1 Data Model for Video Contents

Traditional Relational representation does not allow multiple values (related or not) for an attribute, which forces one to use multiple attributes for extracted video content, such as bounding boxes (*which is a vector that has 4 integer values corresponding to the base coordinates along x and y axis of the frame and two offsets describing the width and height of the rectangular box*) and feature vectors (*typically multi-dimensional vectors whose dimensions vary depending on the feature extraction algorithm used to identify and extract the content, such as histogram, SIFT (Scale Invariant Feature Transform), or SURF (Speeded-Up Robust Features*). Hence, representation as well as expression of queries using the traditional representation is complicated as well as difficult to understand. It also makes computation and optimization of queries difficult and expensive. Another drawback of the traditional relational data model is its inherent row-oriented computation. If one wants to compare elements from two rows, a join is needed to bring them into a single row. Further-

more, lack of ordering for performing computations (as it is a set-based model) does not suit video data that is inherently a sequence of frames (and time stamps.) All of the above are data model issues that need to be overcome for video content analysis.

Ordering of video stream data tuples is important to make sure frames are not processed in arbitrary order, and sequences (of bounding boxes, frames etc.) are used correctly for computing various actions (e.g., movement). If one wants to compute the number of frames in which the same object appears, or duration of an object in a video, one needs to bring together tuples with consecutive frames/timestamps for performing computations of each scenario. This is not possible in SQL or CQL except to perform large number of self-joins. Even then, ordering of these attribute values from different tuples is not available. Hence, one of the data model extensions we propose is a vector or an array data type (whose contents may themselves be array/vector data type) to the existing data model and use it to represent attributes such as bounding boxes as well as ordering of values for other computations. We should also be able to order the contents of each vector values based on one or more attributes of interest. With the ability to convert attribute values into a vector representation based on specified ordered attribute, each object extracted from a frame can be represented as a tuple shown in Table 5.1. Each vector attribute name is enclosed within brackets and bounding box and feature vector attributes as described above are represented as vectors. Underlined attributes denote the key of the relation. Assuming multiple objects in a frame and 30 frames per second, same object may be present in several tuples.

An extension to the relational model that has been proposed to deal with ordered computations (*useful in the financial domain for computing moving averages and in other domains*) is an **arrable** [15]. An arrable combines array (as multi-set) representation with the relational model where an attribute can be an array of scalar

values. All relational operations (e.g., `SELECT, PROJECT, JOIN`) have been defined to be backward compatible on an arrable and additional computations have been defined that can be expressed on vectors by grouping on certain attributes and assuming an order (which is not possible in the traditional relational model.) AQuery [15] is an extension of SQL which was introduced to deal with computations that require order dependence (e.g., moving averages). AQuery has also introduced a number of vector operations (order preserving, size preserving, and size not preserving.) Hence, the arrable and the AQuery language can be viewed as a generalization of the traditional relational model and SQL, respectively. Extending the arrable further with vector and enumerated data types as a primitive attribute value will provide us with a richer model on which continuous queries for situation analysis can be expressed and processed. Detailed explanation of this is described in Arrable Operator under Section 5.2.1.

cid	**Fr#**	**oid**	**[FV]**	**[BB]**	**ts**
1	1	1	[fv1-1]	[0,0,10,12]	1
1	1	2	[fv2-1]	[bb2-1]	1
1	1	3	[fv3-1]	[bb3-1]	1
1	...	...	...	...	...
1	31	1	[fv1-31]	[bb1-31]	21
1	31	2	[fv2-31]	[bb2-31]	21
1	31	4	[fv4-31]	[bb4-31]	21
1	...	...	...	...	...
1	61	2	[fv2-61]	[bb2-61]	31
1	61	4	[fv4-61]	[bb4-61]	31
1	61	5	[f5-61]	[bb5-61]	31
1	...	...	...	...	...
1	91	2	[fv2-91]	[bb2-91]	41
1	91	4	[fv4-91]	[bb4-91]	41

Table 5.1: **VS1: Entry Video Stream as a Relation with Vector Attributes**

Described below is an example of an arrable representation. Given a video stream from a surveillance camera in a parking lot, consider the extracted information from four frames shown in Table 5.1 using the schema shown below. In this table, we are annotating a vector attribute by adding [and]. For example, feature vector attributes is shown as $[FV]$. The size of a feature vector actually depends on its type (*e.g., Histogram,SIFT or SURF described above*). Similarly, a bounding box is represented using coordinates of the frame (*if the camera is fixed*) or can be normalized for the motion of the camera if it is known.

VS1 (Camera_id as cid, Frame# as fr#, Object_id as oid, Feature_Vector as [FV], Bounding_box as [BB], Timestamp as ts)

Table 5.1 shows a sample of the extracted data represented as a relation using vectors. For brevity, assume that object 1 occurs in consecutive frame numbers 1 to 50, object 2 in 1 to 20 and reappears from 31 to 91. Also assume that object 3 appears in consecutive frames from 1 to 40, object 4 from 31 to 90 and object 5 from 61 to 90.

```
SELECT          [fr#], oid, [FV], [BB], [ts]
FROM            VS1
ASSUMING ORDER  fr#
WHERE           consecutive(40, [fr#])
GROUP BY        oid
ORDER BY        vs1.oid ASC
```

Assuming order is used for each group after the group by. Once the input is in an arrable form, we can write AQuery queries to select objects that occur in n consecutive frames. For n as 40, the above mentioned AQuery results in the output shown in Table 5.2.

cid	**Fr#**	**oid**	**[FV]**	**[BB]**	**ts**
[1, ..., 1]	[1, ..., 40]	1	[fv1-1, ..., fv1-40]	[[0,0,10,12], ..., bb1-40]	[1, 21]
[1, ..., 1]	[31, ..., 71]	2	[fv2-31, ..., fv2-71]	[bb2-31, ..., bb2-71]	[21, 31]
[1, ..., 1]	[1, ..., 40]	3	[fv3-1, ..., fv3-40]	[bb3-1, ..., bb3-40]	[1, 21]
[1, ..., 1]	[31, ..., 71]	4	[f4-31, ..., fv4-71]	[bb4-31, ..., bb4-71]	[21, 31]

Table 5.2: **Result of AQuery on VS1 in Arrable Representation**

Ordering and grouping can be done on any attribute(s) to get the values of vectors as a sequence. Various vector-based computations can be performed on them without resorting to multiple joins, making the query easy to understand and optimize.

We also include **enumerated data types** as part of the existing data model to capture predefined attribute values such as, direction of motion. It can have 8 predefined direction values like (*North,South,East,West,North East,South East,North West,South West*). We can also define more values to include later.

However, this by itself is not sufficient for the purposes of expressing meaningful situations on videos. For example, in the example mentioned above, unless the consecutive occurrences are satisfied with respect to a given value, we cannot assume fixed size computations which may be meaningful in other domains (e.g., financial applications) for computing moving averages etc. For the video domain, the number of frames (or duration) in which an object appears is not known in advance. One option is to try and use the current notions of windows (discussed in Section 4.3) for this purpose. It turns out that current window types are not sufficient for this purpose as well.

5.2 New Operators

As described in the previous sections, there is a need to add new operators which specifically support queries on video stream input data. So, in this section we will discuss the design of new operators that have been implemented as part of this thesis to support queries on video stream input. Several of these operators are required to express situations defined in Section 3 as part of Situation **Set 1** and their relevance. They can also be used for how some of the "actions" needed for **Set 2** can be expressed as composite events on the extracted video contents using event operators. Due to our extended representation, we assume that operator and attribute compatibility can be checked by the system. For example, it does not make sense to apply scalar operations on a vector representation and vice versa. Similarly, traditional relational operations such as $>$ is not applicable to certain attribute types like bounding boxes, feature vectors etc.

5.2.1 Arrable Operator

There can be various situations in video stream data where we need to store multiple occurrences of the same element in a sequence of frames. For example, in order to predict the trajectory of a moving object, we need use bounding boxes from a sequence of frames to compute an object's trajectory. This can be achieved by using the arrable representation of extracted data which can be created by using the arable operator on the initial data representation. An Arrable operator integrates the capability of converting a traditional relation into an arrable representation. Arrable representation and its need for expressing and querying video stream data has been described in Section 5.1. It takes two inputs : a group by attribute(s) or gba and an assuming order attribute(s) or aoa. Both these inputs can either be one or multiple

attributes. Arrable operator assigns each tuple to a unique group, based on the values of gba attributes.

Each group is a collection of tuples with the same gba attribute values. Post the formation of all the groups in the current window, tuples in each group are ordered based on the aoa attribute values. Once the ordering is completed, each group is reduced to a single tuple (arrable tuple) where each attribute represents an array consisting of ordered values of that attribute in that group, except the gba attribute(s). It is a Unary Operator, as it only needs a single input stream to process producing a stream output.

An example of an Arrable operator is described below. Note that our extension already supports vectors for attribute values. This extension is also included in the arrable creation. We will assume the gba attribute as oid and aoa attribute as Fr# for this example. Table 5.3 is the input to the arrable operator. For brevity, assume that object 1 occurs in consecutive frame numbers 1 to 50, object 2 in 1 to 90. Also assume that object 3 appears in consecutive frames from 1 to 40 and object 4 from 31 to 90. As we can see in Table 5.4, which is the output of the arrable operator, except oid (*gba attribute*), all other attributes are converted to arrable representation and are ordered based on Fr# (*aoa attribute*).

Once an arable is created, all the vector operators associated with AQuery (e.g., same object in n consecutive frames shown in) can be used on this representations without having to perform joins. This representation can also be used for computing actions in a video (e.g., direction of motion) as well. Arrable representation along with native vector representation permits representation of feature vectors and other aspects of information extracted from a video in an elegant manner.

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
1	1	[fv1-1]	[bb1-1]	ts1
1	2	[fv2-1]	[bb2-1]	ts1
1	3	[fv3-1]	[bb3-1]	ts1
1	...	...	...	...
31	1	[fv1-31]	[bb1-31]	ts31
31	2	[fv2-31]	[bb2-31]	ts31
31	4	[fv4-31]	[bb4-31]	ts31
1	...	...	...	...
61	2	[fv2-61]	[bb2-61]	ts61
61	4	[fv4-61]	[bb4-61]	ts61
1	...	...	...	...
90	2	[fv2-90]	[bb2-90]	ts90
90	4	[fv4-90]	[bb4-90]	ts90

Table 5.3: **Sample Input for an Arrable Operator**

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
[1, ..., 50]	1	[fv1-1, ..., fv1-50]	[bb1-1, ..., bb1-50]	[ts1, ...,ts50]
[1, ..., 90]	2	[fv2-1, ..., fv2-90]	[bb2-1, ..., bb2-90]	[ts1, ..., ts20]
[1, ..., 40]	3	[fv3-1, ..., fv3-40]	[bb3-1, ..., bb3-40]	[ts1, ...,ts40]
[31, ..., 90]	4	[fv3-31, ..., fv3-90]	[bb4-31, ..., bb4-90]	[ts31, ...,ts90]

Table 5.4: **Sample Output for an Arrable Operator**

5.2.2 Compress Consecutive Tuples (CCT)

Since our representation of a video input (preprocessed) involves a tuple for each object from each frame, there is likely to be a lot of repetition of elements in multiple sequential frames. The presence of multiple consecutive tuples containing the same object significantly increases the size of the stream and in turn computation effort needed for operations such as join, grouping, ordering etc. At the same time, if one wants to retain all the information, that should be possible as well (e.g., for accuracy during forensic analysis which may not be subject to response time constraints of real-time processing.) For example, to identify a car crossing a check point, a single occurrence of the car in a check point frame may be sufficient to identify, instead of

using all the occurrences. Also it is possible that the same element appears multiple times in a video over disjoint period (Example : Person reentering a building from the same door multiple times a day). It may be important for us to not lose this information in the video. Hence, the stream input needs to be compressed in a way that temporally disjoint occurrences are preserved.

Even though the DISTINCT Operator is defined to reduce the multiple occurrences of an attribute,our need is to reduce each consecutive occurrences of an object into one or two occurrences. Also we want to keep all other attributes intact, which is not supported by the DISTINCT operator of SQL.

CCT operator helps us achieve the above and can be used further to improve the efficiency of processing. It is defined as :

`CCT (gba, aoa, {first/last/both})`

The first input to this operator as described above is the group by attributes(*gba*). The second input is the assumed order attributes(*aoa*). Ordering can be either ascending or descending. Both these inputs can either be one or more attributes. The last attribute specifies which of the consecutive tuples are selected. It can either be "first,last or both" with first being the default value if nothing is specified by the user. The significance of each of these inputs in described below.

Table 5.5 describes the sample input to the CCT operator. We assume that oid is gba attribute and Fr# is the aoa attribute for this example. For brevity, assume that object 1 occurs in consecutive frame numbers 1 to 50, object 2 in 1 to 90. Also assume that object 3 appears in consecutive frames from 1 to 40 and object 4 from 31 to 90.

CCT Operator takes each tuple as input as described in Table 5.5 and assigns it a group based on the group id value of gba attributes. Post the creation of groups, the tuples are ordered based on the specified aoa attributes. Ordering is needed to

check the consecutiveness of the aoa attributes. This is achieved by converting the tuples into an arrable representation. Once the arrable is generated, the specified occurrence of the tuple in the group is returned as an output. If a user specifies "first" as the third attribute value, only the first occurrence of each gba attributes ordered on the basis of aoa attributes is picked from all, as described in Table 5.6. If "last" is specified by the user as the third attribute value, the last occurrence of each gba attributes ordered on the basis of aoa attributes is picked from all as described in Table 5.7. If "both" is selected as the third attribute value, an arrable is created with the first and last values as elements of the array for each attribute except the gba attributes. Table 5.8 describes the output of the CCT operator when "both" is chosen as the third attribute value. Let's assume a situation query, which says *"Find out if a marathon runner crossed a check point ?"*, then to answer this, we only need a single (perhaps first) occurrence of the runner in the checkpoint frame. Hence, we can use either the "first" or "last" (for all if used) as third attribute to figure out the same. However, if our situation query says *"Find out the time a runner took to cross a checkpoint ?"*, then we need to use both as the third attribute. As described in the sample table 5.8 we can get both, the first and last timestamp value for the runner, and compute the duration.

CCT is a unary operator as it works on a single input video stream data. It can be used with or without a window specification, however if window specification is used, consecutive occurrences of elements across window boundaries will not be taken into consideration as consecutive occurrences.

5.2.3 **Similarity Match(sMatch)**

Relational model provides a suite of arithmetic and Boolean operators for select and join conditions. These operators work on numeric and other kinds of data (e.g.,

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
1	1	[fv1-1]	[bb1-1]	ts1
1	2	[fv2-1]	[bb2-1]	ts1
1	3	[fv3-1]	[bb3-1]	ts1
1	...	...	...	...
31	1	[fv1-31]	[bb1-31]	ts31
31	2	[fv2-31]	[bb2-31]	ts31
31	4	[fv4-31]	[bb4-31]	ts31
1	...	...	...	...
61	2	[fv2-61]	[bb2-61]	ts61
61	4	[fv4-61]	[bb4-61]	ts61
1	...	...	...	...
90	2	[fv2-90]	[bb2-90]	ts90
90	4	[fv4-90]	[bb4-90]	ts90

Table 5.5: **Sample Input for a CCT Operator**

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
1	1	[fv1-1]	[bb1-1]	ts1
1	2	[fv2-1]	[bb2-1]	ts1
1	3	[fv3-1]	[bb3-1]	ts1
31	4	[fv1-31]	[bb1-31]	ts31

Table 5.6: **CCT Operator for "First" occurrence**

date). However, there are various attributes in the preprocessed video stream input data which cannot be compared using the available operators. For example, the feature vector extracted for the same person cannot be compared for equality as they differ slightly based on various aspects due to lighting, motion etc. A threshold is typically used for re-identification of an object which needs to be used as well when we want to compare objects from different video streams. For this purpose, we introduce a similar match function for matching two objects using feature vectors. This threshold value needs to be chosen carefully as well as it is not a universal value even for the same object type (e.g., person vs. vehicle.) Furthermore, the method used for this match is different depending upon the feature vector type (histogram

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
50	50	[fv1-50]	[bb1-50]	ts50
90	2	[fv2-90]	[bb2-90]	ts90
40	3	[fv3-40]	[bb3-40]	ts40
90	4	[fv4-90]	[bb4-90]	ts90

Table 5.7: **CCT Operator for "Last" occurrence**

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
[1,50]	1	[fv1-1,fv1-50]	[bb1-1,bb1-50]	[ts1,ts50]
[1,90]	2	[fv2-1,fv2-90]	[bb2-1,bb2-90]	[ts1,ts90]
[1,40]	3	[fv3-1,fv3-40]	[bb3-1,bb3-40]	[ts1,ts40]
[31,90]	4	[fv4-31,fv4-90]	[bb4-31,bb4-90]	[ts31,ts90]

Table 5.8: **CCT Operator for "Both" occurrence**

versus SIFT.) The sMatch function makes all this transparent to the user including the threshold where possible. This threshold value is needed to identify the extent that can be inferred while preprocessing the video and can be captured as part of extracted info. Similarly, the type of feature vector is also inferred based on the video meta data and is made part of the extracted information.

sMatch is used as part of a join or select condition along with other arithmetic and Boolean operators. sMatch always returns a true or false. sMatch is defined as :

```
sMatch (Stream1.attr, Stream2.attr,{threshold})
```

sMatch takes three parameters, one of them being optional. First parameter is the attribute (typically, a Feature Vector) from first video stream. Second parameter is the corresponding attribute to be sMathched from the second video stream. The optional third parameter is the threshold value to infer similarity of matches. Different match methods are used to calculate the threshold value between the attributes based on the type of attribute. For example, If the feature vector attribute type is SIFT (Scale Invariant Feature Transform) then the Euclidean distance matching between the key feature points are used for performing the match. If the ratio of key points

matched to all the key points is greater than desired threshold value, we call it a match (refer Equation 7.1). Threshold value range for SIFT can be between 0 and 1 with 0 being a no match and 1 being an exact match with all the key points matched. However, if the Feature vector type is a Histogram, then either of chi-square or Euclidean distance matching is used. For this thesis, we have used Euclidean distance matching by calculating euclidean distance between two histograms. If the distance between two histograms is less than the desired value, we call it a match. Threshold value range for histograms can vary depending on the objects. For this thesis, we have used the threshold value as 0.0005 for one of the data sets. Details of this are described in Chapter 7. If the match between two input attributes is made, the two tuples from different streams are joined and output as a single tuple. Preprocessing will generate a threshold value to be used for each object type (or feature vector type) so the burden is not on the user. Of course, it can be supplied as part of the query if desired.

Example of how an sMatch function is used to compute joins is described in the section 5.2.4.

5.2.4 **Consecutive Join(cJoin)**

With the availability of sMatch as a Boolean operator, the traditional join can be used for joining two video streams to identify objects that match. However, this is likely to be inefficient due to the multiple occurrence of the same object which is likely to significantly increase the number of matches. We provide another join operator (termed cJoin) where the cross product computation can be significantly reduced in the presence of multiple occurrences of the same object in a video stream. Use of an sMatch operator either in a selection or in a join can be processed and optimized like any other operator. Since the cost of this operator is higher than tradition

relational comparison of attributes, this comparison may be deferred for efficiency in the presence of other conditions. Also there is likely to be a lot of repetition of elements in multiple frames (*e.g. : a person entering or exiting a building is likely to be in multiple consecutive frames*). As traditional join is defined in terms of Cartesian product followed by selection (although not implemented in that fashion), this kind of join on videos is not only extremely inefficient (due to the cost of sMatch) but also produces enormous results that are not useful. We need to avoid matching all the occurrences of the same object and generating multiple instances of the same object. We need to be able to compare and match the same object once and disregard other *consecutive* occurrences and generating redundant matches. In order to solve this problem, we introduce Consecutive Join (*cJoin*).

cJoin compares and matches the same object just once and disregards all the other occurrences of the same object. cJoin works similar to a traditional Join with two input video data streams. Syntax of cJoin is defined as :

`cJoin(VS1.gba,VS1.aoa,VS2.gba,VS2.aoa,Join-Condition)`

The purpose of this operator is to avoid matches of the same objects multiple times. To support this semantics, there are multiple ways of implementing it.

1. Perform a traditional join. Once a match is found, remember the gba and aoa attribute values of those tuples. If a tuple is encountered with the same gba values and is consecutive based on aoa values (in either stream), discard them without performing a join. That is the join conditions are not applied including sMatch and saves significant computation. This approach to computing cJoin is non-blocking as grouping and ordering need not be performed. This approach will work well with or without a window.
2. Use the gba attributes on both streams individually to apply the CCT operator described earlier using either the first or last or both parameters. Once it is

done, tuples are joined using the traditional join approach for first and last parameters. For the both parameter an arrable is returned. Arrable tuple join is done as is described next. This approach is a blocking approach as grouping and ordering need to be applied and hence is suited for a window-based computation. This approach is more efficient than the previous one as individual windows are grouped and ordered to eliminate tuples in each stream. However, the accuracy may suffer as a result.

3. In this approach group by and ordering is applied on each stream separately without eliminating any tuples. As a result, we get an arrable representation for each stream. For each arrable tuple, elements (essentially feature vectors) are compared until either a match is found or otherwise. The first occurrence of the match outputs a tuple (not an arrable) and discards the rest of the elements in that tuple. Consecutive ordering of aoa attributes is used as well to match temporally disjoint tuples properly. This approach falls in between the above two approaches in efficiency. Also, expected to produce the same accuracy as the first approach. However, this is a blocking approach as compared to the first one.

Consider the situation where we need to *find if a person entering and exiting a building from two different entry and exit areas is the same.* As shown, we have multiple occurrences of objects in consecutive Fr# in both the entry (Table 5.9) and exit (Table 5.10) streams. For brevity, we will assume the first two attributes of sMatch function in Join Condition to be Stream1.FV and Stream2.FV . Also, we will assume FV's of oid 1 from Table 5.9 matches with FV's of oid 11 from 5.10 and FV's of oid 3 from Table 5.9 matches with FV's of oid 13 from Table 5.10. Also, we assume the gba attributes for both the input streams is *oid* and aoa attributes for both the

input streams is *Fr#* and *"last"* is mentioned as the third attribute for applying CCT operator on both streams .

If we use the first approach described above, which returns the joined tuple with first match occurred from both streams, we will get the output as described in Table 5.11. If we decide to use the second approach using CCT with "last" occurrence on both streams, we will get the output as described in Table 5.14. Intermediate results of applying CCT on both entry and exit streams are shown in Tables 5.12 and 5.13. As can be seen from both the output tables, only a single tuple from each stream is matched and returned. However, if we use the third approach described above, which returns the joined tuple with first match occurred after grouping from both streams, we will get the output as described in Table 5.15.

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
1	1	[fv1-1]	[bb1-1]	ts1
1	2	[fv2-1]	[bb2-1]	ts1
1	3	[fv3-1]	[bb3-1]	ts1
1	4	[fv4-1]	[bb4-1]	ts1
2	1	[fv1-2]	[bb1-2]	ts2
2	2	[fv2-2]	[bb2-2]	ts2
2	3	[fv3-2]	[bb3-2]	ts2
2	4	[fv4-2]	[bb4-2]	ts2
3	1	[fv1-3]	[bb1-3]	ts3
3	3	[fv3-3]	[bb3-3]	ts3

Table 5.9: **Entry Stream : Sample Input 1 for cJoin**

Fr#-1	**oid-1**	**[FV-1]**	**[BB-1]**	**ts-1**	**Fr#-2**	**oid-2**	**[FV-2]**	**[BB-2]**	**ts-2**
1	1	[fv1-1]	[bb1-1]	ts1	100	11	[fv11-100]	[bb11-100]	ts100
1	3	[fv3-1]	[bb3-1]	ts1	100	13	[fv13-100]	[bb13-100]	ts100

Table 5.11: **Sample cJoin Output for Entry and Exit Streams using Approach 1**

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
100	11	[fv11-100]	[bb11-100]	ts100
100	12	[fv12-100]	[bb12-100]	ts100
100	13	[fv13-100]	[bb13-100]	ts100
100	14	[fv14-100]	[bb14-100]	ts100
101	11	[fv11-101]	[bb11-101]	ts101
101	13	[fv13-101]	[bb13-101]	ts101
102	11	[fv11-102]	[bb11-102]	ts102
102	13	[fv13-102]	[bb13-102]	ts102

Table 5.10: **Exit Stream : Sample Input 2 for cJoin**

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
3	1	[fv1-3]	[bb1-3]	ts3
2	2	[fv2-2]	[bb2-2]	ts2
3	3	[fv3-3]	[bb3-3]	ts3
2	4	[fv4-2]	[bb4-2]	ts2

Table 5.12: **Intermediate Result of Entry Stream after applying CCT operator with "last" occurrence for cJoin Approach 2**

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
102	11	[fv11-102]	[bb11-102]	ts102
100	12	[fv12-100]	[bb12-100]	ts100
102	13	[fv13-102]	[bb13-102]	ts102
100	14	[fv14-100]	[bb14-100]	ts100

Table 5.13: **Intermediate Result of Exit Stream after applying CCT operator with "last" occurrence for cJoin Approach 2**

Fr#-1	**oid-1**	**[FV-1]**	**[BB-1]**	**ts-1**	**Fr#-2**	**oid-2**	**[FV-2]**	**[BB-2]**	**ts-2**
3	1	[fv1-3]	[bb1-3]	ts3	102	11	[fv11-102]	[bb11-102]	ts102
3	3	[fv3-3]	[bb3-3]	ts3	102	13	[fv13-102]	[bb13-102]	ts102

Table 5.14: **Sample cJoin Output for Entry and Exit Streams using CCT Approach 2**

Fr#-1	**oid-1**	**[FV-1]**	**[BB-1]**	**ts-1**	**Fr#-2**	**oid-2**	**[FV-2]**	**[BB-2]**	**ts-2**
1	1	[fv1-1]	[bb1-1]	ts1	100	11	[fv11-100]	[bb11-100]	ts100
1	3	[fv3-1]	[bb3-1]	ts1	100	13	[fv13-100]	[bb13-100]	ts100

Table 5.15: **Sample cJoin Output for Entry and Exit Streams using Grouping Approach 3**

5.2.5 Order By

For creating an arrable, there is need for ordering tuples on one or more attributes in aoa specification. As timestamps are part of extracted contents, there may be a need to order tuples by timestamp value. For example, in order to compute *"Find out the duration in which a marathon runner crossed a checkpoint"*. To compute this situation query, we will need to sort the tuples in the order of their time stamps as we cannot always rely on video stream input to feed tuples sorted on timestamp basis. Hence, we introduce the operator in MavStream which is similar to the traditional Order By operator defined in SQL. The syntax of Order By Operator is :

OrderBy(aoa,{ASC,DESC})

Order By is used to order tuples based on the aoa attributes and order type specified, i.e. ascending or descending. aoa attributes can be one or more attributes. Once all the tuples in the window are processed, it returns the output tuples in the specified ordered manner. Order By operator can be used with or without window specification, however if window specification is used,occurrences of *aoa* across windows will not be taken into consideration and tuples will be ordered and output for each window separately. Without a window specification, it becomes blocking as sorting needs all input before producing any output.

Table 5.16 describes a sample input to the order by operator. Let's assume that this is a video stream input of a runner crossing a checkpoint. For brevity , we assume ts as aoa and *"ASC"* as OrderType. As can be seen in 5.16, input tuples are not ordered on the basis of timestamps, hence it will be difficult for us to compute the duration from first and last tuple ts difference. Order By operator orders these tuples as can be seen in Table 5.17, hence, making it easier to compute the above mentioned situation query.

Fr#	**oid**	**ts**
1	1	ts1
3	1	ts3
4	1	ts4
3	2	ts2
4	2	ts4
2	1	ts2
6	2	ts6
5	2	ts5

Table 5.16: **Sample Input Stream for Order By Operator**

Fr#	**oid**	**ts**
1	1	ts1
2	1	ts2
3	1	ts3
3	2	ts3
4	1	ts4
4	2	ts4
5	2	ts5
6	2	ts6

Table 5.17: **Sample Output Stream of Order By Operator**

5.2.6 **Direction Operator**

For the class of queries that require either events or actions to be detected as part of the query (e.g., crossing a check post), it is necessary to compute these from the extracted video contents. Another approach is to compute these during preprocessing and express it as part of the data model. However, if this approach is taken, it has to be done on a case by case where as if it can be computed from the canonical extracted data, it becomes more flexible and extensible. Hence, we have taken this approach for inferring events/actions from the extracted video content.

An action from the video that can be used in multiple ways is the direction of motion of an object. This can be used to infer left- right- or even u-turn by composing movement segments. The basic idea is to use event composition to infer complex actions, such as u-turn or getting into a vehicle using simple events that reflect the direction of motion.

Direction is one such operator which helps us find the moving direction of an object from the preprocessed video stream input data. Each object in a video stream can be represented by a minimum rectangle (as a bounding box) that confines the outline of the object. It is represented as a vector of four values [x-coordinate,y-coordinate,length,width]. (x,y)-coordinates represent the position co-ordinates of the object, length and width represent the dimensions of the rectangle. Hence, to compute the direction of a moving object, we use its bounding box data. Syntax of the Direction event generator Operator can be defined as :

Direction(gba,dir-attr)

Direction is a unary operator as it works on a single video stream. It takes as input a window of tuples and applies the CCT operator using both as the parameter. This results in an arrable where each non-gba attribute will have two values in its array. Then the direction is computed using the dir-attr (which is typically a bounding

box.) Using the end points the direction of motion is computed and is assigned as one of the eight predefined values. Current predefined values are : NORTH, SOUTH, EAST,WEST,NORTH EAST,NORTH WEST,SOUTH EAST and SOUTH WEST. In the absence of a window specification, this will be a blocking computation. However, if window specification is used, occurrences of gba across windows will not be taken into consideration and might result into multiple movement detections for the same gba across windows. This behaves like an aggregate operator by adding a new attribute direction to the output. This output can be used to generate an event for composition.

Table 5.18 describes a sample input to the direction operator. For brevity, we assume *oid* as gba for direction operator. As we can see in Table 5.19, a new attribute for movement is added to the input video stream and direction is calculated from first and last occurrence of the gba attribute.

Fr#	**oid**	**[FV]**	**[BB]**	**ts**
1	1	[fv1-1]	[0,0,5,5]	ts1
1	2	[fv2-1]	[4,3,8,8]	ts1
1	3	[fv3-1]	[15,3,10,10]	ts1
2	1	[fv1-2]	[2,2,5,5]	ts2
2	2	[fv2-2]	[4,0,8,8]	ts2
2	3	[fv3-2]	[8,3,10,10]	ts2
3	1	[fv1-3]	[5,5,5,5]	ts3
3	3	[fv3-3]	[5,3,10,10]	ts3

Table 5.18: **Sample Input for Direction Operator**

We can use the composition of output results of direction operator to detect a non primitive event (Events in situations such as Situation Set 2 described in Section 3 can be classified as Non Primitive Content/events in a Video) such as an object turning around or two objects crossing each other etc. Once these events are generated

Fr#	**oid**	**[FV]**	**[BB]**	**ts**	**[direction_of_movement]**
[1,3]	1	[fv1-1,fv1-3]	[[0,0,5,5],[5,5,5,5]]	[ts1,ts3]	"North East"
[1,2]	2	[fv2-1,fv2-2]	[[4,3,8,8],[4,0,8,8]]	[ts1,ts2]	"South"
[1,3]	3	[fv3-1,fv3-3]	[[15,3,10,10],[5,3,10,10]]	[ts1,ts3]	"West"

Table 5.19: **Sample Output for Direction Operator**

along with other attributes, such as oid, Fr#, and ts, they can be combined using event operators of Snoop. Detection of an object "turning around" can be expressed as a composition of several events for the same object and possibly with some time constraints. This can be expressed in Snoop and detected by the event detector that is part of the MavEStream to take an appropriate action (which could result in a stream with the direction of motion for each object). Future work on this can be exploring similar solutions for other interesting high-level events, such as "entered a vehicle", "crossed each other" etc.

5.3 Summary

In this Chapter, we discussed extension to the existing data model to express and process queries on video stream inputs. We also described why the existing data model is not adequate to capture the intricacies of video stream data. Then, we discussed about the relevant operators that are needed for processing queries corresponding to situations in video stream inputs. Use of these operators to express situation queries is described in the next Chapter. This is a work in progress and additional extensions may be needed to handle or deal with more complex situations. This can be considered as future work to this thesis. All the above discussed operators have been implemented in MavVStream (described in Section4.2). Details of the implementation for each operator are described in Chapter 7.

CHAPTER 6

Expressing Situations using New Operators

In this chapter, we will describe how situations described as queries in Section 3 can be expressed and computed using the extended data model and operators proposed in this thesis. All these operators are compatible with the relational model and the arable as well. Preprocessing of video streams is assumed as described in Section 4 and representation as shown in Section 5.1. An existing stream processing system (MavEStream) is being extended to support both the data model and the new operators proposed as MavVStream. AQuery operators [15] will also be incorporated as needed. Current system supports both physical and logical windows with flexible window specification with hop size to support disjoint and tumbling/rolling windows. It accepts operator sequences as a query plan in the form of an operator tree. We use that approach below to express situations. We will use Tables 6.1 and 6.2 for showing sample evaluation of queries. For brevity, assume that in VS1, object 1 occurs in consecutive frame numbers 1 to 50, object 2 in 1 to 90. Also assume that object 3 appears in consecutive frames from 1 to 40 and object 4 from 31 to 90.

6.1 Queries in Situation Set 1

Queries in Situation Set 1 as described in Section 3 are window-based aggregation type queries. Below are some of those queries and how they can be expressed and computed using our MavVStream.

1. **How many people entered every disjoint 30 minutes?**

cid	**Fr#**	**oid**	**[FV]**	**[BB]**	**ts**
1	1	1	[fv1-1]	[bb1-1]	ts1
1	1	2	[fv2-1]	[bb2-1]	ts1
1	1	3	[fv3-1]	[bb3-1]	ts1
1	...	...	...	...	...
1	31	1	[fv1-31]	[bb1-31]	ts31
1	31	2	[fv2-31]	[bb2-31]	ts31
1	31	4	[fv4-31]	[bb4-31]	ts31
1	...	...	...	...	...
1	61	2	[fv2-61]	[bb2-61]	ts61
1	61	4	[fv4-61]	[bb4-61]	ts61
1	1	...	...	...	...
1	90	2	[fv2-90]	[bb2-90]	ts90
1	90	4	[fv4-90]	[bb4-90]	ts90

Table 6.1: **VS1: Entry Video Stream as a Relation with Vector Attributes**

This query can be expressed if time-based tumbling window is available along with grouping and ordering. We need to count the distinct number of objects in the video stream from Table 6.1. We can express this query using CCT (oid, F#, first or last) on VS1 which eliminates all consecutive occurrences of the same object using F#. The output of each 30 minute window from CCT is grouped by oid and a count (*) is applied. Hop size of 30 minutes is used for moving the window. The result of this query applied on VS1 will be 3 (objects 1, 2, and 3) for the window 1 to 15, 4 (objects 1, 2, 3, 4) for the window 31 to 60, and 2 (objects 4, 5) for the window 61 to 90. Figure 6.1 represents the operator tree formed to process this query.

cid	Fr#	oid	[FV]	[BB]	ts
2	10	15	[fv15-10]	[bb15-10]	10
2	10	12	[fv12-10]	[bb12-10]	10
2	12	15	[fv15-12]	[bb15-12]	12
2	55	13	[fv13-55]	[bb13-55]	55
2	57	13	[fv13-57]	[bb13-57]	57
2	60	11	[fv11-60]	[bb11-60]	60
2	70	10	[fv10-70]	[bb10-70]	70
2	...	...	...	...	...
2	90	11	[fv11-90]	[bb11-90	90
2	90	10	[fv11-90]	[bb11-90]	90

Table 6.2: **VS2: Relation for the Exit Video Stream**

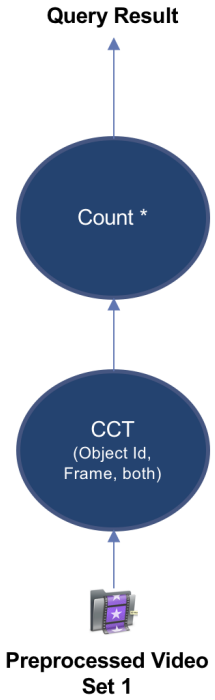


Figure 6.1: Operator Tree for Query 1

Queries that do not use a window but time range can be expressed similarly both in SQL and our approach. First, the SELECT operator is applied specifying the range and the result is fed to the CCT operator as a single window.

2. **Identify the slowest/busiest 30 minutes of the day?**

   We know that the answer to this query is between timestamps 61 to 90 from above VS1 Table 6.1. However, the output for query 1 consists of the 4 values and the aggregate operator min or max is applied on that. But, we also need the interval. AQuery constructs can be used for this purpose. We can apply CCT on VS1 with oid and ts as gba and aoa attributes. This will give an arrable with 4 tuples from which we can project [ts], min(oid) (or [ts] max(oid)) as the final result. Figure 6.2 represents the operator tree formed to process this query.

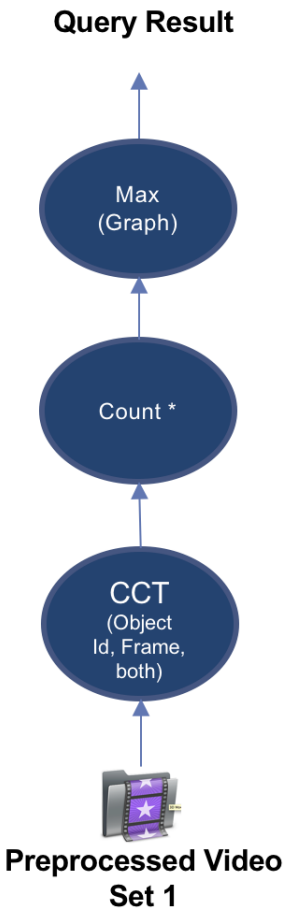


Figure 6.2: Operator Tree for Query 2

3. **List objects that stayed for less than 30 minutes.**

   This query needs another input stream corresponding to the exit video in addition to the video corresponding to entry. Also, the use of sMatch condition is done with cJoin. We will use Table 6.1 as entry video and Table 6.2 as exit video. For ease of understanding we will assume at least one of the feature vectors of the following oids match: 2 & 10, 1 & 11, and 3 & 13. The above query can be expressed as -

   `VS1\{null, VS1.oid, VS1.ts\} cJoin(join\_condition)`

   `VS2\{null, VS2.oid, VS2.ts\}`

   where the join_condition is (VS1 sMatch(VS1.FV, VS2.FV, 0.9) AND VS1.ts + 30 $\leq$ VS2.ts). Output of this can be followed by a project of VS1.oid, VS2.oid, VS1.ts, VS2.ts. The result of this query results in two tuples ($< 2, 10, 25, 70 >$, $< 3, 13, 22, 55 >$). This is assuming matches of the last instances i.e CCT Approach. Although 1 and 10 match, they do not satisfy the second condition. If there was a match of feature vectors of, say, frame 25 of oid 1 and frame 65 of oid 10, it would have come out. Figure 6.3 represents the operator tree formed to process this query. The operator query tree for other approaches are shown as well.

4. **Indicate when 2 different people (given images or feature vectors) enter AND leave with in 10 minutes of each other.**

   Assume feature vectors are used for the given images and oids 1 and 3 match in Table 6.1 and oids 11 and 13 match in Table 6.2. In Table 6.1, oid 1 and 3 enter within 10 minutes of each other. In Table 6.2, oid 11 and 13 exit within 10 minutes of each other. If we further assume that the given images match these two respectively, the result should be objects 1 & 3 and 11 & 13. If there are multiple occurrences, they will also come out. First, we apply select on each

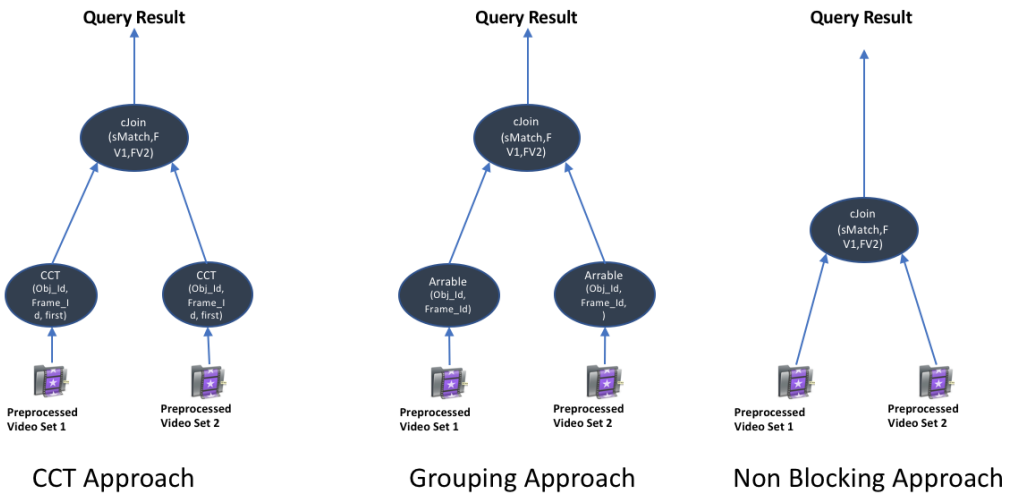


Figure 6.3: Operator Tree for Query 3

stream to isolate tuples that sMatch the feature vector of given images. Then CCT is applied to reduce each object to one consecutive occurrence with first or last on both streams. Then a join is performed for each stream to identify objects that entered within the designated time window for entry and for exit. That will give the above results. Note that the object ids for the same image is different in different streams as each stream is preprocessed independently. Figure 6.4 represents the operator tree formed to process this query.

5. **Indicate when a specific person (whose image or feature vector is given) entered the building.**

   This query is fairly straight forward. sMatch can be used to match the given image or feature vector to VS1 Feature Vectors. If sMatch returns a match for a tuple, return the ts value for that tuple and that will be the result to the query. Figure 6.5 represents the operator tree formed to process this query. Note, this is assuming that tuples appear in an ordered manner of ts in the entry video

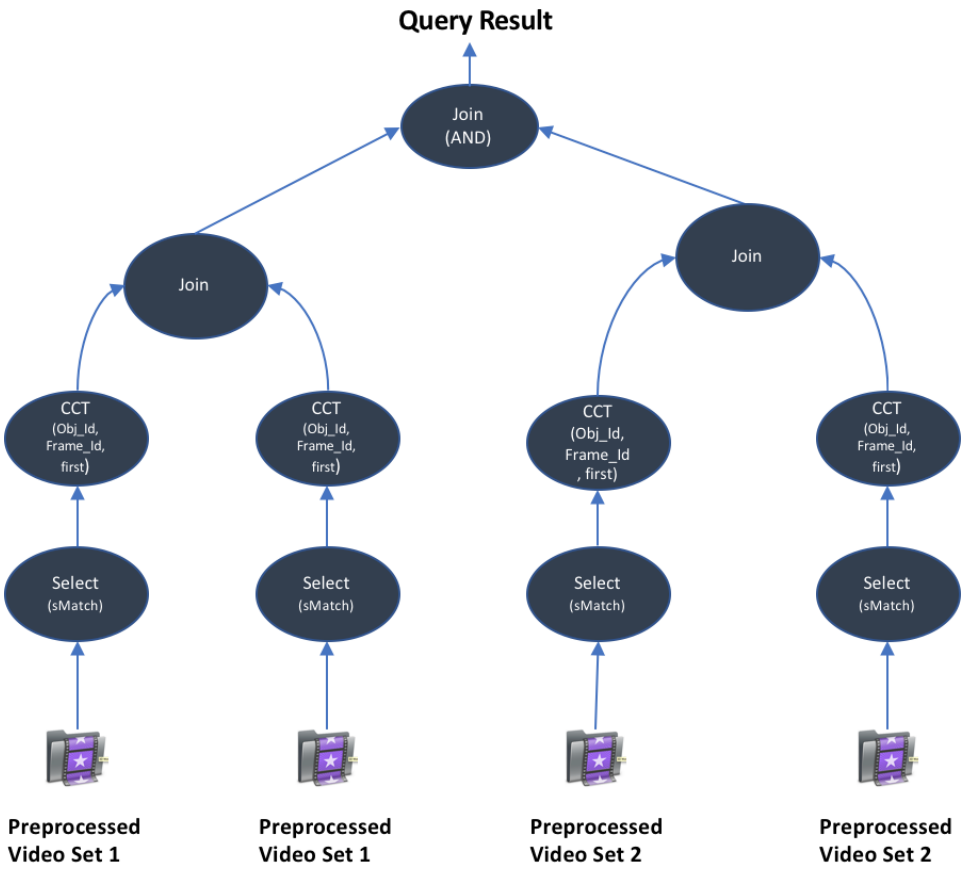


Figure 6.4: Operator Tree for Query 4

stream VS1. If this is not considered, then an order by on ts as aoa needs to be performed on the input video stream before it can be processed by sMatch.

6. **Did two individuals (images or feature vectors given) enter OR exit the building within 5 minutes of each other ?**

   This query is a variant of query 5 and uses and OR clause instead of and AND. Assume feature vectors are used for the given images and oids 1 and 3 match in Table 6.1 and oids 11 and 13 match in Table 6.2. In Table 6.1, oid 1 and 3 enter within 5 minutes of each other. If we further assume that the given

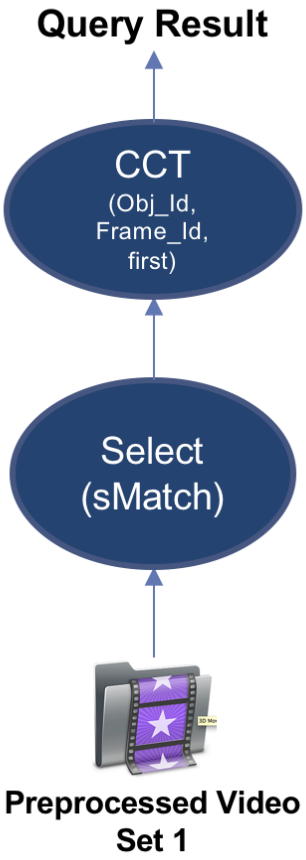


Figure 6.5: Operator Tree for Query 5

images match these two respectively, the result should be objects 1 & 3 and 11 & 13. Note here, we did not assume that oid 11 and 13 in Table6.2 exit within 5 minutes of each other but still got them in output as the query says either entry OR exit within 5 minutes of each other. If there are multiple occurrences, they will also come out. The computation can be done in exact same way as Query 5 with an OR clause at the end. First, we apply select on each stream to isolate tuples that sMatch the feature vector of given images. Then CCT is applied to reduce each object to one consecutive occurrence with first or last on both streams. Then a join is performed for each stream to identify objects that

entered within the designated time window for entry ***or*** for exit. That will give the above results. Figure 6.6 represents the operator tree formed to process this query.

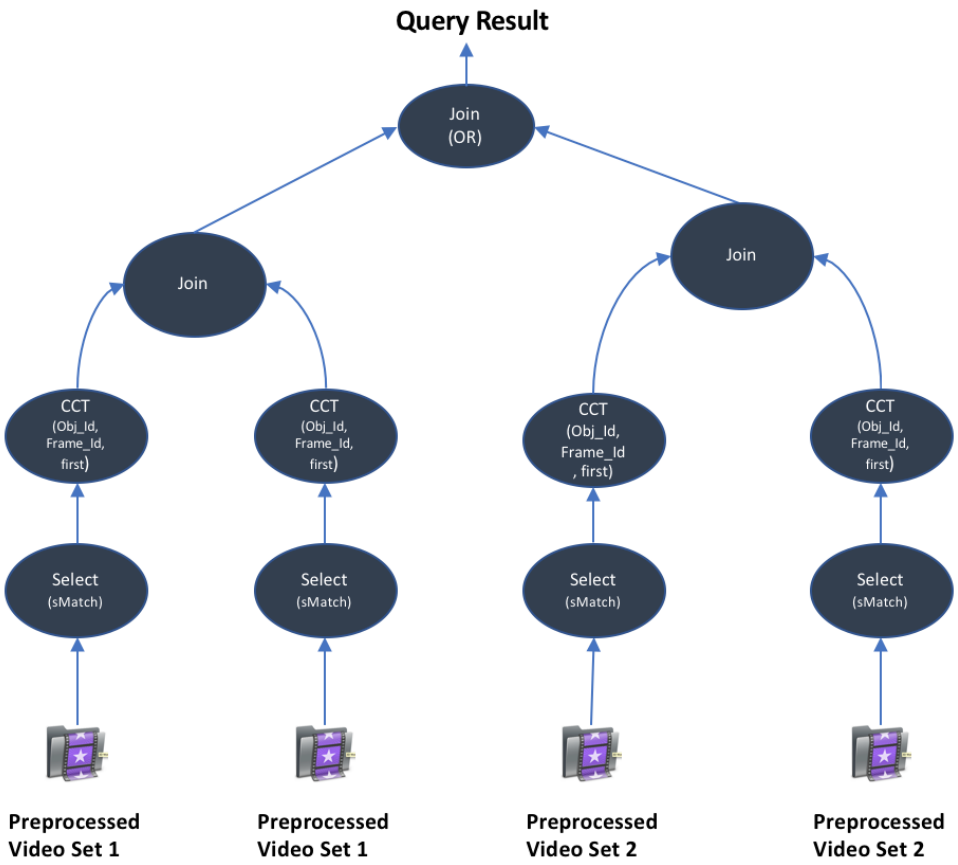


Figure 6.6: Operator Tree for Query 6

6.2 Queries in Situation Set 2

Here, we outline how queries in Situation Set 2 can be computed. As indicated earlier, our approach here is to detect primitive events/actions using a continuous

query and raise corresponding events. These raised events can be further composed to infer higher-level events/actions such as "turns around". "Turns-around" can be expressed as a composition of primitive events: "moving north" SEQ "moving left" SEQ "moving left" SEQ "moving south" and these happening in temporal succession. SEQ is a well-defined sequence event operator. A continuous query computing the direction of motion proposed earlier can raise these primitive events on a specified window. Contexts (here the "recent" context) available in many complex event systems can be used to drop multiple occurrences of any of these events and combine them properly. With this approach, the higher-level event "turns around" can be detected. For this to work, the object re-identification needs to identify objects correctly with front and back images in frames. Our contention is that if that cannot be done, even the extant IVA approaches cannot detect this event. We can detect these subject to the accuracy of the underlying preprocessing accuracy for which we can use best of the extant available techniques.

CHAPTER 7

IMPLEMENTATION

This chapter fleshes out the details on implementation of changes made to the MavEStream system with respect to the data model and all the new operators that are discussed in Section 5.2 to express and process queries on video stream input. Implementation of all these have been done in MavVStream (**Mav**erick **V**ideo **Stream** Processing System) described in Section 4.2. MavVStream has been implemented in Java. Extensions to the existing data model(such as vector and enumerated data types) and new operators have also been implemented in Java and incorporated as part of MavVStream.

All the preprocessing of videos for generating tuples for each object in each frame has been done using Matlab. Each attribute, including the vector attributes, is a Java Object. These objects are wrapped into a tuple object and written to a file as serialized objects. A multi-threaded feeder reads these objects from a file for each video stream and feeds them into input buffers of leaf operators at individually specified rates. Continuous queries (CQs) are processed by the server and generates interesting events or results of CQs.

Briefly, an input CQ is input by writing a Java program that generates a plan object corresponding to that query. The plan object is in the form of an operator computation sequence and contains all the necessary information for processing that CQ. This plan object is instantiated by the instantiator module which creates an executable CQ operator tree by instantiating each operator along with input and output buffers needed for processing that operator. A scheduler is also associated

with each operator for scheduling it using supported scheduling strategies. When a query is activated for execution, feeder populates the input queues and the scheduler starts scheduling the CQ operators. The system has the capability to accept latency and memory usage requirements and supports load shedding. However, this is not currently used for video processing. Details of all the modules of MavVStream and its architecture have been described in Section 4.2. Below, we describe the data model and operator implementations using MavVStream.

7.1 Data Model Extensions

Vector and Enumerated data types have been added to MavVStream as data model extensions. Inclusion of Vector data type allows us to represent bounding box and feature vector as attributes which are generated during preprocessing. In order to support vectors, a new VectorDataItem class has been implemented which extends the DataItem class. This VectorDataItem class contains multi-dimensional vector data represented as a string. A method *getVector()* has been implemented as part of this class which parses the multi-dimensional vector data represented as string and then creates objects of each dimension at a time and recursively continues until all dimension data has been read. It also validates the dimension value matching to number of elements in vector. This VectorDataItem class consists of attributes to capture the details of attributes of type vector in preprocessed video stream data such as bounding box, feature vector etc. These details are as follows -

- **vector_name** *(name for the vector defined in the schema)*
- **vector_type** *(can be Feature Vector, Bounding Box etc.)*
- **vector_datatype** *(can be Integer for bounding box, Double for Feature Vectors, String or Vector etc.)*

- **vector_signature_method** *(method used to create the vector such as Histogram, SIFT, SURF etc.)*
- **vector_attribute_value** *(contains the actual value of the vector)*

The dimensions of each vector depends on the vector_type and the signature_method used to create the vector. These can either be computed during run time in Java, or, can also be notified as part of schema if known previously (such as in bounding boxes (*vector of size 1*4*) or in Histograms (*multi-dimensional vector of size 3*256*). The above representation also allows us to include additional vectors of different types and be able to distinguish them at run time and using them for various operations. One such addition is an arrable described in Section 5.1. VectorDataItem class is also used to support an arrable as they are an attribute value sequence represented as an array/vector. The relevant non-vector attributes of the VectorDataItem are initialized when a vector for an arrable is created. Arrable is further used in implementation of many other operators dealing with multi-dimensional video stream data or where multiple instances of same objects need to be stored for further operations on it. One such example is the Arrable Operator where tuples from video stream input are formed into arrable representation. Details of implementation are described in Section 7.2.1.

The VectorDataItem is created either as part of the Continuous Query(CQ) computation by an operator (e.g., for creating arrables) or is generated as part of preprocessing. As part of a tuple, this attribute is carried as part of the stream unless projected out. This attribute is accessible to all operators although it will be used by operators such as sMatch, cJoin and direction operators described in Section 5.2. Note that the stream tuple composition (or schema) may change after an operation (e.g., join, project) and new stream definitions are generated and passed on.

We have also added Enumerated data types in MavVStream. Enumerated data type allows us to store the direction of movement values discussed in the design of direction operator. Currently, two specific enumerated types – DIRECTION as one of the eight directions *(NORTH, SOUTH, EAST, WEST, NORTH_EAST, NORTH_WEST, SOUTH_EAST, SOUTH_WEST)*, and WEEKDAYS as one of the seven days of the week *(SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY)* – are supported. It will be relatively straightforward to add new ones as needed and make them available for usage within the system. For the implementation, we have used Enum class in Java rather than the typical in-line usage. This is needed to define a stand-alone enumerated object type and is also consistent with our object association for each attribute value. Predefined enum types such as DIRECTION and WEEKDAY can be associated with attributes in the stream definition. We are exploring a generic way of supporting this for new enumerated types needed by the user.

7.2 Operator Extensions

Since MavVStream does not have a query language of it own, a Continuous Query (CQ) is input as a sequence of operators along with their input streams. Each operator outputs a stream of tuples along with its data stream definition *(DSD)* associated to its output buffer. *SELECT, PROJECT, JOIN, GROUP BY and AGGREGATE* are some of the existing operators of MavEStream. Care must be taken to perform an aggregation after performing a grouping to get the effect of aggregation semantics of SQL/CQL. HAVING of SQL is supported as a select on aggregated output. Notion of windows is defined currently for the entire Continuous Query rather than a separate window for every operator or stream. Start point, end Point and hop size (*value by which the window boundaries needs to be shifted to form the next*

window) can be specified in terms of tuples (*physical window*) as well as time (*logical window*). Window inputs are provided to MavVStream as part of the CQ.

Below, we discuss the implementation of the new operators that are defined in Section 5.2. All the new operators are implemented by extending the generic operator class.

7.2.1 **Arrable Operator**

Arrable Operator takes a stream of tuples as input and converts it into an arrable representation. Once the conversion is done, it changes the data stream definition (also known as *schema*) of all the attributes except the gba. Data type of each attribute is changed from its original data type to a Vector¡data type¿. The arrable tuples and new DSD is then put into the output buffer to be returned as output of the operator. This helps the next operators in the operator tree to identify that the schema definition of incoming tuples has been changed. Arrable operator processes each tuple in the input, one at a time, and puts it into a Hash Table using the gba attributes. Hash Table is the best data structure to be used for grouping the tuples with the same value of gba, as tuples in the group can be assessed in O(1) time. The key for this Hash Table is the group by attributes (gba) which are input to this operator. Once all the tuples in the input buffer queue are processed, it takes the list of tuples in the group, orders them on the basis of assuming order attributes (aoa) using the in-built collection sort of Java, and then converts each list of tuple into an arrable representation. That is, takes each tuple attribute, puts it into a separate vector and then outputs the list of attribute vectors as a single arrable tuple. Currently, this operator can be applied to any input where the gba and aoa are not vectors themselves. The operator can work with or without window specification.

If a window is specified, it will only consider the tuples in the current window for the grouping, ordering, and converting the tuples in the window into an arrable.

7.2.2 **Compress Consecutive Tuples (CCT)**

Compress Consecutive Tuples, as the name suggests is used to compress consecutive redundant tuples. As described in the design of CCT, preprocessing video stream input uses a large number of frames and as our representation is object based, each object can appear in a large number of consecutive frames (even hundreds or thousands) depending on the field of view of the camera. This operator is implemented as the final step after the grouping and ordering is done as explained in the implementation of arrable operator. After the ordering is completed, We now have a list of sorted tuples on aoa attributes. Based on the value of the third parameter which signifies the tuple to be chosen from the arrable for output, we select the tuple. If the parameter value is "first" or is not specified, first tuple from the arrable is output, whereas if the parameter value is "last", last tuple from the arrable is output. If the parameter value is specified as "both", both the first and last tuples are chosen and converted into an arrable representation with all the attribute values as vectors of size 2 except the gba attributes. Data stream definition is changed accordingly. This arrable tuple is then returned as output. If there will be only one occurrence of the object in a group, but both is selected as an occurrence option, still an arrable tuple will be created and each vector will have only one element in it.

7.2.3 **Similarity Match (sMatch)**

Similarity Match or sMatch function is applied on attributes (typically Feature Vectors) of two tuples from different input video streams to determine whether they match (i.e., they are the same object if using feature vectors.) sMatch uses a

comparison method based on the vector type and other characteristics (e.g., SIFT or Histogram or SURF) and generates a similarity value between calculated using the inputs. Although the definition for sMatch is same for every type of input attributes, the matching functions can be different. Current matching functions that are implemented as part of MavVStream are - **Hist_Match(fv1,fv2)** and **Sift_Match(fv1,fv2)**. *Hist_Match* is used for calculating the Euclidean distance between two RGB histogram feature vectors. Algorithm 1 describes the process of matching two histogram feature vectors. *Sift_Match* is used to calculate the similarity between key points of two SIFT feature vectors. The algorithm used in Sift Match is described below . Note here, that Hist_Match returns the Euclidean distance between two feature vectors which should be less than the threshold value for them to match with each other, whereas Sift_Match returns the similarity ratio between two feature vectors which should be more than the threshold value for them to match with each other. Once the feature vectors are confirmed as a match, tuples from both the input video streams are joined and output as a single tuple, similar to a traditional join operation when a join condition evaluates to true. Output data stream definition is updated accordingly.

Algorithm used to match two Sift Feature Vectors -

Two images or their Feature Vectors are said to be similar if the the ratio between the number of SIFT feature descriptors (or key points) that matched amongst the images to the total number of descriptors obtained from one of the above two images is greater than or equal to a threshold. If M1 is the number of feature descriptors in the first image, M2 is the number of feature descriptors in the second image and K is the number of feature descriptors that matched between the images, then the similarity value of first image to second image can be expressed as in the Equation 7.1. However, if similarity of second image to first is to be calculated, it can com-

Algorithm 1 Histogram Comparison

Normalize each RGB column of Histogram 1 *using below* 3 *equations.*

$$HistRed1 = \frac{H1[1][1], H1[1][2], ..., H1[1][256]}{256}$$
$$HistGreen1 = \frac{H1[2][1], H1[2][2], ..., H1[2][256]}{256}$$
$$HistBlue1 = \frac{H1[3][1], H1[3][2], ..., H1[3][256]}{256}$$

Normalize each RGB column of Histogram 2 *using below* 3 *equations.*

$$HistRed2 = \frac{H2[1][1], H2[1][2], ..., H2[1][256]}{256}$$
$$HistGreen2 = \frac{H2[2][1], H2[2][2], ..., H2[2][256]}{256}$$
$$HistBlue2 = \frac{H2[3][1], H2[3][2], ..., H2[3][256]}{256}$$

$$HRed = \sum_{i=1}^{256} |HistRed1[i] - HistRed2[i]|^2$$
$$HGreen = \sum_{i=1}^{256} |HistGreen1[i] - HistGreen2[i]|^2$$
$$HBlue = \sum_{i=1}^{256} |HistBlue1[i] - HistBlue2[i]|^2$$

$$H = 0.2989 * HRed + 0.5870 * HGreen + 0.1140 * HBlue$$

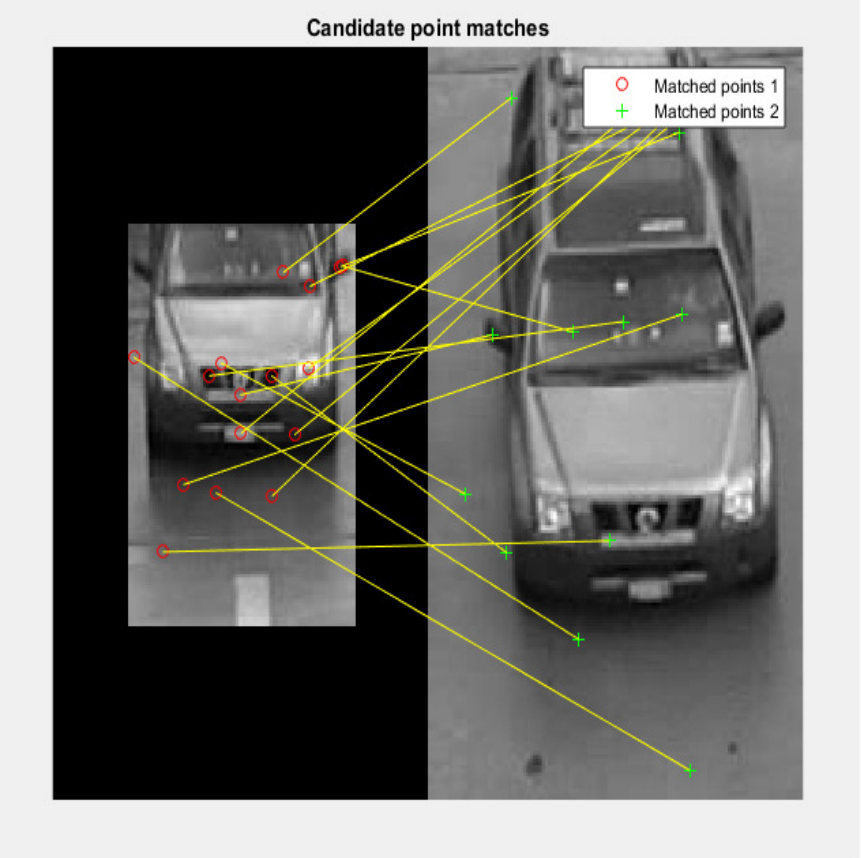


Figure 7.1: Illustration of SIFT key-points match between two images

puted using !7.2. A feature descriptor D1 in first image matches with descriptor D2 in second image, if the distance between them multiplied by 1.5 is not greater than distance of D1 to all other descriptors in the second image (Please refer to [16] and [17] for additional details about SIFT descriptor comparison.). Figure 7.1 illustrates the matching of key-points between two images using SIFT.

$$Similarity(1to2) = \frac{K}{M1} \tag{7.1}$$

$$Similarity(2to1) = \frac{K}{M2} \tag{7.2}$$

7.2.4 **Consecutive Join (cJoin)**

cJoin is implemented in the same way as traditional join on two video input streams, however, its join condition typically contains an sMatch function. The basic idea behind he consecutive join is to reduce the computation complexity of a traditional join in the presence of multiple occurrences of the same object in both the streams. Once a match is found for an object, the same object is not compared with the corresponding same object in the other stream. This is done only if the object appearance is consecutive and hence the name. cJoin takes as two streams as inputs and a number of parameters (group by attributes gba1 and gba2) and assume order attributes (aoa1 and aoa2) from two video inputs streams, respectively. MavVStream supports hash and nested joins implementations of traditional join operator. However, cJoin not being an equality match uses the nested join implementation. The design section on cJoin (Section 5.2.4 describes three approaches for implementing cJoin each having its own advantages and disadvantages. The implementation for all the approaches is described below.

- **Approach 1 - Using Grouping and then Match**

  In this approach, we form two groups of tuples as Hash Tables for each video input stream data, with key as gba1 and gba2 respectively. Once the groups are formed, the list of tuples in first group is ordered on aoa1 and the list of tuples in second group is ordered on aoa2. After the ordered groups are ready, each feature vector from each tuple in the first group is matched with each feature vector from each tuple in the second group. As soon as a match is found between any two feature vectors, tuples for those two feature vectors are joined and returned as output. We do not match any other tuples in those groups any further and proceed to the next group for each input stream. Advantage of

using this approach is that it is more accurate, since it will keep on matching till a match in all occurrences of the object is found.

- **Approach 2 - Using CCT and then Match**

  This approach uses CCT to filter out consecutive occurrences of tuples and only keeps a single occurrence depending on first/last value given to CCT operator. CCT(gba1,aoa1,first/last) is used on video stream 1 and CCT(gba2,aoa2,first/last) is used on video stream 2. Once CCT is complete for each stream, it then joins the two compressed streams . Note here, only single tuple for consecutive occurrence of an object will be matched unlike approach 1. Hence, this approach is much faster and cost effective than approach 2.

- **Approach 3 - Non blocking Approach**

  This approach maintains a hash set of objects. As soon as an object in video stream 1 is matched to object in video stream 2, gba from each stream is stored in the respective hash set and is not compared again. As streams are fed tuple by tuple from the feeder, this makes the approach non blocking.

There can be cases where first occurrence of any object in stream 1 might not match with first occurrence of same object in stream 2. This will result in a no match and will not generate the expected result. Approach 1 would have matched every occurrence of it in both streams and might have predicted a match at some occurrence. This is a drawback of approach 2. It is less accurate than approach 1. Based on the requirement of the user, we can use either of the two approaches. If faster results are expected and accuracy can be compromised, we should use approach 2, however if accuracy is given the priority, we should use approach 1.

cJoin can work with or without an sMatch function in the join condition. If an sMatch function is specified along with any other Boolean expression in the join condition, first sMatch function is applied on tuples. If the sMatch function returns

a pair of tuples matched, Boolean condition is evaluated on those tuples and a joined tuple is output if it evaluates to true. If join condition is without any sMatch function, Boolean expression will directly be evaluated on the tuples. Approach 1 and 2 will both yield the same results for this scenario. Output data stream definition for cJoin is updated in the same way as of traditional join.

7.2.5 **Order By Operator**

Order By operator takes each tuple and stores it in an Array List of tuples until all the tuples are processed. Once all the tuples are processed, it sorts them using the in-built Collections.Sort function of Java. The complete list of tuples needs to be stored first as Collection.Sort takes the complete list as input. Tuples are output based on the *OrderType* attribute specified by the user. If "*asc* or *ascending*" is specified as order type, the tuples are output in the ascending order of aoa. However, if "*desc* or *descending*" is specified as order type, the tuples are output in the descending order of aoa.

Order By is a blocking operator if the stream is unbounded. It is best used with window specification to make it a non-blocking operator.

7.2.6 **Direction Operator**

This operator is implemented to calculate the direction of motion of an object to compute an enumerated value indicating the direction of motion. Since the direction of motion needs to be computed for each object separately a gba attribute(s) is used as a parameter. It uses Enumerated data type defined in MavVStream to figure out the direction of motion as one of the eight directions of motion. Currently used values are described in the design chapter (Section 5.2.6. Direction operator reads each input tuple and assigns it a group by putting it into a Hash Table with key as gba. Once

the groups are formed for each value of gba, it takes into consideration the first and the last occurrence of gba from each group. Current implementation of Direction uses a bounding box to figure out the co-ordinates for computation. (x,y) co-ordinates of both first and last occurrence of gba are compared to figure out the final direction of movement of the gba. Note that this is the net direction of the movement and not the trajectory. A new String attribute, *dir_of_movement* is added to data stream definition of each object tuple and is output as part of the tuple. Based on the current values of direction defined as enumerated types in MavVStream, here is how direction is figured out -

Bounding Box for first instance of Object 1 - (x1,y1,w1,b1)

Bounding Box for last instance of Object 1 - (x2,y2,w2,b2)

- **(y2-y1 > 0) and (x2-x1 == 0)** - Object moves in the **NORTH** Direction
- **(y2-y1 < 0) and (x2-x1 == 0)** - Object moves in the **SOUTH** Direction
- **(y2-y1 == 0) and (x2-x1 > 0)** - Object moves in the **EAST** Direction
- **(y2-y1 == 0) and (x2-x1 < 0)** - Object moves in the **WEST** Direction
- **(y2-y1 > 0) and (x2-x1 > 0)** - Object moves in the **NORTH EAST** Direction
- **(y2-y1 > 0) and (x2-x1 < 0)** - Object moves in the **NORTH WEST** Direction
- **(y2-y1 < 0) and (x2-x1 > 0)** - Object moves in the **SOUTH EAST** Direction
- **(y2-y1 < 0) and (x2-x1 < 0)** - Object moves in the **SOUTH WEST** Direction

Direction operator can work with or without windows. If windowed approach is used, net direction of movement for an object will be calculated for each window separately.

In MavVStream, an event generator operator is added after the root node if there is an event association with that CQ. This produces an additional event output along with the regular stream output. The CQ for event generation can include a condition on the resulting attribute which is used to filter the tuples for generating events. In our example, the event "moving north" is generated if the value of the attribute is "north". Similarly, events "moving left", "moving south" are generated. The event detector uses this event stream to detect composite events which are specified separately and rules can be associated with any composite event.

7.3 Summary

This chapter described the implementation of all the data model extensions proposed in this thesis. It also described the implementation of the new operators that are a part of the first phase of MavVStream and are needed to express and compute situation queries over video stream data. Using this implementation, these data model changes and new operators were tested on a variety of data sets. The experiments conducted to test the validity of these implementations are described in the next chapter. Results for each experiments are shown.

CHAPTER 8

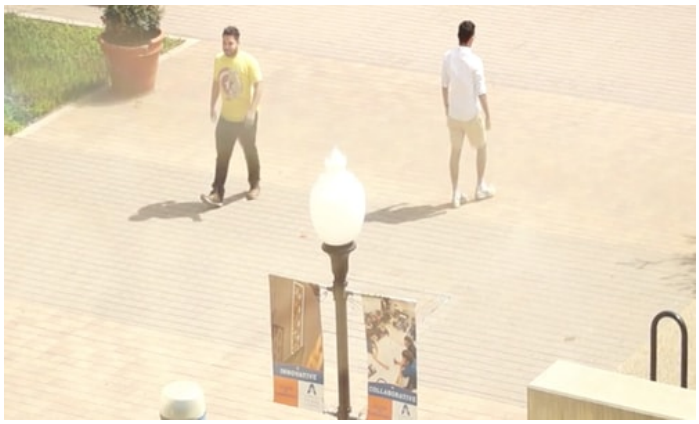
EXPERIMENTS AND RESULTS

As explained in the design section 6, we can express different situations using the richer data model and new operators introduced. This chapter discusses the evaluation of queries using actual video data sets.

8.1 Video Datasets Used

Below are the details of video datasets that were used for experimentation results in this thesis. Note here that these describe the Video Sets. Actual use of these datasets were done after preprocessing from Matlab. Different configurations were used to preprocess different video sets.

1. *Video 1*: Aerial view of people walking on a street (Figure 8.1a). This is a sample video bundled with MATLAB installation. The human images extracted in this video will have *low resolution* as the video is captured from far.
2. *Video 2*: Ground view of people entering inside a room (Figure 8.1b). This is a video recorded at our lab. The human images extracted in this video will have *higher resolution* as compared to Video-1.
3. *Video 3*: Ground view of people exiting a room (Figure 8.1c). This is a video recorded at our lab. The human images extracted in this video are similar in resolution (*higher resolution*) to that to Video-2.
4. *Video 4*: Ground view of cars at Point A in a parking lot (Figure 8.1d) [18]. The car images in this video have *higher resolution*.

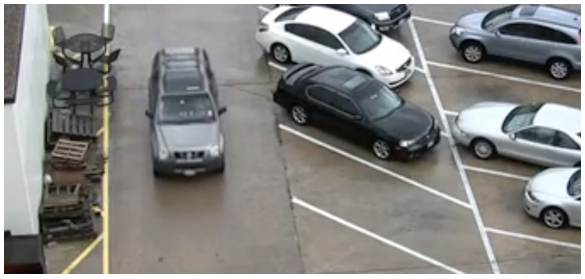


(a) Video-1: People Strolling on a Street

(b) Video-2: People Entering a Lab/Building

(c) Video-3: People Exiting from the Lab/Building shown in Figure 8.1b

(d) Video-4: Cars at Point A



(e) Video-5: Cars at Point B

Figure 8.1: Snapshots of videos used for experimentation

5. *Video 5*: Ground view of cars at Point B in a parking lot (Figure 8.1d) [18]. The car images in this video have *higher resolution*.

8.2 Data Stream Definition for Datasets

Preprocessed video datasets are defined in a generic way for all the video data. Here are the definitions of various tuple attribute fields used in these datasets. Position of each attribute may or may not be the same for each dataset and also it is not necessary that all attributes defined are present in the dataset if they aren't needed for the computation. This makes the computation faster to process. This is fed as a JSON file to the MavVStream server.

FrameId signifies the frame in which a moving object is detected. *ObjectId* is a new sequence assigned to any moving object detected. *PrevObjectId* is the id of the object to which matches the best amongst previously occurred objects. This is significant to check if an object reoccurred across multiple frames. *BB* is the bounding box for an object and is declared as a vector with four dimensions. FV is the Feature Vector for an object and is declared as a multi dimensional vector. *tbSourceTS* is the time stamp at which the object was detected in the frame. *tbSystemTS* is the tuple time stamp generated by the system once a tuple is fed into the system to be processed. Below we describe the generic schema/data stream definition for datasets as a JSON object, used after its video preprocessing.

Generic Schema for Preprocessed streams -

```
{   "tableName": "STREAM_NAME",
    "streamURI": "STREAM_TUPLES.txt",
    "fields": [
      [
        "ATTRIBUTE_NAME",
        "ATTRIBUTE_DATATYPE",
        "ATTRIBUTE_POSITION"
      ]
    ] }
```

8.3 Experiments

1. **How many people entered every disjoint 30 seconds ?**

   This situation query was executed on Video Dataset 2 where 7 distinct people are entering a lab (as detected by preprocessing). The operator query tree used is described in Figure 6.1. The output can be seen in Figure 8.2. As can be seen from the results, 5 people entered the building in first 30 seconds and only 2 people entered in the next. A new window was created for every 30 seconds and people were counted for each window. This was verified to be correct after manually checking the video. Time stamp displayed in the result is the timestamp of the last entry frame for the object.

Frame_Range	Obj_Id_Range	PrevObjectId	TimeStamp	Total Count in 30 Second Window
BeginWindow				
[130,266]	[2,149]	1	16	5
[352,421]	[150,223]	150	15	5
[423,463]	[225,266]	1	16	5
[578,732]	[267,422]	267	25	5
[733,824]	[423,516]	423	28	5
EndWindow				
BeginWindow				
[885,918]	[521,554]	519	31	2
[952,1041]	[588,689]	578	35	2
EndWindow				

Figure 8.2: Output for Query 1

2. **Identify the busiest 30 seconds of the day/video.**

   To compute this, we will use the output of Query 1.We need to find out the max count of all windows of 30 seconds in it. But as MavVStream only takes a single window definition for all operators in a query, using max is not possible. This is because max will require a new window of windows to compute the max value of window, however as windows are already defined, max for each window

will be output by MavVStream which is not needed. Hence, we can plot a graph with counts for each window and check the peak of the graph to identify the busiest 30 seconds. As can be seen from Figure 8.3, the busiest 30 seconds are the first 30 seconds.

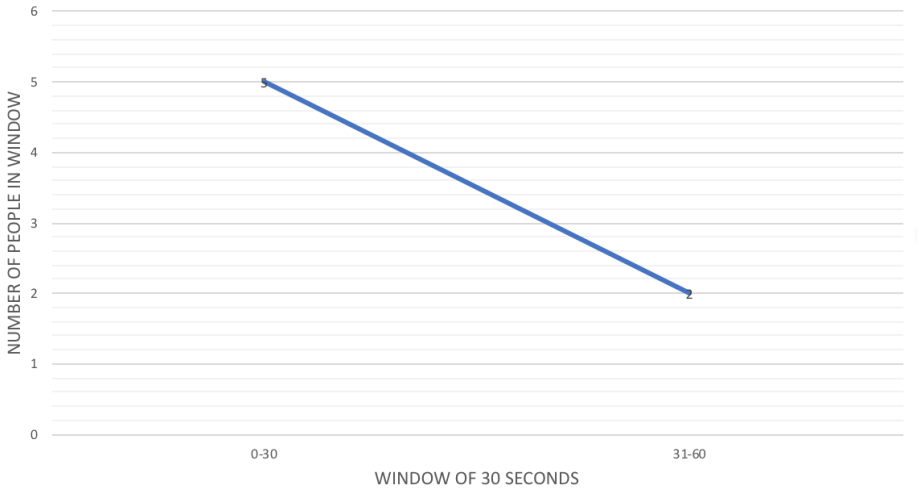


Figure 8.3: Output Graph for Query 2

3. **Indicate when a specific person (whose image or feature vector is given) entered the building.**

   This experiment was conducted on Video Dataset 2 and an sMatch function inside a Select operator was called. A feature vector for an instance of a person similar to Object 1 was passed as a parameter and a threshold value very low(*0.00032*) was kept such that it gives us the exact/few matches to the given feature vector. The operator query tree used is described in Figure 6.3

   Input feature vector belonged to Object 1.

   Intermediate Results obtained are shown in figure 8.4. As we can see, the object matched to nearest 5 instances of it.

Frame_Id	Obj_Id	PrevObjectId	TimeStamp	FV
196	79	1	7	fv1
197	80	1	7	fv2
198	81	1	7	fv3
199	82	1	7	fv4
200	83	1	7	fv5

Figure 8.4: Intermediate output for Query 3

Now we used a CCT(object_id,frame_id,first) on this output to extract first instance to figure out the exact timestamp at which the person entered.

Note here that there can be false positive results as well for some other cases, however for this case we didn't find any false positives.

Frame_Id	Obj_Id	PrevObjectId	TimeStamp	FV
196	79	1	7	fv1

Figure 8.5: Final output for Query 3

4. **Find out the directions in which the people are moving.**

   This experiment was done on Video set 1. There were four people in the video moving in different directions. Direction operator was applied on the stream and direction was computed based on the first and the last instance detected for each person. Based on their movement, their direction was computed.

   Results obtained are in figure 8.6. Some moving shadows and noises were also detected as part of the preprocessed data. However, after manually checking the video, Individual object Ids for peoples were detected and computation was detected as correct. As the dataset is small in size, the window size is kept such as to to include the complete video in a single window.

From Manual Checking of Video	Object Id	Direction of Movement	Timestamp
Person 1	**1**	**Object is moving in the West Direction**	**5**
Shadow/Noise	67	Object is moving in the North East Direction	6
Shadow/Noise	28	Object is moving in the East Direction	7
Shadow/Noise	202	Object is moving in the North East Direction	16
Person 2	**225**	**Object is moving in the East Direction**	**18**
Shadow/Noise	620	Object is moving in the South West Direction	20
Shadow/Noise	711	Object is moving in the South East Direction	24
Person 3	**445**	**Object is moving in the South East Direction**	**24**
Shadow/Noise	755	Object is moving in the North West Direction	25
Person 4	**752**	**Object is moving in the North West Direction**	**28**

Figure 8.6: Final output for Query 4

5. **Indicate when 2 different people (given images or feature vectors) enter AND leave with in 20 seconds of each other.**

   This experiment was conducted on Video Set 2 and Video Set 3. Feature Vectors of 2 people both entering and exiting a room were input to an sMatch function used inside a select operator. These were entered such that entry object 120 matches to exit object 15 and entry object 504 matches to exit object 383. The operator query tree for this query is described in Figure 6.4 .

   Results obtained are for a 20 second window and are shown in figure 8.7. If the window size is changed to 15 this result would not be achieved as difference between exit TS's is not less than 15 seconds.

Entry Fr 1	Entry Obj 1	Entry TS 1	Entry Fr 2	Entry Obj 2	Entry TS 2	Exit Fr 1	Exit Obj 1	Exit TS 1	Exit Fr 2	Exit Obj 2	Exit TS 1
213	120	8	546	504	19	98	15	33	548	383	48

Figure 8.7: Final output for Query 5

6. **Indicate when 2 different people (given images or feature vectors) enter OR leave with in 15 seconds of each other.**

This experiment is similar to experiment 5. Instead of an AND clause, here we have an OR clause, so either an entry or exit between 15 seconds is sufficient to get the result. It was conducted on Video Set 2 and Video Set 3. Feature Vectors of 2 people both entering and exiting a room were input to an sMatch function used inside a select operator. These were entered such that entry object 120 matches to exit object 15 and entry object 504 matches to exit object 383. The operator query tree for this query is described in Figure 6.6 .

Results obtained are for a 15 second window and are shown in figure 8.8. As can be seen here, difference from previous query is that, even for a smaller window, now we will achieve the same result as previous query due to an OR clause instead of AND.

Entry Fr 1	Entry Obj 1	Entry TS 1	Entry Fr 2	Entry Obj 2	Entry TS 2	Exit Fr 1	Exit Obj 1	Exit TS 1	Exit Fr 2	Exit Obj 2	Exit TS 1
213	120	8	546	504	19	98	15	33	548	383	48

Figure 8.8: Final output for Query 6

7. **List people who stayed for less than 30 seconds in a room.**

   This experiment was conducted on video Set 2 and video Set 3. To check the correctness of results, Object Ids of people were identified manually (described in figure 8.9 . The operator tree for this query is described in figure 6.3

ENTRY			EXIT	
NAME	OBJ_ID		NAME	OBJ_ID
Person 1	1		Person 2	15
Person 1	58		Person 5	217
Person 2	120		Person 5	236
Person 3	329		Person 4	383
Person 3	409		Person 1	530
Person 4	504		Person 3	695
Person 5	679			

Figure 8.9: Object Id's for People Entering and Exiting in Video Set 2 and 3

Results obtained are in figure 8.10. Some moving shadows and noises as well as false positives were also detected as part of the preprocessed data. However, based on the manual results from the video, Individual object id's for peoples were detected and computation was detected as correct.

Entry Frame	Entry Object Id	Entry TS	Exit Frame	Exit Object Id	Exit TS	System TS	Diff
130	120	5	54	15	35	1.52325E+12	30
651	504	22	514	383	50	1.52325E+12	28
655	504	22	514	383	50	1.52325E+12	28
657	504	22	514	383	50	1.52325E+12	28
724	679	25	317	217	41	1.52325E+12	16
724	679	25	335	236	44	1.52325E+12	19

Figure 8.10: Final output for Query 7

8. **Find if a car that crossed Point A also crossed Point B**

   This experiment was executed on Video Set 4 and Video Set 5. From Manual Observation, it was noted that Point A object Id was 3 and Point B object Id was 2.

Results obtained are described in figure 8.11. This query was executed using cJoin Grouping approach (i.e. Approach 2) and SIFT matching. Inference from the result is that car also crossed Point B.

Entry Fr #	Entry Obj Id	Entry TS	Exit Fr #	Exit Obj Id	Exit TS
36	3	1	12	2	11

Figure 8.11: Final output for Query 8

CHAPTER 9

CONCLUSION AND FUTURE WORK

This thesis discusses about how stream and event processing can be adapted to analyze situations on a class of videos. It also explains about the class of queries for which our approach will work. Furthermore, it discusses about how current existing data model is unable to express and process queries on video data and what changes are needed to be done. This thesis proposes the design and implementation of the required data model changes and how they are used in operators which are used to process process queries on videos. It demonstrates the design and implementation of operators that are specifically used for the processing of queries on extracted video content, however not limited to those, and how situations can be expressed in a non-procedural manner and detected. Experimentation results on situation queries using proposed data model changes and operators are then included at the end to prove the correctness of our approach.

This work in this thesis can be extended for detecting the non primitive events. This requires integration of current operators with event and rule processors and also come up with additional spatial and temporal operators to further process situation queries in an accurate manner and broaden the classes of situations that can be analyzed. Optimization and improvement of existing operators can also be done as part of future work.

REFERENCES

[1] R. Chellappa, "Frontiers in image and video analysis nsf/fbi/darpa workshop report," in *Workshop*, 2014, p. 120. [Online]. Available: www.umiacs.umd.edu/~rama/NSF_report.pdf

[2] (2008) Virat. [Online]. Available: https://en.wikipedia.org/wiki/VIRAT

[3] "Broad agency announcement, video and image retrieval and analysis tool (virat)," DARPA INFORMATION PROCESSING TECHNIQUES OFFICE (IPTO), Tech. Rep. BAA 08-20, 03 March 2008. [Online]. Available: https://www.fbo.gov/utils/view?id=32f2382440cfb57d2695171885acab57

[4] A. J. Aved, "Scene Understanding For Real Time Processing Of Queries Over Big Data Streaming Video," *Ph.D. Dissertation, UCF Orlando, Florida*, 2013.

[5] A. J. Aved and K. A. Hua, "An informatics-based approach to object tracking for distributed live video computing," *Multimedia Tools and Applications*, vol. 68, no. 1, pp. 111–133, 2014.

[6] B. Liu, A. Gupta, and R. C. Jain, "Medsman: a streaming data management system over live multimedia," in *Proceedings of the 13th ACM International Conference on Multimedia, Singapore, November 6-11, 2005*, 2005, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/1101149.1101174

[7] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[8] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and

S. Zdonik, "Retrospective on aurora," *VLDB Journal: Special Issue on Data Stream Processing*, vol. 13, no. 4, pp. 370–383, 2004.

[9] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, pp. 120–139, August 2003. [Online]. Available: http://dx.doi.org/10.1007/s00778-003-0095-z

[10] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Monitoring Streams - A New Class of Data Management Applications," in *Proceedings of the International Conference on Very Large Data Bases*, August 2002, pp. 215–226.

[11] S. B. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan, "The aurora and medusa projects," *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 3–10, 2003.

[12] M. Annappa, S. Chakravarthy, and V. Athitsos, "Pre-processing of video streams for extracting queryable representation of its contents," in *Advances in Visual Computing - 12th International Symposium, ISVC 2016, Las Vegas, NV, USA, December 12-14, 2016, Proceedings, Part II*, 2016, pp. 301–311. [Online]. Available: https://doi.org/10.1007/978-3-319-50832-0_29

[13] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems." in *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 2002, pp. 1–16.

[14] T. M. Ghanem, W. G. Aref, and A. K. Elmagarmid, "Exploiting predicate-window semantics over data streams," *SIGMOD Record*, vol. 35, no. 1, pp. 3–8, 2006.

[15] A. Lerner and D. Shasha, "-aquery: Query language for ordered data, optimization techniques, and experiments** work supported in part by us nsf grants iis-9988636 and n2010-0115586," in *Proceedings 2003 VLDB Conference*. Elsevier, 2003, pp. 345–356.

[16] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004. [Online]. Available: http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94

[17] G. Bradski, *Dr. Dobb's Journal of Software Tools*, 2000.

[18] MathWorks, *Image Processing Toolbox MATLAB R2016a Sample Video - atrium.avi*. MathWorks, 2016. [Online]. Available: http://www.mathworks.com/help/vision/examples/motion-based-multiple-object-tracking.html

BIOGRAPHICAL STATEMENT

Mayur Arora was born in Dehradun, Uttarakhand, India. He received his Bachelor of Technology degree in Computer Science and Engineering from National Institute of Technology, Hamirpur, India in May 2013. There after he worked as a Subject Matter Expert with Amdocs DVCI Pvt. Ltd., Pune, India , from July 2013 till July 2016. In the Fall of 2016, he started his graduate studies in Computer Science at The University of Texas, Arlington. He worked as an SDE intern at Bazaarvoice Pvt. Ltd., Austin, Texas from Fall - 2017 to Spring - 2018. He received his Master of Science in Computer Science from The University of Texas at Arlington, in May 2018. His research interests include stream processing , cloud computing and deep learning.