Materialized Views over Heterogeneous Structured Data Sources in a Distributed Event Stream Processing Environment

by

Mahesh Balkrishna Chaudhari

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2011 by the
Graduate Supervisory Committee:

Suzanne W. Dietrich, Chair
Susan D. Urban
Hasan Davulcu
Yi Chen

ARIZONA STATE UNIVERSITY

May 2011

ABSTRACT

Data-driven applications are becoming increasingly complex with support for processing events and data streams in a loosely-coupled distributed environment, providing integrated access to heterogeneous data sources such as relational databases and XML documents. This dissertation explores the use of materialized views over structured heterogeneous data sources to support multiple query optimization in a distributed event stream processing framework that supports such applications involving various query expressions for detecting events, monitoring conditions, handling data streams, and querying data. Materialized views store the results of the computed view so that subsequent access to the view retrieves the materialized results, avoiding the cost of recomputing the entire view from base data sources. Using a service-based metadata repository that provides metadata level access to the various language components in the system, a heuristics-based algorithm detects the common subexpressions from the queries represented in a mixed multigraph model over relational and structured XML data sources. These common subexpressions can be relational, XML or a hybrid join over the heterogeneous data sources. This research examines the challenges in the definition and materialization of views when the heterogeneous data sources are retained in their native format, instead of converting the data to a common model. LINQ serves as the materialized view definition language for creating the view definitions. An algorithm is introduced that uses LINQ to create a data structure for the persistence of these hybrid views. Any changes to base data sources used to materialize views are captured and mapped to a delta structure. The deltas are then streamed within the framework for use in the incremental update of the materialized view. Algorithms are presented that use the magic sets query optimization approach to both efficiently materialize the views and to propagate the relevant changes to the views for incremental maintenance. Using representative scenarios over structured heterogeneous data sources, an evaluation of the framework demonstrates an improvement in performance. Thus, defining

the LINQ-based materialized views over heterogeneous structured data sources using the detected common subexpressions and incrementally maintaining the views by using magic sets enhances the efficiency of the distributed event stream processing environment.

I dedicate this dissertation to my father, Balkrishna Chaudhari,

my mother, Usha Chaudhari

and my wife, Seema Talele.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all those who have given a helping hand in my journey to achieve this major goal of my life.

This research and dissertation would have been incomplete without the invaluable guidance and support from my advisor Dr. Suzanne Dietrich. She has played vital role in guiding me to shape my research and my dissertation. She has been an excellent mentor by sharing her knowledge as an expert database researcher as well as an excellent educator. I am grateful to her for showing confidence and trust in me while I performed my job responsibilities as the lab assistant as well as an instructor for the undergraduate classes. I am also grateful to Dr. Dietrich and National Science Foundation to provide me with the opportunity to work on a funded research project that has shaped my dissertation to perfection. Jennifer Ortiz will always be remembered as my colleague who has helped me by listening to my thoughts and ideas, provided me with valuable suggestions and has assisted me in developing prototypes for my research.

I am also thankful to my committee members i.e. Dr. Susan Urban, Dr. Hasan Davulcu, and Dr. Yi Chen for their support and guidance during various stages of my research. I am very grateful to Dr. Roger Berger and the Division of Mathematical and Natural Sciences to provide me with a career-enhancing opportunity to teach ACO 102 course for 2 semesters at the West campus, ASU. I would like to thank Dr. Anshuman Razdan for providing me valuable support at times when I needed it the most during my early years at ASU. I am grateful for him to provide me with an opportunity to work on interdisciplinary research projects in PRISM.

There are many inspirational people who have boosted confidence in me to keep pushing things one step at a time. Dr. Gerald Farin and Dr. Dianne Hansford have continuously inspired me to continue working on my research and they have supported me in numerous different situations. One key confidence booster has been my brother-in-law Dr. Bhalchandra Chaudhari, whose dedication and perseverance taught me to

follow his footsteps and plunge into the exciting journey of the deep ocean of search and research of PhD.

I dedicate this dissertation to my parents, Balkrishna and Usha Chaudhari. They have always been there to morally support me and provide guidance in my pursuit of higher education after my Bachelors degree. My dad and my mother have been the beacons of light in my path of struggle and endurance. Both of them have taught me important lessons of life and how to balance between enjoyment as well as responsibilities.

Finally, this acknowledgement section cannot be complete without a special thanks to my wife, Seema. She has been of immense help and courage to me. She has shown lot of patience in hearing my single answer "Not Yet!!!" to her repeated single question of "Are you done yet?". Her love and care has given a definitive structure to my life. She has always kept a warm welcoming heart during ups and downs in my adventure.

I am thankful to god to introduce me to such loving, caring, dedicated and exhilarating people who have all contributed to this research and dissertation directly and indirectly.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Database applications are becoming increasingly dynamic, requiring the monitoring of streaming data and events in a distributed environment. In the past, research was individually focused on continuous queries over streams [1, 2] and distributed event processing [3]. In stream processing applications, data are received continuously and ordered by their arriving timestamps [4]. The streaming data, which generally arrives at a very high rate, is processed using continuous queries with filtering capabilities and access to the persistent data sources. In contrast to streams, a primitive event is defined as an atomic and instantaneous occurrence of interest at a given time [5]. Composite events detect complex and meaningful relationships among the multiple occurrences of primitive events. In event processing applications, active rules define the behavior of the application in response to the occurrence of the events.

Although event processing and stream processing are similar in the aspect of consuming the generated information, each paradigm is designed for different functional purposes. Stream processing involves the execution of continuous queries over a large volume of data generated at a high rate to extract meaningful information. Event processing typically deals with a lower volume of data or primitive events to establish correlations that form composite events. Thus, stream processing can serve as a processing step before event processing by reducing the volume of relevant data for the event handler. There are many dynamic applications, in domains such as financial stock market monitoring or criminal justice, which have a need to define events over streaming data with access to heterogeneous data sources. Thus, the integration of steams and events is now receiving attention in the research community [6–9].

The framework proposed in this research uses event stream processing as a fundamental paradigm. The distributed nature of the framework with support for access to heterogeneous data sources originated from an active database perspective to use

events and rules to build enterprise applications [10]. The application of the proposed framework to heterogeneous data sources results in a system that may be considered a dataspace support platform [11] where data sources co-exist and the framework provides the developer with an environment that supports the challenges of building their application over disparate and distributed data sources. Dataspaces differ from data integration approaches [12] that semantically integrate the data sources using a common mediated schema. This research is novel in that it utilizes event stream processing within the framework of the dataspace system.

1.1 Event Stream Processing and Materialized Views

A stream and event processing system consumes multiple data streams as well as event streams. These streams need not be generated in the same application. The streams might originate from other event and stream processing systems, sensors, or data acquisition units. Thus, in a distributed event stream processing environment, in addition to streams generated in-house, a system can subscribe to the data and/or event streams generated by other systems. The subscriber does not have any other control over the subscribed streams other than how to consume the data/events coming on the streams. In the processing of the streams, the system can generate new output streams and these streams can be subscribed to by the other event stream processing nodes in the distributed system.

To understand how to process the streaming information, event and stream processing systems need to maintain metadata information regarding the streams. Other components that are typically used along with the streaming information are the continuous queries with expressive filters, primitive and composite event definitions, and active rules associated with systems. Continuous queries with filters are the queries that are executed on every incoming data stream to filter out unwanted information. This also reduces the rate at which the data is made available for the system components to consume. The Continuous Query Language (CQL) allows filters to be defined

over the streams in conjunction with the relational databases to filter out unnecessary data [1]. Event definitions are maintained in the system to capture the atomic occurrences of interest in a given period of time. Active rules are defined in the system to take appropriate actions whenever an event is detected and the optional condition associated with the event is true. Active rules that contain all the three components, i.e. Event, Condition and Action, are called Event-Condition-Action (ECA) rules. In addition to ECA rules, the active rules can take the form of either EA (Event-Action) or CA (Condition-Action) rules [13]. ECA and EA rules are fired whenever the event occurs. However, for CA rules, since there is no way to invoke the rule automatically, a continuous condition monitoring system is necessary to check when the condition of a CA rule is satisfied.

One goal of this research is to allow these different components (continuous queries, filters, event definitions, and rules) to interact with each other seamlessly and efficiently. Continuous queries and filters have the capability to query data from the persistent relational and/or XML databases. In addition, active rules can have conditions that query information from the database and perform actions involving changes to persistent databases. This introduces an important concept of materialized views defined over these persistent sources. In relational databases, a view is a derived relation over a set of base tables or relations. A view in XML databases is a derived XML tree over a set of XML documents. A materialized view is a view in which qualifying tuples are stored in the database and indices can be defined over this view [14]. Thus, in the case of a non-materialized view, tuples are re-computed from the base tables every time a query uses that view. Whereas in the case of a materialized view, the view is populated when it is defined and retrieved upon reference to the view. Hence, there is no need to re-execute the view definition over the base tables every time. Since the view is populated, executing a query over a materialized view is typically faster than executing a query over a non-materialized view. A materialized view is comparable to

a cache mechanism, where the data is kept ready all the time.

A limitation of keeping materialized views is analogous to the problem of keeping a cache coherent. Whenever updates occur to the underlying base tables, it is important to make these changes visible to the materialized view. Updating the materialized view based on the changes in the underlying database is called view maintenance. However, it is expensive to re-compute the entire view every time there is a change to the base tables. Therefore, the materialized view is incrementally updated by propagating these changes to the materialized view. This approach is called incremental view maintenance and is used for materialized views over relational databases [15–17] as well as over XML documents [18, 19].

These changes to the underlying database are also useful in the execution of Condition-Action (CA) rules. In the case of CA rules, there is no explicit event that triggers the check of the condition over the database. Thus, the system has to continuously monitor the changes in the system to verify whether the condition specified in the CA rule is satisfied or not. Thus, in this case, similar to incremental view maintenance, it is not desirable to check the condition every time there is a change in the current database state. Once the CA rule is defined in the system, the condition is checked at that time with the database state at that instance. After that, the changes made to the underlying database are then used incrementally to check whether the condition becomes true or not. This is called condition monitoring. Condition monitoring algorithms have been explored for active relational databases [13] and object-oriented databases [20]. Thus, the view maintenance and condition monitoring algorithms use the changes to the underlying database to incrementally update the views, and check the conditions respectively.

1.2 Overview of the Distributed Event Stream Processing Framework

The research challenges involving the incremental maintenance of materialized views over heterogeneous distributed data sources in an event stream processing environment

are examined within the context of Distributed Event Stream Processing Agents (DEPAs). The concept of a DEPA was first envisioned in [10] but not implemented. This research contributes a proof of concept implementation of a framework built using a DEPA as a fundamental building block, establishing an environment for this exploration [21]. Specifically, a DEPA can subscribe to different event or data streams. DEPAs can handle data streams from different data sources, including application event generators, the output from continuous queries over sensor data, and streams of incremental changes from databases, such as the Oracle Streams [22] or Change Data Capture feature for SQL Server 2008 [23] for monitoring database log files. The event or data streams can be either in relational format or in a structured XML format. Processing these events or data streams often requires access to persistent data sources such as relational databases and XML data.

This dissertation explores the use of materialized views over heterogeneous structured data sources in such a distributed event stream processing environment as shown in Figure 1.1. A materialized view is a query whose results are computed when defined and stored in the database with appropriate indices [15]. Upon reference to the view, the materialized results are retrieved instead of recomputed. Each DEPA maintains its own set of resources, such as data and event streams, active rules, relational and structured XML data sources to which the DEPA subscribes. Even though DEPAs communicate with each other to exchange information, it is crucial to define materialized views for each individual DEPA for efficient stream processing, composite event detection, and the execution of active rules. The vision is that each DEPA will have a specific responsibility within the distributed system, which will increase the probability of common subexpressions on which to define materialized views for improving performance. Providing materialized views closer to the specific responsibilities is expected to reduce the overall latency of the event stream processing system [24].

The DEPA prototype uses Sybase CEP as the event stream processor. This CEP

engine has its own Continuous Computation Language (CCL) for defining different data streams and event streams [25]. CCL has SQL-like syntax and the streams can be either in relational or XML format. Along with stream definitions, CCL also provides the capability of defining continuous queries and composite events with temporal and windowing capabilities over the streams with access to external persistent data sources, such as relational databases and XML documents.

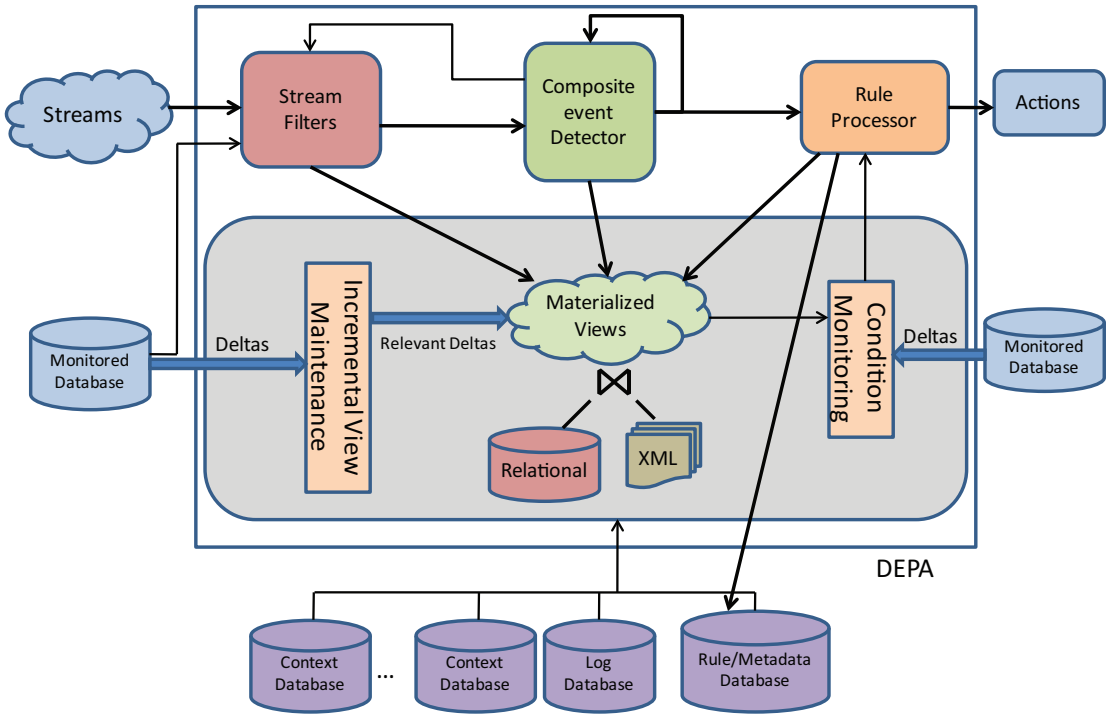


Figure 1.1: DEPA architecture.

This dissertation is exploring a suitable materialized view definition language to define views over the heterogeneous data sources in the DEPA framework. These views will retain the data in their original format and hence the views can be either purely relational or purely XML or a hybrid view over relational and XML data sources. Within the DEPA framework, the event and stream processing often requires access to heterogeneous structured data sources at the same time. Figure 1.2 illustrates a scenario for a query that can process streaming data and events along with access to one relational and one XML data source. However, there are research challenges involved in using

such a query language as a common platform to access heterogeneous data sources. This query language should have the capability to access a collection of: objects in memory or an object-oriented database, tuples in a relational table, or elements in an XML document through a single query.

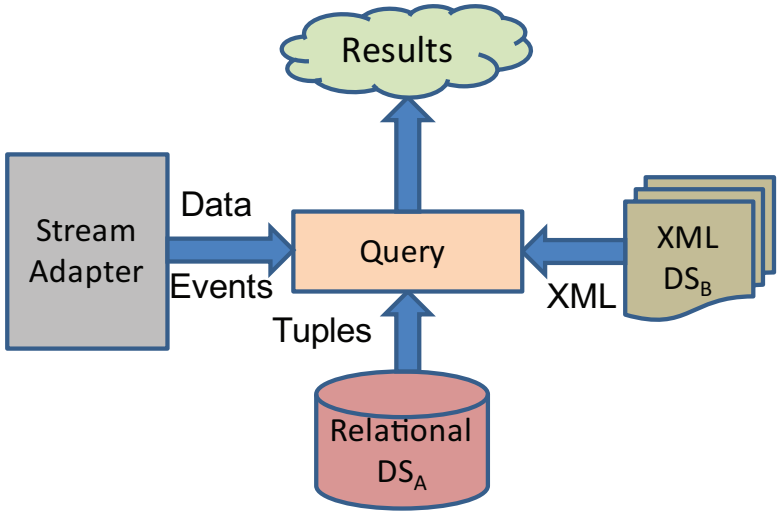


Figure 1.2: Processing streams using query over heterogeneous data sources.

This unifying paradigm of such a query language will also provide a candidate materialized view definition language for the event stream processing framework. In this inherently distributed and heterogeneous environment, the use of materialized views, especially local to a DEPA, is investigated as an optimization technique to speed up multiple query processing. For example, a materialized view can be defined over the relational data source DS_A and XML data source DS_B such that part of the view contains relational data and part of the view contains XML-based hierarchical data. This hybrid materialized view is shown in Figure 1.3 where the original query is rewritten to access the materialized view instead of the base data sources. Determining suitable data structures to store the hybrid materialized views are a contribution of this research.

Finally, this research is also exploring research challenges involved in capturing changes over the base data sources in their native format. These changes can then be used to incrementally update the materialized views or for condition monitoring of the

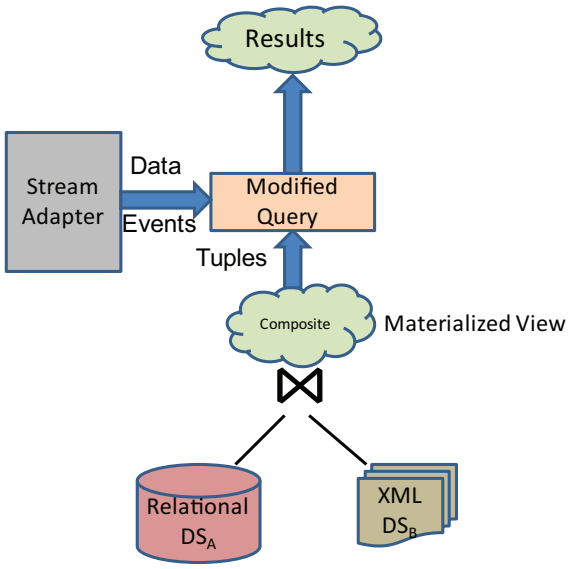


Figure 1.3: Processing streams using modified query and a materialized view over heterogeneous data sources.

Condition-Action (CA) rules. The captured changes can be modeled as data streams so that they can be transported to the respective agent. Thus, there are various specific research challenges involved in developing a distributed event stream processing framework with the capabilities of defining and maintaining materialized views over heterogeneous data sources.

1.3 Research Objectives

A major aspect of this research is the exploration of the use materialized views that are local to the DEPAs for efficient processing of events and data streams. There is a well-known trade-off with the use of materialized views and their maintenance. The materialized view must be updated when the data on which it depends has changed. Rather than recomputing the entire view, an incremental strategy is preferred to modify the materialized view based on the changes to the underlying data sources [[14, 26]]. This work investigates the incremental update of the materialized views within the distributed event stream processing framework.

Prior research on incremental view maintenance for heterogeneous data sources

has converted the different data sources to one format, either the XML data was converted into relational data [27] or the relational data was converted into XML data [28]. Once all the sources were in one format, then the materialized views were defined and established incremental view maintenance approaches were used. However, these techniques require the overhead of converting one source into another. Additional mapping information is also required to reconstruct the results from the converted format to the original source format. One challenge of this research is the investigation of the use of a materialized view definition language that supports a view defined natively over heterogeneous relational and XML data sources, respecting the underlying technology associated with each data store.

Another challenge is the examination of incremental view maintenance in this loosely-coupled distributed event stream processing framework. The changes or deltas to the underlying data sources required to incrementally update the view can be modeled as streams in a DEPA. There are existing mechanisms to individually capture the changes to relational or XML data sources. For example, Oracle has Oracle Streams and SQL Server 2008 has Change Data Capture to identify the changes occurring in the relational database. For XML files, there are change detection algorithms to capture the changes occurring in the XML documents [29,30]. This research explores the challenges involved in modeling the captured deltas in their native format as streams to incrementally update the materialized views.

The perspective of introducing agents for processing distributed events and streams with access to heterogeneous structured data sources with support for incremental view maintenance introduces several specific research challenges:

1. *Dependency analysis across different filtering queries to identify common subexpressions as potential candidates for materialized partial joins*

   An incremental approach to the evaluation of the resulting language that integrates streams, events, and persistent data is essential. Within each DEPA, there

are various query expressions that are accessing the heterogeneous data sources, including continuous queries, queries associated with composite event definitions and active rules as well as queries over persistent data sources. By exploring multiple query optimization across these query expressions, common subexpressions can be detected as the potential candidates for materialized views. The candidate materialized views can represent partial joins across relational and XML data sources.

Multiple query optimization and detecting common subexpressions for Select-Project-Join (SPJ) queries have been explored extensively for SQL queries [[31–34]]. Multiple query optimization reduces memory consumption and improves performance when processing multiple queries. It is a well-known fact that identifying common subexpressions is an NP-hard problem and hence it can be addressed by using a heuristic approach [[31, 32]]. Thus, one of the objectives of this dissertation is to define a multigraph model to represent the different query expressions in a DEPA over relational and XML data sources in the same graph model and to explore multiple query optimization by detecting common subexpressions across these query expressions.

To facilitate multiple query optimization by detecting common subexpressions, each DEPA in the framework requires metadata-level access to the heterogeneous data sources, event and data streams and parser-level access to the registered query expressions. Also, DEPAs communicate with each other using event and data streams to exchange information. DEPAs can request information regarding their registered data sources or streams for their functioning. Thus, it is imperative that each DEPA maintain its own independent metadata repository, which will facilitate in the various functions of the DEPA. This research is exploring the challenge in building and maintaining this metadata repository. The repository should be self-contained and maintained in response to any changes occurring at

the metadata level to the original data sources. The metadata repository should also have access points through which each DEPA can provide access to other agents. Since the DEPAs communicate peer-to-peer in a distributed environment and expose information to other DEPAs, a Service-Oriented Architecture (SOA) is a suitable technology to develop the DEPA framework. One of the research challenges of this work is to investigate different SOA-based technologies and decide which approach is suitable for the environment.

2. *Techniques for selectively materializing the partial joins over relational as well as XML data sources*

   The dependency analysis of the various query expressions leads to identifying common subexpressions as potential candidates for materialized views. One research challenge is the design of a heuristic algorithm that will selectively choose partial joins over heterogeneous data sources that will be beneficial to materialize, using a cost-based approach in a distributed environment.

   These materialized views can be purely relational or purely XML or a hybrid view containing both relational and XML data. In the past, for defining materialized views over relational and XML data sources, different research approaches have first converted one data format into another and then materialized views are defined. This dissertation is focusing on relaxing this conversion constraint to retain the data in their native format and take advantage of the underlying well-established algorithms and mechanisms for each type of data source. One other research objective is to investigate an appropriate materialized view definition language that will facilitate the definitions of such materialized views. This view definition language should be able to access the metadata repository to define these views and provide a mechanism to persist the views. The proposed design for addressing research challenges 1 and 2 is shown in Figure 1.4.

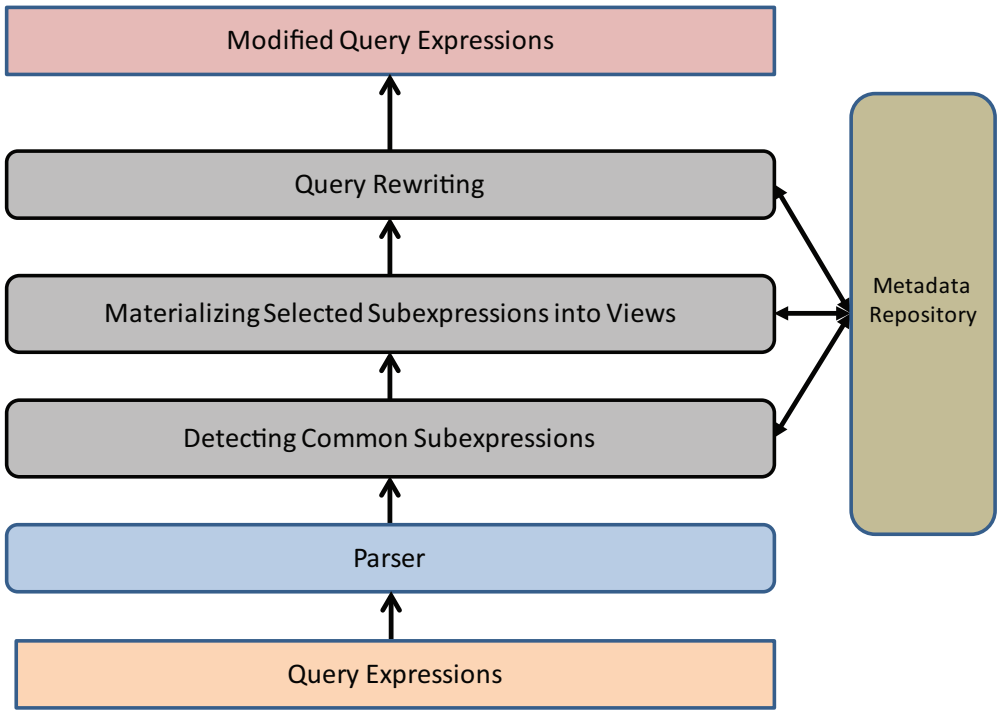


Figure 1.4: Multiple query optimizer.

3. *Incremental Evaluation and Materialized Views for Integrating Streams, Events, and Persistent Data*

   Capturing of deltas or changes to the original data sources is important in incremental view maintenance of the materialized partial joins. Deltas can be captured for Oracle Server using Oracle Streams or for SQL Server 2008 using the Change Data Capture feature. For capturing changes occurring in XML documents, there are various `diff` algorithms to generate deltas. Another research challenge is the capture of the heterogeneous distributed deltas in the native format of the data source and the subsequent use of the delta to incrementally update the materialized view. The general concept of using deltas in their native format to update the view is shown in Figure 1.5.

   In the past, research has focused on incremental view maintenance for relational views or XML views separately. For the views that were defined over relational and XML data sources, the deltas were converted from one format to the other

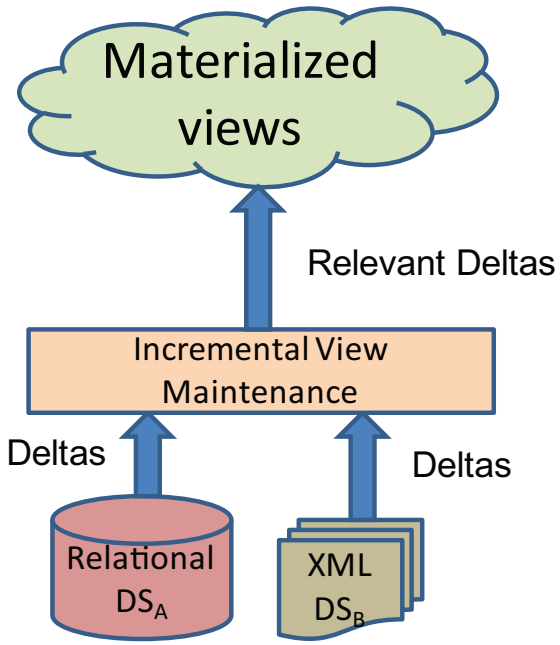


Figure 1.5: Using deltas in native format for incremental view maintenance.

and then used in the incremental maintenance of the views. This approach has similar conversion drawbacks as the views defined over different data sources. One of the objectives of this research is to retain the deltas in their original format and use them in incremental view maintenance. Also, since the data sources are distributed in nature, this research is focusing on streaming these changes from their original source to the relevant agents that need them for view maintenance.

1.4 Dissertation Overview

The objective of this dissertation is to describe the research contributions resulting from the investigation of the incremental maintenance of materialized views over structured data sources, such as relational databases and XML data, within a distributed event stream processing framework. This first chapter has provided an overview of the specific research challenges for incremental view maintenance within the DEPA framework and a proof-of-concept prototype of the framework. Related work in the areas of event stream processing, detecting and using common subexpressions for multiple query optimization, and incremental view maintenance is described in chapter 2. The

detection of common subexpressions over various query expressions within a DEPA requires extensive metadata for each query language, heterogeneous data sources and streams. Chapter 3 describes the design and implementation of a service-based metadata repository for the different data sources and the query expressions. It also provides details on the different services provided by the DEPA to maintain the repository and provide access to the metadata-level information. The different query expressions registered with a DEPA access heterogeneous data sources, such as relational and structured XML data sources. In order to detect common subexpressions across these different query expressions, they have to be represented in a single graph model. Chapter 4 provides details on such a mixed multigraph model that can represent queries over relational and XML data sources. A heuristics-based algorithm to detect common subexpressions as partial joins over relational and XML data sources is also presented in this chapter with a working example over the Criminal Justice data model. These detected common subexpressions can be selectively materialized into views using a materialized view definition language. These views can be used to answer certain queries instead of their base data sources. Chapter 5 provides details on the materialized view definition language and the algorithms used to define the materialized views. This chapter also provides details on how these views are persisted and illustrates the working of these algorithms and the view definition language with an example. The data sources registered with each DEPA may change over time and hence these changes should be captured and used to incrementally update the materialized views defined locally to the DEPA. Chapter 6 provides details on the mechanisms used in the DEPA framework to capture these deltas in their native format and to propagate any relevant changes to the materialized views. The deltas and their use in the incremental view maintenance are illustrated using a detailed example. Details on developing the DEPA environment, its prototype and performance evaluations of the different algorithms designed to define and maintain materialized views over heterogeneous data sources locally to each DEPA

are provided in chapter 7. Finally, the dissertation concludes with chapter 8 that provides a summary of the unique contributions of this research to the database community and a discussion on future research directions that can be pursued with respect to the distributed event stream processing environment.

Chapter 2

RELATED WORK

This research is exploring the use of an agent-based distributed event and stream processing framework that integrates querying heterogeneous data sources and using materialized views for multiple query optimization. The objective of this chapter is to provide high-level details on relevant related research. The DEPA framework uses event and stream processing as the backbone for all the communication between the different agents. Section 2.1 overviews the capabilities typically provided by event and stream processing systems. The DEPA framework is a loosely-coupled system that may be considered a dataspace support platform where data sources co-exist and the framework provides an environment that supports developing applications over heterogeneous distributed data sources. Section 2.2 provides an overview of the shared goals of dataspaces and the DEPA environment. The use of materialized views for multiple query optimization within such a framework requires the identification of common subexpressions. Section 2.3 discusses the related work on the identification of common subexpressions for relational queries and XML queries individually. A contribution of this dissertation, as discussed in chapter 4, is the identification of common subexpressions for queries that combine relational and XML data. Once the common subexpressions are detected as hybrid joins over these heterogeneous data sources, they are materialized into views for efficient query processing. Base data sources of these materialized views may change. These changes must be captured and used to incrementally update the materialized views instead of a complete re-materialization. Prior research has explored incremental view maintenance with respect to views defined over only relational or only XML data, which is described in section 2.4. These approaches first convert one data format into another and then define the materialized views over the homogeneous data sources. This dissertation is exploring the research challenges with defining hybrid materialized views over relational and structured XML data sources. To

define such views, an initial investigation has led to the choice of Language INtegrated Query (LINQ) as the materialized view definition language. Section 2.5 provides a high-level overview of LINQ, its syntax and illustrates its querying capabilities with samples queries.

2.1 Events and Stream Processing

Streams can be considered as inflow or outflow of data, events or combination of both. These streams need to be monitored and consumed at not necessarily real time but under time constraints. Thus, monitoring applications are becoming more and more important in day to day activities. Consider military applications for monitoring locations or other sensors, or consider health-related applications where sensors are used to monitor patients. Even monitoring data continuously in finance-related applications is crucial for corporations. In all these applications, the sensors send a continuous stream of either data or events or both, which are to be analyzed and processed under time and memory restrictions. Current database management systems are not suitable to handle such high volume of data under time and memory restrictions. The monitoring applications have the characteristic of informing about abnormal activity or change in the monitored data. Hence such applications are trigger-oriented. Monitoring applications require a history of values rather than just most recent values from the streams to run their queries, which cannot be achieved easily with traditional database management systems. Sometimes the streamed data is often lost, or is unreadable or sometimes deliberately omitted for processing reasons. Thus, monitoring applications should be able to handle such scenarios and still provide the user with approximate answers based on whatever data is available.

Several event stream processing systems have been developed to process streams using continuous queries and composite event detectors. Some of the open source research prototypes are Aurora [4], Cayuga [7], CEDR [6], Tesla [35], and XChange^{EQ} [36]. There are also commercial event stream processing systems, such as Streambase

[37] and Sybase CEP [25]. As an integral part of the DEPA framework design, this research uses the Sybase CEP as the event stream processor. The remainder of this section describes the main features of event stream processing systems, using only a few of the above referenced systems as examples.

The Aurora system is an event stream processor that continuously monitors streaming data coming from computer programs or hardware sensors [4]. Each stream source has a unique source identifier and the system timestamps the tuple or the record coming through every streaming source. The Aurora system defines a variety of primitive operators for stream processing. There are windowed operators that work on a set of consecutive tuples within a given time interval called a window. In addition to these windowed operators, there are tuple-level operators that work on a single tuple at a time. These different operators can also be classified as either blocking or non-blocking operators. A blocking operator needs to have its entire input data before it can generate any output, whereas, a non-blocking operator can process either an individual tuple or a window of tuples and will continue to generate output. The Aurora system allows the composition of these primitive operators to obtain desirable results. The Aurora Query Model assumes a centralized environment that supports continuous queries, materialized views over relational data sources and ad-hoc queries each of which has process flow based on QoS (Quality of Service) specifications (several performance and quality related attributes).

The Aurora system assumes that the streams are comprised of relational data. However, streams can also be event streams comprising of simple events or composite events. The applications working on such streams detects the occurrence as well as non-occurrences of simple or composite events. Hence, the event language needs to be able to query the streaming events similar to the filtering or querying mechanism that exists in Aurora system. The streaming events are filtered to remove irrelevant primitive events and correlated for composite event detection. However, this becomes difficult

and complex to achieve because of short-life nature of the streaming data and long-lived composite events. SASE (Stream-based And Shared Event processing) is one such event definition language that process real-time data from RFID streams [38]. One advantage for such events coming from the streams is that they are initially time stamped like data streams before they can be consumed. This allows the consumer to look at the true discrete ordering of the events. The events are declared using the declarative event language SASE that combines filtering, correlation and transformation of events [38]. The syntax of this event language is shown in Listing 2.1.

```
EVENT <pattern>
[WHERE  <qualification>]
[WITHIN <window>]
```

Listing 2.1: SASE event syntax.

The event pattern can be either a simple event name or a sequence of multiple simple events to denote a composite event. The WHERE clause helps in filtering the events if the predicate evaluates to false and WITHIN indicates the time frame of the sliding window to be considered. The non-occurrence of an event is indicated by negating that particular event in the composite event description.

In the case of XML streaming data, the XML data can be either a sequence of small sized tuples or a sequence of large semi-structured tuples. It is possible that the entire set of tuples required for processing a query is not available immediately. Also the XQuery queries written over such XML data can be recursive; hence they might need more than one pass over the data, which is not possible all the time. The system developed at Ohio State University has a code generation base that rewrites the XQuery query plan, whenever possible, to execute in one single pass over the dataset [9]. The system analyzes the dependencies in the query using a stream data flow graph and then applies a series of high-level transformations. Based on this analysis, not all the queries can be executed in a single pass. The XQuery queries that involve blocking operators (sorting) with unbounded input and queries that involve progressive blocking operators

(aggregate functions) with unbounded input are the two types of queries which cannot be rewritten to execute in one pass over the dataset.

The communication between the agents in the DEPA framework uses events and data streams, in both relational and XML format. The DEPA framework utilizes Sybase CEP (formerly known as Coral8) as its event and stream processor. Sybase CEP uses CQL queries within the DEPA framework to process events and data streams with the ability to define filters with access to persistent data sources, such as relational databases and XML documents. Data streams are also used to model the flow of captured deltas over the relational and XML data sources from their origin to the relevant agents in the framework for incremental view maintenance and condition monitoring.

2.2 Dataspaces

Recently, the concept of dataspaces has been introduced to access loosely-coupled heterogeneous data sources that are managed locally but are integrated under a common data application [39]. Typically, these heterogeneous data sources are not under the control of a single data management system. The distributed enterprise applications subscribe to these data sources as needed to produce the desired results. Application domains, such as digital libraries, personal information management (PIM), criminal justice and homeland security departments can benefit from these types of application architecture that provides seamless integrated access to loosely-coupled heterogeneous data sources.

Dataspaces provide data management abstraction for the above mentioned application areas and Dataspace Support Platforms (DSSP) provide different abstracted views of the data along with different services [11, 40]. Many different dataspaces can combine to form a network of dataspaces resulting in a larger dataspace or remain as an individual dataspace just interacting with dataspaces under the canopy of DSSP. Since each dataspace registers to all the data sources that are needed for it to function autonomously, it is possible that multiple dataspaces share some common data sources.

With this focus, the first challenge is to provide a querying capability over the relevant data sources registered to the dataspace [[11, 40]]. Hence, it is essential to create a central metadata repository that can be queried at any time instead of querying the data sources individually for their metadata whenever needed. The interesting research challenges that are closely related to the proposed research are of maintaining the data once the data sources are registered with the dataspaces and handling the changes that occur in the system.

Once the resources are registered with a particular dataspace, any subsequent access to the data sources has to guarantee access to the latest data values from the data source. Since dataspace do not have full control over the data sources, changes can occur at the data source locations, which can be distributed. It is very crucial for the dataspace to maintain and provide an updated copy of the data as part of the services the dataspace provide. Thus to maintain such updated copy, dataspaces need to have rules, integrity constraints, unique naming conventions across the data sources, recovery and access control and housekeeping optimized data structures for data as well as metadata [[11, 40]]. Along with getting updated information from the autonomous and heterogeneous data sources, future research work in dataspaces involves providing the capabilities of condition monitoring, complex event detection and complex work-flow patterns. It is quite possible that not all the data sources will be able to provide metadata in the format needed by the dataspaces. Hence, it becomes necessary to maintain an independent metadata repository. Along with maintaining the metadata repository, dataspaces should provide a querying mechanism to find sources that contain specific elements or attributes. Once the metadata repository is created, then this information can be used along with the incremental view maintenance of user defined views and condition monitoring of condition-action rules defined at the dataspace level.

The concept of dataspaces is analogous to the concept of DEPAs. They both share the common goal of working over loosely coupled heterogeneous data sources

without the need of a global schema. Although not inherent to the current framework of a dataspace, dataspaces can benefit from the incorporation of complex event stream processing over data and/or event streams. The following list summarizes the common features of DEPAs and dataspaces in the integration of heterogeneous data sources.

1. Both target autonomous, heterogeneous data sources such as relational databases, XML databases, events and data streams. However, dataspaces are targeted to accommodate even less-structured data sources such as emails and flat files.

2. They both subscribe to other data sources along with the data generated at their own site.

3. Both need to maintain a local metadata repository for different language components and data sources.

4. For efficient search and query mechanisms, both can benefit from materialized view definitions over persistent data sources.

5. To keep the materialized views updated with the changes, both need incremental view maintenance algorithms to synchronize the views with the underlying data sources.

6. Other key services for both include composite event detection, active rules, and condition monitoring to accommodate event detection across multiple DEPAs or dataspaces.

Even though dataspaces and DEPAs are closely related to each other, dataspaces do not satisfy all the requirements of a typical DEPA. Dataspaces are more generalized and targeted towards textual querying of all the data sources. A DEPA can subscribe to streaming sources whereas dataspaces do not have the provision for processing of streaming data. Finally, in the literature available for dataspaces, there is no indication

of the related work on dataspaces that define materialized views over heterogeneous data sources without converting one source format to the other.

2.3 Multiple Query Optimization

In both the relational as well as the XML databases, materialized views are defined over underlying original data sources (tables in relational databases and hierarchical documents in XML databases) and the validate-propagate-update mechanism is used to update these views. In case of views over relational databases, the view definition mainly comprises of simple or complex Select-Project-Join (SPJ) queries. In case of views over XML data sources, a view definition is an XQuery expression encapsulated within a unique root node, which is the view name.

Hence, every view definition consists of a query expression of either a Select-Project-Join expression or an XQuery expression. The collection of such query expressions in a database environment leads to the important concept of Multiple Query Optimization (MQO). MQO optimizes a given set of queries such that the common subexpressions from the query definitions are detected and executed only once [31]. This helps in less consumption of memory and speeds up the processing of multiple queries over single-query processing. It is a well-known fact that identifying common subexpressions is a NP-hard problem and hence it can be addressed only by using a heuristics approach [31]. The main steps in MQO comprises of detecting the common expression either in the form of sub-query, or common data source, or operation and then using these common expressions to construct a global execution plan to facilitate the Multiple Query Processing (MQP). First, the multiple SPJ queries expressed in relational algebra are represented in the form of multiple graphs, and then a series of transformations are applied to these graphs with the help of heuristic rules for detecting common subexpressions to construct a global execution plan [31].

Before detecting the common subexpressions, a given set of queries is classified into different sets of unrelated queries. This guarantees that each set of queries that is

analyzed for common subexpressions has at least one data source in common. Then the queries in each set are executed separately. This helps in reducing the number of queries to be considered for detecting common subexpressions. For any given set of queries, there can be four types of common subexpressions: 1) nothing in common; 2) exactly the same; 3) subsumption; and 4) overlap. Multigraphs generated for every SPJ query are executed using a contraction process [31]. Whenever a relational operation is executed, then the nodes and/or the edges related to that operation are reduced to form a new node. As the relational operations continue to be executed, at the end only one node per query will remain. This node represents the result for that query.

Detecting common subexpressions is important for both MQO as well as for rule execution in active database systems. In active databases, the rules can be represented as rule execution graphs. In these graphs, the intermediate nodes of the SPJ query expression in the condition part of the rule can be viewed as potential candidates for partial joins or views. In a given set of rules in an active database system, efficient pattern matching algorithms can detect common nodes or patterns in the rules that will lead to faster and efficient execution of the rules [41]. These patterns or common subexpressions can be materialized into partial joins and later used to optimize the rule execution. RETE and TREAT are the well-known pattern matching algorithms for active database systems. The RETE algorithm retains the intermediate temporary results for the selection conditions known as $\alpha-$memory nodes and for the join conditions known as $\beta-$memory nodes [41]. Thus, the main advantage of RETE algorithm is that it can reuse these intermediate temporary results for other rules. However, this advantage is its biggest limitation because as the data size grows, the memory consumption for the $\beta-$memory nodes can be huge. Miranker introduced TREAT as another matching algorithm to reduce the huge memory requirement of the $\beta-$memory nodes. The TREAT algorithm reduces the overhead by not maintaining the $\beta-$memory nodes but keeping the $\alpha-$memory nodes. The main reasoning behind the TREAT algorithm

is to determine whether the time required to maintain the $\beta-$memory nodes is less than the time required to compute the join conditions whenever needed [[41]].

The detection of common subexpressions over XML data sources has also been explored as a multiple query optimization technique [[42]]. Multiple XQuery expressions are represented as expression trees of their execution plans. The BEA streaming XQuery processor compiles and parses the XQuery expressions to give access to different parts of the expression trees. These expression trees have nodes that represent the different types of expressions and the edges connecting the nodes represent the data flow dependencies. The expression types includes different variables defined in the query, first order and second order expressions, user-defined functions and constants. All the expression trees are first normalized using rewriting rules based on the XQuery format semantics. A heuristics-based algorithm traverses these expression trees to detect shared subexpressions, which are the candidates for memoization. Memoization is a technique that stores the results of certain chosen expressions in the main memory and for any subsequent access to the same expressions with the same set of arguments, the results are retrieved from the main memory instead of recalculating them.

Multiple query optimization with common sub-expressions to detect partial joins in the past research has been explored in the context of relational databases. This concept has significant importance in the DEPA environment; however, the algorithms need to be redesigned to incorporate XML data sources. The research described in chapter 4 builds on the MQO approaches of [[31]] and [[42]] by defining a mixed multigraph model to represent query expressions over relational and structured XML data sources. An algorithm is presented to detect common subexpressions from this multigraph representation of different queries for multiple query optimization.

2.4 Materialized View Maintenance
Relational Databases

Incremental evaluation in relational databases is comprised of two closely-related concepts i.e. incremental maintenance of materialized views and incremental evaluation for condition-monitoring of CA events defined over the relational databases. These two concepts use the modifications made to the relations to maintain the views or evaluate the conditions.

The materialized views provide access to the view tuples directly rather than recomputing the view every time for the query containing that view. Apart from quicker data access, integrity constraint checking and query optimization can also benefit from materialized views. However, whenever modifications are made to the base relations, it is important to keep these views updated. Thus, the process of using these modifications for updating the materialized views is called view maintenance. Using only relevant changes to update the view is incremental view maintenance. Incremental view maintenance problems are based on the resources used to maintain the view, types of modifications considered and whether the algorithm works for all the instances of databases and modifications ⟦14⟧. Most of the prior research focused on expressing views in different languages such as SQL and Datalog with features like aggregations, duplicates, recursion, and outer joins.

The landmark paper ⟦43⟧ examines materialized view maintenance in the context of Select-Project-Join (SPJ) expressions using SQL as the view definition language in a relational database environment. The algorithm presented by this paper computes differential expressions to identify the tuples that must be inserted or deleted from the view based on the modifications made to the base relations. This approach was enhanced by ⟦44⟧ with a counting algorithm for SQL non-recursive views with duplicates including union, negation, and aggregation. This algorithm keeps a count of the number of derivations for each view tuple as additional information in the view. The

algebraic differencing algorithm [45] differentiated relational algebraic expressions to derive two expressions for each view; one for insertions and the other for deletions. Hence, updates could be modeled using this algorithm as a sequence of deletes and inserts to the view. The Ceri-Widom view maintenance algorithm [46] defines the materialized views using SQL-based syntax and derives production rules in terms of SQL queries that are associated with the view's base tables. Whenever changes occur at the base tables, these production rules are triggered to propagate the changes to the view as a sequence of deletes and inserts to the view. The views defined using this approach do not support duplicates, aggregations, and negations.

Most of the view maintenance algorithms for recursive views were developed in the context of Datalog [15]. Incremental view maintenance of recursive views involves two steps. The first step of propagation or derivation computes an overestimate of change tuples (either insertion or deletion) and the second step of filtering or rederivation determines whether the change tuples computed in the first step are relevant changes necessary to update the the view. The PF (Propagation/Filtration) algorithm [47] and the DRed algorithm [44] were developed for Datalog (or SQL) views with recursion, union, negation and aggregation. These two algorithms differ in how these two steps are performed. The PF algorithm uses top-down memoing whereas DRed uses bottom-up semi-naïve evaluation. In case of deletions, both the algorithms compute the overestimate of tuples to be deleted. However, the PF algorithm filters out those tuples that have alternative derivations before propagating them. The DRed algorithm propagates the potential deletions within the stratum but applies filtering to the overestimated tuples before they are propagated to the next stratum. In case of insertions, the PF algorithm handles insertions similar to deletions. However, the DRed algorithm uses bottom-up semi-naïve evaluation to determine the actual insertions to the materialized view. There are instances where the PF algorithm performs better than DRed and there are other scenarios in which DRed outperforms the PF algorithm.

Another use of incremental evaluation in the context of relational databases can be found in the area of continuous stream processing. Typically, only the new streaming information is considered as the only updates available for incrementally maintaining the materialized views [16]. In such a high-speed streaming processing application, a chronicle is defined as a sequence of records rather than an unordered set of tuples. The chronicle cannot be stored completely in the database because of its size; however, the data that fits in the latest time window are stored in the database for computation (the sliding window protocol). Thus, in this model the only update possible to the chronicle is insertion of tuples. The algorithm to define views over such chronicles and maintain them efficiently makes use of the Summarized Chronicle Algebra (SCA) [16]. This algorithm works on non-recursive views with aggregation and summarization.

XML Databases

XML data, due to its hierarchical structure, provides a different set of challenges in terms of updating views defined over XML data sources. Updating an XML view can lead to recreation of a totally new XML tree where the nodes can not only get modified with new values or new subtrees, but it is possible that these nodes are moved around to a different subtree. Thus, modeling and validating XML updates is an important issue. While updating an XML document, it is necessary to locate the part or subsection of the XML tree structure that is to be updated. Also the view maintenance algorithm should be able to handle batch XML updates, and handle updates with missing information. Once the updates are modeled and validated, propagating and applying the updates to XML views is complex because XML data is a hierarchical semi-structured representation of data with ordering semantics, which is not the case with flat tuples with no ordering [18].

The validate-propagate-apply framework [18] was proposed to solve these problems with XML views defined using XQuery. The framework supports an expressive subset of XQuery views with XPath expressions, FLWOR expressions and element

constructors. In the validate phase, the XQuery updates are transformed into a set of update trees and are checked for their relevance to the views. The potential relevant updates are annotated with information necessary for propagation. This information includes any other nodes needed by the XQuery query to determine the exact location of the update within the view tree. Lastly, all these update trees for different update types are batched into one batch update tree. In the propagate phase, the Incremental Maintenance Plans (IMPs) are derived from the view query. These IMPs process the batch update trees to generate delta update trees, which are used in the apply phase to incrementally update the XML views. The apply phase mainly comprises of merging the delta update trees into the corresponding view trees. In the apply phase, the ordering of the XML tree nodes is very important for the correct merging of delta update trees. This ordering is done by applying semantic identifiers to all the XML nodes [48]. This semantic ID solution helps to keep the XML views distributive because the tag ordering is encoded as part of the semantic IDs.

All the previous work related to incremental evaluation has focused individually on either relational data sources or XML data sources. However, in a distributed environment of heterogeneous data sources, it is necessary to combine relational as well as XML data sources. Past research focused on this aspect has taken the direction of converting one data source to the other and then defining views over the converted data. This approach has the inherent drawbacks of repeated conversion from one format to the other for the original data as well as the deltas captured over the original data sources. Also, to convert one format to the other requires extensive mapping information to be maintained that facilitates the conversion. One of the research challenges that is explored in this dissertation focuses on defining materialized views over heterogeneous data sources within the DEPA framework. These views defined as the partial joins over relational and structured XML data sources will retain the data in their native format, exploiting the well-established technology for each of the data sources.

Thus, the changes over these heterogeneous data sources are captured in their native format and are used in the incremental maintenance of the hybrid views. The deltas are transported over data streams from the base data sources to the relevant agents in the framework for the incremental evaluation of the localized views.

2.5 LINQ

The various research objectives of this dissertation require a materialized view definition language that supports views over heterogeneous structured data sources within the same view definition. Initial investigation has led to the choice of Language INtegrated Query (LINQ) language as the materialized views definition language for the DEPA framework. LINQ is a relatively new language introduced with the .NET 3.5 Framework. LINQ is a declarative, strongly-typed query language that is integrated into an imperative programming language, such as C# and Visual Basic. LINQ's query comprehension syntax is SQL-like with `from`, `where`, and `select` clauses. However, the order of the clauses corresponds to the execution order based on the underlying foundation of functional programming. Specifically, the `from` clause introduces a variable to iterate over a collection, the `where` clause filters the results using dot notation to access properties of the `from` variable, and `select` returns a structured result. An overview of LINQ for the database educator can be found in [49]. As an example of a simple LINQ query to understand its syntax, consider a LINQ query in C# over the TPC-H benchmark database schema [50] as shown in Listing 2.2 that returns the names of the suppliers who have an account balance more than $1000.00.

```
var suppliersBalGreaterthan1000 =
                from s in supplier
                where s.s_acctbal > 1000.00
                select s.s_name;
```

Listing 2.2: Simple LINQ query.

An advantage of LINQ is that the same language can query a collection of: *objects* in memory or an object-oriented database, *tuples* in a relational table, or *elements*

in an XML document. The same dot notation accesses a property of an object, an attribute of a tuple, or a nested element or an attribute. This unifying paradigm allows the database developer to use one language to access the various data sources needed for the application. Thus, the use of LINQ avoids the additional learning curve of the developers to understand the intricacies of the APIs for each database technology.

LINQ provides a layer of abstraction over the different data sources. A LINQ query is represented internally as an expression tree. The LINQ framework then uses a data-source specific LINQ provider to automatically generate an equivalent query over that data source. A LINQ provider for a data source must implement required APIs to facilitate this generation. Microsoft has various LINQ providers as shown in Table 2.1. For example, the LINQ to SQL provider queries SQL Server databases by converting a LINQ query into an equivalent SQL Server query. Also, the LINQ framework is fully extensible. There are numerous third party LINQ providers available to query different types of data sources, such as LINQ to Amazon, LINQ to MySQL, Oracle, and PostgreSql [51].

Name of the Provider	**Type of Data source access**
LINQ to Objects	In-memory collection of Objects
LINQ to SQL	Microsoft SQL Server
LINQ to Entities	Any ODBC-compliant relational data source
LINQ to Dataset	In-memory collection of relational tuples
LINQ to XML	XML data sources
LINQ to XSD	XML data sources with associated XML Schema

Table 2.1: Different LINQ providers by Microsoft.

The **LINQ to Objects** provider queries a collection of objects by defining a variable in the `from` clause. Let us consider that there is a C# class called Supplier that matches the schema from the TPC-H supplier schema and the user has created an array or list called "Suppliers" that contains Supplier objects. Thus, the query from Listing 2.2 can be modified to use the "Suppliers" collection object in the `from` clause and everything else remains the same in the query.

LINQ to SQL provides the mechanism to query and manipulate the data from a SQL Server database. LINQ to SQL is an ORM approach, mapping entity classes (in C#) to tables (in SQL Server). The entity classes provide the metadata for LINQ queries. (Note that the selective mapping of existing SQL Server databases can be automated using tools, such as the visual design tool, LINQ to SQL Classes, within the Visual Studio 2008 environment.) Once the mappings are specified, then LINQ queries are converted to corresponding SQL queries using expression trees and deferred query execution. Assuming that the ORM mappings between the TPC-H database in SQL Server and C# classes have been defined and "supplier" is the name of the collection. (Notice that while using the visual design tool from Visual Studio 2008, certain settings should be configured so that the name of the collection is the same as the name of the table in the SQL Server 2008.) Thus, the LINQ to SQL query remains exactly the same as in Listing 2.2. There are two more providers: LINQ to Dataset and LINQ to Entities, which are very similar to LINQ to SQL. LINQ to Dataset provides querying mechanism over structured relational data collections temporary stored in the main memory for fast execution. LINQ to Entities, which is more stable and provides secured execution than LINQ to SQL is used to more in a production environment, whereas, LINQ to SQL is used for rapid prototyping [[52]].

LINQ to XML query iterates over a collection of XML elements. The XML data is read from an XML file stored in the file system. Whenever LINQ to XML reads an XML file that does not have any XML Schema associated with it, then that particular query is untyped. Considering the TPC-H schema again, assume that an XML file named "customer.xml" has been created with elements having the same name as the attributes of the customer table. The following code Listing 2.3 shows how to open the "customer.xml" file and a LINQ to XML query that returns the names of all the customers who have an account balance greater than $10000.00.

```
XElement customer = XElement.Load("customer.xml");
var customersBalGreaterthan10000 =
```

```
        from c in customer.Elements("customer")
        where Double.Parse(c.Element("c_acctbal").Value) > 10000.00
        select c.Element("c_name").Value;
```

Listing 2.3: LINQ to XML query.

The **LINQ to XSD** provider can be used when an XML Schema Definition (XSD) is available for an XML document. LINQ to XSD creates Object-XML mappings (OXMs) that map entity classes in C# to XML elements and attributes in the XML schema. Assuming that there is an XML schema file "customer.xsd" associated with the XML file "customer.xml", the modified LINQ to XSD query is shown below in Listing 2.4.

```
var customersBalGreaterthan10000 =
          from c in customer
          where c.c_acctbal > 10000.00
          select c.c_name;
```

Listing 2.4: LINQ to XSD query.

The previous discussion illustrated the use of LINQ over disparate data sources individually. However, LINQ can query relations, objects, and XML in the same query. Assume that there is a relational table "supplier" defined in SQL Server 2008 and there is an XML file "customer.xml" without an associated XSD. In addition to these two data sources, there is another collection of objects "Nations" for a C# class "Nation" in accordance with the TPC-H schema. The following query Listing 2.5 returns all the supplier names and customer names from the same nation as "United States" (the nation key for United States is 24).

```
XElement customer = XElement.Load("customer.xml");
var SuppliersCustomersfromUnitedStates =
    from c in customer.Elements("customer")
    from s in supplier
    from n in Nations
    where n.n_nationkey == 24 &&
          n.n_nationkey == s.s_nationkey &&
          Int32.Parse(c.Element("c_nationkey").Value) == n.
             n_nationkey
    select new {CustomerName = c.c_name,
                SupplierName = s.s_name,
```

```
                Nation = n.n_name};
```

Listing 2.5: Single LINQ query over objects, SQL and XML.

Thus, LINQ's unifying paradigm to query heterogeneous data sources in a single query supports the objectives of the materialized view definition language for the DEPA framework.

2.6 Summary

The related work on event stream processing, dataspaces, multiple query optimization, and materialized view maintenance establishes the importance of this work that builds on these various approaches. Chapter 3 describes the metadata requirements for the DEPA framework and presents a service-based implementation of the repository that provides metadata-level access to different language components in the DEPA environment. Chapter 4 presents a heuristics-based algorithm that uses a mixed multigraph model to represent queries over relational and structured XML data sources to detect common subexpressions as the candidates for materialized views. Chapter 5 explores the use of LINQ as the materialized view definition language to define hybrid views over the heterogeneous data sources. An incremental maintenance algorithm based on magic sets query optimization approach to update the materialized views using the deltas captured over relational and XML data sources is presented in chapter 6. Chapter 7 presents the performance evaluation of the different components in the DEPA framework using two different data models.

Chapter 3

SERVICE-BASED METADATA REPOSITORY

The distributed event stream processing environment provides querying capabilities over heterogeneous structured data sources taking advantage of materialized views defined over these data sources. The software agents in the DEPA framework support different query languages, such as SQL, LINQ, and XQuery along with the continuous query language (CQL) provided by the event stream processor. This dissertation is exploring the research challenge of optimizing multiple queries defined in these different languages by detecting common subexpressions over the relational and XML data sources. Detecting common subexpressions requires metadata level access to the different data sources and the various query expressions. Materialized views can be defined using these detected common subexpressions. The view definition and creation algorithms described in chapter 5 also require metadata access to the different data sources, which are used in the view materialization. Thus, the first step in developing the DEPA framework is to collect and provide access to the metadata for the different language components and data sources. The DEPA framework for integrating events and stream processing with access to distributed heterogeneous data sources is based on the concept of co-existence of data sources, similar to a dataspace support platform [11]. This approach differs from a typical data integration framework that semantically integrates the data sources to provide a global reconciled schema over the data sources [12]. The metadata design explained in this chapter coordinates the metadata-level information from the co-existing data sources using services, which provide a protocol for interactions between loosely coupled systems.

The objective of this chapter is to present the design and development of a service-based metadata repository for the heterogeneous data sources registered in a distributed event stream processing framework. These heterogeneous sources include relational databases, XML data, event and data streams. Section 3.1 describes the

metadata repository design and the different components that contribute to the metadata repository. Section 3.2 describes the implementation of the metadata repository and the services exposed for accessing it through a service-oriented architecture. The chapter concludes with a brief discussion of using this metadata repository in the loosely coupled distributed event stream processing framework.

3.1 Metadata Design

The DEPAs are essential to the vision of developing enterprise-level applications by orchestrating events and streams over heterogeneous data sources that are registered with the DEPA. The "application integrator," typically a knowledgeable programmer, can define events of interest, filter streams and pose queries to coordinate these applications.

The goal of this research is to examine multiple query optimization within this context to improve the performance of the system. Specifically, the research uses the metadata level information to identify common subexpressions from the different queries to materialize potential views over distributed data sources locally at the DEPA level to improve performance. This research also uses the metadata level information in conjunction with the incremental view maintenance algorithm to update the views defined within this context.

To extract common subexpressions from these different query expressions, it is important for the DEPA to have access to metadata-level information of the different persistent data sources, such as relational and XML databases, and parsing-level access to various streams, events, continuous queries, SQL, XQuery and LINQ queries. The conceptual metadata repository is shown in Figure 3.1.

DEPAs are autonomic agents that can register/unregister resources at run-time, which requires the design of the metadata repository to be able to handle dynamic updates. Thus, the conceptual repository is designed in two parts: a persistent component and a run-time component as shown in Figure 3.2. The persistent component stores the

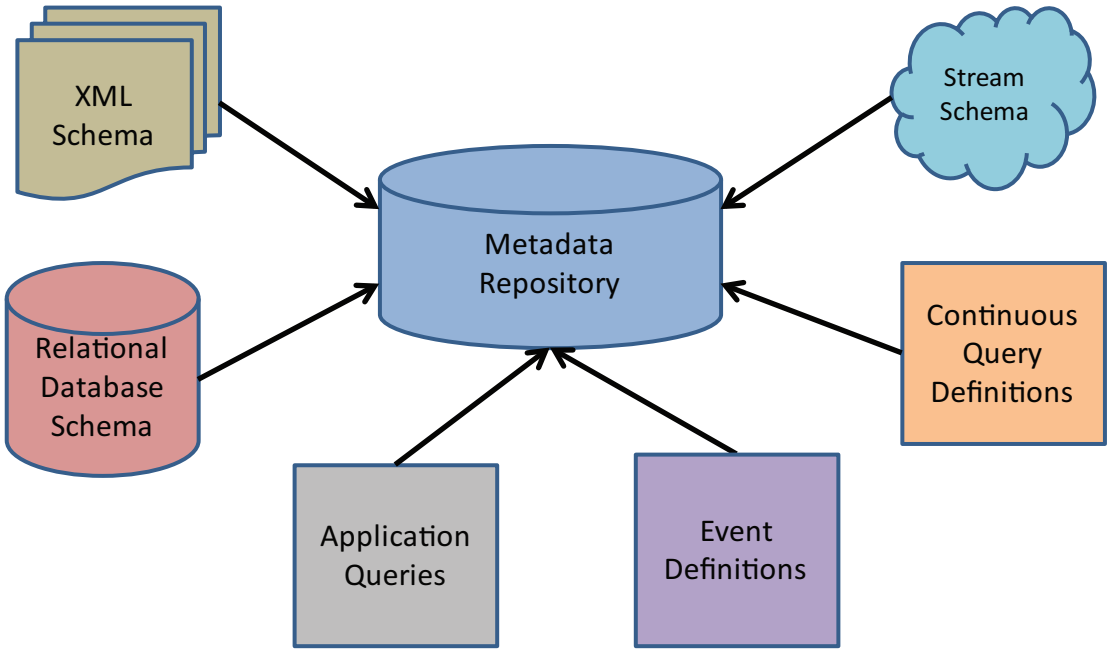


Figure 3.1: Conceptual metadata repository.

necessary access information for the data sources that are used to build the corresponding run-time component within the metadata repository. The run-time component is a collection of C# classes that provides access to the actual metadata through services provided by the DEPA. Maintaining two separate components in the repository provides the capability to update the metadata information in response to schema-level changes.

The persistent metadata consists of query expressions and structured data sources that are registered with the agent. The different types of query expressions that are maintained by the DEPA are LINQ, SQL, XQuery, CQL queries, event definitions, and materialized views. The persistent metadata also keeps access information regarding the different data sources, such as relational databases, XML documents, and stream definitions.

The run-time component uses the information from the persistent component to build the metadata information whenever a new resource is registered with the DEPA. For the different types of data sources, the run-time component is a collection of related

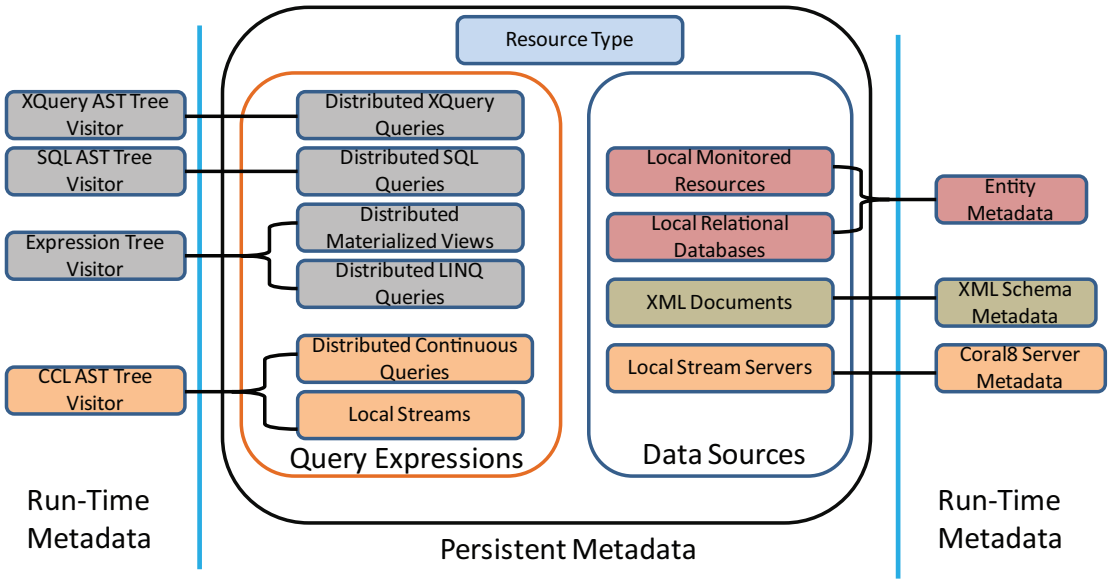


Figure 3.2: Metadata repository.

C# classes that provides access to the metadata level information regarding that type of the data source. For the different query expressions maintained in the persistent component, the run-time component has a set of C# classes generated for the Antlr-based parsers for each language component. These classes provide parsing-level access to different clauses from the query expressions. In the event of an agent going off-line and coming back on-line, then the agent can rebuild the run-time component of the metadata repository by gathering the latest metadata for the relational and XML data sources using the access information from the persistent component.

Since each agent in the framework registers the events and streams to which it subscribes, the metadata repository also provides metadata level information regarding the event and data stream definitions as shown in Figure 3.3. The run-time information regarding the event stream processor, the different workspaces and the schema-level information of the streams and event definitions are acquired through the `Coral8-ServerMetadata` class and its associated classes (see Figure 3.2). The details regarding the metadata for the Sybase CEP server accessed using the API calls are provided by the CEP .NET development kit [25].

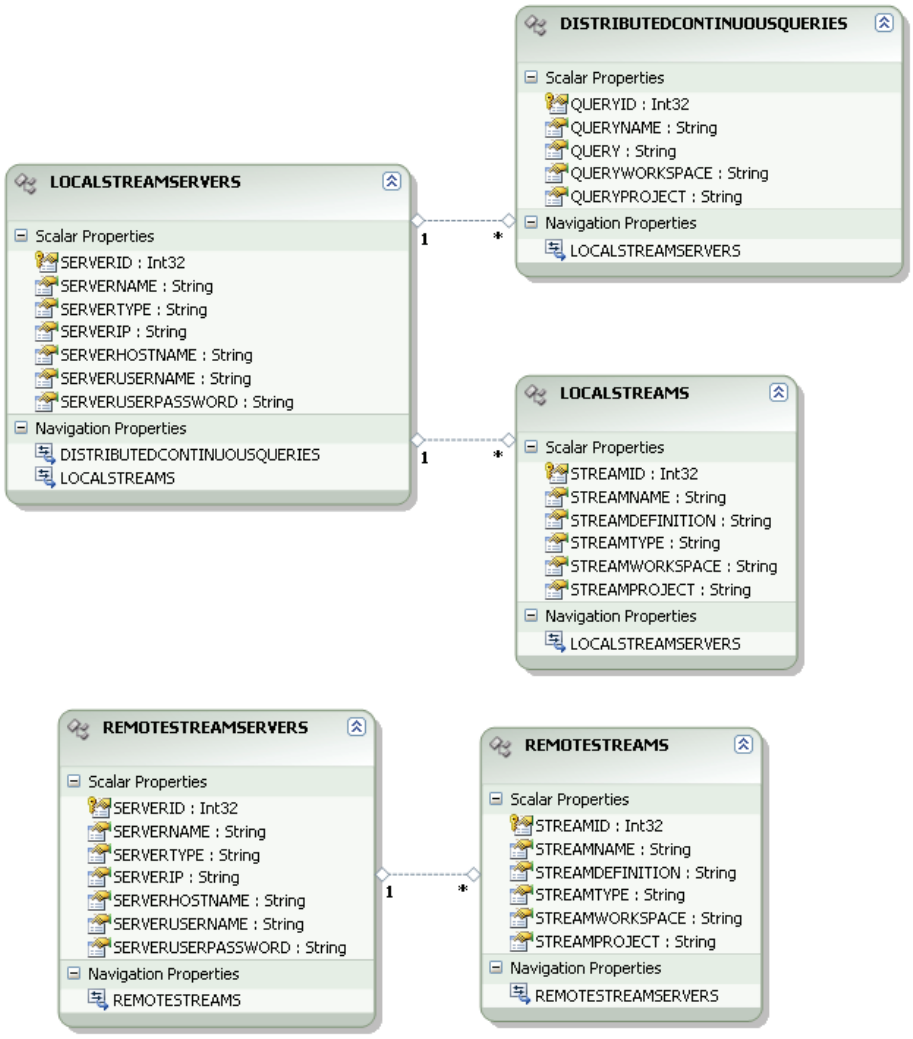


Figure 3.3: Streams metadata information.

The DEPA agents store SQL queries for querying relational data sources and XQuery queries for querying the XML documents. To provide querying capabilities over combined relational and XML data sources, the agents also store the LINQ queries, which can query objects, relations, and XML documents through a single query. In order to detect common subexpressions from these different queries, the agent needs access to information from queries, such as different data sources, predicates used to filter information, and which attributes or elements are projected as the output of the queries. In the DEPA framework, this parsing level information is obtained by using parsers for each query language. A parser written in Antlr parses the

continuous queries defined using the Sybase CQL language into abstract syntax trees that can be used to represent queries in mixed multigraph model. This is a modified parser from the CQL parser, which was provided by the technical support team at Aleri Inc. This CQL parser is a stripped down version of the original parser that is used internally in the Coral8 software. The Coral8 event stream processor, now known as Sybase CEP, was originally developed by Coral8 Inc., which was bought by Aleri Inc., which was then acquired by Sybase. The `CCLASTTreeVisitor` class and its associated Antlr-based classes (see Figure 3.2) provide access to the parsing level information from the CQL queries. Since LINQ is used to access all the relational and XML data sources throughout the development of the DEPA framework, LINQ is also used to access the metadata related to the heterogeneous data sources. The LINQ framework provides access to the expression tree generated by parsing a LINQ query. The `ExpressionTreeVisitor` class and its associated classes from the LINQ framework (see Figure 3.2) provide access to the expression trees. These classes are also used to parse the metadata information from the materialized view definitions since the views are defined using LINQ. Similar to the parser used to parse the CQL queries, there is an SQL parser that accesses the different information obtained by parsing an SQL query. The `SQLASTTreeVisitor` class and its associated Antlr-based classes (see Figure 3.2) provide metadata information related to the SQL queries. The different queries are stored as part of the persistent metadata repository as shown in Figure 3.4. To provide metadata access to the queries expressed in the XQuery language, an Antlr-based XQuery parser is used to access different sections of the query. The `XQueryASTTreeVisitor` class and its associated Antlr-based classes (see Figure 3.2) provide metadata information related to the XQuery queries.

The agents in the DEPA framework are assumed to perform their own tasks or responsibilities. In order to carry our these tasks, each DEPA maintains a set of data sources required for proper functioning and timely output of the different queries regis-

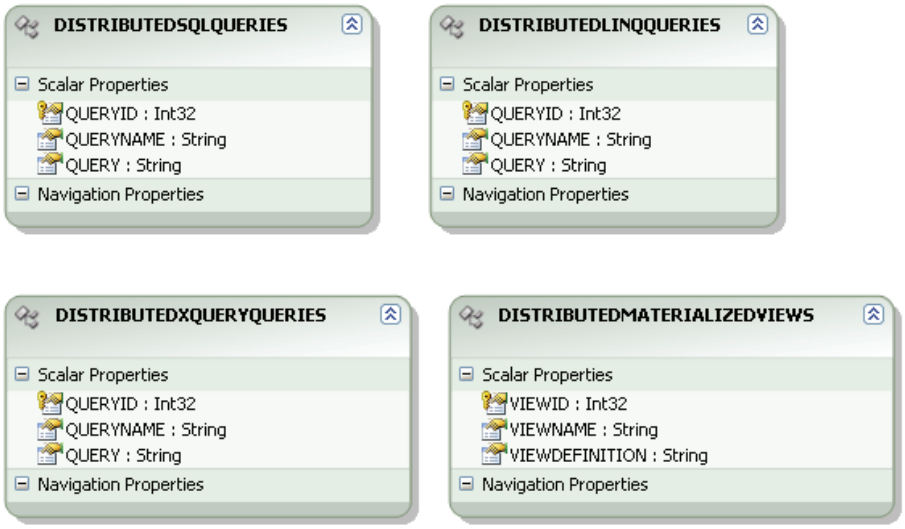


Figure 3.4: Metadata information for query expressions.

tered with the DEPA. As part of the persistent metadata repository, the agent maintains information regarding accessing the data source. For a relational database, this information includes the server name, user name, password, and database name, along with other necessary data required to connect to the database server as shown in Figure 3.5. The run-time component uses this information to access the server for the first time and creates the necessary data structures to store and access the metadata for the database. The run-time component of the metadata repository uses Object-Relational mappings (ORMs) through the LINQ to Entities provider. This ORM provides metadata level information for the classes created for the individual relational table as well as the metadata information of the database server. The classes generated using the ORM technique are compiled into a Dynamic Linked Library (DLL), which can be loaded or unloaded into the main memory of the DEPA at run-time. The `EntityMetadata` class (see Figure 3.2) and its associated LINQ framework classes provide access to the required metadata information on relational data sources. Along with this information, the DEPA also maintains a record of which relational tables are configured for the Change-Data-Capture mechanisms that captures the changes made to the tables. These deltas are used to incrementally update the relevant materialized views. For an XML

document, if the corresponding XML schema is available, then it is used to access the metadata for the XML document. If the XML schema is not available, then the XML schema definition tool called `xsd.exe` [53] can be used to create XML schema from the XML document, attach the XML schema to the document and validate the document at the same time. The same XSD tool is then used to create XML-Object mappings that create C# classes corresponding to the XML schema. These classes are also compiled into a DLL file and the DLL file is used whenever metadata information is required. The `XMLSchemaMetadata` class (see Figure 3.2) and its associated .NET framework classes provide access to the necessary information related to the XML document.

3.2 Implementation of Metadata Services

The DEPAs communicate peer-to-peer in a distributed environment. The DEPAs exchange information about the subscribed resources, as well as the event and data streams they are exposing to other DEPAs. The DEPAs also communicate about the changes occurring in the persistent data sources. Thus, in order to achieve these key functionalities, a Service-Oriented Architecture (SOA) serves as an ideal technology to develop the DEPA framework. Since the research focuses on using LINQ as the view definition language and LINQ is part of the .NET framework, the .NET technology called Windows Communication Foundation (WCF) is a unified framework used to build SOA-based applications [54].

WCF is a Software Development Kit (SDK) provided by Microsoft to develop and deploy services, which can take advantage of the underlying .NET 3.5 framework. Some of the key features of WCF are service instance management, asynchronous calls, transaction management, disconnected message queuing and security. These features distinguish WCF services from traditional web services such that WCF services are stateful and they can retain the client call-back handlers to send messages or data back to the client in an asynchronous way. This is also helpful in achieving server-pushing

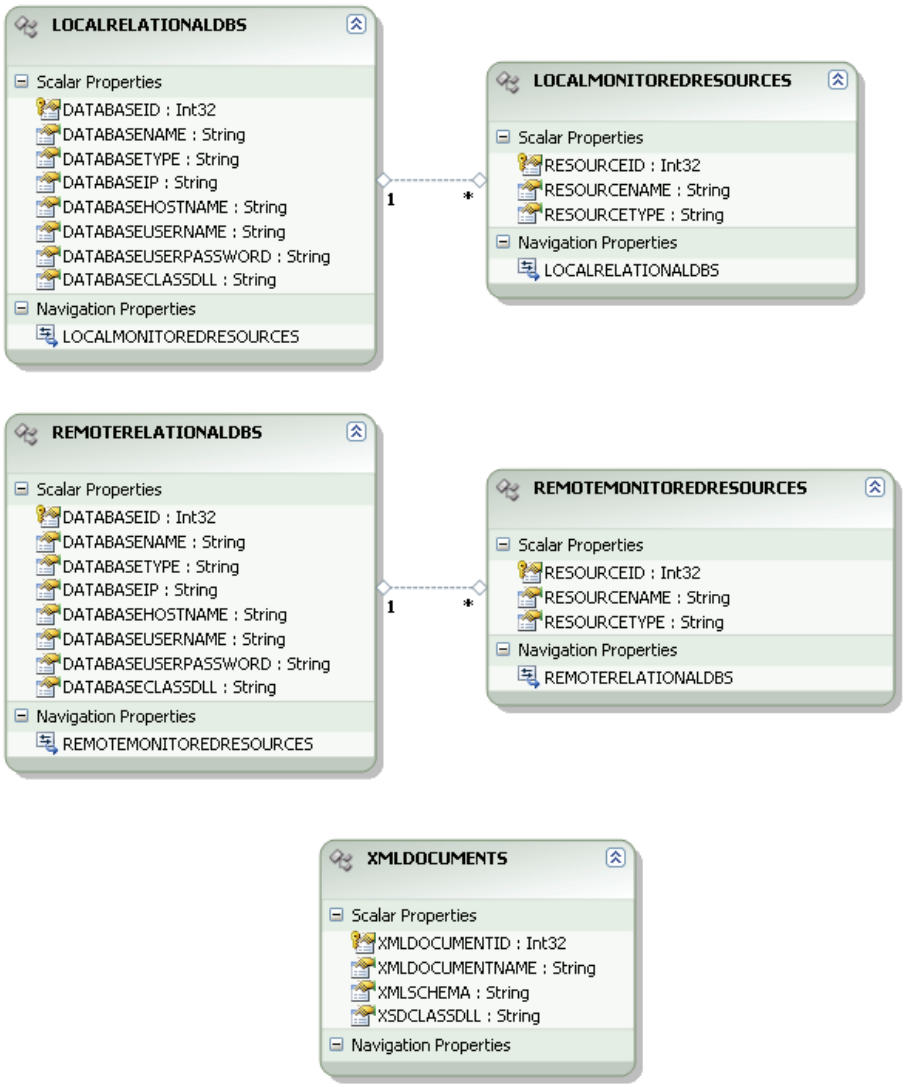


Figure 3.5: Metadata information for heterogeneous data sources.

so that if any metadata or data has changed, then that DEPA service will notify other subscribing DEPAs about the changes. Hence, it is not essential for all the DEPAs to keep polling for the changes occurring at other DEPAs.

Like any typical service, a WCF service has endpoints through which the clients can communicate with the service. The WCF service can be hosted on either an IIS server or Windows Activation Service (WAS) or in a standalone windows process. The same service can have multiple endpoints providing different subsets of services to different clients based on their subscriptions. Each endpoint is characterized by 3 param-

eters: a URL address of the endpoint; a binding mechanism that tells how the endpoint can be accessed; and a contract name that indicates which service contract is exposed at that endpoint. The binding mechanism is the most important criteria through which the client can communicate with the endpoint either synchronously or asynchronously, through TCP, HTTP, MSMQ or Named Pipes protocol.

In the DEPA environment, the metadata repository can be accessed at different levels through different WCF services. Figure 3.6 shows two important WCF services that allow access to the metadata repository. The `Subscription` service is accessible to the clients to register/unregister the persistent resources and different query expressions. The `Metadata provider` service provides actual access to the metadata for the resources registered.

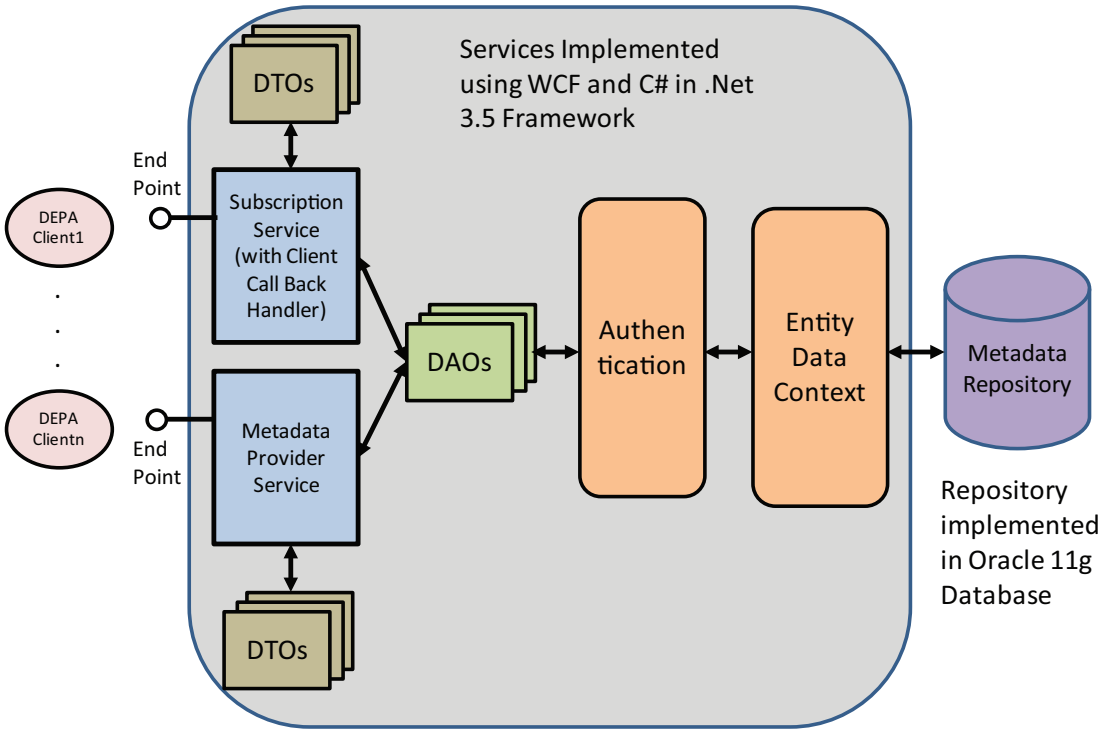


Figure 3.6: Exposing metadata through SOA services.

One of the main issues regarding providing data access over services is marshalling and unmarshalling of the data between the client and the service. In order to avoid heavy objects transfer between the client and service, the metadata services

use the enterprise-level design patterns of Data Access Objects (DAO) [55] and Data Transfer Objects (DTOs) [56]. DAOs are the abstraction-layer objects through which the metadata repository is accessible. DTOs are the light-weight replica objects of actual entities retrieved from the repository. These DTOs are the actual objects that are marshalled and unmarshalled. The mapping between DTOs and the actual entities is handled by DAOs.

The metadata prototype for the DEPA environment is developed using WCF services and C# with the metadata stored in an Oracle 11g database. The service administrator can subscribe/unsubscribe resources at run time. For registering a relational data source, the `EntityMetaData` handler class (see Figure 3.2) takes the access information and creates the Entity Context class for that relational database, converts this class into a Dynamic Linked Library (DLL) and loads the DLL into the WCF service, thus updating the metadata repository at run time. For unregistering a relational data source, the same handler class unloads the DLL file, deletes the file and then removes access information from the metadata repository, thus keeping the repository updated. A similar process is followed for the XML documents. The event stream and continuous query definitions are first registered or unregistered with the Coral8 Streaming server and then the `Coral8ServerMetadata` and the `CCLASTTreeVisitor` classes (see Figure 3.2) can access the metadata.

3.3 Summary

This chapter has discussed the design and implementation of a SOA-based metadata repository for processing events and data streams in a loosely coupled distributed environment in coordination with heterogeneous structured data sources. One aspect of this research is to explore the feature of providing materialized views local to the DEPAs for processing events and data streams. In order to define materialized views, it is important to extract common subexpressions from different query expressions, such as continuous queries, primitive and composite event definitions, SQL queries and existing view

definitions. Designing the metadata repository to provide access to metadata-level information of the distributed sources is the first step in the journey of exploring multiple query optimization for a variety of query expressions to detect common subexpressions and using LINQ as a materialized view definition language over heterogeneous structured data sources while respecting the native format of the data. This metadata repository is the common backbone for the following research challenges:

1. *Dependency analysis across different filtering queries to identify common subexpressions as potential candidates for materialized partial joins.*

   Using an analysis of the various rules, conditions, and queries, materialized views representing the partial joins of common subexpressions can provide the foundation for the efficiency of the incremental evaluation by exploring multiple query optimization within the framework. While identifying these common sub-queries, the metadata repository is essential for query unfolding and to get metadata information about the data sources on which the queries are defined. The metadata database also provides schema-level information for the candidate partial joins across relational and XML data sources.

2. *Techniques for selectively materializing the partial joins over relational as well as XML data sources.*

   After identifying the potential candidates for the materialized views, one research challenge is to design a heuristic algorithm that will selectively choose partial joins over the heterogeneous data sources that will be beneficial to materialize. Once the materialized views are defined in the system, the original query expressions have to be rewritten to use the materialized views instead of the underlying data sources. The metadata repository plays an important role in assisting this process.

3. *Incremental Evaluation and Materialized Views for Integrating Streams, Events,*

and Persistent Data.

Capturing of deltas or changes to the original data sources is important in incremental view maintenance of the materialized partial joins. Using the metadata repository and the materialized views, the native deltas arriving into the system must be analyzed and used to incrementally update the views.

Chapter 4

DETECTING COMMON SUBEXPRESSIONS OVER HETEROGENEOUS DATA SOURCES

This dissertation is focusing on the research challenges involved in developing a distributed event stream processing framework that can take advantage of optimizing multiple queries defined over heterogeneous data sources for the efficient functioning of the individual agent in the framework. A DEPA (an agent) maintains a repository of various types of query expressions including continuous queries over streams with access to heterogeneous data sources, queries over relational and structured XML data sources. Also, every DEPA performs specific tasks that increases the probability of optimizing various query expressions by detecting common subexpressions across them. This multiple-query optimization (MQO) technique analyzes various query expressions to detect common subexpressions across the query graphs and construct a global access plan that will avoid repeated computations of the detected common subexpressions. Once these common subexpressions are identified, they can be computed only once and the results are cached in the main memory for subsequent access [[31, 42]] or the results can be materialized into views which can be referenced for reuse [[15, 33]].

Detecting common subexpressions across relational queries has been studied for centralized databases [[31, 32, 57]] as well as in distributed databases [[58]]. SQL queries can be represented either in relational algebra trees [[32]] or as multigraphs [[31, 59]]. Multiple queries are analyzed and are expressed in terms of a graph that represents the data sources, the operations on the attributes of the data sources and the predicates related to each operation. Detecting common subexpressions across multiple queries is a well-known NP-hard problem [[60]]. However, this problem can be addressed by applying heuristic rules in a specific order. This approach has been explored for Select-Project-Join (SPJ) queries over relational tables in main memory databases using a multigraph model [[31, 59]]. Multiple relational queries can also be optimized by first

obtaining the execution plans from the database engine and then analyzing all the plans for global optimization [33,34]. This approach works well only for SQL queries posed to a specific database based on the execution plan generated by that database.

Processing multiple XQuery queries over XML documents also leads to the detection of common subexpressions involving XPath expressions [42]. XQuery queries are represented as expression trees with nodes that represent different types of expressions and the edges that represent the data flow dependencies. The algorithm described in [42] also uses heuristic rules to detect the common subexpressions and to cache the results in main memory. Subsequent queries are then analyzed to verify whether the cached results can be reused. The above mentioned approaches work well for individual data formats, but these approaches do not handle queries over both relational and XML data under the same graph model. The expression trees or the execution plans generated for XQuery queries are quite different from the execution plans for SQL queries [42, 61]. Thus, combining the two execution plans together to detect common subexpressions for MQO is quite challenging and difficult. Also, the query optimizers provided by the centralized databases cannot be used to optimize queries over distributed heterogeneous data sources. This dissertation proposes a mixed multigraph model to represent SQL, LINQ and XQuery queries over relational and XML data sources in a single graph model.

This chapter focuses on formalizing the multigraph approach to represent SQL, LINQ and XQuery queries and describes a heuristics-based algorithm to detect common subexpressions as potential candidates for view materialization. To motivate the rationale behind the design of the multigraph model, the chapter describes a small scenario from the criminal justice model with sample queries over heterogeneous data sources. These queries exhibit some commonalities in their expressions, which provides the groundwork for detecting common subexpressions. This chapter then describes the formal details of the mixed multigraph model along with the representation

of a sample query over relational and XML data sources. The next section presents the set of heuristic rules and the common subexpression detection algorithm that analyzes the query graph to detect the commonalities from their expressions. A detailed working example to illustrate the use of the algorithm shows that common subexpressions can be purely relational or purely XML or a hybrid join over relational and XML data sources.

4.1 Motivational Example

One of the research challenges of this dissertation is to explore ways for efficient execution of different query expressions over heterogeneous data sources in a distributed event stream processing framework. Each autonomous agent in the framework carries out specific tasks by subscribing to different resources such as events, streams, relational databases and structured XML documents. To query these different data sources, each agent also maintains a repository of metadata level information of the data sources along with various query expressions, such as continuous queries, SQL queries, XQuery queries and LINQ queries. Due to the locality of tasks and their associated data sources and queries, there is a higher probability that the query expressions will exhibit some common subexpressions, which can be used to optimize these queries. These common subexpressions are the potential candidates for materialized views in the DEPA environment. As motivation to illustrate this concept, consider a part of a larger criminal justice model that holds information in different data formats as shown in Figure 4.1. The driver license information is stored in a relational table where as the vehicles associated with the licenses are stored in an XML document.

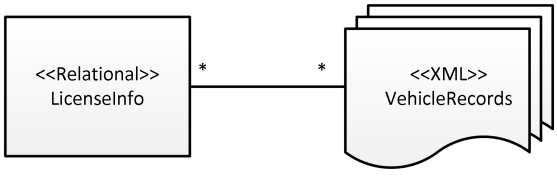


Figure 4.1: Criminal Justice diagram.

Consider a LINQ query $Q1$ in Listing 4.1 that checks for the expired driver

license for all the vehicles registered in the state of Arizona and have more than 10 points on them. Consider another LINQ query $Q2$ from Listing 4.1 that checks for the expired driver licenses of class type D with points greater than 12 for all the vehicles registered in the state of Arizona. These two queries have certain conditions in common over the same set of data sources. Query $Q1$ shares the identical select condition of license status being expired with query $Q2$. Query $Q1$ also shares the common select condition of vehicles registered in the state of Arizona with $Q2$. There is a set of subsumed select conditions on license points in the queries $Q1$ and $Q2$. Finally, Queries $Q1$ and $Q2$ share the common join condition over the driver license number between relational table `LicenseInfo` and the XML document `VehicleRecords.xml`. Thus, it is necessary to represent these queries in a common graph model, where the select and join conditions can be analyzed to detect common subexpressions.

```
Query Q1:
XElement VRDb = XElement.Load("VehicleRecords.xml");
var query1 =
from li in LicenseInfo
from vr in VRDb.Elements("Vehicle")
where li.points > 10 &&
      li.status == "expired" &&
      vr.Element("VehicleLocation").Element("State").Value.Equals("AZ
          ") &&
      vr.Element("DriverInfo").Element("DriverLicense").Value.Equals(
          li.DriverLicense)
select new {/* project needed fields */};

Query Q2:
XElement VRDb = XElement.Load("VehicleRecords.xml");
var query2 =
from li in LicenseInfo
from vr in VRDb.Elements("Vehicle")
where li.class = "D" &&
      li.points > 12 &&
      li.status == "expired" &&
      vr.Element("VehicleLocation").Element("State").Value.Equals("AZ
          ") &&
      vr.Element("DriverInfo").Element("DriverLicense").Value.Equals(
          li.DriverLicense)
select new {/* project needed fields */};
```

Listing 4.1: Sample LINQ queries.

4.2 Multigraph Approach

This research examines query expressions over both relational and XML data sources. A mixed multigraph model is introduced to represent the combined relational and XML query expressions *QE* within a single graph model. This mixed multigraph approach is based on a similar formalism of a multigraph model [[31, 62]], extended to use directed edges for navigating through the XML nodes. Thus, XQuery queries and LINQ queries can be represented in the same graph model along with SQL queries.

Creating multigraphs and defining heuristic rules and algorithms to detect common subexpressions over the full SQL, XQuery, and LINQ languages are complex problems. Hence, the initial focus of this research is to consider queries over a subset of these query languages. For the purpose of this research, Select-Project-Join (SPJ) queries in SQL are considered without self-joins. The restrictions on XQuery include limiting clauses to For-Where-Result (FWR) and XPath expressions to forward axes (/, //, *) and simple boolean predicates. Similar to [[42]], the XQuery queries are assumed to guarantee XML data equivalence. Finally, queries defined using From-Where-Select clauses in the LINQ language are considered.

The definition of the mixed multigraph is as follows:

Let R be a set of relations and X be a set of XML documents.

Assume $R \cap X = \phi$ for simplicity.

Let DS be the set of structured data sources such that,

$DS = R \cup X$.

For a given set of queries Q over DS, define a mixed multigraph $G = (N, SE, JE, NE)$ where N is a set of nodes and SE, JE and NE are the edges where,

- A node $n \in N$ where,

  $N = \{ r \cup x \cup r.a \cup Descendant - or - self(d(x)) \mid r \in R,\ x \in X,\ a \in schema(r),$ and $d(x)$ is the distinguished root element of x $\}$

- A non-directional selection edge $SE(u,u)$ is an edge looping over the node u that corresponds to a selection condition $pred\_cond$ from a query Q_i for the node u. The selection edge is labeled by $SE(Q_i, pred\_cond)$.

- A non-directional join edge $JE(u,v)$ connects two nodes u and v with an edge that represents a join condition $pred\_cond$ from a query Q_i between the two nodes. The join edge is labeled by $JE(Q_i, pred\_cond)$.

- A directional navigation edge $NE(u,v)$ represents a reachable path $path\_expr$ from node u to node v from the query Q_i. For $u,v \in N$, the $path\_expr$ can be either . for relational navigation or a valid XPath expression for XML navigation. The navigation edge is labeled by $NE(Q_i, path\_expr)$.

- Let $E = SE \cup JE \cup NE$ be the set of all the edges defined in the multigraph.

- Define $QL\{E(u,v)\}$ as the list of query IDs from the labels on the edges $E(u,v)$.

With this definition of the multigraph, the different queries over relational as well as XML data sources can be represented in the same graph model. As an example, the multigraph for LINQ queries $Q1$ and $Q2$ over one relational and one XML data source is shown in Figure 4.2. The nodes are represented as rectangular boxes. The selection edges are represented as dashed lines. The solid line represents the join condition between the two data sources and the dotted lines are the navigational edges.

4.3 Detecting Common Subexpressions

Once the multigraph is created by parsing all the queries, then heuristic rules can be applied to detect common subexpressions. However, before applying the common subexpression detection algorithm, it is important to do an early pruning by dividing the query expressions into separate lists of queries that have at least one data source in common [[31]]. These sets indicate that there is no data source common across any query set. The heuristic rules defined in this paper use the following notations:

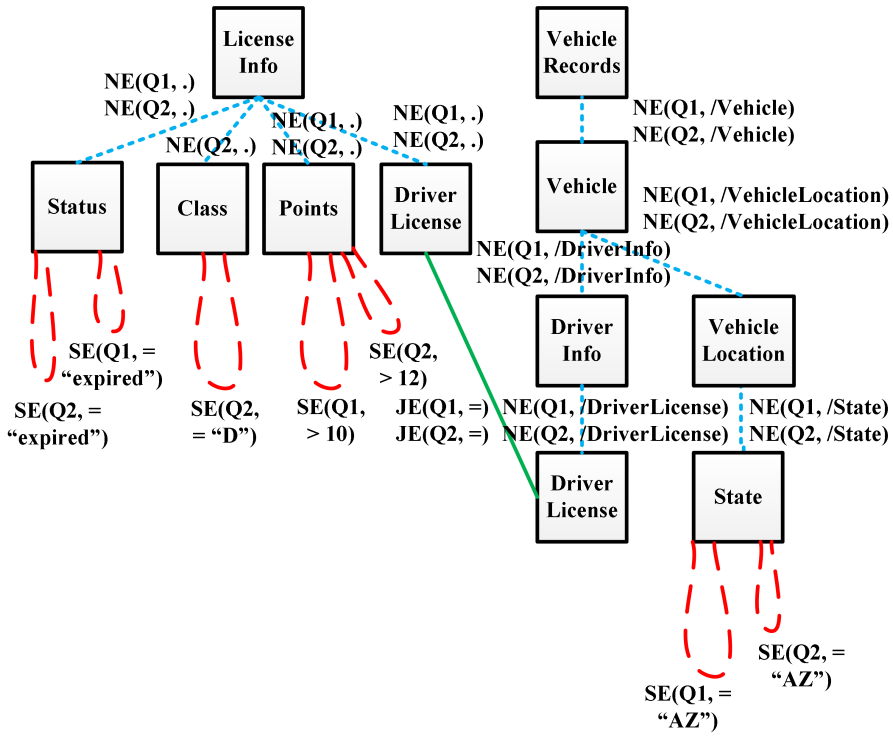


Figure 4.2: Mixed multigraph representing queries over relational and XML data sources.

Define $Root(u)$ as the root node or the data source for the node u.

$$Root(u) = \begin{cases} r & \text{if u} \in \text{schema(r)} \wedge \text{r} \in \text{R} \\ x & \text{if u} \in \text{Descendant-or-self(d(x))} \wedge \text{x} \in \text{X} \end{cases}$$

Let $Path(u,v)$ be the navigational path from node u to node v.

If $Root(u) \in X$, then $Path(u,v)$ = XPath expression from node u to node v

If $Root(u) \in R$, then $Path(u,v) = u.v$

Define *csub* as the detected common subexpression.

Let *CSet* be the set of all the common subexpressions detected in the multigraph.

The heuristic rules defined in this research handle identical and subsumed conditions at the same time whenever they co-exist. In the prior research, heuristic rules were defined separately to handle different conditions [31,32]. These rules accommodate the detection of common subexpressions across heterogeneous data sources and handle the navigational edges to support path navigation in XML trees. The rules are

defined in a specific order so that certain types of edges are selected first in the common subexpression detection algorithm. The first rule handles the selection edges while the second rule handles the join conditions.

1. Selection conditions:

   Let $SE(u,u)_\alpha = \{a | a \in SE(u,u)$ and has the selection condition α on the node $u \in N\}$

   Let $SE(u,u)^\beta_\alpha = \{b | b \in SE(u,u)$ labeled by $SE(Q_i,\beta)$ and $\exists a \in SE(u,u)_\alpha$ labeled by $SE(Q_j,\alpha)$, where α subsumes β and $\beta \neq \alpha\}$

   Let $Root_u = Root(u)$

   Let $QL^\beta_\alpha = QLSE(u,u)_\alpha \cup QLSE(u,u)^\beta_\alpha$ be the list of IDs of the queries that contain the selection conditions α and β.

   IF ($\mid SE(u,u)_\alpha \mid > 1$ *or* $SE(u,u)^\beta_\alpha \neq \phi$) [True if multiple identical or subsumed selection conditions exist]

   - Create a new node u_α representing the selection condition α on u.

     Edge bookkeeping

   - Join edges: move the relevant JEs on α and β selections to the new node u_α.

     – Let $JE(u,v)_\alpha = \{je | je \in JE(u,v) \wedge QL\{JE(u,v)\} \subseteq QL\{SE(u,u)_\alpha\}\}$

       For each $je(u,v) \in JE(u,v)_\alpha$

       $JE = JE \cup \{je(u_\alpha,v)\}$

       $JE = JE - JE(u,v)_\alpha$

     – Let $JE(u,v)^\beta_\alpha = \{je | je \in JE(u,v) \wedge QL\{JE(u,v)\} \subseteq QL\{SE(u,u)^\beta_\alpha\}\}$

       For each $je(u,v) \in JE(u,v)^\beta_\alpha$

       $JE = JE \cup \{je(u_\alpha,v)\}$

       $JE = JE - JE(u,v)^\beta_\alpha$

   - Selection edges:

- Remove $SE(u,u)_\alpha$ edges since they are coalesced into the new node u_α.

  $SE = SE - SE(u,u)_\alpha$

- Move all the edges in $SE(u,u)^\beta_\alpha$ to the new node u_α.

  For each $se(u,u) \in SE(u,u)^\beta_\alpha$

  $SE = SE \cup \{se(u_\alpha, u_\alpha)\}$

  $SE = SE - SE(u,u)^\beta_\alpha$

- Navigation edges:

  Move the relevant NEs from node u to new node u_α.

  - An outgoing edge $ne \in NE(u,v)$ is relevant if $QL\{NE(u,v)\} \subseteq (QL\{SE(u,u)_\alpha\} \cap QL\{SE(u,u)^\beta_\alpha\})$

    Let $NE(u,v)_\alpha = \{ne | ne \in NE(u,v) \wedge QL\{NE(u,v)\} \subseteq QL^\beta_\alpha\}$

    For each $ne(u,v) \in NE(u,v)_\alpha$

    $NE = NE \cup \{ne(u_\alpha, v)\}$

    $NE = NE - NE(u,v)_\alpha$

  - Similarly, an incoming edge $ne \in NE(v,u)$ is relevant if $QL\{NE(u,v)\} \subseteq (QL\{SE(u,u)_\alpha\} \cup QL\{SE(u,u)^\beta_\alpha\})$

    Let $NE(v,u)_\alpha = \{ne | ne \in NE(v,u) \wedge QL\{NE(v,u)\} \subseteq QL^\beta_\alpha\}$

    For each $ne(v,u) \in NE(v,u)_\alpha$

    $NE = NE \cup \{ne(v, u_\alpha)\}$

    $NE = NE - NE(v,u)_\alpha$

- Add the newly detected common subexpression to the set of common subexpressions.

  Let $csub = Path(Root\_u, u_\alpha)$

  $CSet = CSet \cup \{csub\}$

  Node bookkeeping

- Remove node u if there are no selection, join or navigational conditions

involving node u.

if $\{SE(u,u) \cup JE(u,v) \cup NE(u,v) \cup NE(v,u)\} = \phi$

then $N = N - \{u\}$

End if

2. Join conditions:

Let $JE(u,v)_\alpha = \{a | a \in JE(u,v)$ and has the join condition α over the two nodes $u,v \in N\}$

Let $JE(u,v)_\alpha^\beta = \{b | b \in JE(u,v)$ labeled by $JE(Q_i,\beta)$ and $\exists a \in JE(u,v)_\alpha$ labeled by $JE(Q_j,\alpha)$, where α subsumes β and $\beta \neq \alpha\}$

Let $Root_u = Root(u)$ and $Root_v = Root(v)$ be the roots of nodes u and v respectively.

Let $QL_\alpha^\beta = QL\{JE(u,v)_\alpha\} \cup QL\{JE(u,v)_\alpha^\beta\}$ be the list of IDs of the queries that contain the join conditions α and β.

If ($\mid JE(u,v)_\alpha \mid > 1$ *or* $JE(u,v)_\alpha^\beta \neq \phi$) [True if multiple identical or subsumed join conditions exist]

- Create a new node $R_{combined} = (Root_u \bowtie Root_v)$ representing the join condition α on the two data sources involving u and v.
- Collect a set of all nodes in the multigraph that are affected by the join condition α.

  $N_{affected} = \{n | n \in N \wedge QL\{NE(n,x) \cup NE(x,n)\} \subseteq QL_\alpha^\beta \wedge x \in N\}$ - $\{Root_u, Root_v\}$
- Create a corresponding new node $n_{combined}$ for each node $n \in N_{affected}$. Let $N_{combined}$ be the set of new nodes.

  For each $n \in N_{affected}$

  $N = N \cup \{n_{combined}\}$

  $N_{combined} = N_{combined} \cup \{n_{combined}\}$

End For

Edge bookkeeping

- Join edges: move relevant JEs from nodes in $N_{affected}$ to the corresponding nodes in $N_{combined}$.

  $JE_{affected} = \{je | je \in JE \wedge QL\{JE(n,x)\} \subseteq QL^{\beta}_{\alpha} n, x \in N_{affected} \wedge x \neq n\} - JE(u,v)_{\alpha}$

  For each $je(x,y) \in JE_{affected}$

  $JE = JE \cup \{je(x_{combined}, y_{combined})\}$ where $x_{combined}, y_{combined} \in N_{combined}$ are the nodes corresponding to nodes x and y from $N_{affected}$.

  $JE = JE - JE_{affected} - JE(u,v)_{\alpha}$

- Selection edges: move relevant SEs from nodes in $N_{affected}$ to the corresponding nodes in $N_{combined}$.

  $SE_{affected} = \{se | se \in SE \wedge QL\{SE(n,n)\} \subseteq QL^{\beta}_{\alpha} \wedge n \in N_{affected}\}$

  For each $se(n,n) \in SE_{affected}$

  $SE = SE \cup \{se(n_{combined}, n_{combined})\}$ where $n_{combined} \in N_{combined}$ is the corresponding node for $n \in N_{affected}$.

  $SE = SE - SE_{affected}$

- Navigation edges:

  Move the relevant NEs from nodes in $N_{affected}$ to the corresponding nodes in $N_{combined}$.

  $NE_{affected} = \{ne | ne \in NE \wedge QL\{NE(n,x) \cup NE(x,n)\} \subseteq QL^{\beta}_{\alpha} \wedge n, x \in N_{affected} \wedge x \neq n\}$

  For each $ne(x,y) \in NE_{affected}$

  $NE = NE \cup \{ne(x_{combined}, y_{combined})\}$ where $x_{combined}, y_{combined} \in N_{combined}$ are the nodes corresponding to nodes x and y from $N_{affected}$.

- Add the newly detected common subexpression to the set of common subexpressions.

Let $csub = R_{combined}$

$CSet = CSet \cup \{csub\}$

Node bookkeeping

- For each $n \in N_{affected}$

  Remove node n if there are no selection or join conditions involving node n.

  if $\{SE(n,n) \cup JE(n,x)\} = \phi$ where $x \in N$

  Remove the navigational edges on node n.

  $NE = NE - NE(n,x) - NE(x,n)$ where $x \in N_{affected}$

  $N = N - \{n\}$

  end if

  End For

- Repeat the above for loop for nodes in $N_{combined}$.

- Finally, remove the nodes $Root_u, Root_v$ if there are no navigational edges involving these nodes.

  if $\{NE(n,x)\} = \phi$ where $x \in N \wedge n = Root_u$ or $Root_v$

  $N = N - \{n\}$

  end if

End if

The heuristic rules defined above identify a set of qualifying edges (either selection or join edges) from the mixed multigraph representation of various query expressions. These edges and their connected nodes are altered based on certain conditions and a modified multigraph is constructed. The common subexpressions detection algorithm takes the original mixed multigraph representation as the input and executes the heuristic rules in a specific order to modify the multigraph. The final output of the algorithm is the modified multigraph and a set of common subexpressions *CSet* over

heterogeneous data sources. In the multigraph representation of given set of queries, there are a finite number of *SE* and *JE* edges. As the algorithm applies certain rules, a set of edges are considered and processed. Once the edges are processed, then they are marked and are not chosen again. Hence, when the algorithm applies a new rule or the same rule again, then those already examined edges are not considered again. This guarantees the termination of the algorithm in finite time. The algorithm 1 outlines the pseudo code to detect the common subexpressions across multiple queries represented in a multigraph model using the heuristic rules.

Algorithm 1 CommonSubexpressionDetection

Input: Mixed Multigraph $G(N, SE, JE, NE)$ and empty *CSet*.
Output: Modified mixed multigraph $G'(N', SE', JE', NE')$ with no related conditions and *CSet* that contains the list of common subexpressions.

```
1:  repeat
2:      Apply the heuristic rules in the specified order to select the type of edges (e.g.
    select or join)
3:      if there exists some type of commonality (e.g. identical and/or subsumed) then
4:          Take appropriate actions
5:      end if
6:      Modify the multigraph based on the actions performed
7:      if common subexpression is detected then
8:          Add the subexpression to the list CSet
9:      end if
10: until all the edges are examined and no more common edges exist
```

4.4 Example

To illustrate the working of the algorithm, consider the same two queries in Listing 4.1 with the corresponding multigraph in Figure 4.2. Recall that the queries $Q1$ and $Q2$ are defined over one relational table `LicenseInfo` and one XML document `VehicleRecords.xml`. Thus, this example illustrates the importance of detecting a common subexpression over a relational and an XML data source.

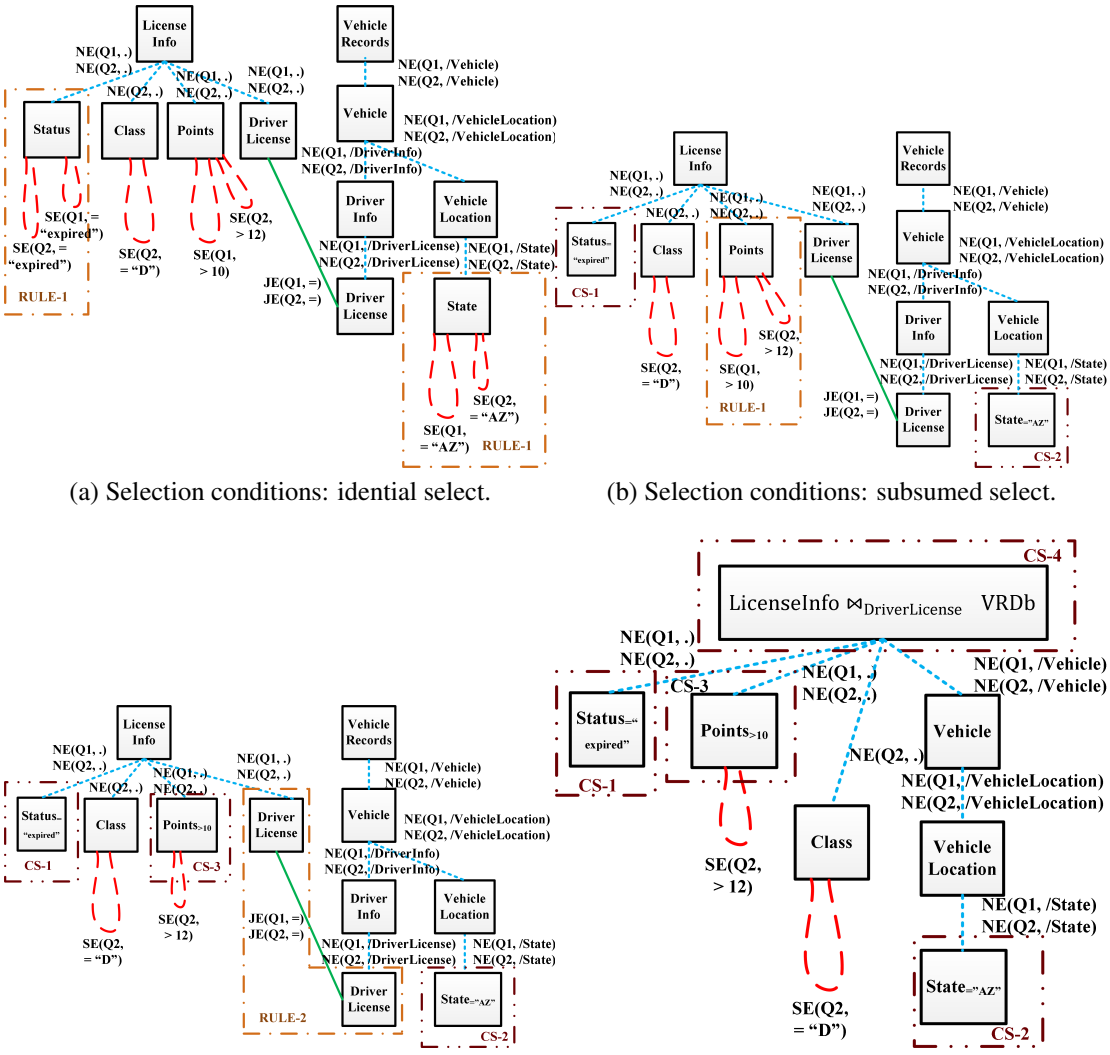
On the multigraph from Figure 4.2, apply the algorithm and heuristic rules from section 4.3. Rule 1 indicates that there are identical selection conditions on the Status and the State nodes and subsumed selection conditions on the Points node. First

analyze the identical selection conditions using dotted boxes as shown in Figure 4.3 (a). After taking the appropriate steps in rule 1, the common subexpression $CS1$ as $LicenseInfo.Status =$"$expired$" is added to the list $CSet$ and the $Status$ node is replaced by a new node $Status_{=}$ "$expired$". Similarly, the common subexpression $CS2$ as $VRDb/Vehicle/VehicleLocation/State =$"$AZ$" is added to the list $CSet$. The State node is also replaced by a new node $State_{=}$"AZ". The modified multigraph is shown in Figure 4.3 (b). The candidate common subexpressions are enclosed in DotDashed boxes.

Now, from Figure 4.3 (b), analyze the subsumed conditions over the Points node. The superset condition $Points > 10$ is applied to the Points node replacing that node with a new node $Points_{>10}$. The remaining subsumed selection condition $Points > 12$ is now applied to the new node. The common subexpression $CS3$ as $LicenseInfo.Points > 10$ is added to the list $CSet$. This change in the multigraph is shown in Figure 4.3 (c).

From Figure 4.3 (c), rule 2 detects identical join conditions $LicenseInfo.DriverLicense = VRDb/Vehicle/DriverInfo/DriverLicense$. In order to process the identical join condition, a new root node $LicenseInfo \bowtie_{DriverLicense} VRDb$ is created. All the qualifying nodes and the relevant selection, join and navigation edges as per the conditions in rule 2 are now associated with the new root node. The modified graph is shown in Figure 4.3 (d). The common subexpression $CS4$ as $LicenseInfo \bowtie_{DriverLicense} VRDb$ is added to the list $CSet$.

Since, no more edges are left to be analyzed; the algorithm terminates with four common subexpressions detected across the two queries. $CS1$ and $CS3$ are purely relational in structure, and $CS2$ is purely hierarchical in structure. However, $CS4$ is a partial join over the relational table and the XML document.



(a) Selection conditions: idential select.

(b) Selection conditions: subsumed select.

(c) Join conditions: identical join.

(d) Final query graph.

Figure 4.3: Detecting common subexpressions from a multigraph.

4.5 Summary

The detection of common subexpressions plays an important role in optimizing multiple queries. Common subexpressions can be selectively materialized into views to answer future queries. In the DEPA environment, each agent registers a set of queries accessing distributed heterogeneous structured data sources. Hence, it is important to detect these common subexpressions to facilitate the efficient execution of the queries. This chapter has presented a mixed multigraph model to represent query expressions over relational and structured XML data sources. Detecting common subexpressions is

a well-known NP-hard problem. Thus to address this problem, this research has contributed the design and implementation of a heuristics-based algorithm that can analyze the multigraph and detect common subexpressions across heterogeneous data sources. As seen in the example, a common subexpression can be a hybrid join over relational as well as XML data source. These common subexpressions are the potential candidates for materialized views. The next chapter in this thesis explores specific research challenges in defining such materialized views using LINQ as the materialized view definition language.

Chapter 5

DEFINING MATERIALIZED VIEWS OVER HETEROGENEOUS DATA SOURCES

The distributed event stream processing environment integrates data from heterogeneous data sources using various types of query expressions, including continuous queries over events and data streams, queries over relational and XML data sources, and primitive and composite event definitions with access to heterogeneous data sources. The goal of this research is to materialize views of common subexpressions across these heterogeneous data sources to improve performance. This chapter first describes the language features required for the materialized view definition language in the DEPA framework and then discusses the capabilities provided by the LINQ language as the materialized view definition language. The chapter then presents an algorithm to define a view in LINQ for the identified common subexpressions. The original queries can then be rewritten to use the newly defined view. Once the view is defined in LINQ, there must be an appropriate data structure for storing the persistent results of a query over heterogeneous data sources. An algorithm is presented that generates the statements required to persist the view. One challenge of this work is the persistence of hybrid views defined in LINQ over relational and XML data sources. The presented algorithm contributes an approach to persist such views in a relational table with XML columns for XML data. Examples are provided to illustrate both the view definition and creation.

5.1 Materialized View Definition Language

The distributed stream processing environment described in chapter 1 integrates data coming from heterogeneous data sources especially relational and structured XML sources. Such an environment uses different query expressions such as continuous queries, SQL queries, XQuery queries and LINQ queries to access the heterogeneous data sources subscribed by the individual agent. The focus of this research is to pro-

vide capabilities of extracting common subexpressions across these different queries and materialize these partial joins into views that will provide better performance in executing the queries. These materialized views can be over purely relational or purely XML or even over a hybrid combination of the two. In order to define such kinds of materialized views, following are the necessary features that a materialized view definition language (MVDL) should have:

1. **Querying Relational Sources:** The MVDL should be able to fully accommodate querying relational tables from different data sources. Each DEPA can subscribe to individual relational tables from different relational database systems in order to gain access to different data needed for its initial view materialization or for subsequent incremental view maintenance. The MVDL should be able to obtain the metadata level information of each of these sources from the metadata repository of that DEPA.

2. **Querying XML Sources:** The MVDL should also be able to query XML documents (with or without) XML Schema. As with relational sources, the MVDL should be able to access the metadata information from the repository for its initial view materialization or for subsequent incremental view maintenance.

3. **Querying Collection of Objects:** The DEPA maintains collection of objects (lists, arrays) to store data such as metadata repository, temporary data and data received over the streams by Sybase CEP processor. The temporary data can be in terms of Datasets while the streaming data can be either a single object or a list. The DEPA is capable of processing data received over the streams to answer certain queries. In addition to this, the changes occurring over the base data sources are streamed to the DEPA that contains materialized views over these base data sources. The DEPA uses these deltas in incremental view maintenance of the views defined locally to the agent. Thus, MVDL should be able to access

these delta objects and use them in the view maintenance algorithm.

4. **Querying Heterogeneous Sources Together:** The main required feature of MVDL suitable for the DEPA environment is the capability of MVDL to query heterogeneous data sources within a single query. Thus, MVDL should be able to access data from relational as well as from XML data sources using a single view definition.

In the past, SQL has been considered as the main MVDL for defining and maintaining materialized views over relational sources [15]. Recently, for defining and maintaining XML views over XML data, XQuery is chosen as the MVDL [18]. However, for defining views over both relational and XML sources, researchers have explored the use of either SQL or XQuery as the MVDL. Some of the work that used SQL, first converted the XML data into relational format and then used SQL to define materialized views over the relational data sources [19, 27]. While others who used XQuery as the MVDL, converted the relational data into XML and then used XQuery to define views which are materialized into different XML documents [28]. However, both the approaches have certain drawbacks. The data has to be converted back and forth from one format to the other which is time consuming. Also, additional mapping information is required to transform the results from the converted format back to the original source format. The time required to convert data from one format to the other increases during incremental view maintenance whenever deltas are received in smaller chunks. This research is relaxing this constraint by taking the advantage of established technology behind each data format. Thus, the materialized views defined in the DEPA framework will retain the data in their original format. Considering these different requirements for MVDL, this dissertation is exploring the use of LINQ as the language accessing multiple heterogeneous data sources through a single query and defining and maintaining LINQ-based materialized views.

5.2 LINQ as Materialized View Definition Language

LINQ is a declarative query language that extends object-oriented programming languages like C# and Visual Basic with a native language syntax for querying different types of data sources or collections. LINQ uses the familiar SQL-like syntax with `from`, `where` and `select` clauses to query heterogeneous data sources through a single query. The `from` clause iterates over a collection, the `where` clause defines the filters over the collection and the `select` clause defines the desired output. An introduction to LINQ is given in chapter 2.

LINQ provides a layer of abstraction over different data sources with the help of an extensive set of class libraries. A typical LINQ query is compiled and internally represented as an expression tree. Based on the type of data source, this expression tree is translated into the corresponding query language to take advantage of the underlying technology for each of the individual data sources to return the desired results. One of the challenges of this research is the materialization of the heterogeneous results while retaining the corresponding data formats. Prior approaches have handled this problem by first converting one format to the other and then defining the materialized views over the now homogeneous data sources. However, this dissertation is exploring the storage of heterogeneous query results as materialized views. An extensive literature search did not reveal any existing work that addresses this research challenge.

As a motivational example for this research challenge, consider a LINQ query *Q* over one relational data source `LicenseInfo` and an XML document `Vehicle-Records.XML` that returns all the active license numbers along with the vehicle information associated with each license number as shown in Listing 5.1.

```
Query Q:
XElement VRDb = XElement.Load("VehicleRecords.xml");
var query =
from vr in VRDb.Elements("Vehicle")
from li in LicenseInfo
where li.Points == 10 &&
      li.Status == "active" &&
```

```
      vr.Element("VehicleLocation").Element("State").Value.Equals("AZ
          ") &&
      vr.Element("DriverInfo").Element("DriverLicense").Value.Equals(
          li.DriverLicense)
select new {DriverLicense = li.DriverLicense,
            Status = li.Status,
            VehicleInfo = vr
           };
```

Listing 5.1: A sample LINQ query over relational and XML data sources.

The output of this particular query contains partial results from the relational data source `LicenseInfo` and partial results from the XML document `Vehicle-Records.XML`. Sample output with one tuple is shown in Table 5.1.

The research challenge is to take this type of query output and materialize it into a persistent view retaining the columns `DriverLicense` and `Status` as relational data and `VehicleInfo` column as the XML data. Any typical materialized view definition language has two counterparts. There is a semantic definition of the view that decides which qualifying tuples from the base data sources are used to populate the view and a persistence definition that denotes how the view is materialized or persisted. The LINQ query provides the semantic definition of the view. However, the LINQ framework does not provide any mechanism to persist this view. This research presents a solution for this problem by constructing a view creation statement that will persist the qualifying tuples into an appropriate data structure that will retain the data in their native format. This research has explored the use of the `Create Table` statement to persist a relational or hybrid view. For the hybrid view, the XML data will be stored in an XML column. For a pure XML view, an `XML Schema` will be created so that the qualifying XML records will be stored in an XML document validated against the schema definition. The corresponding `Create Table` statement for the above sample LINQ query over relational and XML data sources is shown in Listing 5.2. The algorithms to create LINQ-based materialized views are described in the next section.

```
Create Table Q (DriverLicense string,
                Status string,
```

```
                VehicleInfo XML);
```

Listing 5.2: The corresponding create statement for hybrid view.

DriverLicense	Status	VehicleInfo
G97910467	Active	<Vehicle id="11"> <VehicleInformation> <Vin>0ANSU809MIJ610291</Vin> <PlateNumber>CJE-3154</PlateNumber> <Make>Audi</Make> <BodyStyle>Convertible</BodyStyle> <Color>Average red</Color> <Year>2000</Year> <Model>Impreza</Model> <ListPrice>$34191.32</ListPrice> <Fuel>Hybrid</Fuel> </VehicleInformation> <RegistrationInformation> <StartDate>2/26/2009</StartDate> <EndDate>2/21/2021</EndDate> </RegistrationInformation> <DriverInformation> <DriverLicense>G97910467</DriverLicense> </DriverInformation> <VehicleLocation> <Street>2665 Post Avenue</Street> <City>Sedona</City> <State>AZ</State> <Zip>86336</Zip> <Country>United States</Country> </VehicleLocation> </Vehicle>

Table 5.1: Sample output of LINQ query containing part relational and part XML result.

5.3 View Definition and Creation

Each agent in the DEPA framework selectively materializes the common subexpressions across heterogeneous data sources for efficient processing of different queries registered with it. Common subexpressions are detected from various queries represented using a mixed multigraph model as described in chapter 4. Since the common subexpressions are across heterogeneous data sources, the materialized views can be

either pure relational, pure XML or a hybrid combination of the two. This research is exploring the use of LINQ as the MVDL for defining and persisting these views. Any typical MVDL has two counterparts. There is a semantic definition of the view that decides which qualifying tuples from the base data sources are used to populate the view and there is a persistence definition that denotes how the view is materialized or persisted for later use. Similarly, in order to use LINQ as the MVDL, the LINQ query serves as the semantic definition for the qualifying tuples. However, the LINQ framework does not offer any capabilities/functionalities to persist the results from the LINQ query. Thus, this chapter focuses on designing two algorithms: one view definition algorithm to construct a LINQ query from the set of common subexpressions that will serve as the semantic view definition; and the second view creation algorithm that will create an appropriate data structure to persist the output from the LINQ query. Since the views can be either purely relational, purely XML or a hybrid of relational and XML data sources, the algorithm detects that particular scenario and generates either a `Create Table` statement for a relational or a hybrid view or an `XML Schema` for a pure XML view. Both the algorithms access the metadata repository and the modified multigraph to generate the two statements from the common subexpressions. Figure 5.1 shows the high-level process view of the two algorithms indicating the input requirements and the output generated.

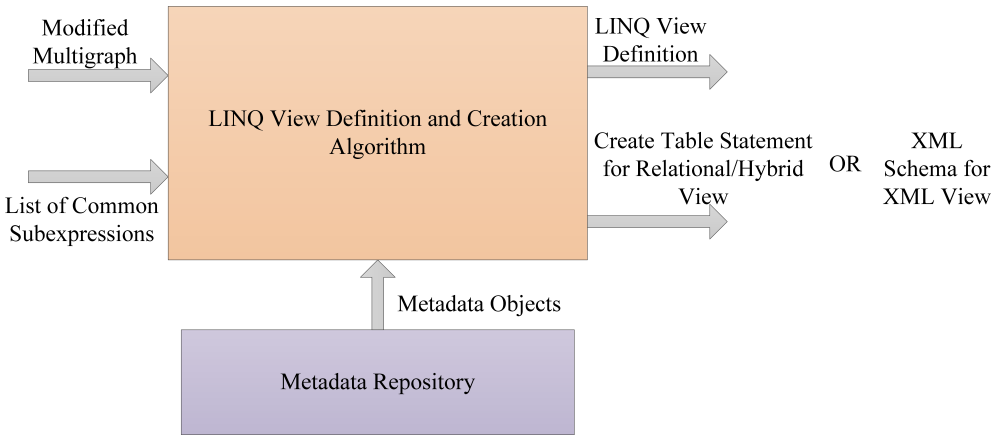


Figure 5.1: Process for defining and creating views.

The view definition and creation statements are created using the common subexpressions, and the metadata objects for the corresponding base data sources from the metadata repository. While creating these view statements, the algorithm uses different data structures to fill the relevant information into the `View` structure described in Figure 5.2. The View data structure comprises of the information necessary to identify (`ViewName` and `ViewType`) and create the view (`ViewDefinition` and `ViewCreation`). The View structure also maintains a list of common subexpressions from which the view is created (`csubs`), the list of view attributes that defines the structure of the view (`attributes`), the list of base data sources from which the view is derived (`BaseDataSources`). To create the LINQ-based view definition, the common subexpressions are transformed into the corresponding `where` clauses, the base data sources are used to create the `from` clauses and the `select` clauses are created using the attributes of the view. The detailed algorithm for the view definition and creation statements is described in the following section along with a working examples that illustrates the use of the data structures.

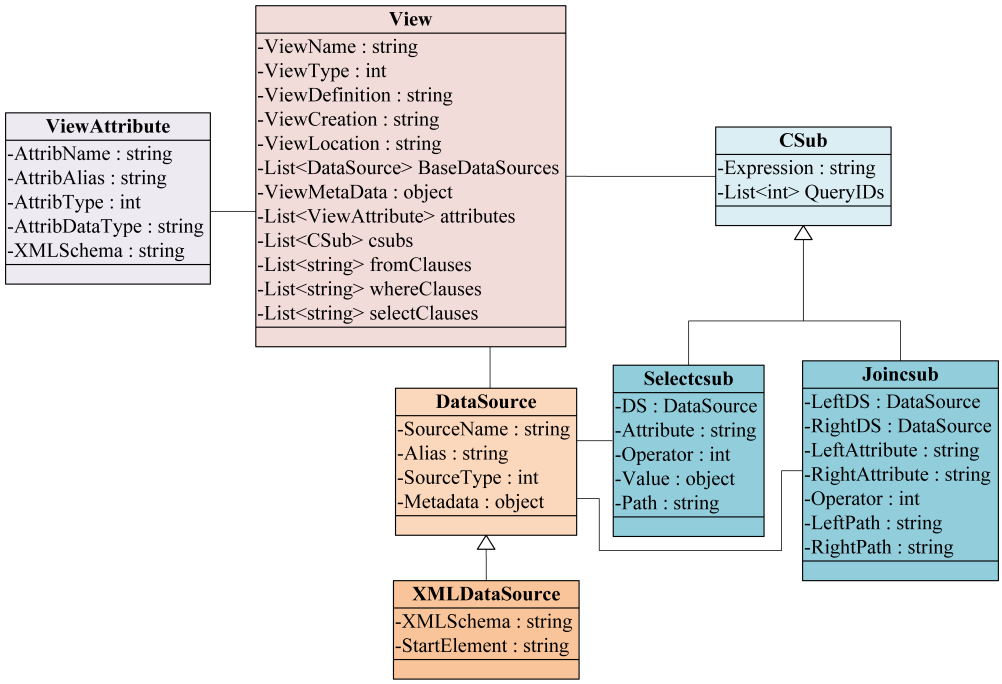


Figure 5.2: UML diagram for view data structures.

Materialized View Definition Algorithm

Let Q be the set of the queries that share all the common subexpressions *csub* in the set *CSet*. Each *csub* can be either of selection type $Selectcsub$ or of join type *Joincsub*. The View data structure is used to maintain all the relevant information regarding the original data sources from which the view is derived, the set of common subexpressions *CSet* which defines the semantics of the view and other necessary information to build and maintain the view. The details are shown in the UML diagram in Figure 5.2.

The materialized view definition algorithm has two parts. The first part of the algorithm takes the partially filled view structure V and creates the view definition $V.ViewDefinition$ in LINQ language. This LINQ query decides which tuples from the base data sources qualify to be in the view V. The view V defined by the LINQ query needs to be persistent in nature. The second part of the algorithm creates the view creation statement $V.ViewCreation$ using the view structure V and corresponding meta-data level information of the base data sources. The following pseudo code describes the algorithm for creating the view definition $V.ViewDefinition$.

Algorithm: $ViewDefinition$

Input: The final modified Multigraph $G'(N', SE', JE', NE')$, the set of common subexpressions *CSet*, the partially filled view structure V and its corresponding required data structures shown in Figure 5.2, and the original queries Q.

Output: Materialized View Definition in LINQ language $V.ViewDefinition$.

Pseudo code:

Let $V.ViewName$ be the name of the materialized view V. Let $V.ViewDefinition =$"".

Let $V.BaseDataSources$ be the set of base data sources from which the view V is derived. Each object in $V.BaseDataSources$ contains the information regarding the base data source and its corresponding metadata object.

/* Since the views can be of hybrid nature and in order to execute the hybrid LINQ queries correctly, it is required that for such hybrid LINQ queries the first base data

source should be an XML data source (only applicable to hybrid LINQ queries).*/

Sort $V.BaseDataSources$ based on $V.SourceType$ to group all the XML data sources together at the beginning of the list.

For each $ds \in V.BaseDataSources$

Generate an alias $dsalias$ for $ds.SourceName$ to be used in the LINQ from clauses.

/* For example: Create alias $T0$ for first data source in the list, $T1$ for the second data source in the list and so on. */

$ds.Alias = dsalias$;

For each $ds \in V.BaseDataSources$

$V.fromClauses = V.fromClauses \cup \{\text{“}from\text{”} + ds.Alias + \text{“}in\text{”} + ds.SourceName\}$

If $ds.SourceType = 0$ /*Source is a relational table */

/* Get all the attributes and their data types for ds and add the attributes to the list of view attributes. */

$attribList = ds.Metadata.getAttributes()$;

For each $attrib \in attribList$

Generate an alias $attribAlias$ for $attrib$ to be used in the LINQ select clauses.

$V.attributes = V.attributes \cup new\ ViewAttribute(attrib, attribAlias, 0, attrib.datatype, null)$;

$V.selectClauses = V.selectClauses \cup \{attribAlias + \text{“}=\text{”} + ds.Alias + \text{“}.\text{”} + attrib\}$;

Else if $ds.SourceType = 1$ /*Source is an XML document */

/* Get the XML schema for ds, extract partial XML schema relevant for the view and a new XML attribute to the list of view attributes. */

Let $attribPartialXMLSchema$ be the extracted partial XML Schema.

$V.attributes = V.attributes \cup new\,ViewAttribute(null,\ ds.Alias,\ 1,\text{“}XML\text{”},$
$attribPartialXMLSchema)$;

$V.selectClauses = V.selectClauses \cup \{ds.Alias+\text{“}=\text{”}+ds.Alias\}$;

End if

EndFor

For each $csub \in CSet$

/*From each *csub*, replace each data source name with its corresponding alias from *V.BaseDataSources*.*/

If *csub* is a *Selectcub*

$tempExpression = csub.DS.Alias + csub.Path + csub.Attribute+\text{“ ”} +$
$csub.Operator +\text{“ ”} + csub.Value$;

Else if *csub* is a *Joincub*

$tempExpression = csub.LeftDS.Alias + csub.LeftPath + csub.LeftAttribute$
$+\text{“ ”} + csub.Operator+\text{“ ”} + csub.RightDS.Alias + csub.RightPath +$
$csub.RightAttribute$;

End if

$V.whereClauses = V.whereClauses \cup \{tempExpression\}$;

EndFor

/* Build the final where clause string by concatenating all the individual where clauses. */

Let $w = |V.whereClauses|$;

$finalwhereClause =\text{“}where\text{ ”}$;

For $i = 0\ to\ w-2$

$finalwhereClause = finalwhereClause + v.whereClauses[i]+\text{“ }and\ \backslash n\text{”}$;

$finalwhereClause = finalwhereClause + v.whereClauses[w-1]+\text{“ }\backslash n\text{”}$;

/* Build the final select clause string by concatenating all the individual select clauses.

*/

Let $s = |V.selectClauses|$;

$finalselectClause =$“$selectnew\{$”;

For $i = 0\ to\ s-2$

$finalselectClause = finalselectClause + v.selectClauses[i]+$“$,\backslash n$”;

$finalselectClause = finalselectClause + v.selectClauses[s-1]+$“$\};$”

/* Build the final from clause string by concatenating all the individual from clauses.

*/

Let $f = |V.fromClauses|$;

For $i = 0\ to\ f-1$

$finalfromClause = finalfromClause + v.fromClauses[i]+$“$\backslash n$”;

$V.ViewDefinition = finalfromClause + finalwhereClause + finalselectClause+$“;”;

To illustrate the working of the materialized view definition algorithm, this section continues with the same example from chapter 4 section 4.4. To recapitulate, the following Listing 5.3 shows the two queries under consideration for detecting common subexpressions.

```
Query Q1:
XElement VRDb = XElement.Load("VehicleRecords.xml");
var query1 =
from li in LicenseInfo
from vr in VRDb.Elements("Vehicle")
where li.points > 10 &&
      li.status == "expired" &&
      vr.Element("VehicleLocation").Element("State").Value.Equals("AZ
          ") &&
      vr.Element("DriverInfo").Element("DriverLicense").Value.Equals(
          li.DriverLicense)
select new {/* project needed fields */};

Query Q2:
XElement VRDb = XElement.Load("VehicleRecords.xml");
var query2 =
from li in LicenseInfo
from vr in VRDb.Elements("Vehicle")
where li.class = "D" &&
      li.points > 12 &&
```

```
      li.status == "expired" &&
      vr.Element("VehicleLocation").Element("State").Value.Equals("AZ
         ") &&
      vr.Element("DriverInfo").Element("DriverLicense").Value.Equals(
         li.DriverLicense)
select new {/* project needed fields */};
```

Listing 5.3: Sample LINQ queries.

After applying the common subexpression detection algorithm from chapter 4 section 4.2, four common subexpressions are detected as shown in Listing 5.4 below:

```
CS1 : LicenseInfo.Status == "expired"
CS2 : VRDb/Vehicle/VehicleLocation/State == "AZ"
CS3 : LicenseInfo.Points > 10
CS4 : LicenseInfo.DriverLicense == VRDb/Vehicle/DriverInfo/
   DriverLicense
```

Listing 5.4: Common subexpressions.

The corresponding filled data structure of common subexpressions *csubs* used in conjunction with the view structure is shown in Table 5.2.

Consider that these common subexpressions are to be materialized into a view with *ViewName* as $V1$ (the name of the view is automatically generated based on how many views are previous defined in that DEPA. As the view definition algorithm executes, the view data structure gets filled with appropriate values and they are as shown below in multiple tables. Since the view is based on two data sources, relational data source `LicenseInfo` and XML data source `VRDb`, the *ViewType* is 2. It is difficult to show all the values for the view structure V in one single table and hence for clarity, the values are grouped together and shown separately. The BaseDataSources and their attributes are shown in Table 5.3. Note that not all the attributes are shown here. But the actual view data structure contains all the attributes from the $LicenseInfo$ relational table and one attribute for the XML column for the data from XML data source $VRDb$ (`VehicleRecords.xml`).

The algorithm uses the common subexpressions *csubs*, *BaseDataSources* and their *Attributes* to create the various LINQ clauses required to construct the view defi-

Selectcsubs

csub #	DS	Attribute	Opt	Value	Path
CS1	LicenseInfo	Status	==	“expired”	null
CS3	LicenseInfo	Points	>	10	null
CS2	VRDb	State	==	“AZ”	Path(VRDb, State) = “/Vehicle/VehicleLoc”

Joincsubs

csub #	LeftDS	LeftAttrib	RightDS	RightAttrib	Opt	LeftPath	RightPath
CS4	LicenseInfo	DriverLicense	VRDb	DriverLicense	==	null	Path(VRDb, DriverLicense) = “/Vehicle/DriverInfo”

Table 5.2: Common subexpressions in the view data structure.

BaseDataSources

SourceName	**Alias**	**SourceType**	**MetaData**	**XMLSchema**
LicenseInfo	T1	0 - (Rel)	Metadata Object	null
VRDb	T0	1- (XML)	Metadata Object	<Vehicles> <Vehicle> ... </Vehicle> </Vehicles>

Attributes

AttribName	**AttribAlias**	**AttribType**	**AttribDataType**	**XMLSchema**
LicenseNumber	LicenseInfo_LicenseNumber	0 - (Rel)	string	null
⋮ all the	⋮ attributes	⋮ of	⋮ LicenseInfo	⋮ Table
null	T0	1 - (XML)	XML	<Vehicle> <VehicleInfo> ... </VehicleInfo> <VehicleLocation> ... </VehicleLocation> <DriverInfo> ... </DriverInfo> </Vehicle>

Table 5.3: Base data sources and their attributes.

nition statement. The *from* clauses, *where* clauses and the *select* clauses are shown in Table 5.4.

Finally, all the different clauses are combined together to form the final LINQ-based view definition. This statement is shown in Table 5.5.

Materialized View Creation Algorithm

The view definition statement constructed by the previous algorithm will determine the qualifying tuples for the view. However, the LINQ framework does not provide any mechanism or operator to materialize this view. This dissertation has designed an algorithm that will generate the corresponding view creation statement that will assist in persisting the view. Based on the type of the view, the following pseudo code uses all the partially filled data structures from the view definition algorithm to generate materialized view creation statement $V.ViewCreation$, which will persist the view locally to the DEPA.

Algorithm: *ViewCreation*

Input: The final modified Multigraph $G'(N', SE', JE', NE')$, the set of common subexpressions $CSet$, the original queries Q, and the materialized View definition in LINQ language $V.ViewDefinition$ created using the algorithm $ViewDefinition$, and the view structure V and its corresponding required data structures shown in Figure 5.2.

Output: The view creation statement $V.ViewCreation$ that can be used to create the actual view in the database. If the view is pure relational or hybrid in structure, then the $V.ViewCreation$ statement is a create table statement that will define a table in relational database. If the view is pure XML in structure, then the $V.ViewCreation$ statement is the XML Schema corresponding to the XML view.

Pseudo code:

From the $ViewDefinition$ algorithm, the view structure V and its corresponding relevant data structures contain all the necessary information regarding the structure of the view.

from Clauses	**finalfromClause**
from T0 in VRDb.Elements(“Vehicle”)	from T0 in VRDb.Elements(“Vehicle”) from T1 in LicenseInfo
from T1 in LicenseInfo	

where Clauses	**finalwhereClause**
T0.Element(“VehicleLocation”).Element(“State”) == “AZ”	where T0.Element(“VehicleLocation”). Element(“State”) == “AZ” && T1.Status == “expired” && T1.Points $>$ 10 && T1.DriverLicense == T0.Element(“DriverInfo”). Element(“DriverLicense”). Value()
T1.Status == “expired”	
T1.Points $>$ 10	
T1.DriverLicense == T0.Element(“DriverInfo”). Element(“DriverLicense”). Value()	

select Clauses	**finalselectClause**
LicenseInfo_DriverLicense = T1.DriverLicense	select new { LicenseInfo_DriverLicense = T1.DriverLicense, LicenseInfo_Class = T1.Class, LicenseInfo_Points = T1.Points, . . . , T0 = T0 }
LicenseInfo_Class = T1.Class	
LicenseInfo_Points = T1.Points	
. . . . . . all the attributes of LicenseInfo Table	
T0 = T0	

Table 5.4: LINQ clauses.

ViewDefinition	from T0 in VRDb.Elements(“Vehicle”) from T1 in LicenseInfo where T0.Element(“VehicleLocation”).Element(“State”) == “AZ” && T1.Status == “expired” && T1.Points $>$ 10 && T1.DriverLicense == T0.Element(“DriverInfo”).Element(“DriverLicense”). Value() select new { LicenseInfo_DriverLicense = T1.DriverLicense, LicenseInfo_Class = T1.Class, LicenseInfo_Points = T1.Points, . . . , T0 = T0 };

Table 5.5: View definition statement.

Let $V.ViewCreation =$“”.

Let $R_V = |ds| \mid ds \in V.BaseDataSourcesandds.SourceType = 0$;

Let $X_V = |ds| \mid ds \in V.BaseDataSourcesandds.SourceType = 1$;

If $X_V \geq 1 and R_V \geq 1$

$V.ViewType = 2$ /* View is of Hybrid type */

Else if $X_V \geq 1 and R_V = 0$

$V.ViewType = 1$ /* View is of XML type */

Else if $X_V = 0 and R_V \geq 1$

$V.ViewType = 0$ /* View is of Relational type */

End if

If $R_V \geq 1$ /* The view V is created from at least one relational base data source */

$CreateStatement =$“$Createtable$ ”$+V.ViewName+$“$(\backslash n$”;

Let $attribcount = |V.Attributes|$;

For each $i = 0$ to $attribcount - 2$

$attrib = V.Attributes[i]$;

$CreateStatement = CreateStatement + attrib.AttribAlias+$“ ” $+$

$attrib.AttribDataType +$ “$,\backslash n$”;

$attrib = V.Attributes[attribcount - 1]$;

$CreateStatement = CreateStatement + attrib.AttribAlias+$ “ ” $+$

$attrib.AttribDataType +$ “$\backslash n$”;

$CreateStatement = CreateStatement+$“);”

$V.ViewCreation = CreateStatement$;

End if

/* Create a single XSD from all the individual partial XML Schemas in $V.Attributes$.

The root node of this resultant XML Schema is $V.ViewName$.*/

If $X_V \geq 1 and R_V = 0$ /* The view V is pure XML in structure */

/* LINQ query to add *V.ViewName* as the root level element, add "Subtree" as the XML element to combine all the partial XML Schemas in a sequence. */

V.ViewCreation =XML Schema output from the LINQ query;

End if

To illustrate the working of this algorithm, consider the partially filled view structure *V* and the additional supporting data structures that were filled during the execution of the view definition algorithm. In this case, the view is based on two data sources i.e. one relational data source `LicenseInfo` one XML data source `VehicleRecords.xml`. Since the bases data sources contain at least one relational data source, the *ViewType* is 2 and hence, the view creation statement is a `Create Table` statement. This statement will create a table in the relational database with relational data columns for all the attributes from the `LicenseInfo` table and one XML type column for the XML data from `VehicleRecords.xml`. The view creation statement is shown in Table 5.6.

Thus, the view definition and creation algorithms generate a LINQ-based view definition that can be used to semantically determine the qualifying tuples going into the view persisted using the view creation statement. The `View` data structure is stored in the metadata repository since the `View` structure will play an important role in the incremental maintenance of the materialized view.

5.4 Summary

This chapter has explored the use of LINQ as the materialized view definition language to define materialized views local to each agent in the DEPA environment. LINQ provides the capabilities to query heterogeneous data sources through a single query and allow the data to be retained in their original format. Thus, a LINQ-based materialized view can be either purely relational or purely XML or a hybrid combination of the two. The view definition algorithm described in this chapter uses the detected common

ViewCreation	Create Table V2 (LicenseInfo_DriverLicense string, LicenseInfo_Class string, LicenseInfo_Points int, . . . , T0 XML);

Table 5.6: View creation statement.

subexpressions across the heterogeneous data sources to generate a LINQ query that can be used to populate the materialized view with the qualifying tuples. This algorithm accesses the metadata for the base data sources from the metadata repository and populates the `View` data structure with the necessary information to create the LINQ view definition. This research has proposed a view creation algorithm that will persist the results from LINQ queries depending on the type of the base data sources to enhance the functionalities of the LINQ framework. The example in the chapter illustrated the workings of both the algorithms and the sample LINQ-based view definition and the corresponding creation statement. After the materialization of the views, the changes occurring over the base data sources need to be captured and used in the incremental maintenance of these views, which is described in the next chapter.

Chapter 6

INCREMENTALLY MAINTAINING MATERIALIZED VIEWS

A distributed event stream processing framework can take advantage of materialized views for efficient processing of the various queries defined over heterogeneous data sources. Materialized views are defined over such heterogeneous data sources so that the original queries can be rewritten to access the materialized views instead of recomputing the view using the base data sources. This will avoid long trips to original data sources since the views are materialized locally in the system. However, as the base data sources change, it is necessary to update the materialized views in order to reflect these changes. Materialized views can either be re-derived completely from the updated base data sources or incrementally updated using the changes. This chapter is focusing on the incremental maintenance of materialized views defined over heterogeneous data sources, such as relational tables and structured XML documents. The chapter first explains the change-data-capture mechanisms and the typical structure of the logical change records in commercial database management systems used in this research. A common delta structure is proposed that can be used to update the views incrementally. The chapter explains the use of triggers and stored procedures to transform the logical change record structures from the relational databases into this common delta format. Once the changes are captured in this delta structure, then the event stream processor can stream these changes to their respective DEPAs within the framework. This chapter then introduces the concept of magic sets and their use in optimizing non-recursive queries in relational database systems. This dissertation is exploring the use of magic sets to efficiently materialize the views over both relational and structured XML data sources as well as the subsequent use of the magic tables to propagate the relevant changes to the views and thus, incrementally updating the materialized views. Finally, the working of the algorithm is illustrated by a detailed example over the criminal justice data model.

6.1 Capturing Changes in DEPA

The changes or deltas to the underlying heterogeneous data sources must be captured and propagated to a DEPA at which a view is materialized based on the changed data sources. The delta will then be used in the incremental view maintenance algorithm described later in the chapter. Since one of the goals of this research is to provide a proof-of-concept implementation within the DEPA framework, this section addresses how changes can be captured for both relational and XML data sources. Specifically the change data capture process is described for two commercial database systems, SQL Server 2008 enterprise edition and Oracle 11g enterprise edition. The mapping of the specific formats of the change information for each database system is performed to a common delta structure, which is then streamed to the appropriate DEPA. In addition, the changes to XML documents must also be captured. These changes are modeled as individual source update trees (SUTs) based on the XQuery update language [63]. The SUTs are incorporated in a proposed XML delta structure and streamed within the DEPA framework. An overview of the process is shown in Figure 6.1 with the details of the capturing and streaming of the changes being described in the remainder of the section.

Common Relational Delta Structure

Each agent in the DEPA framework takes advantage of the locally defined materialized views for efficient query processing. The changes occurring on the base data sources are captured and are used to incrementally update these views. This research is exploring the research challenge of capturing the deltas in their native format and streaming these changes to the respective agents using an event stream processor. Since the change capture mechanisms vary for each database management system, this research is proposing a common relational delta structure as shown in Table 6.1 for the use within the DEPA framework. The first attribute `deltakey` is the unique delta id that serves as the primary key for the delta table. The second attribute `start_lsn` is the

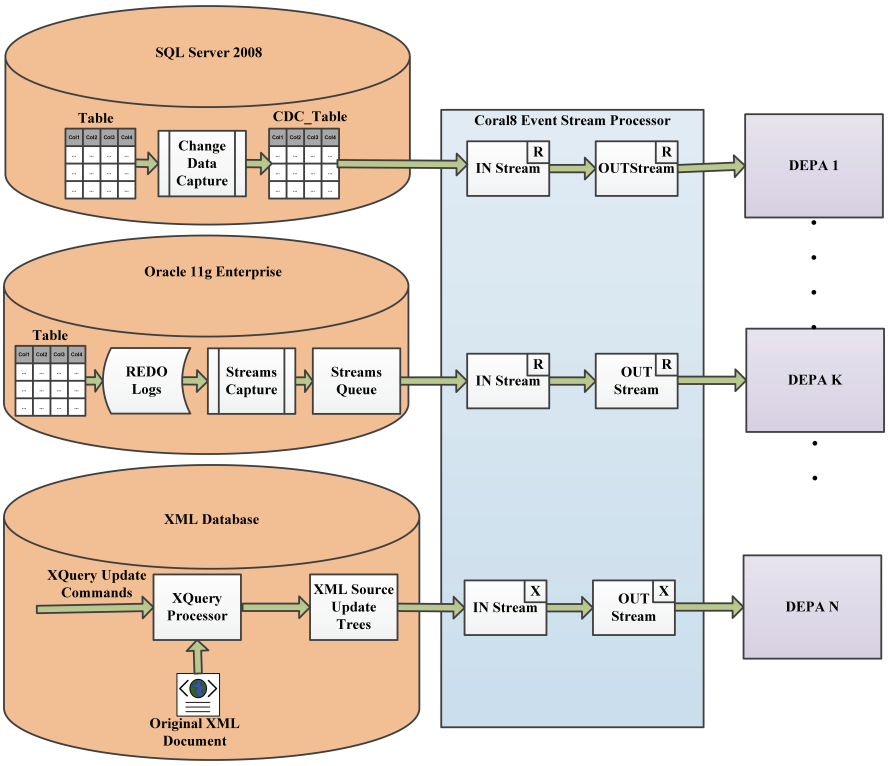


Figure 6.1: Capturing changes in heterogeneous data sources.

unique transaction id assigned by the individual database management system. This transaction id can be used to group all the changes captured during a single transaction. The `operation` attribute identifies the type of the change operation. The changes can be either delete, insert, or update, which includes both the old values and the new values. The rest of the attributes are the column attributes from the original relational table that is configured for capturing changes.

There are advantages of using a common relational delta structure within the DEPA framework. The deltas captured by the commercial systems contain additional log information that is needed for the internal workings of the specific database system. This information is not required for the incremental view maintenance. Thus, the size of the common delta structure is less than the size of the original deltas captured by the database systems. Also, a stream processor needs either a unique timestamp or a primary key to determine the order of the tuples received or processed. The change

Column Names	Data Type	Description
deltakey	int	Unique sequence number associated with the commit transaction for the change.
start_lsn	int	Transaction ID associated with the commit transaction for the change. This column is important because the transaction ID is required to associate the two update rows for old values and the new values.
operation	int	Identifies the data manipulation language (DML) operation associated with the change. Can be one of the following: 1 = delete 2 = insert 3 = update (old values) Column data has row values before executing the update statement. 4 = update (new values) Column data has row values after executing the update statement.
Source Table Columns	Varies	The remaining columns in the Delta table are the columns from the original table that were identified as captured columns when the capture instance was created.

Table 6.1: Proposed common relational delta structure.

data capture mechanisms provided by the commercial systems do not have a primary key that is understandable by the stream processor. This common delta structure has a primary key that will allow the deltas to be streamed to the agents. The stream processor can use either the pooling method (pull method) to grab the new deltas or use the notification method (push method) that will notify the stream server of the new changes so that CEP server can get the new deltas and pass them on to the DEPA. With the use of the common relational delta structure, the changes from different relational database systems can be transformed into a common format that is understandable by the DEPAs and that can be streamed within the framework.

This section elaborates on how two popular commercial relational database systems capture changes and how the change information is mapped to the common relational delta structure for use in the DEPA framework. SQL Server 2008 enterprise edition uses a feature called Change Data Capture (CDC). The database must be enabled to use this feature by running a script through a `sysadmin` privilege account, as shown in Listing 6.1.

```
-- ================================
-- Enable Database for CDC template
-- ================================
USE <Database Name>
GO

EXEC sys.sp_cdc_enable_db
GO
```

Listing 6.1: Script to enable a database for CDC.

Thus, to enable the CriminalJustice database or TPCH Database, replace <Database Name> with "CriminalJustice" or "TPCH". To disable or stop the database for CDC, run the script shown in Listing 6.2.

```
-- ================================
-- Disable Database for CDC Template
-- ================================
USE <Database Name>
GO

EXEC sys.sp_cdc_disable_db
GO
```

Listing 6.2: Script to disable a database for CDC.

Once the database is enabled, then the DEPA user or programmer can enable a particular table or all the tables from that database registered with the DEPA. In order to enable a particular table for CDC, run a script through the regular database user account as shown in Listing 6.3.

```
-- ================================================
```

```
-- Enable a Table for All and Net Changes Queries template
-- ======================================================
USE <Database Name>
GO

EXEC sys.sp_cdc_enable_table
@source_schema = N'<Schema Name>',
@source_name   = N'<Table Name>',
@role_name     = N'MyRole',
@supports_net_changes = 1
GO
```

Listing 6.3: Script to enable a table for CDC.

In order to disable a particular table for CDC, run a script through the regular database user account as shown in Listing 6.4 .

```
-- ======================================================
-- Disable a Table for All and Net Changes Queries template
-- ======================================================
USE <Database Name>
GO

EXEC sys.sp_cdc_disable_table
@source_schema = N'<Schema Name>',
@source_name   = N'<Table Name>'
GO
```

Listing 6.4: Script to disable a table for CDC.

Once the database and the tables are configured for CDC, there are series of action steps performed by SQL Server 2008. For each table enabled for CDC, there is a corresponding CDC table with the name <Schema Name>_<Table Name>_CT created within the `cdc` schema. For example, if a supplier table from the `dbo` schema is enabled for CDC then the corresponding cdc.dbo_supplier_CT table is created. The typical structure of a CDC table is shown in Table 6.2.

The change information from SQL Server 2008 needs to be mapped to the common delta format. Listing 6.5 shows the trigger that maps a tuple inserted into the CDC table to a tuple in the delta table. This will also reduce the size of the delta and remove unwanted information.

```
CREATE TRIGGER <Trigger Name>
   ON  cdc.<CDC Table Name>
```

```
   AFTER INSERT
AS
Begin
  INSERT into dbo_<Delta Table Name>
  select [__operation],
         cast([__start_lsn] as int),
         ,all the columns from the original table on which CDC is
            enabled
  from inserted
End
```

Listing 6.5: Script to map CDC table to delta structure.

Column Names	Data Type	Description
__start_lsn	binary(10)	Log sequence number (LSN) associated with the commit transaction for the change.
__end_lsn	binary(10)	Reserved
__seqval	binary(10)	Sequence value used to order the row changes within a transaction.
__operation	int	Identifies the data manipulation language (DML) operation associated with the change. Can be one of the following: 1 = delete 2 = insert 3 = update (old values) Column data has row values before executing the update statement. 4 = update (new values) Column data has row values after executing the update statement.
__update_mask	varbinary (128)	A bit mask based upon the column ordinals of the change table identifying those columns that changed.
Source Table Columns	Varies	The remaining columns in the change table are the columns from the source table that were identified as captured columns when the capture instance was created. If no columns were specified in the captured column list, all columns in the source table are included in this table.

Table 6.2: A typical Change-Data-Capture table in SQL Server 2008.

The remainder of this section provides details on the Oracle Logical Change Records (LCRs), which contain the deltas captured over relational tables, and the process to transform them into the common relational delta format. Configuring and setting up the change data capture mechanism in the Oracle server is more complex and requires certain steps with admin privileges and some steps with regular user privileges. It is beyond the scope of this dissertation to provide all the details on configuring the CDC for Oracle. However, there is good documentation by Oracle as well as by various external websites and blogs that provide step-by-step commands and scripts to run and setup the Oracle database server for capturing the deltas [[22, 64, 65]]. A typical structure of the LCR captured in Oracle server is shown in Table 6.3.

Similar to the CDC structure in SQL Server 2008, the LCR in Oracle 11g server contains extra information, which is not required for view maintenance. Thus, this LCR structure is transformed into the common delta structure previously shown in Table 6.1. This transformation to a common delta structure has the advantage of keeping the deltas coming from different data sources in the same format. In order to do this transformation, there are 2-3 steps to be performed. The first step is to create a corresponding table to store the delta. Once the table is created and the Oracle streams are configured, then a stored procedure can be written to capture the different types of operations from the LCRs and transform them into the common delta structure. The general outline of the stored procedure is shown in Listing 6.6.

```
CREATE OR REPLACE PROCEDURE <stored proc name as dml handler>(in_any
   IN ANYDATA) IS
lcr           SYS.LCR$_ROW_RECORD;
rc            PLS_INTEGER;
command       VARCHAR2(30);
old_values    SYS.LCR$_ROW_LIST;
new_values    SYS.LCR$_ROW_LIST;
BEGIN
    -- Access the LCR
    rc := in_any.GETOBJECT(lcr);
    -- Get the object command type
    command := lcr.GET_COMMAND_TYPE();
    -- Perform actions based on the type of the command
    IF command IN ('INSERT') THEN
```

```
        -- Set the operation type = 2
        -- Obtain the new values and format the new_values records
           accordingly and insert the row into the delta table
    END IF;
    IF command IN ('DELETE') THEN
        -- Set the operation type = 1
        -- Obtain the old values and format the new_values records
           accordingly and insert the row into the delta table
    END IF;
    IF command IN ('UPDATE') THEN
        -- Perform two step process
        -- Step 1
            -- Set the operation type = 3
            -- Obtain the old values and format the new_values
               records accordingly and insert the row into the delta
                table
        -- Step 2
            -- Set the operation type = 4
            -- Obtain the new values and format the new_values
               records accordingly and insert the row into the delta
                table
    END IF;
END;
```

Listing 6.6: Stored procedure to transform LCR into the common delta structure.

Once the stored procedure is compiled without any errors, then this stored procedure has to be associated with the original base data source on which changes will occur. This is called associating the DML handler with the database object. A typical DML handler is shown in Listing 6.7. Since there are three types of changes (INSERT, DELETE and UPDATE) that can occur to a database object, there has to be 3 separate commands to associate a DML handler to the database object. Listing 6.7 shows an example of an INSERT DML handler. Similarly, remaining two DML handlers can be written.

```
BEGIN
  DBMS_APPLY_ADM.SET_DML_HANDLER(
    object_name          => <data source name>,
    object_type          => 'TABLE',   -- this research is monitoring
       only objects of type ``Table'' in the Oracle database
    operation_name       => 'INSERT',  -- Change this to DELETE and
       UPDATE for rest of the operations
    error_handler        => false,
    user_procedure       => <DML handler name>,
    apply_database_link => NULL);
END;
```

Listing 6.7: Associating the DML handler to the base data source.

Column Names	Data Type	Description
source_database_name	VARCHAR2	The database where the DDL statement occurred.
command_type	VARCHAR2	The type of command executed in the DDL statement. Can be one of the following: delete, insert, update. There are other command_types possible. However, they are not relevant for this research.
object_owner	VARCHAR2	The user who owns the object on which the DDL statement was executed.
object_name	VARCHAR2	The database object on which the DDL statement was executed.
tag	RAW	A binary tag that enables tracking of the LCR.
transaction_id	VARCHAR2	The identifier of the transaction.
scn	NUMBER	System Change Number. The SCN at the time when the change record for a captured LCR was written to the redo log.
old_values	LCR$_ROW_LIST	List of the old values for the columns of the object that is changed.
new_values	LCR$_ROW_LIST	Corresponding list of the new values for the columns of the object that is changed.

Table 6.3: Logical Change Record structure in Oracle 11g.

Once the setup is ready, then as the changes occur to the base data source, then they are queued over the Oracle Stream queue in the form of LCRs. These LCRs are grabbed by the DML handler and based on the operation type, the DML handler inserts the appropriate delta in the common delta format into the delta table. Now, the Sybase CEP stream processor can be configured to access this delta table, procure the new deltas and stream them over to the respective DEPA for incremental view maintenance.

Deltas over XML Sources

Deltas for XML data sources are more difficult to capture. One approach is to apply a `diff` process that compares the revised XML document to the original XML

document. The structure of the output from the `diff` algorithm is implementation dependent [30, 66, 67].

Another approach is based on using XQuery's update language [63] to update the XML document. The XQuery update specification defines a Source Update Tree (SUT) that represents a valid change on the source document. Unfortunately, the structure of the SUT is also dependent on the XQuery processor implementation.

For the purpose of this dissertation, the assumption is that the changes are made over the XML documents using the XQuery update language. The SUT is assumed to be in a format based on the work in [68]. This delta format encompasses the typical changes to the XML documents and provides the information needed to support the proposed incremental view maintenance in the DEPA framework. The change operations that are valid for an XML document are insert, delete, and replace. The replace operation is similar to the update operation in relational databases. There are other types of operations, such as rename, and move, which can be applied to an XML tree structure. However, for the purpose of this dissertation, only the 3 basic operations of insert, delete and replace are considered. The typical XML delta structure that is used in this research is shown in Listing 6.8. Similar to the relational delta structure, the XML delta operation can be either 1 for delete, 2 for insert, 3 for replace with old values, and 4 for replace with new values. The TransID is a binary transaction ID generated to distinguish deltas coming from different transactions and also to group the deltas with update operations (3 and 4) together. Since this research is exploring incremental view maintenance over structured XML data sources only, the SUT contains all the relevant nodes needed to traverse to the exact node location in the XML document. Also, it is assumed that the insert operation will insert the subtree at the end of the document.

```
<DELTA Root Node>
  <Delta>
    <DeltaKey>1</DeltaKey>
    <Operation>2</Operation>
    <TransID>x734ab65f</TransID>
    <Source Document root node>
```

```
    <!-- subtree that is to be inserted into the XML document -->
    </Source Document root node>
  </Delta>
</DELTA Root Node>
```

Listing 6.8: XML delta structure.

6.2 Using Magic Sets for Incremental View Maintenance

The magic-sets transformation (MST) is a well-known query optimization technique that transforms the original query to use multiple auxiliary tables, called magic tables, that filter irrelevant data. The concept of magic sets was first introduced for recursive queries in deductive databases [69–71]. However, most of the commercial database systems follow the relational model, and hence, MST has also been explored in terms of optimizing relational queries [72–74].

The MST algorithm transforms a query in two steps. In the first step, the query is adorned with annotations to the predicates that indicate which arguments are bound and which are not [75]. The bound (b) and free (f) adornments distinguish between the bound and free arguments of a goal [76]. There is also a condition (c) adornment for propagating constraints other than equality [77]. Once the query is adorned, then the magic-sets transformation rules are applied in the second step that generate the auxiliary tables, which help in filtering out irrelevant data. As the MST transformations are applied, the original query is rewritten to use the newly created auxiliary tables. Experimental results have shown that applying MST to the relational queries sometimes transforms the nonrecursive query into a recursive one. This has to be avoided since most of the commercial relational database systems do not support recursion [78]. In order to solve this problem, a variation of MST has been proposed to avoid introducing recursion in SQL queries [78, 79].

This extended magic-sets transformation (EMST) is a variation of MST that is used in purely nonrecursive systems [72, 79]. Prior implementations of the magic-sets transformations were for recursive queries that needed two steps to optimize the query: the first step to adorn the query and the second step to apply magic sets transforma-

tion rules. The EMST algorithm for non-recursive queries combines these two steps into one single step. This approach has been implemented in the Starburst relational database system [78, 79]. The EMST algorithm creates the magic tables while the original query is adorned with `b`, `c`, and `f` adornments. This one-step EMST algorithm reduces the complexity of the adornment process and allows arbitrary conditions (equality or non-equality) to be pushed down into the auxiliary tables providing better and stable optimization for relational queries. The queries are represented in a Query Graph Model (QGM) before applying the EMST or MST algorithm [80]. The EMST algorithm traverses the query graph in depth-first order applying the adornment process and transformation rules to each box in the graph. Once the adorning process is complete, auxiliary magic tables are created while applying the transformation rules.

There are 3 types of magic tables depending on the predicates that are pushed down to create the magic tables. A magic-box contains bindings relevant to the query including the equality predicates and the join ordering from the query definition. Typically, the magic-box is created when the adornment does not contain any c adornment. The magic-box can be either a select box, a join box or a union box. A condition-magic-box contains the bindings relevant to the inequality predicates from the query definition. The condition-magic-box is constructed to handle the inequality predicates that are pushed down during the EMST processing of a box that contains a `c` adornment. Finally, the supplementary-magic-box contains the bindings related to the equality constraints to filter out irrelevant tuples early in the execution plan. The supplementary-magic-box is also created during the EMST processing of a box from which the equality predicates can be pushed down close to the data source to facilitate the intermediate computation that contributes relevant tuples to the associated magic box.

The EMST processing of each box from the query graph depends on the type of the operation associated with the box. The operation type decides whether the box allows a quantifier (table reference to a new table) to be added to the box for the new

magic table replacing the existing quantifier for the original table. The box that allows such rewriting of quantifiers possible is called an AMQ (Accepts Magic Quantifier) box while the box that does not allow this rewrite is called an NMQ (No Magic Quantifier) box. A Select-box is always an AMQ box, whereas union, groupby, difference are NMQ boxes.

Magic sets transformations have also been applied to queries over XML data sources [81,82]. The XQuery queries are compiled and represented in the query graph model, similar to the model used to represent relational queries [82]. The query rewriting rules from [80] are extended to support transformations on XML queries. Using these rewriting rules, the predicates and the navigational constructs (XPath expressions) are pushed down close to the XML data source to take the advantage of XML index. This XQuery optimization is implemented in DB2 pureXML which is a hybrid relational and XML database management system [82]. Magic sets query optimization has also been explored for the XPath language by translating the XML document into a logic program using Prolog [81]. Once the XML document is represented using facts and rules into a Prolog-based logic program, then the magic sets transformations are applied to various XPath expressions. Both these approaches along with the magic sets algorithm for non-recursive queries have explored the use of magic sets for either XML data sources only or relational data sources only. The LINQ-based materialized view definitions in the DEPA framework may be defined over both relational and XML data sources. Hence, the research challenge is to explore the use of magic rules on the queries over combined relational and XML data sources.

6.3 Incremental View Maintenance Algorithm

The Incremental View Maintenance (IVM) algorithm presented in this dissertation uses the extended magic sets transformation rules to optimize the queries defined over combined relational and XML data sources. The algorithms described in chapter 5 created a LINQ-based materialized view definition from the detected common subexpressions

over heterogeneous data sources. The IVM algorithm presented in this section applies the EMST rules for non-recursive queries to this view definition to create the magic (auxiliary) tables. The data structures that are maintained to define and create LINQ-based materialized views are reused in the IVM algorithm. The magic tables created during the magic sets transformations will retain the data in their native format depending on the type of the data sources from which the magic tables are created. Thus, if a magic table is derived from only relational data sources, then the magic table will be in relational format. If the base data sources are in XML format, then the magic table will be in XML. For hybrid combinations of relational and XML data sources, the magic tables will use the same data structure as the materialized view defined in chapter 5. For the purpose of this research, Select-Project-Join (SPJ) queries in SQL are considered without self-joins. The XQuery queries are restricted to For-Where-Result (FWR) clauses and XPath expressions to forward axes (/, //, *) and simple boolean predicates. Finally, LINQ queries are restricted to using From-Where-Select clauses. Thus, the view definitions derived from these types of queries also hold similar restrictions. These view definitions are represented in QGM and then the EMST rules are applied. Since, the view definitions are simple, the boxes in QGM for this research are AMQ boxes. The IVM algorithm based on the magic sets transformations is first illustrated by an example before the presentation of the details of the algorithm.

Working of the IVM Algorithm

This section illustrates the incremental view maintenance algorithm using the running Criminal Justice example in chapters 4 and 5. The example illustrated in this section takes the same view definition and applies the IVM algorithm using magic sets. The view definition is shown in Listing 6.9. The view V1 is derived from two data sources: $LicenseInfo$ is the relational table and $VRDb$ (`VehicleRecords.xml`) is the XML document. The query is represented in QGM as shown in Figure 6.2. $LicenseInfo$ has the adornment of f for all of its attributes. The node for the XML document has one

single f adornment.

```
View V1:
var V1 =
from T0 in VRDb.Elements("Vehicle")
from T1 in LicenseInfo
where T0.Element("VehicleLocation").Element("State") == "AZ" &&
      T1.Status == "expired" &&
      T1.Points > 10 &&
      T1.DriverLicense.Equals(T0.Element("DriverInfo").Element("
         DriverLicense").Value)
select new { LicenseInfo_DriverLicense = T1.DriverLicense,
             LicenseInfo_Class = T1.Class,
             LicenseInfo_Points = T1.Points,
             ... ,
             T0 = T0 };
```

Listing 6.9: Step 1 queries.

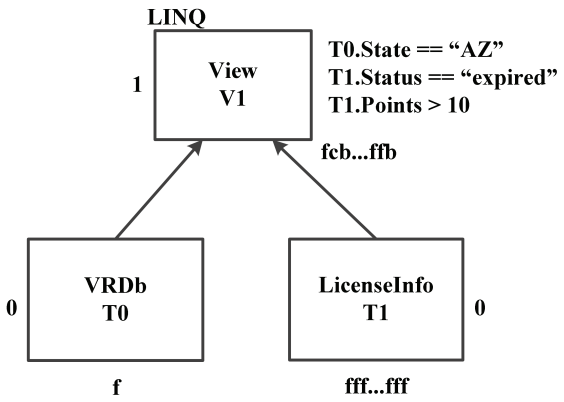


Figure 6.2: Initial query graph : step 1.

The IVM algorithm using magic sets traverses the QGM in depth-first order. As the algorithm proceeds, it first considers the View V1 box which is an AMQ box. Thus, all the predicates that it has can be pushed down. The adornment for this box is `fffbb...ffb` since there are three variables, which are bound and rest of them are free. The equality predicate `T1.Status == ''expired''` is pushed down to create a supplementary magic-box `sm_LicenseInfo`. Once the box is created, the quantifier `T1` in the original view box is replaced by a new quantifier `M1` created for the magic-box `sm_LicenseInfo`. The adornment for the magic-box `sm_LicenseInfo` is `fffbff...f` since Status is the only variable that is bound

and rest of the attributes from `LicenseInfo` are free. This is shown in Figure 6.3. The queries corresponding to the new boxes are shown in Listing 6.10.

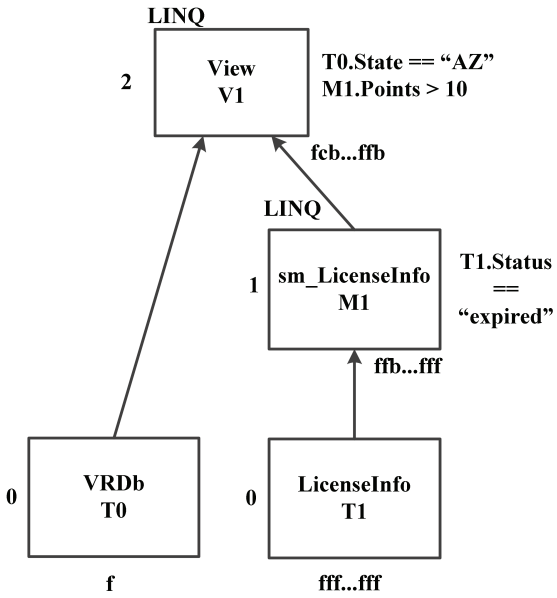


Figure 6.3: Modified query graph: step 2.

```
View V1:
var V1 =
from T0 in VRDb.Elements("Vehicle")
from M1 in sm_LicenseInfo
where T0.Element("VehicleLocation").Element("State") == "AZ" &&
      M1.Points > 10 &&
      M1.DriverLicense.Equals(T0.Element("DriverInfo").Element("
          DriverLicense").Value)
select new { LicenseInfo_DriverLicense = M1.DriverLicense,
             LicenseInfo_Class = M1.Class,
             LicenseInfo_Points = M1.Points,
             ... ,
             T0 = T0 };

var sm_LicenseInfo =
    from T1 in LicenseInfo
    where T1.Status == "expired"
    select T1;
```

Listing 6.10: Step 2 queries.

There is another inequality predicate `M1.Points > 10` for the View V1 box, which can be pushed down. This inequality predicate creates a condition-magic-box `cm_LicenseInfo` over the already existing supplementary magic-box `sm_Lice-`

nseInfo. Once this box is created, the quantifier M1 in the original view box is replaced by a new quantifier M2 created for the magic-box cm_LicenseInfo. Since the magic-box cm_LicenseInfo uses the quantifier over the magic-box sm_LicenseInfo, the adornment for cm_LicenseInfo is fffbbff...f since there are two variables (Status and Points), which are bound and the rest of the attributes are free. The modified QGM and its corresponding revised queries are shown in Figure 6.4 and Listing 6.11, respectively.

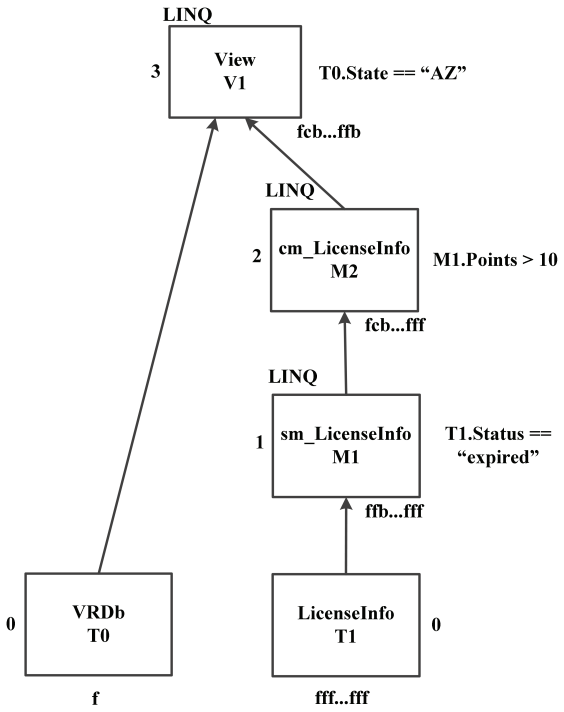


Figure 6.4: Modified query graph: step 3.

```
View V1:
var V1 =
from T0 in VRDb.Elements("Vehicle")
from M1 in cm_LicenseInfo
where T0.Element("VehicleLocation").Element("State") == "AZ" &&
      M1.DriverLicense.Equals(T0.Element("DriverInfo").Element("
          DriverLicense").Value)
select new { LicenseInfo_DriverLicense = M1.DriverLicense,
             LicenseInfo_Class = M1.Class,
             LicenseInfo_Points = M1.Points,
             ... ,
             T0 = T0 };
```

```
var cm_LicenseInfo =
    from M2 in sm_LicenseInfo
    where M2.Points > 10
    select M2;

var sm_LicenseInfo =
    from T1 in LicenseInfo
    where T1.Status == "expired"
    select T1;
```

Listing 6.11: Step 3 queries.

Finally, the equality predicate `T0.Element(''VehicleLocation'').Element(''State'') == ''AZ''` can be pushed down to create a supplementary magic-box `sm_VRDb`. The adornment for this box `sm_VRDb` is `b` since the XML file is bound by a certain element `State`. The revised QGM and its queries are shown in Figure 6.5 and Listing 6.12, respectively.

```
View V1:
var V1 =
from M3 in sm_VRDb
from M2 in cm_LicenseInfo
where M2.DriverLicense.Equals(M3.Element("DriverInfo").Element("
   DriverLicense").Value)
select new { LicenseInfo_DriverLicense = M2.DriverLicense,
             LicenseInfo_Class = M2.Class,
             LicenseInfo_Points = M2.Points,
             ... ,
             M3 = M3 };

var sm_VRDb =
    from T0 in VRDb.Elements("Vehicle")
    where T0.Element("VehicleLocation").Element("State") == "AZ"
    select T0;

var cm_LicenseInfo =
    from M1 in sm_LicenseInfo
    where M1.Points > 10
    select M1;

var sm_LicenseInfo =
    from T1 in LicenseInfo
    where T1.Status == "expired"
    select T1;
```

Listing 6.12: Step 4 queries.

After analyzing the View V1 box, the algorithm traverses down to the `cm_LicenseInfo` box. The condition magic-box is derived from the supplementary

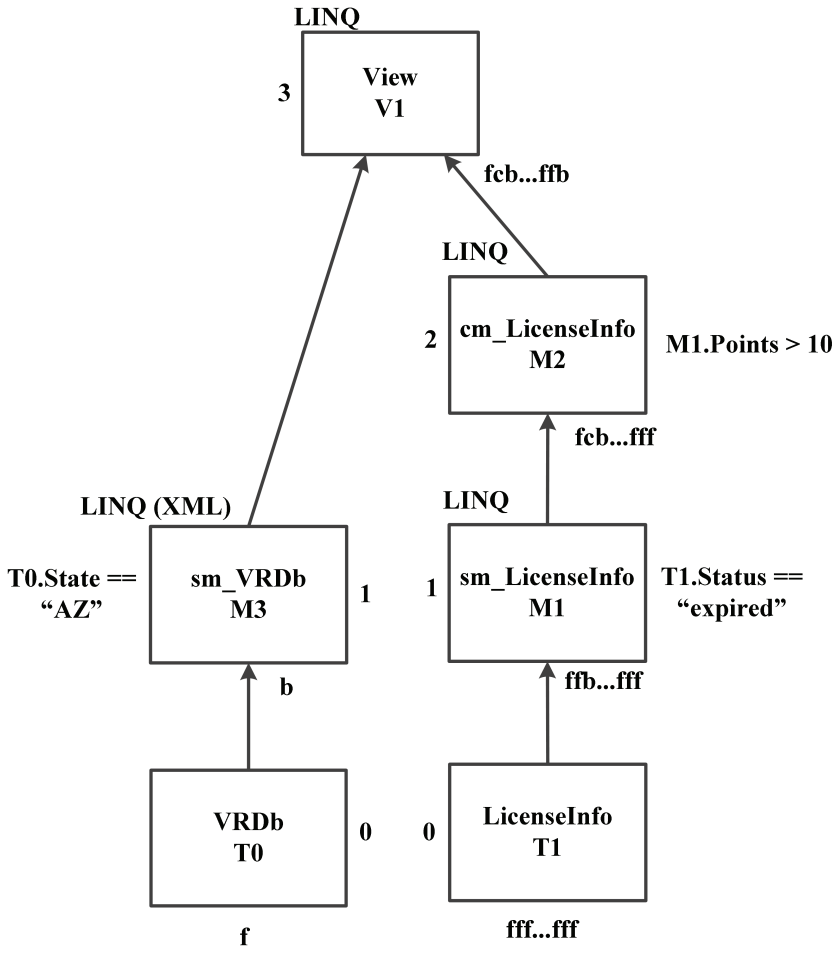


Figure 6.5: Modified query graph: step 4.

magic-box `sm_LicenseInfo` and the supplementary magic-box is not used anywhere else in the QGM. Hence these two boxes can be combined together to avoid the creation of one extra magic-box. Thus, a new combined magic-box `scm_LicenseInfo` is created, replacing the two magic-boxes `cm_LicenseInfo` and `sm_LicenseInfo`. A new quantifier `M4` replaces the quantifier `M2`. The new revised and optimized QGM and the corresponding queries are shown in Figure 6.6 and Listing 6.13, respectively.

```
View V1:
var V1 =
from M3 in sm_VRDb
from M4 in scm_LicenseInfo
where M4.DriverLicense.Equals(M3.Element("DriverInfo").Element("
    DriverLicense").Value)
```

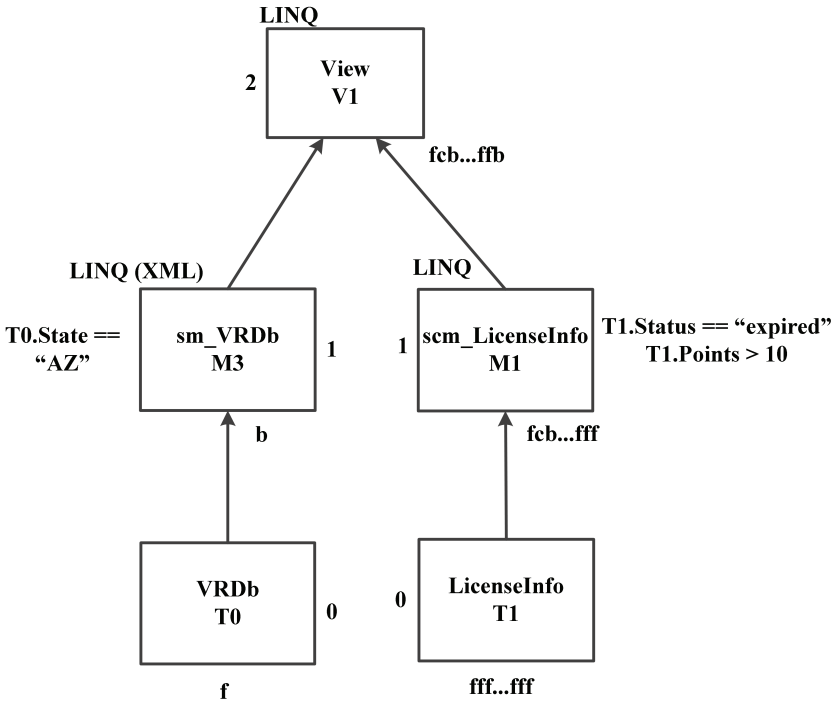


Figure 6.6: Modified query graph: step 5.

```
select new { LicenseInfo_DriverLicense = M4.DriverLicense,
             LicenseInfo_Class = M4.Class,
             LicenseInfo_Points = M4.Points,
             ... ,
             M3 = M3 };

var sm_VRDb =
    from T0 in VRDb.Elements("Vehicle")
    where T0.Element("VehicleLocation").Element("State") == "AZ"
    select T0;

var scm_LicenseInfo =
    from T1 in LicenseInfo
    where T1.Status == "expired" &&
          T1.Points > 10
    select T1;
```

Listing 6.13: Step 5 queries.

The algorithm continues to traverse the boxes one after the other but does not find anything to push down or the necessity to create additional magic-boxes. The algorithm stops when all the boxes are analyzed and no new modifications are made to the QGM.

The final set of queries shown in Listing 6.13 indicates that there are two magic tables created. One for the relational table `LicenseInfo` and the other created for the XML document *VRDb* (`VehicleRecords.xml`). The data format of the magic tables is based on the type of the base data source. Thus, the magic table `scm_LicenseInfo` is relational in structure since the base data source `LicenseInfo` is relational. Similarly, the magic table `sm_VRDb` is an XML document. These magic tables are populated only once after the `IVMMagicsets` algorithm is applied. Whenever future deltas are captured and streamed to the DEPA, the changes are first used to update the magic tables and then are propagated to the view.

Assume that the deltas for `LicenseInfo` are streamed over to the DEPA and are captured as a collection of objects in the list `delta_LicenseInfo`. This collection of deltas can be used in place of the `LicenseInfo` table in the query `scm_LicenseInfo`. This will return any qualifying tuples for the magic table that need to be propagated up to the view. The set of queries that will access the `delta_LicenseInfo` collection to propagate the changes in the view due to changes in the `LicenseInfo` table are shown in Listing 6.14.

```
var deltascm_LicenseInfo =
    from deltaT1 in delta_LicenseInfo
    where deltaT1.Status == "expired" &&
          deltaT1.Points > 10
    select deltaT1;

var sm_VRDb =
    from T0 in VRDb.Elements("Vehicle")
    where T0.Element("VehicleLocation").Element("State") == "AZ"
    select T0;

var V1 =
    from M3 in sm_VRDb
    from deltaM4 in deltascm_LicenseInfo
    where deltaM4.DriverLicense.Equals(M3.Element("DriverInfo").
       Element("DriverLicense").Value)
    select new { LicenseInfo_DriverLicense = deltaM4.DriverLicense,
                 LicenseInfo_Class = deltaM4.Class,
                 LicenseInfo_Points = deltaM4.Points,
                 ... ,
```

```
                    M3 = M3 };
```

Listing 6.14: Set of queries propagating deltas over LicenseInfo table.

In this case, only the delta object for the `LicenseInfo` table is used. The magic table for `VRDb` is the same. Thus, only qualifying delta tuples are propagated to the view definition. Similarly assume that the deltas streamed to a DEPA for the XML document `VRDb` are collected in the list `delta_VRDb` and hence the queries in Listing 6.15 can be used to propagate the relevant changes to the view.

```
var deltasm_VRDb =
    from deltaT0 in delta_VRDb.Elements("Vehicle")
    where deltaT0.Element("VehicleLocation").Element("State") == "AZ"
    select deltaT0;

var scm_LicenseInfo =
    from T1 in LicenseInfo
    where T1.Status == "expired" &&
          T1.Points > 10
    select T1;

var V1 =
    from deltaM3 in deltasm_VRDb
    from M4 in scm_LicenseInfo
    where M4.DriverLicense.Equals(M3.Element("DriverInfo").Element("
       DriverLicense").Value)
    select new { LicenseInfo_DriverLicense = M4.DriverLicense,
                 LicenseInfo_Class = M4.Class,
                 LicenseInfo_Points = M4.Points,
                 ... ,
                 M3 = deltaM3 };
```

Listing 6.15: Set of queries propagating deltas over VRDb.

The same list `delta_LicenseInfo` of deltas is used to update the magic table `scm_LicenseInfo` or the list `delta_VRDb` is used to update the magic table `sm_VRDb` incrementally. Since the deltas can be either inserts, deletes or updates, the magic table can be updated using the following rules. The rules presented below are for updating the magic table `scm_LicenseInfo`, however, similar rules can be written for updating the XML-based magic table `sm_VRDb`.

Let $\Delta LicenseInfo$ be the set of deltas captured over the base data source `LicenseInfo` and streamed to the DEPA. Thus,

$\Delta LicenseInfo = \Delta LicenseInfo^{+} \cup \Delta LicenseInfo^{-} \cup \Delta LicenseInfo^{changes}$ where

$\Delta LicenseInfo^{+}$ are the tuples inserted into `LicenseInfo`,

$\Delta LicenseInfo^{-}$ are the tuples deleted from `LicenseInfo` and,

$\Delta LicenseInfo^{changes}$ are the tuples where certain attributes of those tuples in `License-Info` were changed.

Thus, `scm_LicenseInfo` will be updated as follows:

$scm\_LicenseInfo = scm\_LicenseInfo \cup deltascm\_LicenseInfo$

where $deltascm\_LicenseInfo$ is the set of qualifying tuples.

Thus, the changes captured over the base data sources can be used first to incrementally update the corresponding magic table if it is present for that particular view definition. Once the qualifying deltas for the magic table are determined, those relevant deltas are propagated to the view or the next magic table based on the set of queries and the stratum number. This algorithm propagates only the relevant tuples thus avoiding unnecessary tuples propagating to the view. Hence, the time taken to update the view is very less as compared to naive approach of re-derivation of the view from the updated base data sources.

Pseudo code of the IVM Algorithm

Let V be the materialized view that is defined using the detected common subexpressions *csubs* over the heterogeneous data sources. The view definition is also represented in a graph QG based on QGM [80]. The algorithm traverses the graph in depth-first traversal order starting from the top query box, which, is the view definition.

Algorithm: $IVMMagicsets$

Input: The query graph QG representation of the view definition $V.ViewDefinition$. The view structure V and its supporting data structures are shown in Figure 5.2. The supporting data structures will provide access to the metadata repository for the base data sources.

Output: The modified query graph QG' and the set of magic tables.

Pseudo code:

Start traversing the *QG* graph in depth-first traversal order. Repeat /* Start processing the node (box) */

Let B be the box under consideration.

/* EMST rules can be applied only to the boxes which are not the base data sources */

If $B \notin V.BaseDataSources$

For each q $\in$ list of quantifiers in box B

/* All the predicates related to quantifier q can be pushed down only for AMQ boxes*/

If B is an AMQ box

/* These predicates can be pushed down */

Mark each *csub* related to q $\in$ *CSet*

/* This is applied only to those QGMs which have join-orders. Recollect that the join-ordering in LINQ is determined by the ordering of the `from` clauses*/

Determine the quantifiers that are eligible to pass information into q.

Use predicate push-down rules to push down all the above determined predicates i.e. each marked *csub* $\in$ *CSet*.

Based on the group of predicates for the quantifier q, select an appropriate bcf adornment α.

End If

Make q range over a box B_q with adornment α. If the box B_q does not exist already, then create a new box B_q.

If $\alpha \neq ff...f$

If B is an AMQ box

If q has equality predicate

Create a supplementary magic-box by attaching "s_" to the name of the base data source referenced by the quantifier q. Push all the valid predicates down to the supplementary magic-box. Remove the quantifier q from box B and add a new quantifier for the newly created supplementary magic-box.

Adjust the stratum numbers accordingly.

End If

If q has at least one condition predicate

Create a condition magic-box by attaching "c_" to the name of the base data source referenced by the quantifier q. Push all the valid equality as well as the condition predicates down to the condition magic-box. Remove the quantifier q from box B and add a new quantifier for the newly created condition magic-box.

Adjust the stratum numbers accordingly.

End If

End If

End For

End If

Until the graph *QG* is traversed completely.

Once the magic tables are created using the algorithm `IVMMagicsets`, it is possible that the QGM has some magic-boxes that can be optimized by combining them together. This can be done especially with the supplementary magic-boxes and condition magic-boxes on the same data source. This step traverses again through the QGM starting from the root node, which is the view definition node and detects

those magic-boxes, which are derived over the same data source and the magic-boxes can be combined into single magic-box without affecting other boxes. For example, there is a join magic-box `m_Person_LicenseInfo` that defines a join query over `Person` table and `LicenseInfo` table. Consider that while applying the algorithm `IVMMagicsets`, a supplementary magic-box `s_Person` was created over the `Person` table that filters people below the age of 25. Thus, the magic-box `m_Person_LicenseInfo` is modified to do a join over `s_Person` and `LicenseInfo` table. Since no other QGM box is using `s_Person` box, one magic table creation can be avoided by combining magic-box `m_Person_LicenseInfo` and `s_Person` box into one single box `m_s_Person_LicenseInfo` that will contain the join condition as well as the predicate condition over the `Person` table. Thus, the magic sets derivations can be optimized to create and retain less magic tables.

One last step in using magic sets for incremental view maintenance is to create rule-sets for each base data source such that each rule-set will consume the deltas coming over streams for that particular data source and propagate the changes up the QGM till the changes reach the view. Once the deltas are propagated to the view level, then they can be merged into the view to update the view incrementally. The stratum numbers associated with each level in the QGM can be used to determine the order of the rule execution. The queries for all the boxes that have the same stratum number (same depth in the QGM) can be executed in any order whereas the queries for the boxes at different levels are ordered based on the ascending order of the stratum numbers.

6.4 Summary

Defining and maintaining materialized views using common subexpressions over heterogeneous data sources is expected to improve the performance of the system. Rewriting the original queries to access the locally defined materialized views avoids the long trips to the distributed base data sources in the DEPA framework. Since the base data sources may change, these changes need to be captured to incrementally update the ma-

terialized views. This chapter first elaborated on how these changes are captured in both relational and structured XML data sources. The changes from two popular commercial database systems were captured and mapped to a common relational delta format for streaming within the DEPA framework. The changes for XML documents were captured as source update trees and streamed as XML deltas. Both relational and XML deltas were used in the developed view maintenance algorithm that uses the magic sets query optimization approach to incrementally update the materialized views.

Chapter 7

PROTOTYPE SETUP AND EVALUATION

This dissertation has focused on research challenges involved in defining and maintaining materialized views over heterogeneous data sources in a distributed event stream processing framework. Chapter 1 described an an overview of the DEPA framework and a simple proof-of-concept implementation. Chapters 3 through 6 presented a series of algorithms and data structures as part of the solution to these research challenges. This chapter describes the overall architectural design of the DEPA framework and its implementation details. The distributed event stream processing prototype is implemented using the C# language with .NET Framework 3.5, Coral8 Server 5.6.2, SQL Server 2008, Oracle 11g, and XML files stored on a hard drive. The prototype proof-of-concept DEPA framework is evaluated using sample scenarios based on the Criminal Justice and the modified TPC-H enterprises.

7.1 DEPA Architecture

This dissertation research was partially supported by a grant from the National Science Foundation (CSR 0915325). Through this support, the research group was able to purchase three Servers and the necessary software to develop the prototype implementation of the DEPA framework to illustrate the definition and maintenance of materialized views over heterogeneous data sources. The high-level architectural diagram of the DEPA framework is shown in Figure 7.1.

The agents in the DEPA framework communicate with each other using event and data streams. Each DEPA performs its own specific tasks and hence it maintains a local repository of all the heterogeneous data sources, such as relational databases, structured XML documents, event and data streams, and different query expressions that access these distributed data sources. The agent also receives deltas from the monitored data sources over streams so that these changes can be used in the incremental maintenance of the materialized views. To develop the prototype environment for the

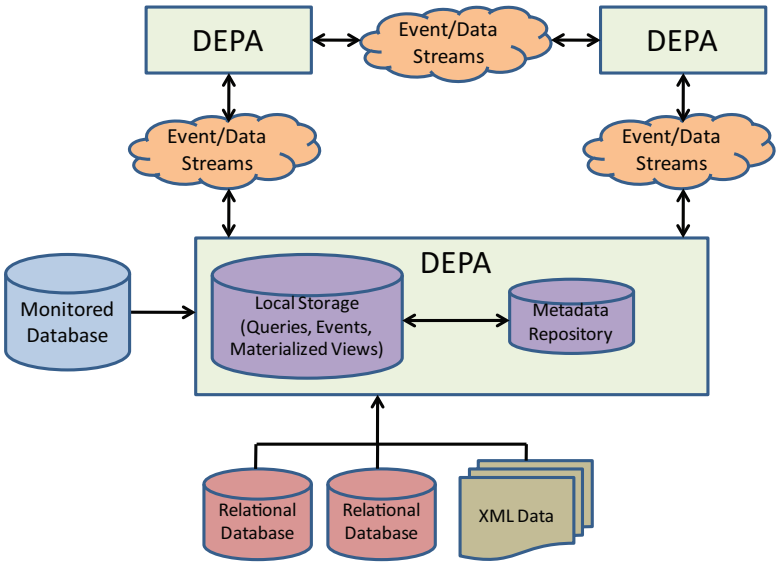


Figure 7.1: High-level overview of DEPA framework.

DEPA framework, the three servers are configured as part of Arizona State University's WAN network so that the servers can communicate with each other. The hardware configuration of the prototype system is shown in Figure 7.2. The three servers are Dell PowerEdge R300 with Xeon 3.0 GHz Processor with 4 GB ram and 160 GB hard drive space. One of the servers provides access to the Oracle 11g Enterprise Database Server installed on the Linux Enterprise Operating system. This version of the Oracle server has the Oracle Streams enabled to capture the changes over the relational tables. Another server has a standalone SQL Server 2008 Enterprise edition running on top of the Windows Server 2003 R2 64-bit operating system. This edition of SQL Server has the capabilities of configuring and using the Change-Data-Capture feature to gather the changes over the relational tables. The third server is configured as the actual DEPA that communicates with the other two servers. The third server uses Sybase CEP (formerly known as Coral8) as the event stream processor. XML Stylus studio is used to create and manipulate the XML documents. The Oracle LINQ provider by Devart is a third-party LINQ provider used to access the Oracle database using LINQ to Entities. The DEPA framework prototype is developed in the C# programming language using Visual Studio 2008 SP1.

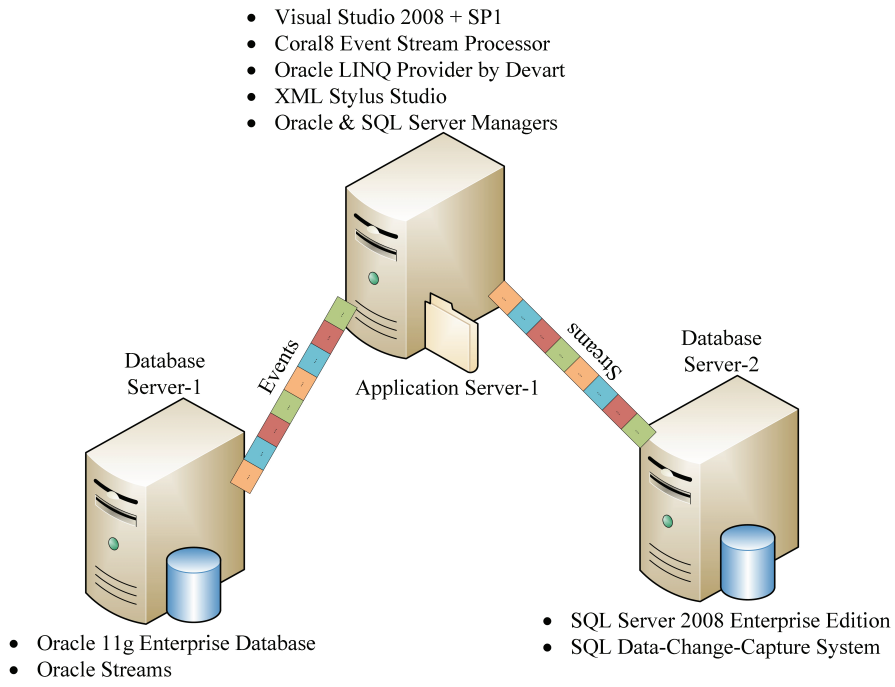


Figure 7.2: DEPA architecture hardware and software specifications.

7.2 Evaluation using Criminal Justice Enterprise

The Criminal Justice enterprise was developed as part of this research to create sample scenarios for evaluation purposes. The enterprise is designed based on the Global Justice XML Data Model (GJXDM) [83]. GJXDM is designed for the exchange of information across law enforcement agencies at the municipal, county, state, and federal levels . However, GJXDM is complex and a vast data model and hence, a smaller Criminal Justice enterprise was created to evaluate the DEPA framework. The Criminal Justice enterprise consists of relational as well as XML data sources. The UML diagram is shown in Figure 7.3. The enterprise stores the personal information regarding people, such as their name, age, residential address, etc. Missing people records and records related to parolees are also maintained as the part of this enterprise.The enterprise also stores the driving license information for those people who have one. The vehicle information is stored as XML document that contains vehicle description, registration information, license plate number, and the people who are eligible to drive

that vehicle. To illustrate a sample set of charges related to the GJXDM, this enterprise stores information regarding missing vehicles, missing persons, and parolees. The enterprise also handles the different types of traffic violations and parole violations. The classes in the UML diagram are annotated to indicate the XML data sources whereas the rest of the classes are relational in structure. This model has been used to develop and test the algorithms and the data structures throughout the chapters 1 through 6.

The system is first assessed to evaluate the performance of processing streams over heterogeneous data sources using a LINQ query. The experiment is conducted to find out the query execution time of processing streaming data over one relational and one XML data source using a LINQ query with and without the use of a materialized view. Consider a sample scenario involving speeding tickets cited by the photo radar system on freeways. Assume that the photo radar speeding tickets are streamed over an XML stream (`SpeedingTickets`). The Motor Vehicle Division (MVD) stores vehicle records in an XML file (`VehicleInfo`) and the driver license information in a relational database (`LicenseInfo`). The personal information (`PersonalInfo`) is stored in an associated state-level relational database. In order to process each speeding ticket, data is required from both the XML file as well as the relational tables so that the speeding tickets can be mailed directly to the vehicle owner's home address. The LINQ query uses the `speedingPlateNum` from the `SpeedingTickets` stream to process each violation as shown in Listing 7.1.

```
var VehicleInfo = XElement.Load(@"VehicleRecords.xml");

var VehicleDetailedInfo =
    from Vehicles in VehicleInfo.Elements("Vehicle")
    from Licenses in DBCon.LicenseInfo
    from Persons in DBCon.PersonalInfo
    where Licenses.LicenseNumber.Equals(Vehicles.Element("
       DriverInformation").Element("DriverLicense").Value.ToString()
       )
          && speedingPlateNum.Value.ToString().Equals(Vehicles.
             Element("VehicleInformation").Element("PlateNumber").
             Value.ToString())
          && Persons.SSN.Equals(Licenses.SSN)
```

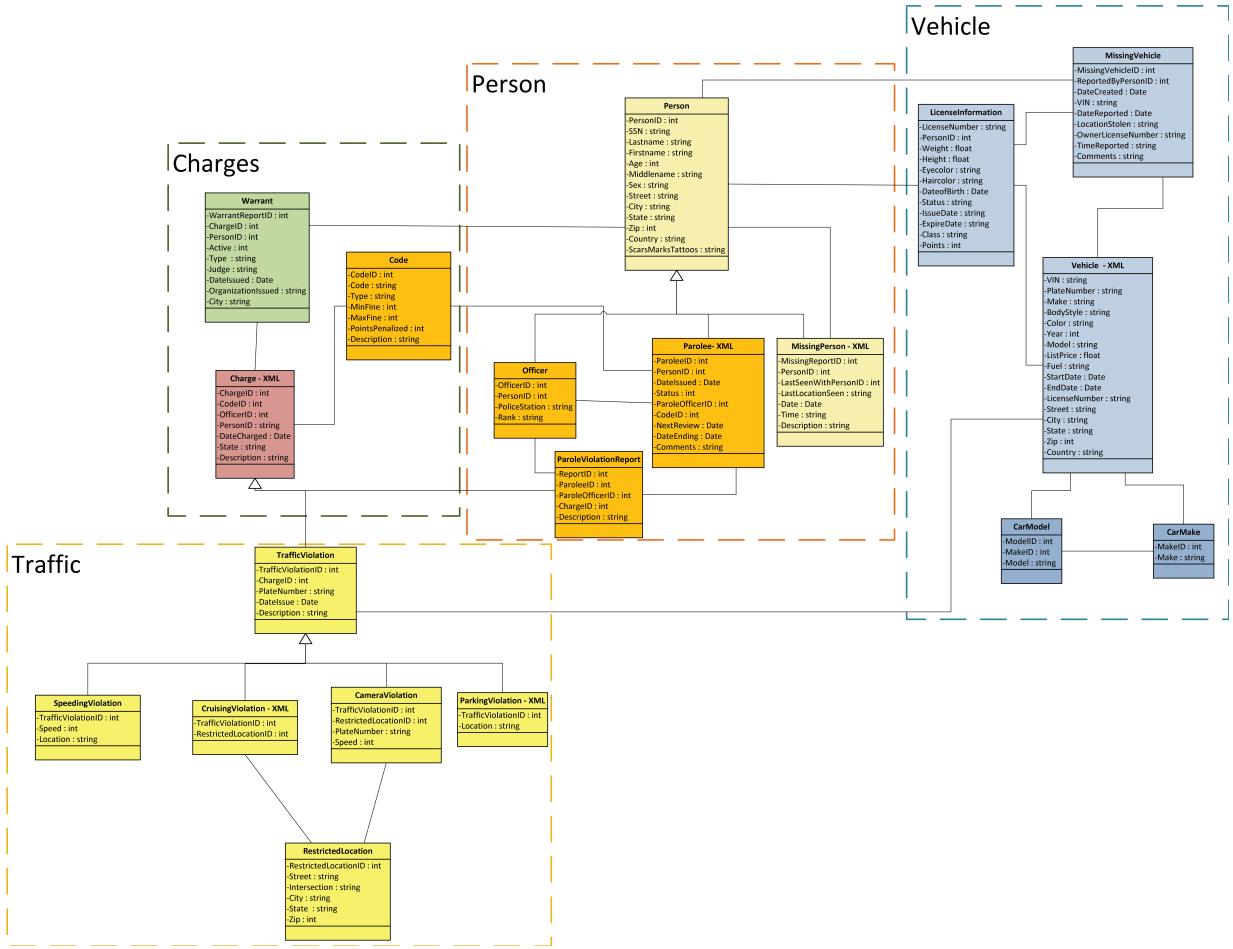


Figure 7.3: Criminal Justice enterprise UML diagram.

```
    select new {/* project needed fields */};
```
Listing 7.1: LINQ query processing streaming data over relational and XML data sources.

Assume that a materialized view is defined in the system such that it contains related tuples for a vehicle with the license and the personal information of its registered owner. Such a view can be used to answer the query defined in Listing 7.1. This materialized view combines both relational and XML data sources. Once the materialized view is registered with the system, the original LINQ query can be rewritten to query the materialized view instead of the original data sources. The modified LINQ query that accesses the materialized view `ViewPersonVehicle` is shown in Listing 7.2.

```
var VehicleInformation = XElement.Load(@"VehicleRecords.xml");

var VehicleDetailInfo =
    from mv in DBCon.ViewPersonVehicle
    where speedingPlateNum.Equals(mv.VehicleInfoPlateNum)
    select new {/* project needed fields */};
```
Listing 7.2: LINQ query processing streaming data over a hybrid relational and XML view.

As expected, query execution in the DEPA environment using the materialized view shows significant performance improvement. Figure 7.4 shows the performance difference between the execution time of both the LINQ queries shown above as the XML speeding citations are received as streaming data. The graph clearly indicates the expected performance improvement using the materialized view. The `PersonalInfo` table contains 14500 tuples and the `LicenseInfo` table contains 500 tuples. The `VehicleInfo` XML file contains 500 records. The XML speeding citations were received at a rate of 1 citation per second. The average time to execute 120 XML speeding citations using the LINQ query over the original data sources is 324 milliseconds. The average time to execute the same 120 citations using the LINQ query with the materialized view is 50 milliseconds. For this example, the use of the materialized view for stream processing over heterogeneous data

sources increases the query efficiency by an approximate factor of 6. The materialized view `ViewPersonVehicle` contains 500 tuples as a result of the join between the `PersonalInfo` table, the `LicenseInfo` table, and the `VehicleInfo` XML file.

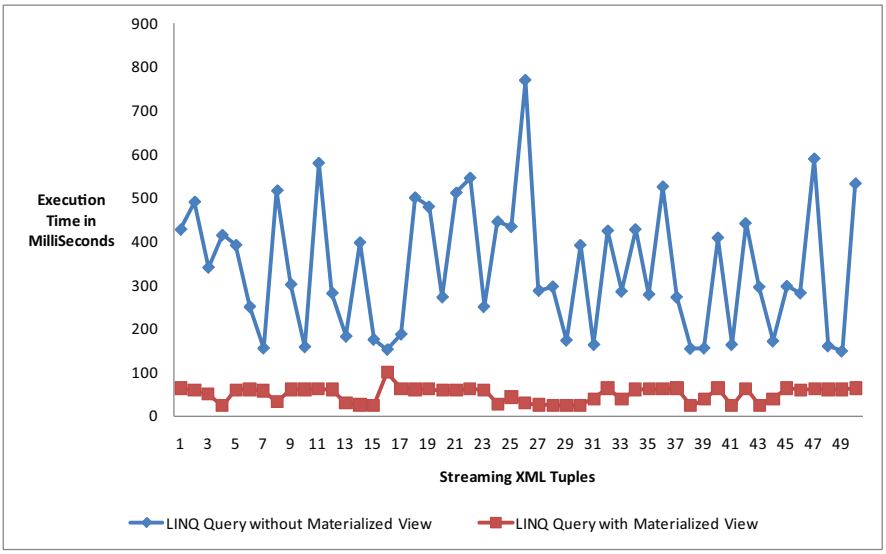


Figure 7.4: Criminal Justice: Comparing query execution time with and without using materialized view.

The second performance evaluation is done to compare the time required for the materialization of a hybrid view. The time required to materialize the view directly from the base data sources without using magic tables is compared with the time required to materialize the view using temporary magic tables. The third entry in the evaluation is to persist the magic tables for the first time in a Dataset and then the subsequent view materialization is done using the magic tables from the Dataset. This evaluation is done for materializing the view definition as shown in Listing 7.3.

```
var VRDb = XElement.Load("VehicleRecords.xml");

View V2:
var V2 =
    from T0 in VRDb.Elements("Vehicle")
    from T1 in LicenseInfo
    where T0.Element("VehicleLocation").Element("State") == "AZ" &&
          T1.Status == "expired" &&
          T1.Points > 10 &&
          T1.DriverLicense.Equals(T0.Element("DriverInfo").Element("
              DriverLicense"). Value)
```

```
    select new { LicenseInfo_DriverLicense = T1.DriverLicense,
                 LicenseInfo_Class = T1.Class,
                 LicenseInfo_Points = T1.Points,
                 ... ,
                 T0 = T0 };
```

Listing 7.3: LINQ definition for a hybrid view over a relational and XML data source.

To measure the time required to materialize the view using magic tables, which are not stored in a Dataset, the set of queries shown in Listing 7.4 are used. These set of queries are obtained by applying the magic sets algorithm from chapter 6.

```
View V2:
var V2 =
    from M3 in sm_VRDb
    from M4 in scm_LicenseInfo
    where M4.DriverLicense.Equals(M3.Element("DriverInfo").Element("
       DriverLicense"). Value)
    select new { LicenseInfo_DriverLicense = M4.DriverLicense,
                 LicenseInfo_Class = M4.Class,
                 LicenseInfo_Points = M4.Points,
                 ... ,
                 M3 = M3 };

var sm_VRDb =
    from T0 in VRDb.Elements("Vehicle")
    where T0.Element("VehicleLocation").Element("State") == "AZ"
    select T0;

var scm_LicenseInfo =
    from T1 in LicenseInfo
    where T1.Status == "expired" &&
          T1.Points > 10
    select T1;
```

Listing 7.4: Magic sets LINQ queries to materialize the view.

Finally, in the third option, while materializing the view for the first time, the two magic tables `sm_VRDb` and `scm_LicenseInfo` are stored in a Dataset. Subsequent re-materialization of the view is done by accessing the magic tables from the Dataset. The queries for the magic tables `sm_VRDb` and `scm_LicenseInfo` remain the same as shown in Listing 7.4, however, the final view query now accesses the magic tables stored in the Dataset. Thus, the modified query for the view materialization using magic tables from the Dataset is shown in Listing 7.5.

```
View V2:
var V2 =
    from M3 in sm_VRDbDataset
    from M4 in Dataset.scm_LicenseInfo
    where M4.DriverLicense.Equals(M3.Element("DriverInfo").Element("
        DriverLicense"). Value)
    select new { LicenseInfo_DriverLicense = M4.DriverLicense,
                 LicenseInfo_Class = M4.Class,
                 LicenseInfo_Points = M4.Points,
                 ... ,
                 M3 = M3 };
```

Listing 7.5: LINQ query to materialize the view using magic tables stored in a Dataset.

The graph comparing different execution times captured over multiple runs is shown in Figure 7.5. The `LicenseInfo` table contains 7000 license information and the `VRDb` XML document contains 7000 XML records of vehicle information. The total number of qualifying tuples for the view `V2` is 722 records. The average time taken to materialize the view without the use of magic sets is 02 minutes and 34 seconds. The average time taken to materialize the view using magic sets (without storing them in the Dataset) is 26 seconds. The average time taken to materialize the view using magic sets stored in the Dataset is 5 seconds. This analysis indicates that the use of magic sets (without Dataset) to materialize the view is approximately 6 times faster than without the use of magic sets. The use of magic sets with Dataset to materialize the view is approximately 5 times faster than using magic sets without the Dataset feature. Thus for this example, using magic sets with storing them in a Dataset is approximately 30 times faster than view materialization without the use of magic sets. This is a significant increase in efficiency in materializing the view using magic tables.

The final performance evaluation conducted over the criminal justice data model scenario is to determine the number of tuples propagated to incrementally update the view. The incremental view maintenance algorithm using magic sets is compared against the semi-naive incremental view maintenance algorithm and the re-derivation algorithm to persist the view from the updated base data sources. To perform the test runs, deltas are generated at random over the two base data sources. Once these deltas

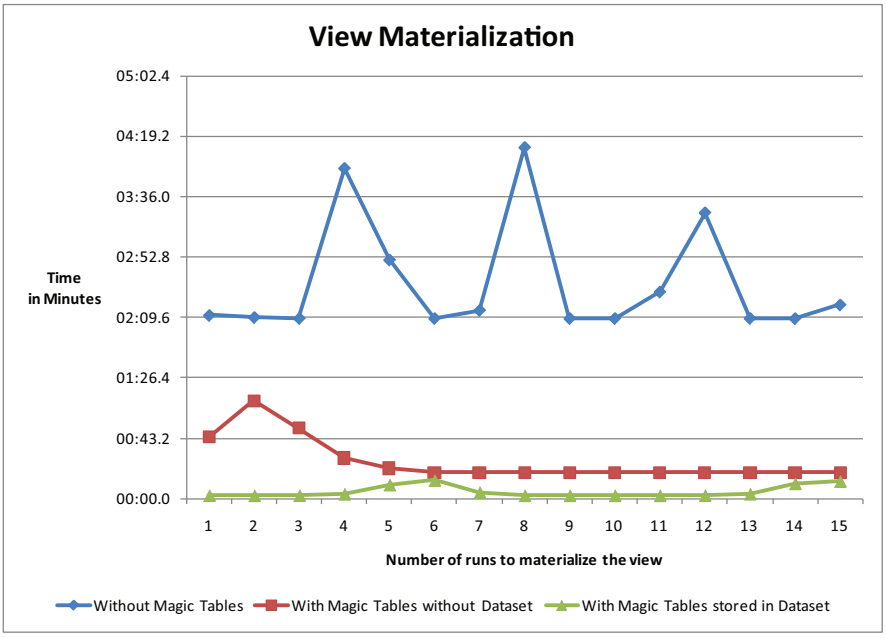


Figure 7.5: Criminal Justice: Comparing view materialization time without using magic tables, with magic tables (no Dataset) and with magic tables stored in Dataset.

are captured and transformed into the common delta structure, the event stream processor streams these deltas to the respective agent. As the agent receives the deltas, the agent uses the different algorithms to update the view. The changes are generated totally at random so as not to fabricate the results. Table 7.1 shows the different runs for changes captured over the `LicenseInfo` table during the incremental maintenance of the view using the different algorithms. One of the columns shows the total number of changes captured during each run. There is one column for the total number of changes propagated using magics tables. Another column indicates the total number of relevant changes propagated using the semi-naive approach, which is always the total number of changes captured. Finally, the last column shows the number of tuples used in the re-derivation of the view from the updated `LicenseInfo` table. The database is returned to its default state between every evaluation of the different algorithms. Thus, for every test case, the total number of starting tuples in the `LicenseInfo` table is 7000. The analysis indicates a significant reduction in the tuples that are propagated using magic tables as compared to the semi-naive algorithm. The graphical compar-

ison of the total number of relevant changes propagated using magic tables and the semi-naive algorithm is shown in Figure 7.6.

Run	Total # of Changes	# of changes propagated by deltascm_LicenseInfo		Re-derivation from updated LicenseInfo
		Using Magic Sets	Naive Approach	
1	60	9	60	6982
2	80	12	80	7000
3	120	14	120	7003

Table 7.1: Criminal Justice: Comparing # of changes propagated using IVM algorithms.

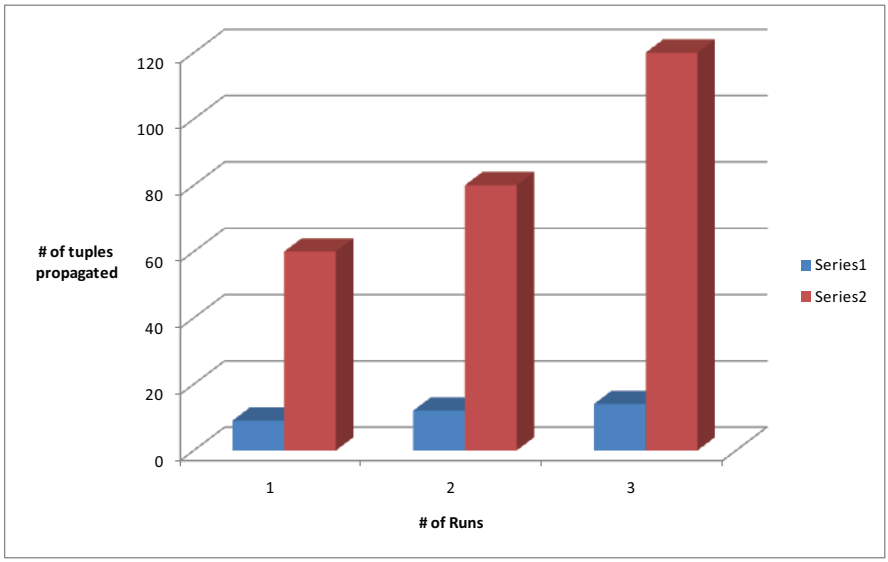


Figure 7.6: Criminal Justice: Comparing number of tuples propagated using magic tables algorithm and semi-naive algorithm.

7.3 Evaluation using TPC-H Data Model

Most of the database research community has been using data models provided by the Transaction Processing Performance Council (TPC) for benchmarking different database systems [[50]]. This research is using the TPC-H data model as the other data model for testing the DEPA environment. The TPC-H data model is specifically designed for benchmarking ad-hoc, decision support systems. Since the TPC-H is a

relational benchmark, this research has modified the enterprise by converting some of the relational tables into XML documents. The UML diagram for the hybrid TPC-H enterprise is shown in Figure 7.7. Similar to the Criminal Justice UML diagram, the TPC-H UML diagram is also annotated to indicate the XML data sources. This section illustrates the workings of the different algorithms and data structures using a more complex example scenario over the TPC-H data model.

Consider the experimental setup as shown in Figure 7.19. The setup has access to six heterogeneous data sources, three relational tables and three XML documents. The `Supplier` relational table contains information related to suppliers who supplies parts for certain projects. The information for each part is stored in the `Part` relational table and the supply information for each part and its supplier is in the relational table `PartSupp`. Customers order certain parts from the suppliers for their projects. The `Customers.xml` file contains the customer information. The orders placed by the customers are stored in the `Orders.xml` file and details on each item ordered is stored in the `LineItems.xml` file.

Assume that there are four queries: one SQL query, one XQuery query and two LINQ queries defined in one DEPA within the framework. These queries access different data sources as shown in Listing 7.6.

```
var lineitemsXML =
    XElement.Load("lineitems.xml").Elements("LINEITEM");
var ordersXML =
    XElement.Load("orders.xml").Elements("ORDER");
var customersXML =
    XElement.Load("customers.xml").Elements("CUSTOMER");

SQL Query Q1:
select /* project needed fields */
from supplier s, partsupp ps
where s.s_suppkey = ps.ps_suppkey and
      s.s_nationkey = 24 and
      ps.ps_availqty > 9000;


LINQ Query Q2:
var Q2 =
    from li in lineitemsXML
```

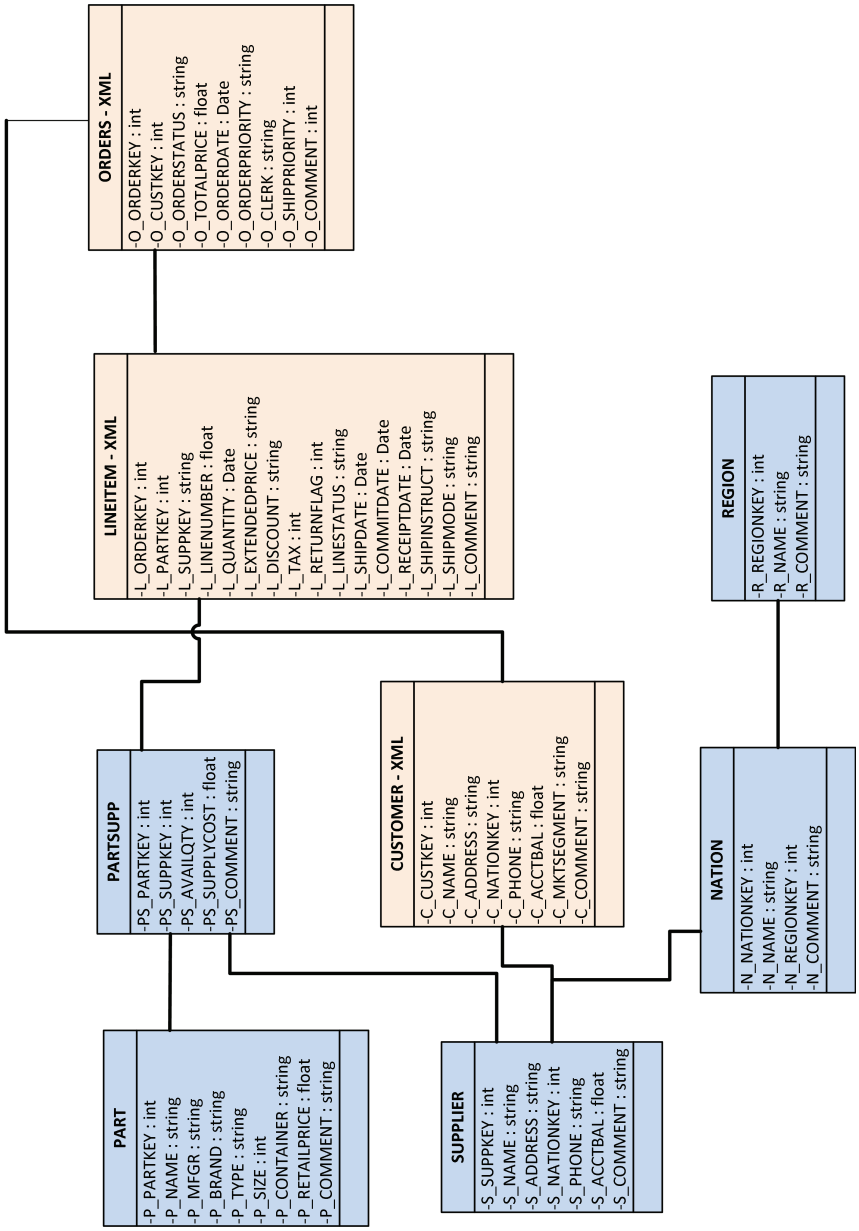


Figure 7.7: Hybrid TPC-H enterprise UML diagram.

```
    from ps in PARTSUPP
    from p in PART
    from s in SUPPLIER
    where p.P_PARTKEY == ps.PS_PARTKEY &&
          ps.PS_AVAILQTY > 9000 && ps.PS_PARTKEY == Int32.Parse(li.
              Element("L_PARTKEY").Value) &&
          Double.Parse(li.Element("L_QUANTITY").Value) > 10 &&
          s.S_NATIONKEY == 24 && s.S_SUPPKEY == ps.PS_SUPPKEY
    select /* project needed fields */;

LINQ Query Q3:
var Q3 =
    from li in lineitemsXML
    from o in ordersXML
    from ps in PARTSUPP
    from s in SUPPLIER
    where Double.Parse(li.Element("L_QUANTITY").Value) > 10 &&
          ps.PS_PARTKEY == Int32.Parse(li.Element("L_PARTKEY").Value)
              && ps.PS_AVAILQTY > 9000 &&
          li.Element("L_ORDERKEY").Value == o.Element("O_ORDERKEY").
              Value &&
          s.S_SUPPKEY == ps.PS_SUPPKEY && s.S_NATIONKEY == 24
    select /* project needed fields */;

XQuery Query Q4:
for $c in doc("customers.xml")/Customers/Customer
for $li in doc("LineItems.xml")/LineItems/LineItem
for $o in doc("Orders.xml")/Orders/Order
where $c/c_custkey = $o/o_custkey and
      $o/o_orderkey = $li/l_orderkey and
      $li/l_quantity > 10
return /* project needed fields */;
```

Listing 7.6: Sample queries defined over hybrid TPC-H model.

Once the queries are registered with the DEPA, the DEPA analyzes this set of queries for multiple query optimization. The queries are represented in the mixed multigraph model by parsing each query. There are three separate Antlr-based parsers that parse each query to provide access to different parts of the queries. The multigraph representation of the four queries is shown in Figure 7.8. The DEPA then starts analyzing the multigraph to detect any common subexpressions, which are marked as the candidates for materialized views. The multigraph representation and the heuristics-based algorithm to detect common subexpressions are presented in chapter 4.

The common subexpression detection algorithm starts selecting the edges based on the heuristic rules described in the chapter 4. Rule 1 detects and selects the identi-

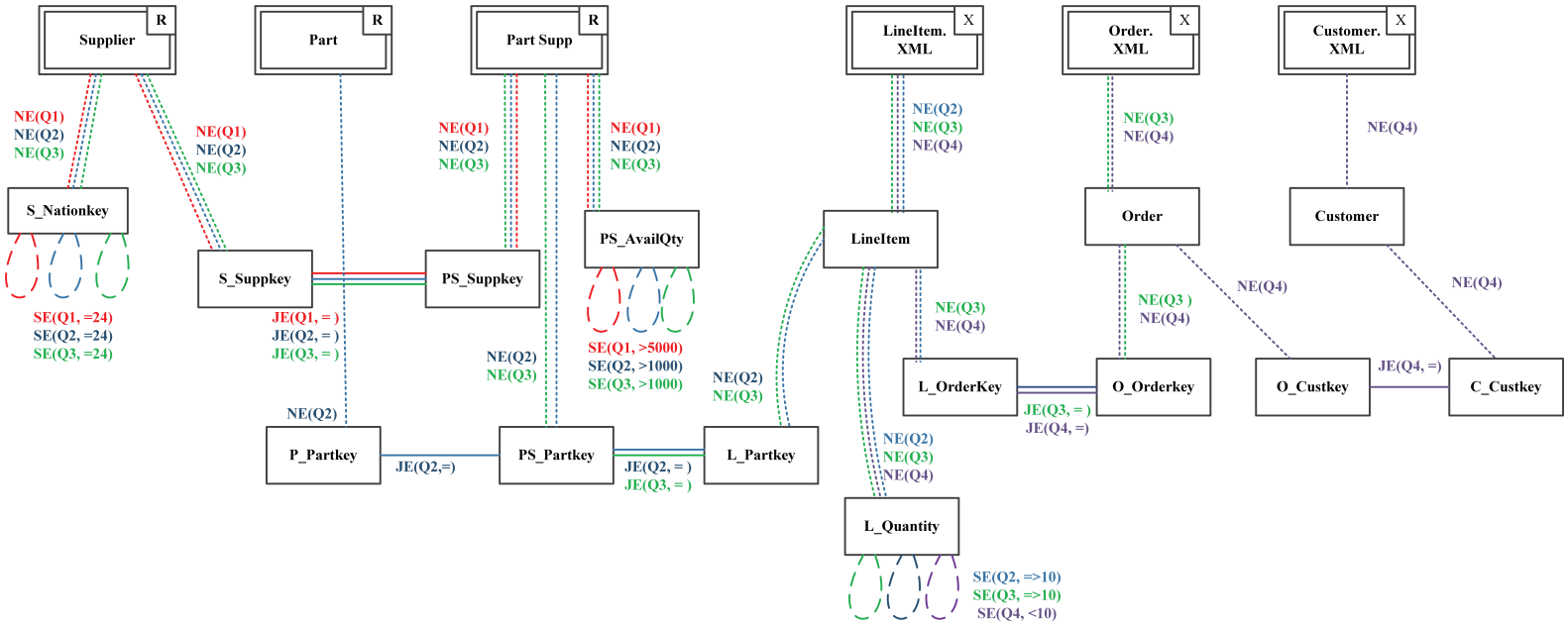


Figure 7.8: Multigraph representation of the TPC-H queries.

cal selection conditions on the nodes S_NationKey and L_Quantity. Rule 1 also detects identical and subsumed selection conditions on the node PS_AvailQty. The identical and subsumed selection conditions are indicated using dot-dash boxes as shown in Figure 7.9. After taking the appropriate steps of actions from rule 1, the common subexpression *CS*1 as *Supplier.S_NationKey* ="24" is added to the list *CSet* and the *S_NationKey* node is replaced by a new node *S_NationKey*= "24". Similarly, the node PS_AvailQty is processed for combined identical and subsumed selection conditions and the common subexpression *CS*2 as $PartSupp.PS\_AvailQty > 1000$ is added to the list *CSet* and the *PS_AvailQty* node is replaced by a new node $PS\_AvailQty{>}1000$. The node L_Quantity is also processed for identical selection conditions and the common subexpression *CS*3 as $LineItem.XML/LineItem/L\_Quantity >= 10$ is added to the list *CSet* and the *L_Quantity* node is replaced by a new node $L\_Quantity{>=}10$. The modified multigraph is shown in Figure 7.10.

The algorithm continues on to use rule 2 to detect any identical and subsumed join conditions from the graph. There are three identical join conditions in the graph. Rule 2 processes the first join condition *Supplier.S_Suppkey* = *PartSupp.PS_Suppkey* by creating a new node $Supplier \bowtie_{S\_Suppkey=PS\_SuppKey} PartSupp$. After applying the conditions from rule 2, the qualifying nodes, relevant selection, join and navigational edges are now associated with the newly created node. The modified multigraph is shown in Figure 7.11. The common subexpression $Supplier \bowtie_{S\_Suppkey=PS\_SuppKey}$ *PartSupp* is added to the list *CSet*.

The second identical join condition $LineItem.XML/LineItem/L\_Orderkey$ = $Order.XML/Order/O\_Orderkey$ is processed to create a new node $LineItem.XML \bowtie_{L\_Orderkey=O\_Orderkey} Order.XML$. The qualifying nodes and the edges are transferred to this newly created node. The common subexpression $LineItem.XML \bowtie_{L\_Orderkey=O\_Orderkey} Order.XML$ is added to the list *CSet*. The modified multigraph is shown in Figure 7.12.

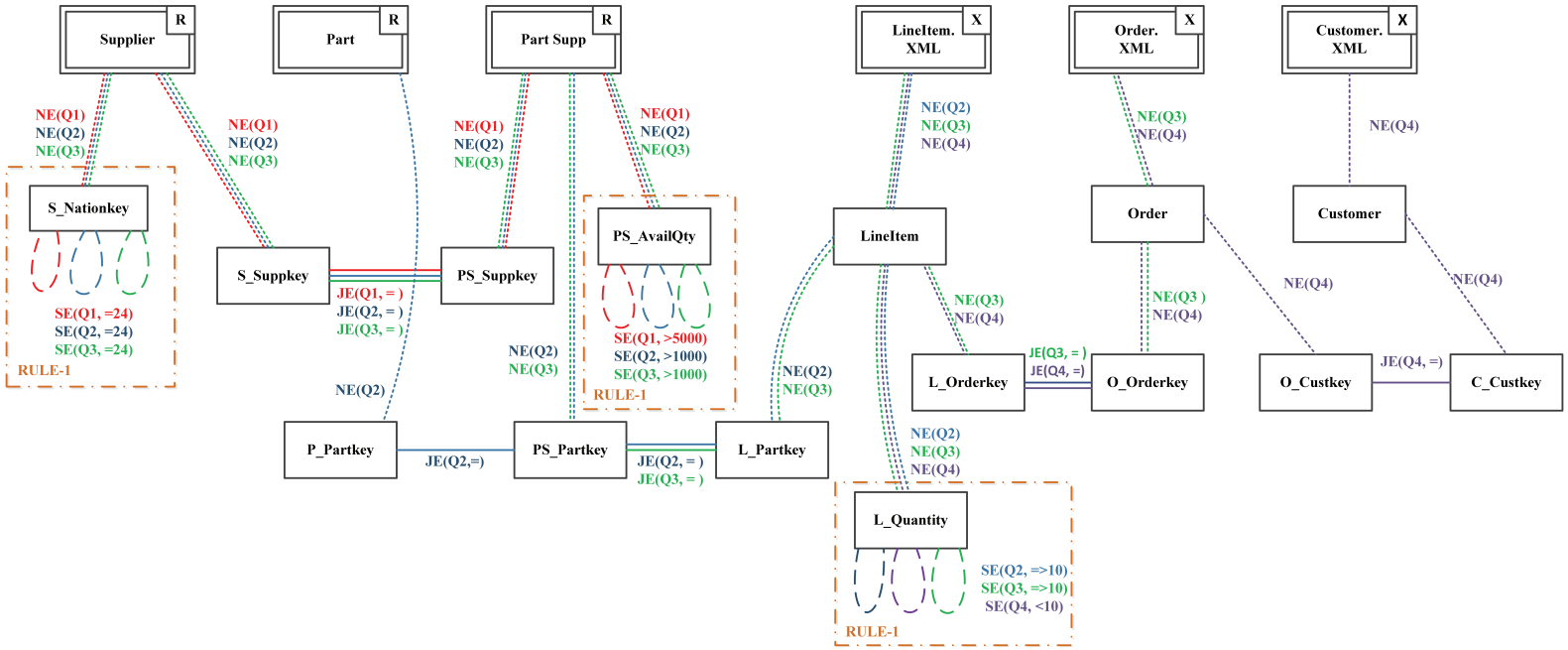


Figure 7.9: Multigraph with chosen identical and subsumed selection conditions.

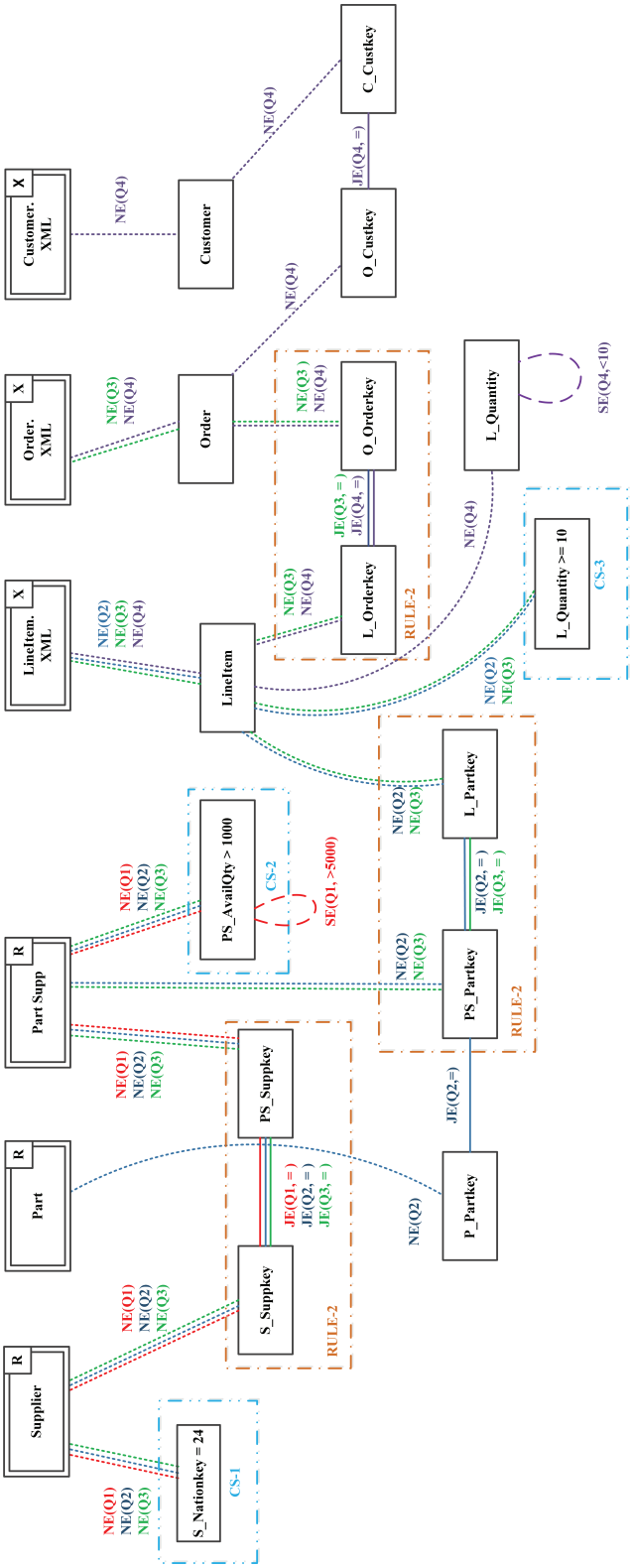


Figure 7.10: Modified multigraph after applying rule 1.

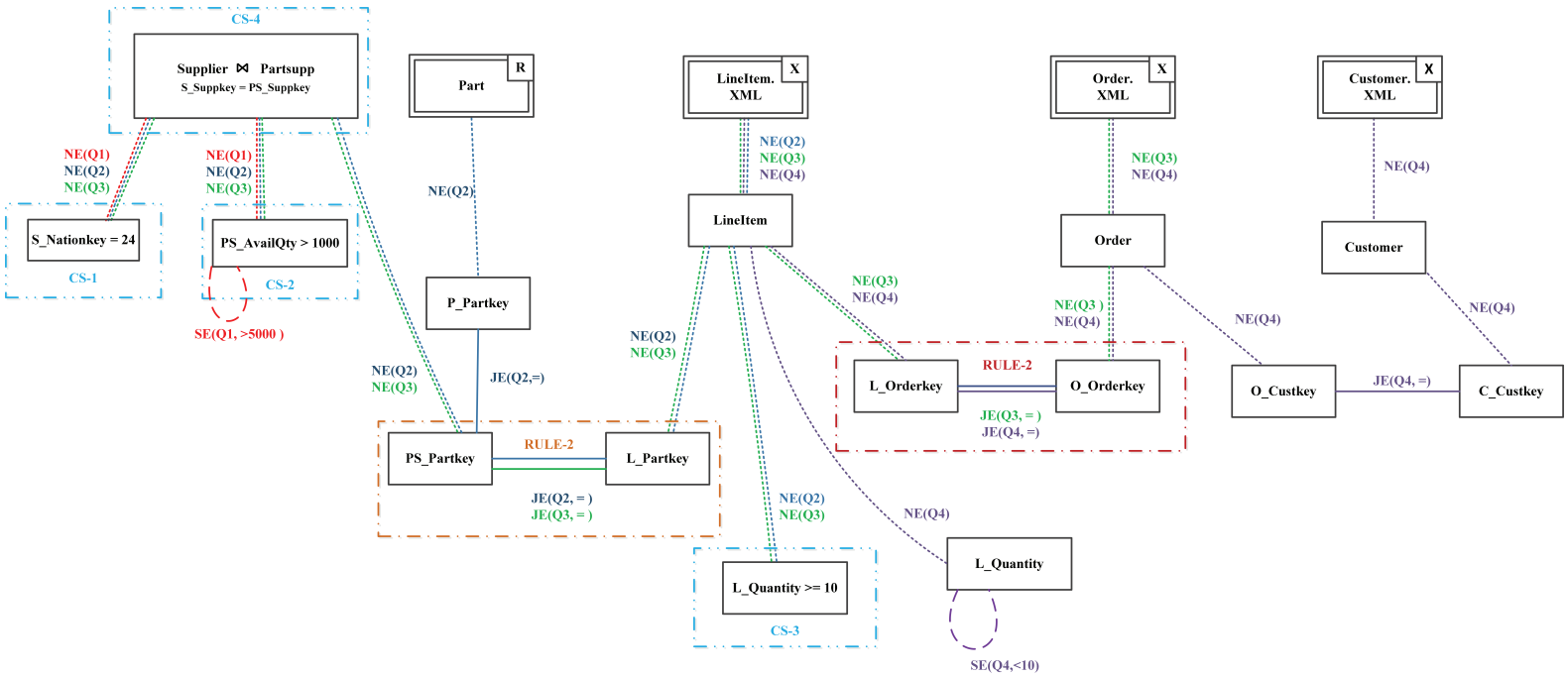


Figure 7.11: Modified multigraph after applying rule 2 step (a).

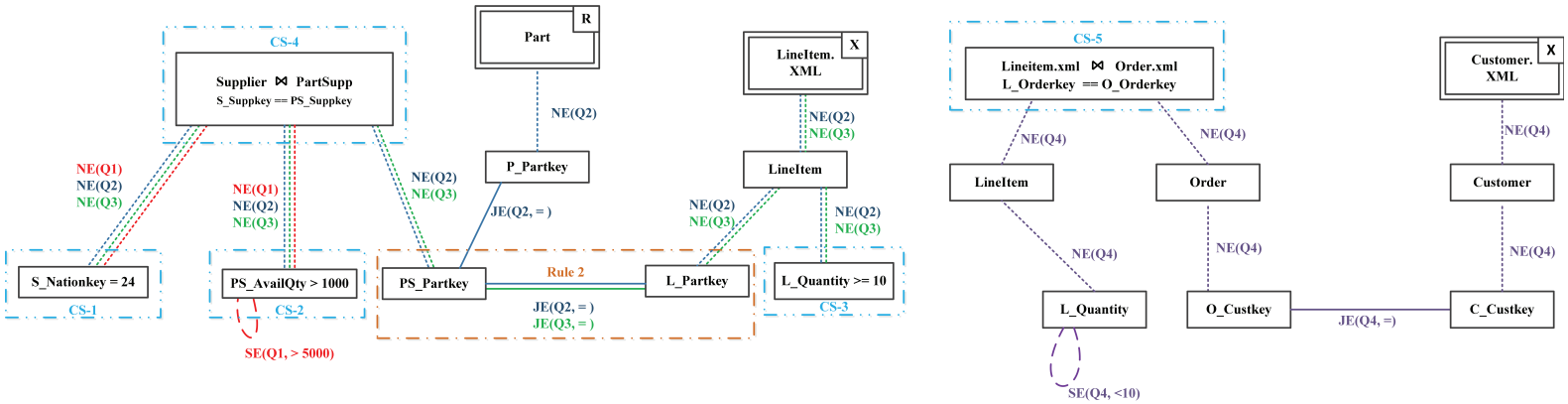


Figure 7.12: Modified multigraph after applying rule 2 step (b).

Finally, the remaining identical join condition $Supplier \bowtie_{S\_Suppkey=PS\_SuppKey} PartSupp.PS\_Partkey = LineItem.XML/LineItem/L\_Partkey$ is processed. A new node $Supplier \bowtie_{S\_Suppkey=PS\_SuppKey} PartSupp \bowtie_{PS\_Partkey=L\_PartKey} LineItem.XML$ is created. In this case also, the qualifying nodes and the edges are transferred to the newly created node. The common subexpression $Supplier \bowtie_{S\_Suppkey=PS\_SuppKey} PartSupp \bowtie_{PS\_Partkey=L\_PartKey} LineItem.XML$ is added to the list $CSet$. The final modified multi-graph is shown in Figure 7.13.

Since, there are no more identical or subsumed selection conditions or join conditions to be analyzed, the algorithm terminates with six common subexpressions. $CS1$, $CS2$, $CS3$, and $CS4$ are purely relational in structure where as $CS5$ is purely XML in structure. $CS6$ is a hybrid join over two relational tables and one XML document. Since this dissertation is focusing on defining and maintaining hybrid materialized views, consider only those common subexpressions that contribute towards defining such a view. The common subexpressions $CS1$, $CS2$, $CS3$, and $CS6$ can be materialized into a hybrid view to answer the queries $Q2$ and $Q3$. The view definition and creation algorithms described in the chapter 5 are applied to this set of common subexpressions. The view definition and the creation statements for the materialized view based on the four common subexpressions are shown in Listing 7.7.

```
var lineitemsXML =
    XElement.Load("lineitems.xml").Elements("LINEITEM");

View V2:
View Definition =
    from T0 in lineitemsXML
    from T1 in SUPPLIER
    from T2 in PARTSUPP
    where T2.PS_AVAILQTY > 9000 && T2.PS_PARTKEY == Int32.Parse(T0.
        Element("L_PARTKEY").Value) &&
           Double.Parse(T0.Element("L_QUANTITY").Value) > 10 &&
           T1.S_NATIONKEY == 24 && T1.S_SUPPKEY == T2.PS_SUPPKEY
    select new {SUPPLIER_S_NATIONKEY = T1.S_NATIONKEY,
                SUPPLIER_S_SUPPKEY = T1.S_SUPPKEY,
                .... /* all the remaining columns from the SUPPLIER
                   table */
                PARTSUPP_PS_AVAILQTY = T2.PS_AVAILQTY,
                PARTSUPP_PS_PARTKEY = T2.PS_PARTKEY,
```

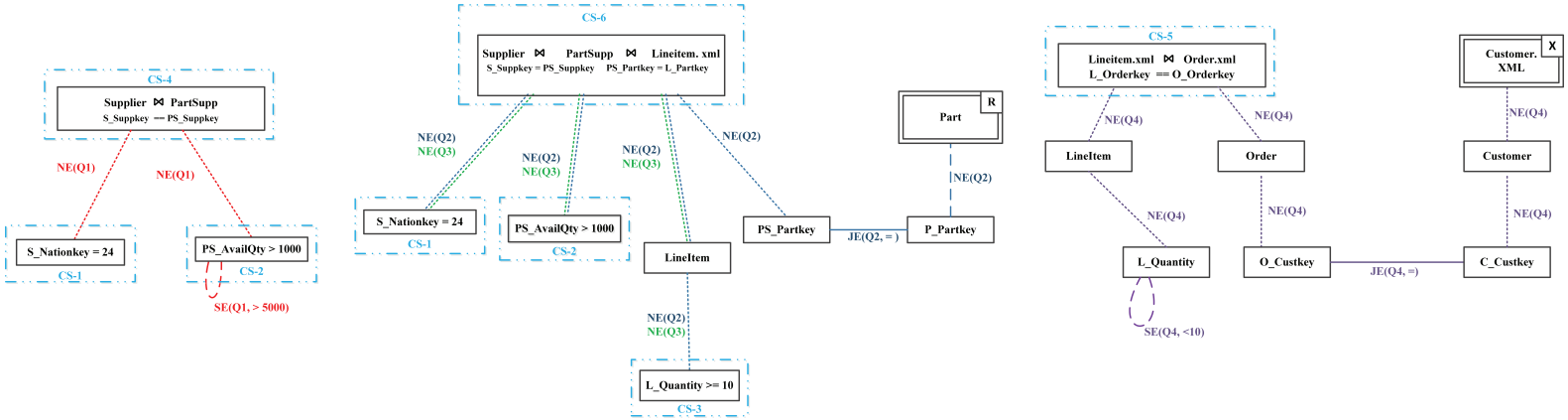


Figure 7.13: Modified multigraph after applying rule 2 step (c).

```
                .... /* all the remaining columns from the PARTSUPP
                   table */
                T0 = T0
                };

View Creation:
Create Table V2
        (SUPPLIER_S_NATIONKEY int,
        SUPPLIER_S_SUPPKEY string,
        .... /* all the remaining columns from the SUPPLIER table */
        PARTSUPP_PS_AVAILQTY int,
        PARTSUPP_PS_PARTKEY string,
        .... /* all the remaining columns from the PARTSUPP table */
        T0 XML);
```

Listing 7.7: View definition and creation statements.

Once the view definition and creation statements are defined in the system, then the magic-sets optimization technique can be applied to the view definition to create magic tables. These magic tables are used to materialize the view as well as to propagate the updates to incrementally update the view. The view maintenance algorithm from the chapter 6 accepts the view definition from Listing 7.7 represented in a query graph model (QGM) as shown in Figure 7.14. Since the view definition contains joins over three data sources, the join ordering is determined by the order in which tuples are utilized in the query computation. LINQ uses the right to left ordering, thus, the `PartSupp` table is joined with `Supplier`, the result of which is then joined with the `LineItem.XML` document. The revised queries are shown in Listing 7.8.

```
var lineitemsXML =
    XElement.Load("lineitems.xml").Elements("LINEITEM");

var m_Supplier_PartSupp =
    from T1 in SUPPLIER
    from T2 in PARTSUPP
    where T1.S_SUPPKEY == T2.PS_SUPPKEY
    select new {SUPPLIER_S_NATIONKEY = T1.S_NATIONKEY,
                SUPPLIER_S_SUPPKEY = T1.S_SUPPKEY,
                .... /* all the remaining columns from the SUPPLIER
                   table */
                PARTSUPP_PS_AVAILQTY = T2.PS_AVAILQTY,
                PARTSUPP_PS_PARTKEY = T2.PS_PARTKEY
                .... /* all the remaining columns from the PARTSUPP
                   table */
                };
```

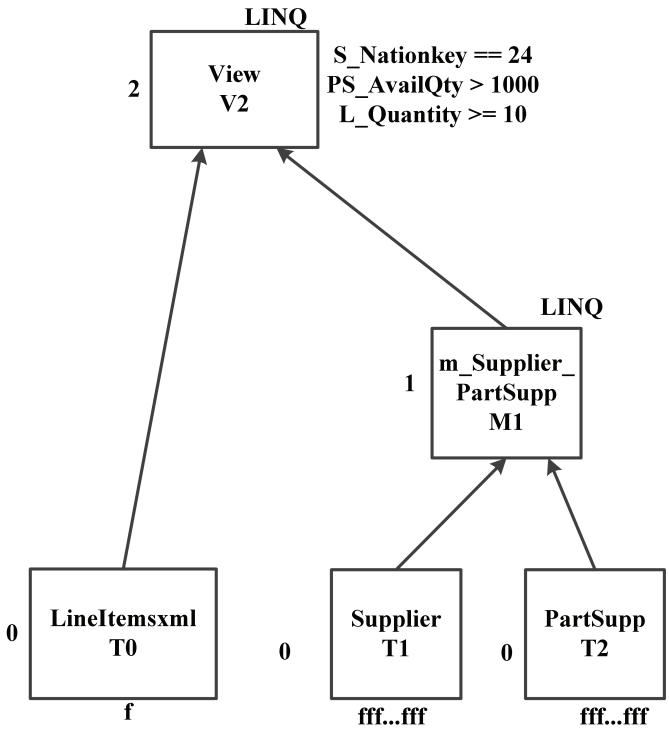


Figure 7.14: Query graph model for the view definition.

```
var ViewDefinition =
    from T0 in lineitemsXML
    from M1 in m_Supplier_PartSupp
    where M1.PARTSUPP_PS_AVAILQTY > 9000 &&
          M1.PARTSUPP_PS_PARTKEY == Int32.Parse(T0.Element("L_PARTKEY
              ").Value) &&
          Double.Parse(T0.Element("L_QUANTITY").Value) > 10 &&
          M1.SUPPLIER_S_NATIONKEY == 24
    select new {SUPPLIER_S_NATIONKEY = M1.SUPPLIER_S_NATIONKEY,
                SUPPLIER_S_SUPPKEY = M1.SUPPLIER_S_SUPPKEY,
                PARTSUPP_PS_AVAILQTY = M1.PARTSUPP_PS_AVAILQTY,
                PARTSUPP_PS_PARTKEY = M1.PARTSUPP_PS_PARTKEY,
                .... /* all the remaining columns from the
                   m_Supplier_PartSupp table */
                T0 = T0
                };
```

Listing 7.8: View queries for the TPC-H query graph model

As the magic sets incremental view maintenance algorithm traverses the QGM in depth-first order, the box containing the view definition is processed first. Since the box View V2 is an AMQ box, all the predicates inside this box can be pushed down. The adornment for this box is `fff...bbb...f`, since there are three bound variables

and rest of the attributes are free. Thus, the inequality predicate `T0.Element("Line-Item").Element("L_Quantity") >= 10` is pushed down to create a condition magic box `cm_LineItemsXML`. Once this box is created, the quantifier T0 in the original view box is replaced by a new quantifier M2 created for this newly created magic box. The adornment for the magic box `cm_LineItemsXML` is b since there is one variable binding. Similarly, the two predicates `M1.S_NationKey == 10` and `M1.PS_AvailQty > 1000` are pushed down to the magic box `m_Supplier_PartSupp`. The adornment for the magic box `m_Supplier_PartSupp` is `fff...bb...f` since the two variables `M1.S_NationKey` and `M1.PS_AvailQty` are bound and rest of the variables are free. The modified QGM and the revised queries are shown in Figure 7.15 and Listing 7.9, respectively.

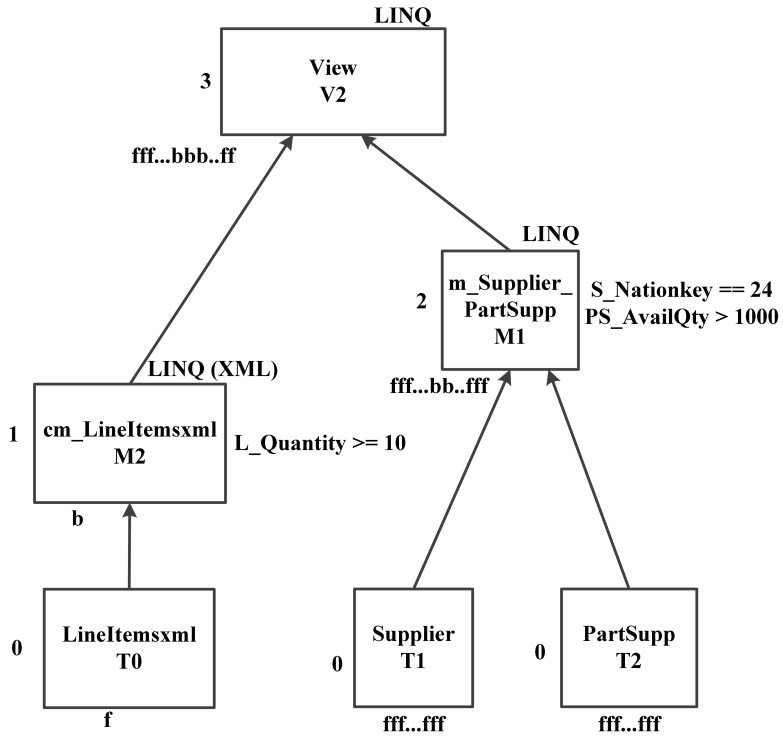


Figure 7.15: Modified query graph model - step 1.

```
var lineitemsXML =
    XElement.Load("lineitems.xml").Elements("LINEITEM");

var cm_LineItemsxml =
```

```
    from T0 in lineitemsXML
    where Double.Parse(T0.Element("L_QUANTITY").Value) > 10
    select new {T0 = T0};

var m_Supplier_PartSupp =
    from T1 in SUPPLIER
    from T2 in PARTSUPP
    where T1.S_SUPPKEY == T2.PS_SUPPKEY && T1.S_NATIONKEY == 24 &&
          T2.PS_AVAILQTY > 9000
    select new {SUPPLIER_S_NATIONKEY = T1.S_NATIONKEY,
                SUPPLIER_S_SUPPKEY = T1.S_SUPPKEY,
                .... /* all the remaining columns from the SUPPLIER
                   table */
                PARTSUPP_PS_AVAILQTY = T2.PS_AVAILQTY,
                PARTSUPP_PS_PARTKEY = T2.PS_PARTKEY
                .... /* all the remaining columns from the PARTSUPP
                   table */
              };

var ViewDefinition =
    from M2 in cm_LineItemsxml
    from M1 in m_Supplier_PartSupp
    where  M1.PARTSUPP_PS_PARTKEY == Int32.Parse(M2.Element("
       L_PARTKEY").Value)
    select new {SUPPLIER_S_NATIONKEY = M1.SUPPLIER_S_NATIONKEY,
                SUPPLIER_S_SUPPKEY = M1.SUPPLIER_S_SUPPKEY,
                PARTSUPP_PS_AVAILQTY = M1.PARTSUPP_PS_AVAILQTY,
                PARTSUPP_PS_PARTKEY = M1.PARTSUPP_PS_PARTKEY,
                .... /* all the remaining columns from the
                   m_Supplier_PartSupp table */
                T0 = M2
                };
```

Listing 7.9: View queries for the TPC-H query graph model - step 1

As the algorithm continues to traverse down the query graph, there is nothing more to process on the magic box `cm_LineItemsXML`. The algorithm directly processes the magic box `m_Supplier_PartSupp`. Since this magic box is also an AMQ box, the two predicates `T1.S_NationKey == 10` and `T2.PS_AvailQty > 1000` are pushed down. The first equality predicate `T1.S_NationKey == 10` creates a supplementary magic box `sm_Supplier` and the quantifier `T1` in the magic box `m_Supplier_PartSupp` is replaced by a new quantifier `M3` for the newly created supplementary magic box. Similarly, the inequality predicate `T2.PS_AvailQty > 1000` creates a condition magic box `cm_PartSupp`. The quantifier `T2` is also replaced by the new quantifier `M4` in the magic box`m_Supplier_PartSupp`. The final

modified QGM and the revised queries are shown in Figure 7.16 and Listing 7.10, respectively.

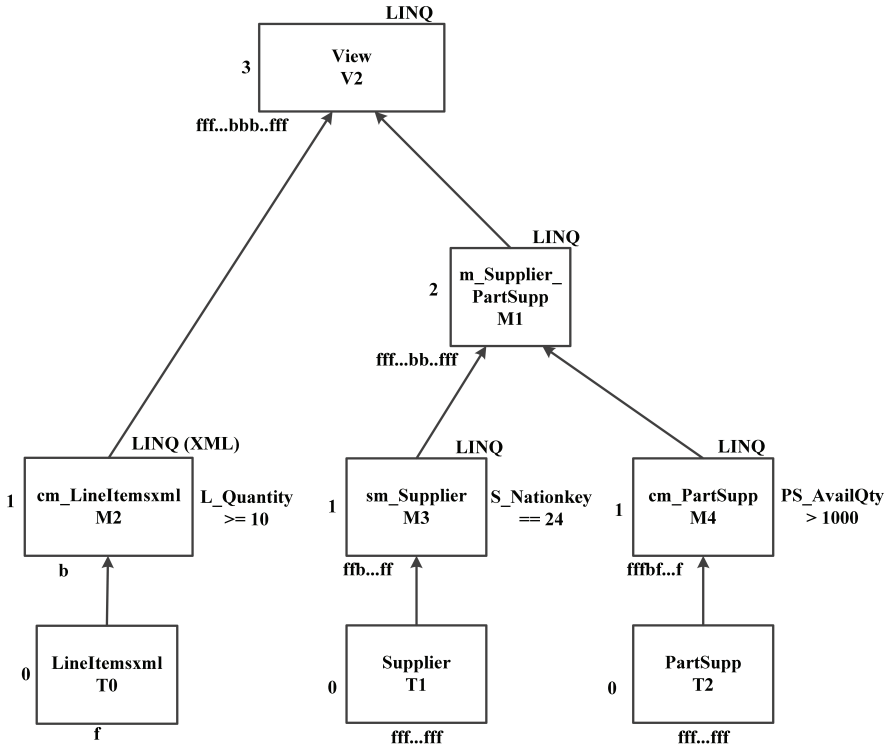


Figure 7.16: Modified query graph model - step 2.

```
var lineitemsXML =
    XElement.Load("lineitems.xml").Elements("LINEITEM");

var cm_LineItemsxml =
    from T0 in lineitemsXML
    where Double.Parse(T0.Element("L_QUANTITY").Value) > 10
    select new {T0 = T0};

var sm_Supplier =
    from T1 in SUPPLIER
    where T1.S_NATIONKEY == 24
    select new {SUPPLIER_S_NATIONKEY = T1.S_NATIONKEY,
                SUPPLIER_S_SUPPKEY = T1.S_SUPPKEY,
                .... /* all the remaining columns from the SUPPLIER
                   table */
               };

var cm_PartSupp =
    from T2 in PARTSUPP
    where T2.PS_AVAILQTY > 9000
    select new {PARTSUPP_PS_AVAILQTY = T2.PS_AVAILQTY,
                PARTSUPP_PS_PARTKEY = T2.PS_PARTKEY,
```

```
                PARTSUPP_PS_SUPPKEY = T2.PS_SUPPKEY,
                .... /* all the remaining columns from the PARTSUPP
                   table */
                };

var m_Supplier_PartSupp =
    from M3 in sm_Supplier
    from M4 in cm_PartSupp
    where M3.SUPPLIER_S_SUPPKEY == M4.PARTSUPP_PS_SUPPKEY
    select new {SUPPLIER_S_NATIONKEY = M3.SUPPLIER_S_NATIONKEY,
                SUPPLIER_S_SUPPKEY = M3.SUPPLIER_S_SUPPKEY,
                .... /* all the remaining columns from the
                   sm_Supplier table */
                PARTSUPP_PS_AVAILQTY = M4.PARTSUPP_PS_AVAILQTY,
                PARTSUPP_PS_PARTKEY = M4.PARTSUPP_PS_PARTKEY,
                .... /* all the remaining columns from the
                   cm_PartSupp table */
              };

var ViewDefinition =
    from M2 in cm_LineItemsxml
    from M1 in m_Supplier_PartSupp
    where  M1.PARTSUPP_PS_PARTKEY == Int32.Parse(M2.Element("
       L_PARTKEY").Value)
    select new {SUPPLIER_S_NATIONKEY = M1.SUPPLIER_S_NATIONKEY,
                SUPPLIER_S_SUPPKEY = M1.SUPPLIER_S_SUPPKEY,
                PARTSUPP_PS_AVAILQTY = M1.PARTSUPP_PS_AVAILQTY,
                PARTSUPP_PS_PARTKEY = M1.PARTSUPP_PS_PARTKEY,
                .... /* all the remaining columns from the
                   m_Supplier_PartSupp table */
                T0 = M2
                };
```

Listing 7.10: View queries for the TPC-H query graph model - step 2.

Similar to the performance evaluation on the Criminal Justice data model, performance of the different components of the DEPA framework was tested using the TPC-H data model. The time required to process the streaming data over one relational table and one XML document from the TPC-H data model is compared with the time required to process the streaming data using a materialized view over the same relational table and the XML document. The graph of both the recorded timings is shown in Figure 7.17. The average time required to process 110 streaming tuples without the use of materialized views is approximately 252 milliseconds, whereas the time required to process the same 110 streaming tuples with the use of materialized views is approximately 53 milliseconds. Thus, on this example, the streaming tuples are processed 5

times faster with the use of a materialized view.

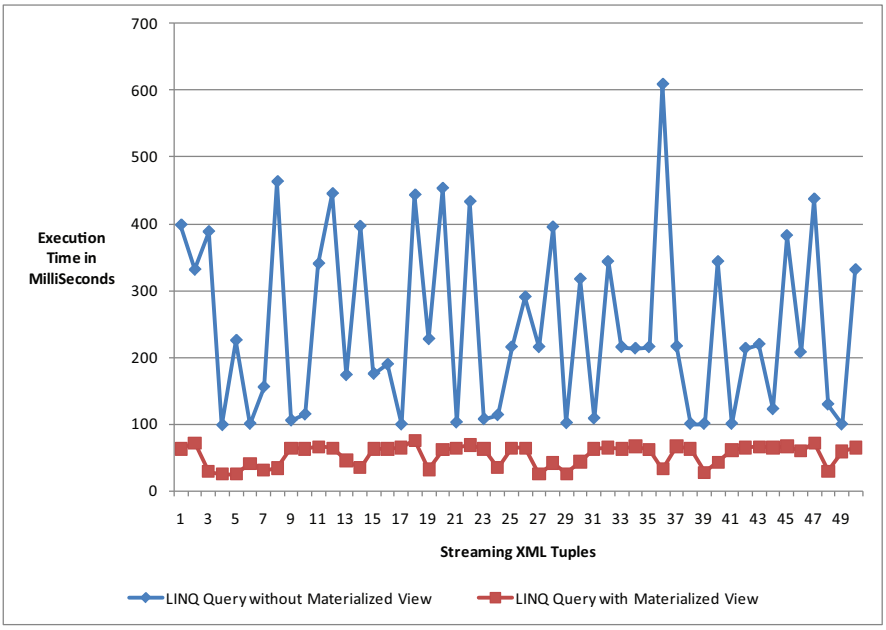


Figure 7.17: TPC-H: Comparing time to process streaming data with and without using materialized view.

The second performance evaluation is done to compare the time required for materializing the view with and without the use of magic tables. Also these timings are compared with the time required to materialize the view using magic tables stored in the Dataset. The graph showing these different timings is shown in Figure 7.18. The average time taken to materialize the view using magic tables stored in the Dataset is approximately 2.4 seconds. The average time taken to materialize the same view using magic tables (not stored in the Dataset) is approximately 22.8 seconds, whereas the time taken to materialize the same view directly from the base data sources is approximately 3 minutes and 22 seconds. Thus, for this test scenario, it can be concluded that the view materialization using magic tables stored in Dataset is 7 times faster than using magic tables, which are not stored in the Dataset. Similarly, the time required to materialize the view using magic tables not stored in the Dataset is 8 times less than the time required to materialize the view directly from the base data sources. All these test evaluations show increased efficiency in processing streams over heterogeneous data

sources using materialized views wherever possible and materializing the view using magic tables stored in the Dataset.

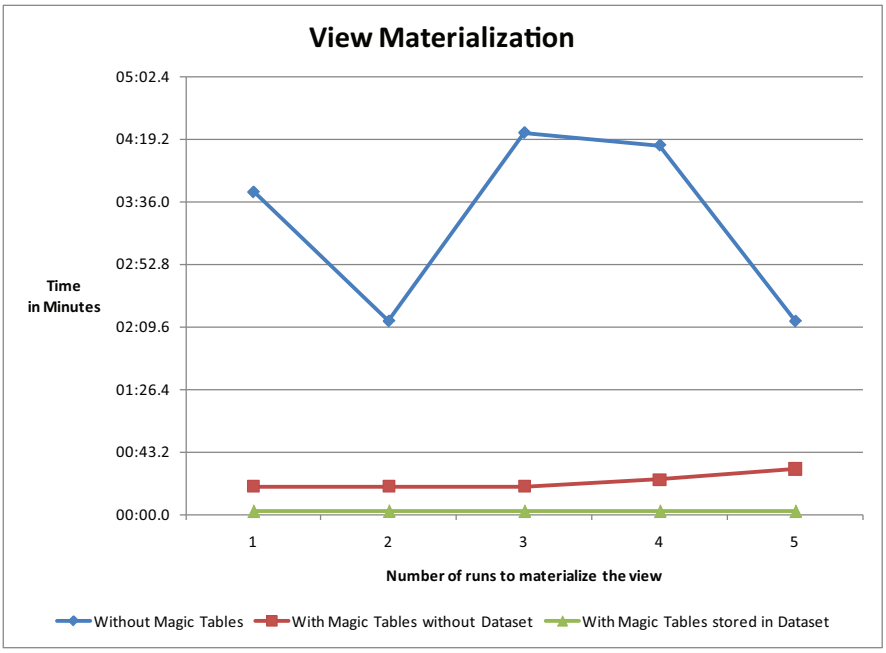


Figure 7.18: TPC-H: Comparing view materialization time with no magic tables, with magic tables (no Dataset) and with magic tables stored in Dataset.

Similar to the test scenario for the Criminal Justice enterprise, a test case is designed to compare the number of tuples propagated to the view definition using magic sets and semi-naive algorithm. Table 7.2 compares the number of relevant change tuples propagated to the view definition using the magic tables algorithm, the semi-naive algorithm and the re-derivation algorithm. The graphical comparison of the number of change tuples propagated using magic tables and the semi-naive algorithm is shown in Figure 7.20. This test analysis also shows a significant reduction in the number of tuples that are propagated using magic tables.

7.4 Summary

The distributed event stream processing framework supports materialized views over heterogeneous data sources for efficient query evaluation, stream processing and event

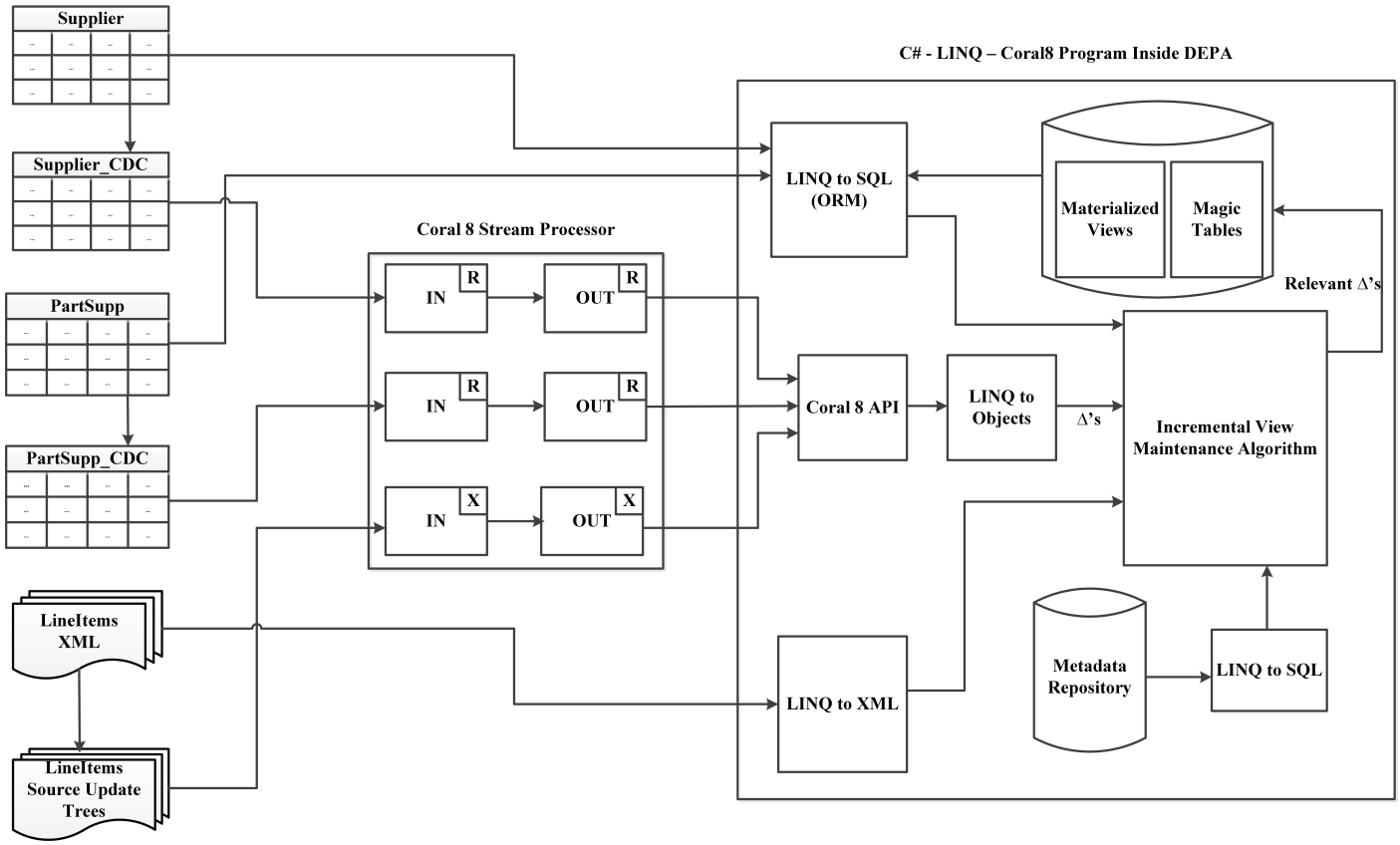


Figure 7.19: Experimental setup based on the hybrid TPC-H data model.

Run	Total # of Changes	# of changes propagated by deltasm_Supplier		Re-derivation from updated Supplier
		Using Magic Sets	Naive Approach	
1	40	7	40	103
2	60	4	60	100
3	80	10	80	115

Table 7.2: TPC-H: Comparing # of changes propagated using IVM algorithms.

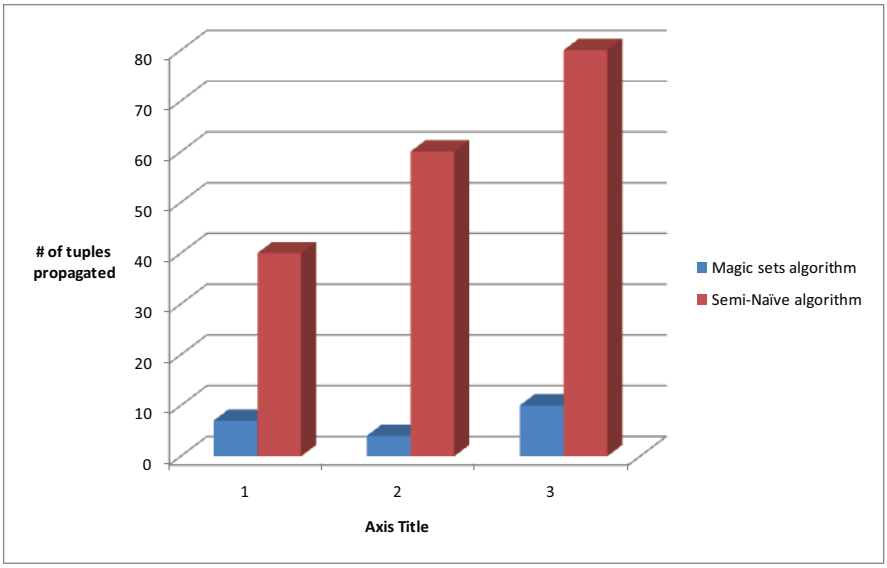


Figure 7.20: TPC-H: Comparing number of tuples propagated using magic tables algorithm and semi-naive algorithm.

detection. This chapter has first described the prototype developed as the proof-of-concept for the DEPA framework. The prototype is built using two popular commercial database systems and XML documents as the distributed heterogeneous data sources. The performance of the various algorithms presented throughout this dissertation was evaluated using two different data models. The detailed evaluation of the algorithms was presented using the Criminal Justice data model. Later in the chapter, similar test scenarios are developed and evaluated using the modified hybrid TPC-H data model. The DEPA framework was evaluated for processing streams through a LINQ query over heterogeneous data sources against processing the streams through the same LINQ query using a hybrid materialized view over the heterogeneous data sources. The test results indicated that use of materialized views improves the time for processing the

streams. The materialization of the view that retains the data in their native format by using the magic-sets query optimization technique is faster than the naive approach of view computation directly from the base data sources. The magic tables decreases the query evaluation time by early pruning process and propagating only relevant tuples for the view materialization. The view materialization using magic sets is faster than the naive materialization. However, in the original algorithm, magic tables are not stored in the main memory for subsequent access. The incremental view maintenance algorithm presented in this chapter stores these magic tables in the main memory using Datasets. The view materialization using the in-memory magic sets is faster as compared to view materialization using magic tables not stored in the main memory. Finally, this chapter has evaluated the incremental view maintenance algorithm using magic tables against the semi-naive approach for maintaining the views incrementally. This evaluation is based on a metric of the number of change tuples propagated to the view definition. Various evaluation scenarios have shown that using magic tables typically propagates a significantly fewer number of qualifying delta tuples to the view for incremental maintenance than the semi-naive approach. Thus, defining and maintaining LINQ-based materialized views significantly improves the performance of the DEPA framework.

Chapter 8

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

8.1 Summary

Enterprise applications are becoming increasingly dynamic, requiring the monitoring of streaming data and events in a distributed environment. The streaming data is processed using continuous queries with access to persistent data sources. Events detect complex and meaningful relationships among the data occurring over the streams in conjunction with heterogeneous data sources. The active rules define the behavior of the application in response to the detected events. Finally, there are queries posed to the applications to generate meaningful results from various different data sources. All these different query expressions access heterogeneous distributed data sources. Thus, there are research challenges associated with developing these applications to improve their performance by enhancing the querying capabilities in such systems. This research has explored a framework using distributed event stream processing agents that supports such application functionalities. Specifically, this dissertation has addressed research challenges involved in defining and maintaining materialized views over heterogeneous data sources to support multiple query optimization.

In the DEPA framework, each agent maintains a set of different types of query expressions, such as continuous queries, event definitions, and SQL, XML and LINQ queries over heterogeneous data sources, such as relational databases and structured XML documents. In order to provide metadata level access to different components of the various query languages and the data sources, this research has designed and implemented a SOA-based metadata repository. This repository can be updated at runtime and is a useful component throughout the development of different components of the DEPA environment. To support multiple query optimization, DEPA agents detect common subexpressions across these various query expressions. The query expressions are represented in a mixed multigraph model and a heuristics-based algorithm is

used to detect the common subexpressions. The detected common subexpressions are either relational or XML or a hybrid join over relational and XML data sources. These common subexpressions are the candidates for materialized views.

In any database system, materialized views store the results of the computed view locally to avoid the cost of recomputing the view every time the view is accessed. This improves the query evaluation by reducing the cost of the query computation considerably. One of the challenges addressed by this research is the definition and materialization of views when the heterogeneous data sources are retained in their native format, instead of converting the data to a common model. LINQ is used as the materialized view definition language to define materialized views locally to each DEPA. LINQ integrates the querying capabilities over heterogeneous data sources in a single query and provides the option of retaining the data in their native format. The view definition algorithm uses the detected common subexpressions to generate a LINQ query that generate the qualifying tuples for the materialized views. An algorithm is developed that uses LINQ to create a data structure for the persistence of these hybrid views.

Materialized views are typically used to improve the performance of data oriented applications. However, the views must be maintained against any changes occurring in the base data sources. The materialized views can be updated either by rederiving the view completely from the updated base data sources or by capturing only the changes occurring in the base data sources and using these changes to incrementally update the views. A common delta structure is proposed to stream the relational changes from two commercial database systems used within the DEPA framework. The changes in XML documents are captured as source update trees and are also streamed to the respective DEPAs. The incremental view maintenance algorithm presented in this dissertation uses the concept of magic-sets query optimization to capture the streaming changes in their native format to incrementally update the materialized views. The magic-sets based algorithm uses the captured deltas to first incrementally update the

magic tables and then propagate the relevant tuples for updating the view.

The prototype of the DEPA framework uses the Sybase Event Stream Processor, SQL Server 2008, Oracle 11g Server, XML documents and the C# programming language with .NET framework 3.5. The framework and the associated algorithms in this dissertation were evaluated using two different scenarios. One scenario is based on a Criminal Justice enterprise and queries developed as the part of this research. Another evaluation scenario is based on a modification of the TPC-H decision support benchmark to replace some relational sources with XML. The evaluation results indicate a performance improvement of approximately 6 times on the representative scenarios tested. Thus, defining LINQ-based materialized views and incrementally maintaining them by using magic tables over relational and structured XML data sources improves the performance of applications involving distributed event and stream processing.

8.2 Research Contributions

This dissertation has explored key database areas ranging from multiple query optimization, the detection of common subexpressions, and incremental view maintenance over heterogeneous data sources in a distributed event stream processing environment. This research has resulted in a working prototype of distributed event stream processing agents as envisioned in [[10]]. The prototype has been successfully tested and evaluated based on the Criminal Justice and hybrid TPC-H scenarios. This dissertation research has provided unique contributions in the area of multiple query optimization in a distributed event stream processing environment.

The service-based metadata repository developed for the DEPA framework is unique. The repository provides extensive metadata for various data sources and query expressions. Metadata is maintained for relational data sources, XML documents as XML schema, as well as the event and data streams. In addition, the repository provides metadata for the queries expressed in SQL, LINQ, XQuery, and the Sybase CEP CQL languages. The design and implementation of this metadata repository has already been

published [84].

Another contribution of this research is the definition of a mixed multigraph model to represent the heterogeneous query expressions and its use in a heuristics-based algorithm to detect common subexpressions over relational and structured XML data sources. Prior research has investigated the representation and detection of common subexpressions over a single data model - either relational or XML. The mixed multigraph model introduced in this research handles queries expressed over both relational and XML data sources. The developed heuristics-based algorithm uses this distinctive multigraph model to detect common subexpressions, which are the candidates for view materialization.

The use of LINQ as a materialized view definition language over heterogeneous data sources in a distributed event stream processing environment is novel. LINQ provides the capabilities to query heterogeneous data sources through a single query and allow the data to be retained in their original format. Thus, a LINQ-based materialized view can be either purely relational or purely XML or a hybrid combination of the two. The presented view definition algorithm uses the common subexpressions to generate a LINQ query that specifies the qualifying tuples for the materialized view. The view creation algorithm defines an appropriate data structure that will persist these query results, which are potentially the hybrid views.

Once materialized, these views must be updated to reflect changes to their base data sources. This research has developed an algorithm to incrementally update the materialized views defined in LINQ in response to either relational or XML changes. A common relational delta structure is defined to capture and stream changes from two popular commercial relational database systems. The XML deltas are also streamed in the form of source update trees. Using the magic sets query optimization approach, the view maintenance algorithm applies these deltas in their native format to incrementally update the view.

The implementation of the proof-of-concept DEPA framework using commercially available software is another contribution of this work. A hybrid Criminal Justice enterprise that consists of relational and XML data sources was developed along with sample scenarios to test the different algorithmic components of the DEPA environment. The TPC-H decision-support relational benchmark was modified to support XML data sources and the benchmarking SQL queries were rewritten in LINQ to query the relational and XML data sources. Representative scenarios from the hybrid TPC-H enterprise were also used in the assessment of the algorithms and the framework with respect to the developed technologies, which demonstrates an improvement in performance.

8.3 Future Research Directions

This dissertation has taken a first step towards representing SQL, LINQ and XQuery queries in the same multigraph model to detect common subexpressions across the different query languages. Also, this dissertation has provided algorithms to define LINQ-based materialized views over heterogeneous data sources and to use deltas in their native format to incrementally update the materialized views using magic sets transformations. However, in order to develop a fully functional distributed event stream processing framework, there are additional research avenues to be explored.

There are open research issues to further explore with respect to detecting common subexpressions over the mixed multigraph model in which different types of query expressions such as LINQ, SQL, and XQuery queries can be represented under the same graph model. In selectively choosing the common subexpressions, either they can be chosen to maximize the number of queries that can benefit from the materialized view or they can be chosen to maximize the number of common subexpressions to be materialized. Various test scenarios can be used to determine which option is beneficial to choose a set of common subexpressions to be materialized as views.

There are also future research opportunities with respect to the use of LINQ.

There are limitations imposed by the default LINQ provider from the .NET framework 3.5 that restricts access to only one persistent relational database, requiring other data sources to be in the main memory. However, this framework is extensible so that these limitations can be addressed through the implementation of a custom LINQ provider.

Another research avenue to explore is the application of this work to monitoring conditions within the DEPA framework. The monitoring of conditions in an active rule specified using a Condition-Action format is similar to the incremental maintenance of materialized views. When a change to a base data source occurs on which the condition is defined, the system must use the delta to determine whether the condition is satisfied. If the condition becomes true, then the relevant bindings from the change can be passed to the execution of the corresponding action.

REFERENCES

[1] S. Babu and J. Widom, "Continuous queries over data streams," *SIGMOD Rec.*, vol. 30, no. 3, pp. 109–120, 2001.

[2] L. Golab and M. T. Özsu, "Issues in data stream management," *SIGMOD Rec.*, vol. 32, no. 2, pp. 5–14, 2003.

[3] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed event-based systems*. Berlin; New York: Springer-Verlag, 2006, 9783540326519; Gero Mhl, Ludger Fiege , Peter Pietzuch.; :ill. ;24 cm; Includes bibliographical references and index.

[4] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams: a new class of data management applications," in *VLDB '2002: Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 215–226.

[5] S. Chakravarthy and D. Mishra, "Snoop: an expressive event specification language for active databases," *Data Knowledge Engineering*, vol. 14, November 1994. [Online]. Available: http://portal.acm.org/citation.cfm?id=197053.197054

[6] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing," in *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*, January 7-10, 2007 2007, pp. 363–374, crossref: DBLP:conf/cidr/2007. [Online]. Available: http://www.drdb.org/dr2007/papers/dr07p42.pdf

[7] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A general purpose event monitoring system," in *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*. www.crdrdb.org, 2007, pp. 412–422.

[8] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "Mavestream: Synergistic integration of stream and event processing," in *Proceedings of the Second International Conference on Digital Telecommunications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 29–. [Online]. Available: http://portal.acm.org/citation.cfm?id=1270389.1271026

[9] X. Li and G. Agrawal, "Efficient evaluation of xquery over streaming data," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 265–276.

[10] S. Urban, S. Dietrich, and Y. Chen, "An xml framework for integrating continuous queries, composite event detection, and database condition monitoring for multiple data streams," in *Event Processing*, ser. Dagstuhl Seminar Proceedings, M. Chandy, O. Etzion, and R. von Ammon, Eds., no. 07191. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/1142

[11] M. Franklin, A. Halevy, and D. Maier, "From databases to dataspaces: a new abstraction for information management," *SIGMOD Rec.*, vol. 34, no. 4, pp. 27–33, 2005.

[12] M. Lenzerini, "Data integration: a theoretical perspective," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ser. PODS '02. New York, NY, USA: ACM, 2002, pp. 233–246. [Online]. Available: http://doi.acm.org/10.1145/543613.543644

[13] J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules For Advanced Database Processing*. San Francisco, Calif.: Morgan Kaufmann, 1996, edited by Jennifer Widom, Stefano Ceri.; :ill. ;24 cm; Includes bibliographical references (p. 303-324) and index.

[14] A. Gupta and I. S. Mumick, "Materialized views," in *Materialized views: Techniques, Implementations, and Applications*, A. Gupta and I. S. Mumick, Eds. Cambridge, MA, USA: MIT Press, 1999, ch. Maintenance of materialized views: problems, techniques, and applications, pp. 145–157. [Online]. Available: http://portal.acm.org/citation.cfm?id=310709.310737

[15] A. Gupta and I. S. Mumick, Eds., *Materialized views: Techniques, Implementations, and Applications*. Cambridge, MA, USA: MIT Press, 1999.

[16] H. V. Jagadish, I. S. Mumick, and A. Silberschatz, "View maintenance issues for the chronicle data model (extended abstract)," in *PODS '95: Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM, 1995, pp. 113–124.

[17] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," in *Proceedings of the Eleventh International Conference on Data Engineering*, ser. ICDE '95. Washington, DC,

USA: IEEE Computer Society, 1995, pp. 190–200. [Online]. Available: http://portal.acm.org/citation.cfm?id=645480.655434

[18] M. El-Sayed, E. A. Rundensteiner, and M. Mani, "Incremental maintenance of materialized xquery views," in *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*. Washington, DC, USA: IEEE Computer Society, 2006, p. 129.

[19] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan, "Incremental maintenance of path-expression views," in *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2005, pp. 443–454.

[20] A. Sundermier, "Condition monitoring in an active deductive object-oriented database." Master's Thesis, Arizona State University, 1999.

[21] M. Chaudhari, "A distributed event stream processing framework for materialized views over heterogeneous data sources," in *VLDB 2010 Ph.D. Workshop*, 13-17 September 2010, pp. 13–18.

[22] Oracle Corporation, "Oracle streams - features overview," 2009. [Online]. Available: http://oracle.com/technology/products/dataint/htdocs/streams_fo.html

[23] Microsoft Corporation, "Tracking data changes: Sql server 2008 books online," 2009. [Online]. Available: http://msdn.microsoft.com.ezproxy1.lib.asu.edu/en-us/library/bb933994.aspx

[24] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Rec.*, vol. 34, pp. 42–47, December 2005. [Online]. Available: http://doi.acm.org/10.1145/1107499.1107504

[25] Sybase, "Complex event processing," http://www.sybase.com/products/financialservicessolutions/complex-event-processing, September 2010. [Online]. Available: http://www.sybase.com/products/financialservicessolutions/complex-event-processing

[26] S. W. Dietrich, "Maintenance of recursive views," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer US, 2009, pp. 1674–1679.

[27] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk, "Querying xml views of relational data," in *VLDB '01: Proceedings of the 27th*

International Conference on Very Large Data Bases. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 261–270.

[28] L. Wang, E. A. Rundensteiner, and M. Mani, "Updating xml views published over relational databases: towards the existence of a correct update mapping," *Data Knowl. Eng.*, vol. 58, pp. 263–298, September 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1162457.1162461

[29] E. Leonardi and S. S. Bhowmick, "Detecting changes on unordered xml documents using relational databases: a schema-conscious approach," in *Proceedings of the 14th ACM international conference on Information and knowledge management*, ser. CIKM '05. New York, NY, USA: ACM, 2005, pp. 509–516. [Online]. Available: http://doi.acm.org/10.1145/1099554.1099693

[30] Y. Wang, D. J. DeWitt, and J. Y. Cai, "X-diff: an effective change detection algorithm for xml documents," in *Proceedings.19th International Conference on Data Engineering, 2003.*, 2003, pp. 519–530.

[31] F.-C. F. Chen and M. H. Dunham, "Common subexpression processing in multiple-query processing," *IEEE Transactions on Knowledge and Data Engineering.*, vol. 10, no. 3, pp. 493–499, 1998.

[32] J. Park and A. Segev, "Using common subexpressions to optimize multiple queries," in *Proceedings of the Fourth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1988, pp. 311–319.

[33] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," *SIGMOD Rec.*, vol. 29, pp. 249–260, May 2000. [Online]. Available: http://doi.acm.org/10.1145/335191.335419

[34] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner, "Efficient exploitation of similar subexpressions for query processing," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 533–544. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247540

[35] G. Cugola and A. Margara, "Tesla: a formally defined event specification language," in *DEBS*, J. Bacon, P. R. Pietzuch, J. Sventek, and U. Çetintemel, Eds. ACM, 2010, pp. 50–61.

[36] F. Bry and M. Eckert, "Rule-based composite event queries: The language xchangeeq and its semantics," in *Web Reasoning and Rule Systems*, ser. Lecture

Notes in Computer Science, M. Marchiori, J. Pan, and C. Marie, Eds. Springer Berlin Heidelberg, 2007, vol. 4524, pp. 16–30.

[37] StreamBase Systems, "Streamsql," 2008. [Online]. Available: http://streambase.com/products-streamsql.htm

[38] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 407–418.

[39] I. Elsayed, P. Brezany, and A. M. Tjoa, "Towards realization of dataspaces," in *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 266–272.

[40] A. Halevy, M. Franklin, and D. Maier, "Principles of dataspace systems," in *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM, 2006, pp. 1–9.

[41] Y.-W. Wang and E. N. Hanson, "A performance comparison of the rete and treat algorithms for testing database rule conditions," in *Proceedings of the Eighth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1992, pp. 88–97.

[42] Y. Diao, D. Florescu, D. Kossmann, M. J. Carey, and M. J. Franklin, "Implementing memoization in a streaming xquery processor," in *XSym*, ser. Lecture Notes in Computer Science, Z. Bellahsene, T. Milo, M. Rys, D. Suciu, and R. Unland, Eds., vol. 3186. Springer, 2004, pp. 35–50.

[43] J. A. Blakeley, P.-A. Larson, and F. W. Tompa, "Efficiently updating materialized views," *SIGMOD Rec.*, vol. 15, pp. 61–71, June 1986. [Online]. Available: http://doi.acm.org/10.1145/16856.16861

[44] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," *SIGMOD Rec.*, vol. 22, pp. 157–166, June 1993. [Online]. Available: http://doi.acm.org/10.1145/170036.170066

[45] X. Qian and G. Wiederhold, "Incremental recomputation of active relational expressions," *IEEE Trans. on Knowl. and Data Eng.*, vol. 3, pp. 337–341, September 1991. [Online]. Available: http://dx.doi.org/10.1109/69.91063

[46] S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance," in *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 577–589.

[47] J. V. Harrison, "Condition monitoring in an active deductive database." Ph.D. dissertation, Arizona State University, 1992.

[48] M. El-Sayed, E. A. Rundensteiner, and M. Mani, "Incremental fusion of xml fragments through semantic identifiers," in *IDEAS '05: Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 369–378.

[49] S. W. Dietrich and M. Chaudhari, "The missing linq between databases and object-oriented programming: Linq as an object query language for a database course," *Journal of Computing Sciences in Colleges*, vol. 24, pp. 282–288, April 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1516546.1516592

[50] Transaction Processing Performance Council, "Tpc-h," 2011. [Online]. Available: http://www.tpc.org/tpch//

[51] C. Calvert and D. Kulkarni, *Essential LINQ*. Boston, MA: Addison-Wesley, 2009.

[52] Dot Com Infoway, "Dot com infoway," http://www.dotcominfoway.com/blog/linq-to-sql-vs-ado-net-entity-framework, September 2010. [Online]. Available: http://www.dotcominfoway.com/blog/linq-to-sql-vs-ado-net-entity-framework

[53] Microsoft Corporation, "Xml schema definition tool (xsd.exe)," 2011. [Online]. Available: http://msdn.microsoft.com/en-us/library/x6c1kb0s(v=vs.71).aspx

[54] J. Löwy, *Programming WCF Services*. O'Reilly Media, Inc., 2007.

[55] Oracle b Corporation, "Core j2ee patterns - data access object," 2010. [Online]. Available: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

[56] Oracle a Corporation, "Core j2ee patterns - transfer object," 2010. [Online]. Available: http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html

[57] T. K. Sellis, "Multiple-query optimization," *ACM Transactions on Database Systems*, vol. 13, no. 1, pp. 23–52, 1988.

[58] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, pp. 422–469, December 2000. [Online]. Available: http://doi.acm.org/10.1145/371578.371598

[59] U. S. Chakravarthy and J. Minker, "Multiple query processing in deductive databases using query graphs," in *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 384–391.

[60] M. Jarke, "Common subexpression isolation in multiple query optimization," in *Query Processing in Database Systems*. Springer, 1985, pp. 191–205.

[61] A. Weiner, T. Harder, and R. Oliveira da Silva, "Visualizing cost-based xquery optimization," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, march 2010, pp. 1165 –1168.

[62] V. Vizing, "On incidentor coloring in a partially directed multigraph," *Journal of Applied and Industrial Mathematics*, vol. 3, pp. 297–300, 2009, 10.1134/S1990478909020161. [Online]. Available: http://dx.doi.org/10.1134/S1990478909020161

[63] W3C, "Xquery update facility 1.0," http://www.w3.org/TR/2011/REC-xquery-update-10-20110317/, March 2011. [Online]. Available: http://www.w3.org/TR/2011/REC-xquery-update-10-20110317/

[64] D. T. Liu, "Oracle database 10g: Implement streams," 2007. [Online]. Available: http://www.nocoug.org/download/2007-08/NoCOUG_Aug_16_2007_Presentation_Oracle_Streams.pdf

[65] L. R. Cunningham, "Oracle streams: Step by step," 2008. [Online]. Available: http://filedb.experts-exchange.com/incoming/2008/11_w48/81772/Oracle-Streams-Step-by-Step.pdf

[66] K.-H. Lee, Y.-C. Choy, and S.-B. Cho, "An efficient algorithm to compute differences between structured documents," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 8, pp. 965–979, 2004, member-Kyong-Ho Lee and Member-Yoon-Chul Choy and Member-Sung-Bae Cho.

[67] G. Cobena, S. Abiteboul, and A. Marian, "Detecting changes in xml documents," in *18th International Conference on Data Engineering, 2002.*, 2002, pp. 41–52.

[68] F. Yuan, "Materialized view maintenance for xml documents," Master's Thesis, National University of Singapore, 2004.

[69] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic sets and other strange ways to implement logic programs (extended abstract)," in *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, ser. PODS '86. New York, NY, USA: ACM, 1986, pp. 1–15. [Online]. Available: http://doi.acm.org/10.1145/6012.15399

[70] C. Beeri and R. Ramakrishnan, "On the power of magic," *J. Log. Program.*, vol. 10, pp. 255–299, March 1991. [Online]. Available: http://portal.acm.org/citation.cfm?id=114549.114552

[71] R. Ramakrishnan, "Magic templates: a spellbinding approach to logic programs," *The Journal of Logic Programming*, vol. 11, no. 3-4, pp. 189 – 216, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0J-45G0HB9-1/2/7f576f6c76aabe55a9f947e22ac90bc2

[72] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan, "Magic is relevant," *SIGMOD Rec.*, vol. 19, pp. 247–258, May 1990. [Online]. Available: http://doi.acm.org/10.1145/93605.98734

[73] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan, "The magic of duplicates and aggregates," in *Proceedings of the 16th International Conference on Very Large Data Bases*, ser. VLDB '90. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 264–277. [Online]. Available: http://portal.acm.org/citation.cfm?id=645916.671834

[74] I. S. Mumick, "Query optimization in deductive and relational databases," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1991, uMI Order No. GAX92-17860.

[75] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan, "Adornments in database programs," *Computers and Artificial Intelligence*, vol. 13, pp. 201–231, 1994.

[76] J. D. Ullman, *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. New York, NY, USA: W. H. Freeman & Co., 1990.

[77] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan, "Magic conditions," in *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, ser. PODS '90. New York, NY, USA: ACM, 1990, pp. 314–330. [Online]. Available: http://doi.acm.org/10.1145/298514.298584

[78] A. Gupta and I. S. Mumick, "Magic-sets transformation in nonrecursive systems," in *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, ser. PODS '92. New York, NY, USA: ACM, 1992, pp. 354–367. [Online]. Available: http://doi.acm.org/10.1145/137097.137908

[79] I. S. Mumick and H. Pirahesh, "Implementation of magic-sets in a relational database system," *SIGMOD Rec.*, vol. 23, pp. 103–114, May 1994. [Online]. Available: http://doi.acm.org/10.1145/191843.191860

[80] H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/rule based query rewrite optimization in starburst," *SIGMOD Rec.*, vol. 21, pp. 39–48, June 1992. [Online]. Available: http://doi.acm.org/10.1145/141484.130294

[81] J. Almendros-Jimnez, A. Becerra-Tern, and F. Enciso-Baos, "Magic sets for the xpath language," *Journal of Universal Computer Science*, vol. 12, no. 11, pp. 1651–1678, 2006, http://www.jucs.org/jucs_12_11/magic_sets_for_the.

[82] F. Özcan, N. Seemann, and L. Wang, "Xquery rewrite optimization in ibm db2 purexml," *IEEE Data Eng. Bull.*, vol. 31, no. 4, pp. 25–32, 2008.

[83] GJXDM, "Global justice xml data model," http://www.it.ojp.gov/jxdm/, November 2007. [Online]. Available: http://www.it.ojp.gov/jxdm/

[84] M. Chaudhari and S. Dietrich, "Metadata services for distributed event stream processing agents," in *19th International Conference on Software Engineering and Data Engineering*, San Francisco, 16-18 June 2010, pp. 307–312.