ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

OPTIMIZATION OF SEMANTIC NEWS FEED QUERY

By

ANDINET ASSEFA BEKELE

A Thesis Submitted to the School of Graduate Studies of Addis Ababa University in Partial Fulfillment for the Degree of Master of Science in Computer Science

April, 2013

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

COLLEGE OF NATURAL SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

OPTIMIZATION OF SEMANTIC NEWS FEED QUERY

By

ANDINET ASSEFA BEKELE

Approved By

MEMBERS OF THE EXAMINATION BOARD:

1. Fekade Getahun (Ph.D.), Advisor ________________________
2. Solomon Atnafu (Ph.D.), Examiner ________________________
3. ____________________________ ________________________

DEDICATION

In Memory of My Beloved Father

Able Seaman Assefa Bekele

who was Kind and Generous

To My Beloved Mother Yeshi Sime

For Her Motivation and Encouragement

To my Academic Success

ACKNOWLEDGEMENTS

First of all, I would like to thank the Lord God for He hides me in the secret of His tabernacle in the time of trouble and set me up upon a rock. Great praise be to the Lord for He has quickened me together with Christ even when I was dead in sins and saved me with grace.

Next, I would like to express my deepest gratitude to my advisor Dr. Fekade Getahun for his motivation and guidance right from the moment of problem formulation to the completion of this research work. Many thanks and appreciations go to him for the discussions with him always made me think that things are possible. His enthusiasm and encouragement has always inspired me to the fruitful completion of this research work.

I am also very thankful to my instructors and all staffs members of the department of Computer Science for their contribution in one way or another for the success of my study and great thank to Addis Ababa University for the sponsorship granted to present and publish part of this research work on the 4th *International Conference on Management of Emergent Digital Ecosystems* (MEDES`12) prepared by ACM in cooperation with Addis Ababa University IT Doctoral Program on October 28-31, 2012.

Finally, I am very grateful to my mother Yeshi Sime who played great role to a success in my academic endeavor especially in my early childhood. And I thank my brothers Misikir, Minwuyelih, Lijalem and Lawugalih as well as my sisters Selam Assefa and Lidia Lemma for their motivation and encouragement has always inspired me in my academic life.

TABLE OF CONTENTS

TABLE OF CONTENTS .... **I**

LIST OF TABLES .... **III**

LIST OF FIGURES .... **IV**

LIST OF ALGORITHMS .... **V**

LIST OF ACRONYMS .... **VI**

ABSTRACT .... **VII**

1 INTRODUCTION **1**

1.1. Background 1

1.2. Motivation 2

1.3. Statement of the Problem 4

1.4. Objective 5

1.5. Methodology 5

1.6. Significance of the Study 6

1.7. Roadmap 7

2 LITERATURE REVIEW **8**

2.1 Introduction 8

2.2 RSS News Feed 8

2.3 Semantic News Feed Query and RSS Operators 10

2.4 Data Stream Management System 11

2.5 Publish/Subscribe Systems 13

2.6 Query Optimization in DSMS 14

3 RELATED WORKS **18**

3.1 Introduction 18

3.2 Multi-Query Optimization for Stream Queries 18

3.3 Load Shedding 19

4 SEMANTIC NEWS FEED QUERY PROCESSING **22**

4.1 Introduction 22

4.2 System Architecture 23

4.2.1 Static Query Optimizer 26
4.2.2 Execution Manager 38
4.2.3 Run Time Optimizer 39
4.2.4 Multi-Query Executer 46
4.3 Database Schema Design 47
4.3.1 Profile Database 47
4.3.2 Knowledge Base 48
4.4 Summary 50
5 IMPLEMENTATION AND EXPERIMENTAL RESULTS 51
5.1 Introduction 51
5.2 User Interfaces 52
5.3 Experimental Results 55
5.3.1 Network Access Time (NAT) 55
5.3.2 MQE of SNF Queries 56
5.3.3 Load Shedding 58
6 CONCULUSION AND FUTURE WORKS 60
6.1 Conclusion 60
6.2 Future Works 61
7 REFERENCES 63
8 APPENDIX 67
Sample Microsoft C# Codes Taken From Prototype Development 67
Publication 70

LIST OF TABLES

Table 1-1: List of semantic news feed queries registered by users 3

Table 2-1: Summary of semantic aware RSS operators...11

Table 4-1: Vector generated when comparing two SNF queries...29

Table 4-2: Primitive relationships between time intervals [32]...30

Table 4-3: Windows Relations for of queries identified using WinRelation algorithm 34

LIST OF FIGURES

Figure 2-1: Sample RSS News Feed 9

Figure 2-2: Architecture for data stream management system 12

Figure 2-3: High level Architecture for Multi-Query optimizer 16

Figure 4-1: Architecture of the Semantic News Feed Query Processing Engine 25

Figure 4-2: Query relationship and corresponding threshold 28

Figure 4-3: Equality threshold and its corresponding *minTR* and *maxTR* 31

Figure 4-4: Topology for window relationship type 32

Figure 4-5: Sample Execution table/schedule generated by *Execution Manager* 39

Figure 4-6: Non-overlapping sliding window with variable size 41

Figure 4-7: QuickDrop operator placement in Multi-Query Execution plan 46

Figure 4-8: Logical database model of profile database for SNF query engine 49

Figure 4-9: Logical database model of profile database for SNF query engine 50

Figure 5-1: User preferences subscription interface 52

Figure 5-2: SNF query definition interface 53

Figure 5-3: Query scheduling interface 54

Figure 5-4: Query execution status displaying interface 54

Figure 5-5: Interface for displaying query execution result 55

Figure 5-6: Graph for experimental result of NAT 57

Figure 5-7: Graph for MQE timing of queries based on window relationship type 57

Figure 5-8: Experimental result for query execution before and after load shedding 59

Figure 5-9: Experimental result for query execution with different precision level 59

LIST OF ALGORITHMS

Pseudo Code 4-1: WinRelation Identifier Algorithm 33

Pseudo Code 4-2: Chain Generator Algorithm..35

Pseudo Code 4-3: Algorithm for *quickDrop* Operator..44

LIST OF ACRONYMS

MQO	Multi-Query Optimization
MQE	Multi-Query Execution
SNF	Semantic News Feed
NAT	Network Access Time
DSMS	Data Stream Management System
RSS	Really Simple Syndication
RDF	Resource Description Framework
DBMS	Database Management System
ACQ	Aggregate Continuous Queries
LWF	Larger Window First
SWF	Smaller Windows First
QOS	Quality of Service
LCS	Lowest Common Subsume

ABSTRACT

RSS feeds are text-content rich, semantically heterogeneous, dynamic XML element and streamed in an asynchronous and pull strategy. Hence, for efficient retrieval of RSS feed, window-based and semantic-aware feeds querying operators has been proposed recently. It is shown that the use of semantic information improves the relevance of query result at the cost of degrading the efficiency of the system. To benefit from query execution on semantic information while keeping the efficiency of the system, we have proposed multi-query optimization strategy and semantic load shedding technique. The proposed MQO strategy accepts multiple semantic news feed queries and examines the relationship that exist between each queries with respect to their defined window and their semantic similarity. Then, a MQE chain is generated based on the relationship examined for the efficient manipulation of queries at run time. And each time a new query arrives, it is added to its appropriate chain. On the other hand, the proposed semantic load shedding technique drops non-relevant data elements from a shared window and generates a new window with optimal size. The approach first extracts a set of representative keys from multiple query's search term and then the keys are enhanced semantically with concepts retrieved from knowledge base. Then before Multi-Query execution, using the enhanced key set, the news items in the shared window are classified into relevant and non-relevant classes based on their key content. Those elements in the relevant class is directly chosen to be a member of the new window and among the elements in the non-relevant class a representative sample is selected at random with a controlled sampling precision level to be included into the new window. The resulting new window contains reduced data elements and yet satisfies the need of all queries in the given MQE chain. The approach is implemented as an operator called quickDrop operator. The operator `quickDrop (W, T, K, e)` *takes four parameters;* `W` *denotes a shared window among a set of queries for Multi-Query execution,* `T` *denotes the attribute of the news feed for shedding i.e. title or description,* `K` *denotes a set of enhanced keys and* `e` *denotes the level of precision to be used for sampling and it returns a window with optimal number of elements. Coordinated with the MQO approach, the proposed semantic load shedding technique performs efficiently for the performance increase of the system. To validate the proposed approach, we have developed a prototype and experimental results show a significant performance increase.*

Key Words: SNF Query, NAT, Windows Boundary Similarity Measure, MQE Chain Generator, Window Relationship, MQE Rules, Semantic Load Shedding, QuickDrop Operator.

CHAPTER ONE

1 INTRODUCTION

1.1. Background

RSS (originally RDF Site Summary, often dubbed *Really Simple Syndication*) is a family of web feed formats used to publish frequently updated information such as blog entries, news headlines, audio, and video in a standardized format. An RSS document (which is called a "news feed", "web feed", or "channel") includes summarized text, plus metadata such as publishing dates and authorship. RSS feed is formatted in XML and benefit publishers to syndicate content automatically, and allow clients to subscribe to timely updates from favorite providers rather than roaming from site to site.

RSS feeds can be read using software called an "RSS reader", "feed reader", or "aggregator", which can be web-based, desktop-based, or mobile-device-based. The RSS reader checks the user's subscribed feeds regularly for new work, downloads any updates that it finds, and provides a user interface to monitor and read the feeds. In addition, it allows users to avoid manually inspecting all of the websites they are interested in, and instead subscribe to websites such that all new content is pushed onto their browsers when it becomes available [1].

RSS feeds are used to distribute dynamic data and are integrated as part of the new web applications such as web blogs and content sharing applications. In addition, feeds have a streaming nature as compared to other web documents since feed providers send dynamic information when available. Hence, the content is available immediately to the feed reader and also to feed search engines. In contrast, web documents are only accessible to public once after it is found by a crawler and indexed by search engines. Moreover, RSS news items are text-content rich, semantically heterogeneous, dynamic XML document and their contents are dependent on the authors' style of writing.

Taking the behavior of RSS feeds into consideration, for efficient retrieval of news feed, a system with a functionality based on concept similarity is required. In [2], semantic-aware and window-based RSS querying operators have been proposed to retrieve set of news items satisfying a given condition while considering semantic information. It is known that semantic based query

processing strategy improves the relevance of query result. However, this process degrades the efficiency of the system highly. As we have already mentioned that web feeds are dynamic and its content changes through time and have streaming nature. And processing long running persistent user queries over this dynamic stream of information makes the performance of the system even worst. Thus, to benefit from semantic based query processing while keeping the efficiency of the system, query optimizer should be designed. Briefly, the optimizer should consider the system resource sharing opportunities of queries, similarity of queries, and it should also adaptively decide on the arrival rate of data to the system.

1.2. Motivation

To motivate our work we have presented two scenarios concerning semantic news feed query optimization problem. The first scenario deals with the need to use Multi-Query optimization approach and the second deals with the need to gracefully decrease non-relevant data and only processing user queries on most relevant data to increase the performance of the system.

Scenario: [Multi-Query Optimization and Load Shedding]

EasyRSSManager, which is RSS management software with a semantic-aware RSS Reader having a window-based RSS query components developed by Fekade G. and Richard C., is installed on a proxy server to process users' queries [2]. Table 1-1 shows some of the registered user queries. Each query is scheduled to be executed at a particular time on a given source. Queries Q_1 and Q_2 are semantically similar and are defined on the same source even if they are defined on relatively different window[1]boundary. Even though, query Q_3 is neither similar to Q_1 nor to Q_2, both share the same feed source. Similarly, queries Q_4 and Q_6 are defined on the same feed source. Thus, rather than executing these queries independently, it is advisable to execute those queries that share the same resource (feed source and window) simultaneously to increases the efficiency of query processing system.

Analyzing the relationship between windows, windows defined for queries Q_1 and Q_2 are included in the window defined for query Q_3. What if we execute the queries only on one window? Doing so reduces network access time and resource/memory utilization. This is also the

[1]Window: it is a collection of elements defined on a given feed source having start and end boundary time.

case for queries Q_4 and Q_6. In addition, queries Q_1 and Q_2 are semantically similar (i.e., similar *Search Values, Attributes,* and *Operators*) what if we execute only for Q_1 and apply the result for query Q_2 too?

This scenario shows the need to execute several queries having same feed source on a single shared window and identifying semantically similar queries helps for saving computation cost and for the betterment of the performance of the system in general.

In addition, query processing on semantic information degrades the performance of the system and it is worst as the number of queries increases and when the data arrival rate is high. Hence, in addition MQO solution, the optimal performance is still dependant on the data load arriving continuously to the system. In streaming applications, sometimes the data arrival rate increase and exceed the system‘s processing capacity. So the immediate solution is to drop some of the load gracefully to get rid of load congestion. We know that all the data arriving to the system are not equally relevant to the result of query execution. Thus, to get an optimal performance, employing a load shedding technique to drop non-relevant data from the incoming stream is trivial.

Table 1-1: List of semantic news feed queries registered by users

Query	**Search Value**	**Attribute**	**RSS Operator**	**Feed Source**	**Window Boundary Time(In GMT)**	
					Start	**End**
Q_1	Car crush killed two people in Iran	Title	Select	BBC	01:30:10	18:30:01
Q_2	Dead in Tehran car accident	Description	Select	BBC	00:30:30	15:00:00
Q_3	Euro 2012 football latest score	Description	TopK	BBC	00:05:30	19:05:00
Q_4	Iran nuclear programme	Title	Join	REUTERS, FRANCE 24	00:00:05	16:00:30
Q_5	South Sudan border town Bentiu bombed	Description	TopK	CNN	01:00:00	18:05:00
Q_6	US president surprise world bank nominee	Title	Select	REUTERS	03:10:00	15:30:10

Motivated by the above observation, this research work aims at finding query optimization approach which helps to increase the performance of the system keeping the relevance of query processing result.

1.3. Statement of the Problem

Liu et al. stated that RSS news items are dynamic and the update rate of RSS feeds is widely distributed and 55% of RSS feeds are updated hourly [3]. In addition, they are text-content rich and semantically heterogeneous. Hence, to improve the relevance of the result of query processing on news feed, semantic aware retrieval functionality is required. In [2], Fekade G. and Richard C. have proposed window-based and semantic-aware feeds querying operators to retrieve the set of news items satisfying a given condition. It is known that query optimization helps the system to evaluate queries efficiently with best execution plan possibly among a set of other plans. Therefore, the authors also have stated some equivalent rules that would be used by query optimizer and have applied heuristic-based query optimization strategies that make use of the equivalence rules to transform a given algebraic query to less costly equivalent one. However, other than the query rewriting strategy stated by the researchers, the design of query optimizer should consider a number of issues which are not considered by the authors. Therefore, in this research work, we have raised the following points which are not addressed by the researchers but yet crucial to improve the efficiency of the query processing system:

- *Optimal window size*: a window is a set of data elements having a start and end boundary taken from a continuous stream of data posted from a particular source in a particular date and time. And since the contents of web feeds are dynamic and the arrival rate is not known appriori, the size of window is not known until the required data arrives at the system. Therefore, at the time when the data arrival rate is high and hence the window size increases, processing a query on semantic information is costly. We know that all the data arriving are not equally relevant to the query processing result and dropping non-relevant data decreases the size of the window. Thus, optimal window size should have to be decided by the optimizer to perform cost effective query optimization.

- *Resource sharing among a set of queries:* as there may be a chance for queries to share system resources such as CPU and memory, the query optimizer should use such opportunities to execute a set of queries on a shared resources.

- *Query similarity:* the optimizer should consider similar queries to avoid redundant execution.

The query optimizer in the query engine should consider the above issues to come up with efficient query execution plan that minimizes system resource utilization and maximizes the performance of the system.

1.4. Objective

The general objective of the study is to design query optimizer for semantic news feed query that consider the optimal window size, resource sharing among queries and similarity of queries before the actual query execution.

In light of this general theme, the specific objectives of the research are the following:

- Studying and analyzing the optimization techniques for semantic based news feed query,

- Designing a query optimizer for semantic based news feed query,

- Finding the optimum window size for efficient query execution,

- Measuring the similarity of queries,

- Developing a prototype to realize the proposed approach.

1.5. Methodology

To achieve the objectives of this research work, different literature reviews and programming tools are employed.

1.5.1. Literature Review

Different literatures that are considered to be relevant for the research is reviewed and adopted for this research work. Since the research work is on optimization of semantic based news feed query, relevant literature on the same issue is reviewed. Furthermore, the query optimization strategies stated in [2] will be revised to understand the approach and to look for further improvements in optimization of semantic based news feed query.

1.5.2. Analysis and Design

After revising relevant research works about optimization of news feed query and detailed analysis, we have designed query optimizer. If any, relevant open source software will be customized and used for the implementation of the system. An experiment will be conducted to evaluate the performance and efficiency of the system. Finally, the conclusions and recommendations will be drawn from the evaluation results.

1.5.3. Development Environment

Prototype will be developed to show the applicability of the approach of the research work practically. Hence, for development, we have employed the following development environments:

- Microsoft C# is employed to develop the prototype.
- WordNet 2.1 Knowledge base stored in MySQL database.
- MySQL 5.5 relational database

1.6. Significance of the Study

The cost difference between two alternatives of query execution can be enormous. Without query optimization, the query processor can choose a plan with a high cost of execution. As stated in the problem statement section, the use of semantic information in executing the user query improves the relevance of query result at the cost of degrading the system efficiency. Hence, to keep the balance between efficiency of query processing on semantic information and query processing speed, optimization is trivial. Accordingly, the result of the study brings optimization approach that enables to find efficient and equivalent query execution plan which considers the semantic

similarity of SNF queries and decide on the optimal window size so as to improve the query processing speed of the system.

1.7. Roadmap

The rest of this thesis report is organized as follows. Chapter 2 reviews concepts relevant to the realm of the proposed approach. Chapter 3 presents related works. Chapter 4 presents the proposed semantic news feed query processing emphasizing on the proposed optimization approach. Prototype development and experimental results are presented in Chapter 5. Finally, Chapter 6 concludes the overall work presented in this research work and draws future directions.

CHAPTER TWO

2 LITERATURE REVIEW

2.1 Introduction

This chapter presents a review of concepts relevant to obtain basic understanding of the ideas of the proposed research work and related research works towards optimization of semantic news feed query. The review presented in this sub-section deal on concepts important to the realm of the proposed approach. Specifically concepts on fundamentals of RSS News Feed, Semantic News Feed (SNF) Query and Semantic Aware RSS Operators, Data Stream Management System/DSMS, Publish/Subscribe systems, and optimization issues in DSMS are presented in detail.

2.2 RSS News Feed

RSS (originally RDF Site Summary, often dubbed *Really Simple Syndication*) is a family of web feed formats used to publish frequently updated information such as blog entries, news headlines, audio, and video in a standardized format. An RSS document (which is called a "news feed", "web feed", or "channel") includes summarized text, plus metadata such as title, description, publishing dates and authorship. RSS feed is formatted in XML and benefit publishers to syndicate content automatically, and allow clients to subscribe to timely updates from favorite providers rather than roaming from site to site.

RSS feeds can be read using software called an "RSS reader", "feed reader", or "aggregator", which can be web-based, desktop-based, or mobile-device-based. The RSS reader checks the user's subscribed feeds regularly for new work, downloads any updates that it finds, and provides a user interface to monitor and read the feeds. In addition, it allows users to avoid manually inspecting all of the websites they are interested in, and instead subscribe to websites such that all new content is pushed onto their browsers when it becomes available [1].

RSS document contains a list of *items* or *entries*, each of which is identified by a link. Each item can have any amount of other metadata associated with it as well. The most basic metadata for an

entry in ATOM includes a title for the link and a description of it; when syndicating news headlines, these fields might be used for the story title and the first paragraph or a summary, for example. For example, Figure 2-1 shows sample RSS 2.0 news feed.

RSS document has various structures depending on the given version of RSS. There are two well-known formats of RSS; RSS 2.0 and Atom 1.0. These formats have different structure on how to tag a given feed item. A particular news feed in RSS 2.0 has elements such as *title*, *description*, *link*, and *publication date* [3].

```
<?xml version="1.0"?>
<rss version="2.0">
<channel>
<title> BBC Channel</title>
<link>http://example.com/</link>
<description>Example BBC channel</description>
<item>
        <title>London Olympic </title>
        <link>http://BBC/rss/sport</link>
        <description>Issues onthe start of London Olympic </description>
</item>
         <item>
         <title>Iran nuclear programme</title>
         <link>http://BBC/rss/worldnews/</link>
         <description>Issues on Iran nuclear programme</description>
         </item>
</channel>
</rss>
```

Figure 2-1: Sample RSS News Feed

News feed is proposed to assist the aggregation of distributed and dynamic information. Since the content is in XML format, software known as RSS/feed readers/aggregators (which can be either web-based application e.g., Google Reader, client-oriented e.g., Microsoft Office outlook, or plug-in to Web Browser) allow a user to subscribe, read, and access feed content originating from different providers in a place rather than roaming site to site. The RSS reader checks the user's subscribed feeds regularly for new work, downloads any updates that it finds, and provides a user

interface to monitor and read the feeds [1]. In addition, it allows users to avoid manually inspecting all of the websites they are interested in, and instead subscribe to websites such that all new content is pushed onto their browsers when it becomes available. It can be also integrated easily with web applications such as blogs and content sharing applications. News feed has streaming nature which allows publishers to distribute dynamic contents and updates to feed readers. Hence, news feed is getting more attention recently.

2.3 Semantic News Feed Query and RSS Operators

Semantic news feed (SNF) query [2] is a persistence query which is intended to be processed on semantic information. A particular semantic news feed query Q have attribute A, RSS operator θ (*extraction*, *set membership* or *merge*) and search term T. The attribute refers to the news feed‘s element i.e. title, description, link or publication date. The RSS operators are categorized under extraction set membership or merge operators. The extraction operators are dedicated to retrieve data from the stream and it includes *Selection*, its extension *TopK*, and *Join*. Each of these operators accepts a set of windows and a supportive parameter set. The set membership operators are binary and accepts two windows defined on streams. The *Merge* operator accepts two windows and returns the result of merging (i.e. putting them together in a given pattern) w.r.t. the merging rule.

Queries in data stream management are relatively permanent and are stored in profile database as part of the user’s preferences. Hence, as a persistent query, SNF queries are stored in the profile database and are executed when a new stream data is available.

Given Query Q, Q.S, Q.P, and Q.θ returns source, predicate and operator associated respectively to the query. Table 2-1 summarizes semantic aware RSS algebraic operators proposed in [2].

Table 2-1: Summary of semantic aware RSS operators

Symbol	Meaning	Operator Category
$\boldsymbol{\delta_p}$	Similarity Selection	Extraction
$\rhd\lhd$	Similarity Join	
$\boldsymbol{\delta_p^k}$	TopK	
$\cup$	Union	Set Membership
$\cap$	Intersection	
$\setminus$	Difference	
$\Cup$	Additive Union	
$\Cap$	Additive Intersection	
$\oplus$	Merge	Merging
$\overleftrightarrow{\ominus}$	Symmetric Operator	-

2.4 Data Stream Management System

Database management systems (DBMS) are designed to solve business application problems in which data are stored relatively permanent in database and altered when user initiates transactions. The DBMS is suitably designed to process dynamic queries on stored data, which seldom changes. However, DBMS fail to fulfill the need of applications which need processing of persistent queries against continuous, real-time stream of data. To fulfill the needs of such applications, there comes a data stream management system (DSMS).

Data stream management systems processes persistent queries on a time dependent stream data. The source of the stream sends data in either asynchronous or synchronous manner using a push or pull strategy and the query processer evaluate the query continuously on the incoming stream of data. In traditional DBMSs queries are one-time queries, i.e., a query is evaluated once and the result at that point in time is returned to the user. In contrast this, queries in data stream management system (DSMS) are persistent which requires that once a query is registered then it should be continuously evaluated until it is removed from the system. On the other hand, the data

in traditional DBMS is persistent whereas the data in streaming application is dynamic and time dependent [3].

Figure 2-2 shows the general architecture of data stream management system. From the figure, user queries are relatively permanent and registered as part of user preference in database. The data stream management system processes user queries continuously on the incoming stream of data coming from stream sources and produces a stream of results as output.

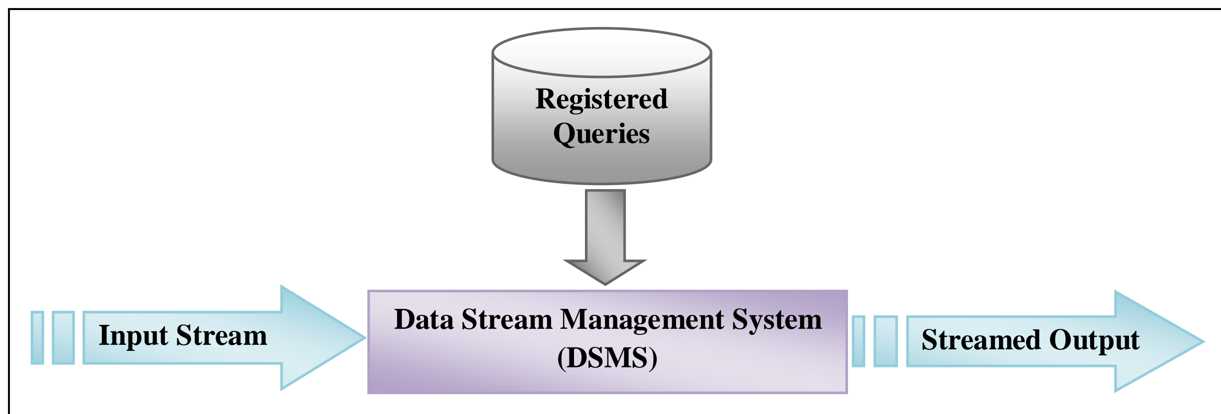

Figure 2-2: Architecture for data stream management system

Since the incoming data streams are unbounded and practically impossible to execute user query on all incoming stream of data, the system requires specifying a set of data elements that form the stream of data over which the query should be computed via *windows*. A window is a collection of elements defined on a given stream data and having a type *sliding* [27], or *tumbling*[28], and satisfies a boundary condition (i.e., the start and end condition). In sliding window, the basic idea is that rather than running computations on all of the data seen so far, or on some sample, we can make decisions based only on *recent data*. More formally, at every time t, a new data element arrives. This element expires at time $t + w$ to maintain the window size when new data element arrives, where w is the window "size" or length. Tumbling window stores incoming data elements until the window is full and then all stored data elements are removed for news set of data elements before starting over from scratch. The difference is that tumbling windows do not overlap and sliding windows overlap but never have disjoint first items.

The application of DSMS is vast and some of the streaming applications are in sensor network, road traffic analysis, performance analysis, and anomaly detection. In addition the proposed SNF

query processing system is an application of DSMS. As a data stream processing system, in SNF query processing system the user queries (SNF queries) are registered in database and are relatively permanent. The system reads queries from database and evaluates semantically on a stream of RSS feed items transmitted from RSS feed publishers. The semantic processing capacity of the system is supported by a knowledge base[2]. SNF query processing system is designed with a publish/subscribe architecture. The next section presents issues on publish/subscribe systems.

2.5 Publish/Subscribe Systems

Publish/Subscribe is a term used to define an application model in which the provider of some information is decoupled from the consumers of that information [24]. Publish/subscribe systems have become very popular in recent years as a way of distributing data messages from publishing computers to subscribing computers. Such systems are especially useful where data supplied by a publisher is constantly changing and a large number of subscribers needs to be quickly updated with the latest data. In such systems, publisher applications of data messages do not need to know the identity or location of the subscriber applications which will receive the messages. Similarly, the subscribing applications do not need to know the identity or location of the publishing application providing their information. In this sense the providers and consumers are said to be loosely-coupled.

Publish/subscribe systems are classified into two main categories [25]; subject based and content based systems. In subject-based systems, a message belongs to one of a fixed set of what are variously referred to as groups, channels, or topics. Subscription targets a group, channel, or topic, and the user receives all events that are associated with that group. Brokering a connection between publishers and subscribers is the act of connecting a channel supplier with a channel consumer, similar to the reader-writer problem in that the buffer is the communication medium.

[2]A knowledge base is a special kind of database for knowledge management. It provides a means for information to be collected, organized, shared, searched and utilized. In this research work, it is used to assist relatedness and is formally defined as KB = (C, E, R, f) where C is the set of concepts (synonym sets of words/expressions as in WordNet), E is the set of edges connecting the concepts, $E \subseteq C \times C$, R is the set of semantic relations i.e. *hyponym*, *hypernym*, *meronym*, *holonym* and *antonym* relations, f is a function designating the nature of edges in E.[4]

Content-based systems, on the other hand, are not constrained to the notion that a message must belong to a particular group. Instead, the decision of to whom a message is directed is made on a message-by-message basis based on a query or predicate issued by a subscriber. The advantage of a content-based system is its flexibility. It provides the subscriber just the information he/she needs. The subscriber need not have to learn a set of topic names and their content before subscribing.

Publish/subscribe systems differ on their architectures as well. The terms “push based” and “pull based” are also becoming increasingly popular when describing the flow of information between applications [25]. In push-based, messages are automatically broadcast to subscribers. This model provides tight consistency and stores minimal data. Pull-based models can be more responsive to user needs in that only user requested data is pulled to the subscriber.

In SNF query processing system, users publish their preferences and subscribe for queries on their sources and the system processes the queries on a stream of data coming from the news feed publishers. Thus, SNF query processing systems integrates both the DSMS and publish/subscribe systems behavior.

2.6 Query Optimization in DSMS

Given a user query Q, there may be more than one alternative query execution plans that return the same result. These plans may differ by their order of operators’ execution but yet they produce same result. A good query plan utilizes minimum system resources and maximizes the efficiency of the system. Hence, the objective of query optimization is to choose a plan which utilizes minimum system resources and maximizes system efficiency among a set of alternative query plans. To achieve this objective, the query processing engine has query optimizer component which performs the query optimization operation.

In traditional database management systems, cost based query optimization has been employed to minimize the size of intermediate results. Cost based optimization utilizes cardinality as a metrics to determine the cost of a query evaluation strategy. Cardinality is effective since the data is relatively permanent and disk bound. Hence disk I/O and cardinality is a good approximation of how much disk I/O is needed. But, for stream data cardinality is not a good metrics as it is unknown until the data is read fully and it is no use for optimization. Even if cardinality is known

at optimization time, the data is not readily available when query execution starts. Even worse, cardinality as a cost metric is not even applicable in the case of unbounded streams. The obvious argument here is that the cost of any alternative evaluation strategy, as estimated by a cardinality-based cost model, for a query over infinite streams is infinite [6]. Hence, data stream management systems need for novel optimization approaches.

Among the optimization approaches proposed in the literature, multi-query optimization [7, 8, 9, 10, 11] and load shedding [6, 12] are the main approaches devised to handle the issue of query optimization in stream system and are main concern of this research work. Next we have presented these approaches.

I. Multi-Query Optimization

Instead of one time queries in a relational DBMS, a stream system is processing a number of continuous queries simultaneously as discussed in Section 2.4. These queries are active for long periods of time and they process massive streams in real time. A poor query implementation choice can negatively affect system performance for the lifetime of the query. The key to achieving good stream processing performance is to optimize multiple queries together, rather than individually. In a stream query workload, it is often the case that multiple concurrently active queries can share system resources such as CPU and memory. Query evaluation techniques that exploit this property are referred to as Multi-Query Optimization (MQO) techniques**.** The importance of MQO for stream processing is widely accepted and various stream MQO techniques have been proposed [7, 8, 9, 10, 11].

To meet the objective of Multi-Query optimization, an important component in the query engine is called Multi-Query optimizer. The high level architecture of Multi-Query optimizer is given in Figure 2-3. From the architecture, the Multi-Query optimizer accepts a set of queries and generates a global query execution plan. The global query execution plan satisfies the needs of all the given set of queries.

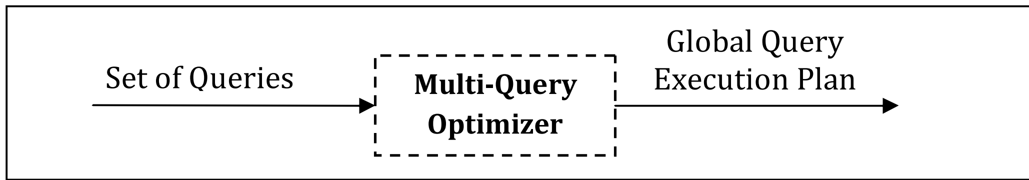

Figure 2-3: High level Architecture for Multi-Query optimizer

II. Load Shedding

One of the potential problems of evaluating queries over streaming sources arises when the arrival rate of data stream exceeds the processing capacity of the CPU. Since no execution schedule can minimize the size of data elements in queue, the only alternative is to artificially decrease the length of the queue by “dropping” some of the data elements in it. This is called load shedding [6].

As optimization problem, the conceptual solution to the problem for load shedding is the introduction of drop-boxes in the query plan. These can be thought of as operators inserted into the execution plan between successive operators, or between an incoming stream and the first operator to process it. The function of these operators is to selectively drop a percentage of incoming data elements before they reach the subsequent operator. Introducing drop-boxes in the execution plan leads to another subtle decision that needs to be made. Not only do we need to determine how much load each drop-box should shed, we also need to determine the location of these drop boxes in the query plan. The solution to both these problems makes use of Loss/Gain ratio modeling [6]. A Loss/Gain ratio is the ratio between the loss in utility incurred by the introduction of a drop-box, over the gain in QoS. Each alternative drop-box location is associated with a Loss/Gain ration. Once all drop locations are enumerated and their Loss/Gain ratios are computed, they are sorted in Loss/Gain ratio ascending order. The system then iterates over each location adding drop-boxes. After the addition of a drop box, new Loss/Gain ratios are computed, since the introduction of a drop box may affect them. The iteration stops once the optimization objective is met.

An important decision is how data elements are dropped once the drop-boxes are in place. The simplest method is, of course, *random dropping*. That is, once the percentage of data elements to be dropped is identified, these data elements are randomly dropped from the input, so long as the

dropped data elements percentage quota is met. An alternative approach and one that has been proven to work better in practice, is *semantic dropping*. When employing semantic dropping, the drop-box effectively becomes a selection operator evaluating a predicate on the values of the incoming stream. The utility of each data element is computed, based on the data element's values. The semantic predicate of the selection operator then becomes one that drops the least useful data elements, propagating only those having a higher utility as stated in [12].

CHAPTER THREE

3 RELATED WORKS

3.1 Introduction

In this Chapter, we have presented research works which are related to the proposed approach. There are a lot of research works on optimization in streaming applications. Among the works, Multi-Query optimization and load shedding are the two main common approaches of optimization and the main concern of the proposed research work. Thus, we have presented the most related research works on Multi-Query Optimization and Load Shedding.

3.2 Multi-Query Optimization for Stream Queries

In streaming applications, a number of research works have been conducted on the idea of MQO [7, 8, 9, 13, 14, 15]. Among these works, we have discussed some of the most related works to our approach. In [13], a Multi-query optimization with schedule synchronization has been proposed in which synchronizing the re-execution times of similar queries takes advantage of computation sharing. However, the approach do not considers periodic tasks in stream applications and don't considers dissimilar queries having the same feed sources. In [14] a paradigm for the sharing of window join queries has been proposed in which slicing window states of a join operator into fine-grained window slices and forms a chain of sliced window joins. This paradigm enables to push selections down into the chain and flexibly select subsequences of such sliced window joins for computation sharing among queries with different window sizes. But it is the cases that in streaming applications like publish/subscribe systems, queries are posted continuously to the system and massive queries exist with different window definition. Trying to generate a slice of window states for such a case is not computationally efficient. In addition, in [15] a Massively Multi-Query Join Processing technique for processing a large number of XML stream queries involving value joins over multiple XML streams and documents is proposed. These techniques enable the sharing of representations of input to multiple join, and the sharing of join computation.

MQO by partial aggregation of aggregate continuous queries (ACQ) is also considered in [10, 16, 17]. The Panes [16] scheme splits the slide into equal sized fragments, to be processed using the partial aggregation operator. Paired windows, [10] improve Panes by splitting the slide into exactly two fragments, minimizing the processing needed at the final-aggregation level. And Weave Share, a cost-based multiple ACQs optimizer, which exploits weaveability to optimize the shared processing of ACQs is proposed in [17].

Though these works contribute to the growing of stream query processing to a full potential, to the best of our knowledge, no work has been proposed that consider a shared execution of stream queries based on their semantic similarity of the queries and their defined windows. Thus, the use of semantic similarity measure in our approach to identify the relatedness of queries ensures a highly shared computation among a set of queries. This is especially important for the case of query execution on semantic information [2] which has a high computation cost. In addition, we have introduced a formal semantic measure for windows to create different classes of window relationship types for effective sharing of a given window among multiple queries.

Shared execution addressed in [18] focus on execution of multiple queries on larger window first (LWF) and alternatives to consider smaller windows first (SWF) strategy. However, overlapping windows are the case in streaming applications and should be treated wisely other than LWF and SWF strategies. And our novel approach also has a solution to shared execution on overlapping windows. Moreover, our approach brings efficient way of sharing a window among a set of queries defined on same source whether they are semantically similar or not. In addition, our approach can handles periodic queries which executes periodically in a given schedule and provides user schedule aware execution of multiple queries.

3.3 Load Shedding

Load shedding is an optimization approach which is a common way to reduce the system workload in stream systems. It has been first proposed in Aurora [26]. Aurora introduces two types of load shedding; random and semantic. Based on the analysis of the loss/gain rate, random load shedding determines the amount of data to shed to guarantee the output rate. For semantic drop, it assumes that different data element values vary in terms of utility to the application. In XML streams, instead of a simplistic model of certain domain value denoting utility, we consider

the complexity as well as importance of XML result structures in order to make shed query decisions. There are several other works related to load shedding in DSMSs in general, including memory allocation among query operators [30] load shedding for aggregation queries [29], and overload-sensitive management of archived streams [31].

In the Load Star system [33], statistical approaches are employed to identify the data to shed. However, statistical approaches only cannot insure that a given data is not fully irrelevant. Thus, our approach classifies all the incoming data into relevant and less relevant data class based on their information contents applying semantic analysis technique. While data in the relevant class are consumed directly, some sample of data from less relevant is selected using statistical approach. In aurora [34], three main questions about load shedding are raised: when, where, and how much to shed. Aurora answers these questions by checking system load periodically and triggering load shedding when excess amount of data arrives. A pre-computed Load Shedding Roadmap (LSRM) which determines possible shedding plans is used to predict where to shed load by minimizing the system utility loss. In our approach these questions are critical too. Thus, we shed data when the arrival rate is more than the system capacity based on the relevancy of each of data arriving to the system.

In [29], Babcock et al. proposed load shedding technique for systems that process continuous queries over data streams. The idea of the approach is that when the data arrival rate is high, placing load shedders in different places of the query plan, in such a way that the maximum relative error through all queries is to be minimized. Unlike this approach, our approach works for join operations. In our technique of load shedding a load is minimized from a sharable window applying load shedder on the incoming data before the actual evaluation of multiple queries.

In addition, other works which are closely related to our work are active mining and concept drifting for mining data streams is proposed. In [36], an active mining method that detects potential changes in data streams is proposed and in [35], Wang et.al. proposed an algorithm for mining concept-drifting data streams using weighted ensemble classifiers. Das et al. [37] proposed two heuristics for dropping data elements from window-join input streams. The first heuristic sets the priority of a data element being retained in memory based on the probability

that a counterpart joining data element arrives from the other data stream. The second heuristic favors a data element based on its own remaining lifetime. Kang et al. [38] proposed a cost-based load shedding mechanism for evaluating window stream joins by adjusting the size of the sliding windows to maximize the number of produced data elements. Ayad and Naughton [39] proposed techniques for optimizing the execution of conjunctive continuous queries under limited computational resources by placing random drop boxes throughout the query plan to maximize the plan throughput.

Approximate query evaluation approaches have been also researched for both traditional static data and continuous data streams [41]. The first approach mostly depends on pre-computed data synopsis whereas the second technique makes one-pass summaries when data streams arrive. Online aggregation [42] falls in-between the two approaches implementing sampling with query evaluation on static data. On the other hand the data triage technique has proposed to shed data load on streams making a summary of excess data into synopsis data structures rather than simply dropping it [43]. However, keeping synopsis data locally creates extra burden on the system as the rate data arriving to the system is huge and it removes the streaming nature of the data. Hence, to keep the streaming nature of the data and be sure of the data dropped are irrelevant, we have employed a highly guaranteed technique called semantic analysis technique.

Finally, to the best of our knowledge, our work is the first to systematically use linguistic analysis for semantic load shedding on a shared window among multiple queries.

CHAPTER FOUR

4 SEMANTIC NEWS FEED QUERY PROCESSING

4.1 Introduction

In this chapter we have presented the proposed approach for optimization of semantic news feed query in detail. Throughout the chapter, the different techniques proposed for optimization of SNF queries are discussed and these techniques are incorporated as components of our SNF query processing engine.

As presented in Chapter 2 of this thesis, though the objective of query optimization in general is to reduce query execution cost while keeping the relevance of query result, optimization of stream queries is more difficult than the traditional queries in relational database. To deal with such difficulties, various related optimization approaches using multi-query optimization and load shedding techniques for stream queries have been proposed as detailed in Chapter 3 of this paper. However, most of the approaches cannot be applied directly for our news feed query processing system for two main reasons. First, our SNF query is processed semantically on concepts retrieved from knowledge base. Secondly, the data is continuous and XML in nature and streamed in an asynchronous and pull strategy. Hence, we need an optimization approach capable of processing semantic SNF query on XML stream data. To do so, we have devised a new approach to amortize cost using multi-query optimization and load shedding.

Given a set of news feed queries; the proposed MQO accomplished in three main steps. First, the semantic similarity between queries is computed to generate a Multi Query Execution (MQE) plan for the set of semantically equivalent queries. This step helps to apply the execution result of one SNF query to the other semantically equivalent SNF queries to reduce execution cost. Secondly, the window relationship type that exists between queries is identified to generate a shared window among a set of SNF queries and as a result network access cost and memory space are minimized. Finally, MQE chains of queries are generated which are used to create singly linked list for efficient and faster execution of multiple queries at run time.

On the other hand, our load shedding works also in three steps. First a set of representative key terms of a set of SNF queries are identified. These set of keys are used to filter out non-relevant data elements from the incoming stream of data. These terms are enhanced with concepts extracted from knowledge base and this helps to identify relevant data elements from the incoming stream having semantically similar meaning. Finally, data elements from the incoming steam are examined to drop the non-relevant ones using the expanded key sets as a filter key and this task is incorporated into an operator called *QuickDrop* shedding operator.

Generally, the benefit of the proposed MQO approach is threefold. First it helps to identify semantically equivalent SNF queries to avoid redundant execution of similar queries. Secondly, it helps to generate common windows to execute a set of SNF queries simultaneously and avoids redundant holdings of same data in memory and reduces network access time and finally, the MQE chain data structure increases the speed of query execution. In addition, the proposed load shedding technique helps to reduce the data load from shared window by dropping non-relevant data elements.

The rest of this Chapter is organized as follows. Section 4.2 presents the architecture of the proposed SNF query engine emphasizing on the optimizer subcomponent. Section 4.3 is on multi query execution management. Section 4.4 describes the database schema and finally Section 4.5 summarizes this chapter.

4.2 System Architecture

The proposed architecture of SNF query processing is shown in Figure 4-1. The SNF query processing engine performs query optimization in two phases; *Static Query Optimization* and *Run Time Optimization*. In stream processing system, user queries are relatively permanent and thus, it is stored as preference in *profile database*. The query engine reads a set of queries from the profile database and starts the optimization process and SNF query execution too.

The query engine utilize *Static Query Optimizer* component first. This optimizer reads queries from the profile database and performs static optimization using its two sub-components *MQO Manager* and *Shed Key Extractor*. The *MQO Manager* is responsible to generate Multi-Query Execution (MQE) plan based on semantic similarity of queries and their window relation. The

MQO Manager performs its task in three main steps; *Query Similarity Evaluation, Window Relationship Type Identification* and *MQE Chain Generation*. Query similarity evaluation is the task of identifying the semantic similarity between queries. If queries are semantically equivalent, the execution result of one can be applied to the other. Doing so will reduce processing cost and memory space. In the second step, the relationship between query execution boundaries (also called 'windows) is identified; and this helps to generate shared windows later at run time. We have identified five type of window relationship types; *Includes, Included-In, Similar, Disjoint, and Overlap*. Hence, based on the relationship type that exists between windows, queries can be executed on shared windows simultaneously and this helps to save network access time and also saves memory space. The last step is MQE chain generation in which a base linked list data structure called *MQE Chain* is generated and a particular chain contains queries which share a resource and hence have potential to be executed together.

The second subcomponent in static SNF query optimization is *Shed Key Extractor* and it is responsible to filter out set of key words from search terms of set of queries. After extracting key words, it semantically enhances each keys of the set to generate a larger set of keys. The keys set are used to filter out non-relevant data elements from shared windows among set of queries which are represented by key in the key set. Finally, the optimized query plan is written to the profile database to be used later at run time and this is the end of the first phase of SNF query optimization.

The second phase of the optimization process starts just at the actual query execution time. The *Run-Time Query Optimizer* is the component to perform the optimization tasks in this phase. The optimization is triggered by the *Execution Manager* component in the query engine. The *Execution Manager* reads optimized MQE plan prepared at the first phase of static optimization and user preferences from the profile database. After the *Execution Manger* read the MQE plan, it gives the *Run-Time Query Optimizer* the feed source with window boundary information and key set for load shedding. Then the optimizer generates shared window with the interaction of its subcomponents; *Load Shedder* and *Window Generator.*

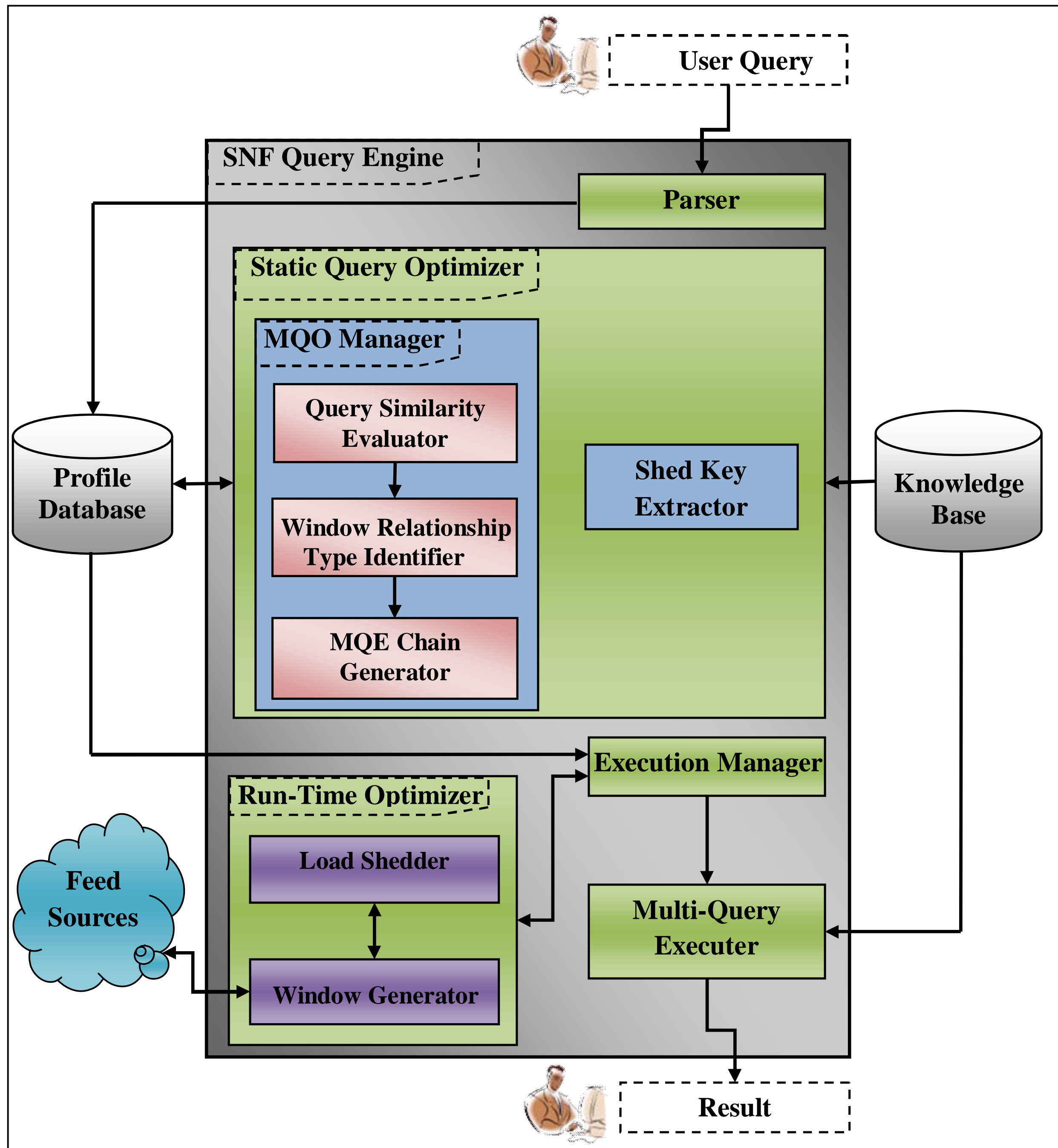

Figure 4-1: Architecture of the Semantic News Feed Query Processing Engine

Finally, the *Execution Manager* passes the MQE plan with the generated shared window by *Run-Time Query Optimizer* to the *Multi-Query Executer* component. The *Multi-Query Executer* component performs its task using multi-query execution rules identified and with thread based execution strategy.

4.2.1 Static Query Optimizer

In the first phase of SNF query optimization, the *Static Query Optimizer* performs two main tasks. First, it prepares Multi-Query optimization plan after analyzing query similarity and their windows relation. Then, it extracts key terms/concepts embedded in each query for load shedding. These activities are performed by *MQO Manager* and *Shed Key Extractor* subcomponents of Static Query Optimizer component. The next subsections present these subcomponents and their core tasks in detail.

4.2.1.1 MQO Manager

Given a set of SNF queries, *MQO Manager* prepares MQO plan in three main steps. First it evaluates the semantic similarity between queries in the database to generate MQE plan for set of semantically equivalent queries. Then, it identifies window relationship types that exist between pair of queries in the profile database. Finally, it generates MQE chain of similar queries and represents them in a base singly linked list for efficient and faster execution of multiple queries at run time. Next, these steps are discussed in detail.

STEP 1. Query Similarity Evaluation

In stream systems, queries are relatively persistent and changes seldom and thus are registered persistently in profile database. It is likely that the profile database may contain large number of queries with possible redundancy in query definition. Processing such redundant queries degrades the performance of the system. To avoid such anomalies, the *Query Similarity Evaluator* subcomponent of *MQO Manager* Component applies semantic similarity measure given in Definition 4-1 to examine the extent to which user queries in the profile database are semantically equivalent.

Definition 4-1: [SNF Query Similarity Measure]

Given two pair of queries Q_1 and Q_2, each predicate having a source, operator and value part, the semantic similarity in between is denoted as *SimQuery*(Q_1, Q_2) and is defined as follows:

$$SimQuery(Q_1, Q_2) = \begin{cases} 0 & \text{if } Q_1.S_1 \neq Q_2.S_2 \text{ and } Q_1.\theta_1 \neq Q_2.\theta_2 \\ SemRel(Q_1.P_1, Q_2.P_2) & \text{otherwise} \end{cases}$$

Where: -

- S_1, S_2 are feed sources of Q_1 and Q_2,
- θ_1, θ_2 are operators of Q_1 and Q_2,
- P_1, P_2 are predicates of Q_1 and Q_2,
- *SemRel()* returns similarity between queries predicates.

Two queries are different if the source and operators are different; otherwise their similarity is dependent on the similarity between their corresponding predicates. To compute the similarity between predicates of two queries, we have employed cosine based approach, which is a vector-based semantic analysis technique.

Vector-based semantic analysis is the practice of representing word meanings as semantic vectors, calculated from the co-occurrence statistics of words in large text data. There are many vector based approaches to measure semantic meaning between texts. Among the approaches, the cosine measures the angle that separating the vector representing the text [5].

When comparing two texts T_1 and T_2, each would be represented as a vector $\vec{V}$ ($\vec{V_1}$ and $\vec{V_2}$ respectively) with weights underlining concept occurrences and descriptive degrees in their corresponding concept Sets, $CS(T_1)$ and $CS(T_2)$, taking into account global semantic neighborhood [23], which is defined as the set of concepts in a given knowledge base, related with the hyponymy or meronymy semantic relations, directly or via transitivity. And the global semantic neighborhood of a given concept is the union of each semantic neighborhood w.r.t. all synonymy, hyponymy and meronymy relations altogether. Accordingly, we employed the revised cosine analysis approach [2] to measure the semantic relatedness *SemRel,* which is a value between 0 and 1, of two texts given by Equation 4-1.

$$SemRel = Sem(T_1, T_2) = Cos(\vec{V_1}, \vec{V_2}) = \frac{\vec{V_1} \cdot \vec{V_2}}{|\vec{V_1}| \times |\vec{V_2}|} \qquad 4\text{-}1$$

Where: -

- $\vec{V_1} \cdot \vec{V_2}$ is the dot product of the texts\` vectors,
- $|\vec{V_1}|$ is the norm of vector V_1,
- $|\vec{V_2}|$ is the norm of vector V_2.

Accordingly, one can compute the similarity between two texts simply by computing the cosine distance between their feature vectors such that if the score (*SemRel*) between concepts of texts is above a given *Equality Threshold,* T_{Equal}, value, then the two texts are said to be semantically similar.

Given an equality threshold, T_{Equal}, if the similarity between two queries Q_1 and Q_2 $SimQuery(Q_1,Q_2)$ *is above* the threshold, then the two queries are semantically equivalent as shown in Figure 4-2.

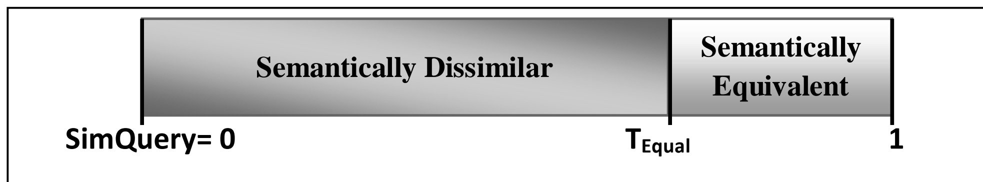

Figure 4-2: Query relationship and corresponding threshold

If two queries are semantically equivalent, we filter out one as a representative and apply the execution result to the other query. Thus, query processing cost can be minimized. The query processing cost is a sum of network access cost, memory access cost, and execution cost. Hence, the decision to evaluate one of semantically similar queries reduces network access time and avoids redundant holding of the same data in memory which directly reduces the query processing cost. Example 4-1 explains the concept briefly.

Example 4-1: [Semantically Equivalent Queries]

Referring to queries Q_1 and Q_2 in Table 1-1, vectors shown in Table 4-1, and using equality threshold T_{Equal} of 0.9, the semantic similarity, *SemRel*, in between is:

$$SemRel(T_1,T_2) = Cos(\overrightarrow{V_1},\overrightarrow{V_2}) = \frac{\overrightarrow{V_1} \cdot \overrightarrow{V_2}}{|\overrightarrow{V_1}| \times |\overrightarrow{V_2}|} = 0.94$$

Therefore, $SemRel >= T_{Equal}$ *and* as a result the two queries are semantically equivalent. Accordingly, we can select one as a representative and apply the execution result to the other. As a result, execution cost and network access cost can be minimized and memory space can be used efficiently.

Table 4-1: Vector generated when comparing two SNF queries

	Search terms									Attribute		Operators
	Car	*crush*	*kill*	*Two*	*people*	*Iran*	*dead*	*Tehran*	*accident*	*Title*	*Description*	*Select*
$\overrightarrow{V_1}$	1	1	1	1	1	1	0.5	1	0.8	1	1	1
$\overrightarrow{V_2}$	1	0.8	1	0	0.5	1	1	1	1	1	1	1

On the other hand, given dissimilar queries defined on the same feed source, it is possible to minimize query execution cost as there is a chance to share network access cost. Thus, the next step in the Static Optimizer component is to identify the relationship between windows of queries in the database for effective sharing of network access cost, as detailed in the next step.

STEP 2. Windows Relationship Type Identification

Whether they are semantically similar or not, queries defined on the same feed source might share a window at the definition of their window boundary time. Such queries may be executed on a single shared window simultaneously.

Generating a window and sharing it among a set of queries defined on the same source have the following advantages:

- It significantly decreases the network access time (NAT[3]) since a data accessed once will be used by a set of queries.
- Efficient utilization of memory space. Sharing a single window among a set of queries eliminate redundant storage of same data in memory.

As described in Chapter 2 of this thesis, a window has a start and end boundary time. Hence identification of windows semantic relationship is related to the issue of the time ontology. In [32], the author defined the basic set of mutually exclusive primitive relationships that exists between two time intervals. Accordingly, for a given two time intervals t_1 and t_2, the relations that can holds are given in Table 4-2.

[3]NAT is the time elapsed while retrieving feed items from a specified feed source.

Table 4-2: Primitive relationships between time intervals [32]

Relation Type	Denoted as	Meaning
t_1 *during* t_2	$t_1\ d\ t_2$	time interval t_1 is fully contained within t_2
t_1 *starts* t_2	$t_1\ s\ t_2$	t_1 shares the same beginning as t_2, but ends before t_2 ends
t_1 *finishes* t_2	$t_1\ f\ t_2$	interval t_1 shares the same end as t_2, but begins after t_2 begins
t_1 *before* t_2	$t_1 < t_2$	t_1 is before interval t_2, and they do not overlap in any way
t_1 *overlap* t_2	$t_1\ o\ t_2$	interval t_1 starts before t_2, and they overlap
t_1 *meets* t_2	$t_1\ m\ t_2$	t_1 is before t_2, but there is no interval between them
t_1 *equal* t_2	$t_1 = t_2$	intervals t_1 and t_2 are the same interval

The primitive relations given in Table 4-2 compare time intervals with exact matching of boundary times. However, it is likely that two windows might be defined with some boundary time variations and yet we need to take them as similar. Therefore, we need to enhance the definition of primitive relationships between time intervals to make it flexible and handle negligible variations. To enhance the primitive definitions of relationships between time intervals, we have given a formal semantic measure between windows boundary as presented in Definition 4-2.

Definition 4-2: [Similarity Measure of Boundary Time]

Given two windows W_1 and W_2, with start boundary time S_1, S_2 and end boundary time E_1, E_2 defined in minutes respectively, and equality threshold value *TEqual*, the similarity between the boundary times of the two windows is denoted as $BoundarySim(W_{1,}W_2)$ and returns a similarity vector B_V having normalized start, end and end to start time difference and is formally given as follows:

$$\mathrm{BoundarySim}(\mathrm{W_1}, \mathrm{W_2}) = \mathrm{B_v} = [\mathrm{S}, \mathrm{E}, \mathrm{D}]$$

Where:

- $S = \frac{S_1 - S_2}{60}$, normalized start time difference
- $E = \frac{E_1 - E_2}{60}$, normalized end time difference,
- $D = \begin{cases} \frac{E_1 - S_2}{60} & if\ S_2 > E_1 \\ \frac{E_2 - S_1}{60} & if\ S_1 > E_2 \\ 0 & otherwise \end{cases}$, normalized end to start time differnece

60 minute, an Hour, is used as normalization value. The normalized values are used later in identifying the magnitude of the windows similarity vector. Accordingly, window relationship type is identified after building a row vector of similarity between each windows` boundary of given set of queries in profile database.

Based on Definition 4-2, a value of 0 for all S, E, and D means that the two windows boundary time's are defined exactly on the same time. However, to handle the time variation that might happen during window definition, we define minimum and maximum equality threshold values (i.e, $minTR$ and $maxTR$); and if the boundary time similarity value falls within this range, then the two boundary time definitions are similar.

Therefore, given an equality threshold T_{Equal}, $minTR$ is equality threshold below zero and $maxTR$ is equality threshold above zero are defined as given in Figure 4-3.

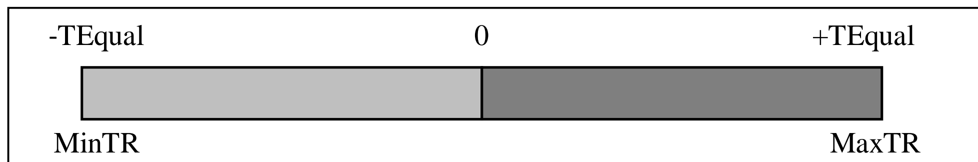

Figure 4-3 : Equality threshold and its corresponding *minTR* and *maxTR*

In light of Definition 4-2 and the primitive relationships between time intervals given in Table 4-2, windows semantic relationship type is refined in Definition 4-3.

Definition 4-3: [Window Semantic Relationship Type]

Given two queries Q_1 and Q_2 defined on associated windows W_1 and W_2, and boundary similarity vector B_V containing S, E, and D, the relationship between W_1 and W_2 is exclusive which is either *Similar, Includes, Included-In, overlaps* or *Disjoint* (as shown in Figure 4-4) is identified using two threshold values $minTR$ and $maxTR$. It is denoted as $WinRelation(W_1, W_2)$ and defined as follows:

Relation I. The relationship between two windows is disjoint, W_1 *Disjoint* W_2, denoted as $W_1 \rhd\lhd W_2$: if W_1 and W_2 are defined in different window boundary time as shown in Figure 4-4 (C). Formally defined as:

$$WinRelation(W_1, W_2) = 'Disjoint'$$
$$\Longrightarrow \forall_{t \in W_1}, t \notin W_2 and\ if(D < \ minTR\ or\ D = \ 0)$$

Relation II. The relationship between two windows is includes, W_1 *Includes* W_2, denoted as $W_1 \supset W_2$: if the window boundary time of W_2 falls within the window boundary time of W_1 as shown in Figure 4-4 (A).Formally defined as:

$$WinRelation(W_1, W_2) = 'Includes'$$
$$\Longrightarrow \forall_{t \in W_2}, t \in \ W_1 and\ if(S < \ minTR \ \ and\ \ minTR \leq E \leq maxTR) or$$
$$(minTR \leq S \leq maxTR\ and\ E \ > \ \ maxTR)\ or(S < \ minTR\ and\ E > \ maxTR)$$

Relation III. The relationship between two windows is Included-In, W_1 *Included-In* W_2 , denoted as $W_1 \subset W_2$: if the window boundary time of W_1 falls within the window time boundary of W_2 as shown in Figure 4-4 (B) or if $W_2 \supset W_1$.

$$WinRelation(W_1, W_2) = \text{“Included-In”}$$
$$\Longrightarrow WinRelation(W_2, W_1) = 'Includes'$$

Relation IV. The relationship between two windows is overlap, W_1 *Overlap* W_2, denoted as $W_1 \Cap \ W_2$: if the window boundary time of W_1 and W_2 intersect each other as shown in Figure 4-4 (E). Formally defined as:

$$WinRelation(W_1, W_2) = 'Overlap'$$
$$if\ (S < \ \ minTR \ \ and\ E \ < \ \ minTR) or(S > \ maxTR\ and\ E > \ maxTR)$$

Relation V. The relationship between two windows is similar, W_1 *Similar* W_2 denoted as $W_1 \equiv W_2$: if the boundary time definitions of W_1 and W_2 falls within *minTR* and *maxTR* as shown in Figure 4-4 (D). And it is formalized as:

$$WinRelation(W_1, W_2) = \text{’Similar’}$$
$$\Longrightarrow \forall_{t \in W_1}, t \in W_2\ and\ \ if(minTR \leq \ S \leq maxTR\ and\ minTR \leq \ E \leq \ maxTR \ )$$

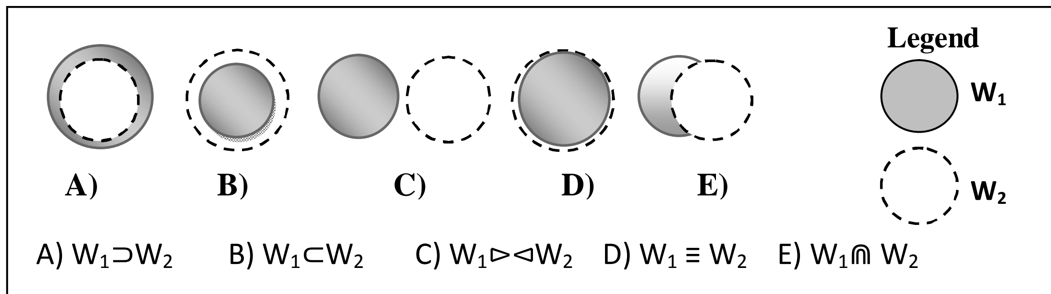

Figure 4-4: Topology for window relationship type

The algorithm presented in Pseudo Code 4-1 identifies the window semantic relationship type that exists between queries as discussed above. The algorithm accepts windows, associated feed sources and equality threshold as inputs. Computing similarity between windows boundaries is pre-condition to relationship type identification. In line 7 the algorithm compares feed sources and if they are defined on the same source then it computes similarity values of the given windows boundary time as a vector having three components; S, E, and D based on Definition 4-2 in line number 8 and 9. Then in line 11 – 26, it identifies the window relationship type that exists between the given windows based on Definition 4-3. And if the given feed sources are dissimilar, the window relationship type is *Disjoint* as shown in line number 28.

Pseudo Code 4-1 :WinRelation Identifier Algorithm

```
     Input:
1.     W1, W2:Window                                  // Windows of queries
2.     FS1,FS2: String                            // Feed sources of queries
3.     TEQ: Decimal                                  // Equality threshold
     Intermediate:
4.     minTR, maxTR, S, E, D :Decimal
5.     BV: Row-Vector
     Output:
6.     WinRel: String
     Begin:
7.   IF(FS1 = FS2) Then      // Same feed source thus apply Definition 4-3
8.     BV = BoundarySim(W1, W2)                    //using Definition 4-2
9.     S = BV[0],E = BV[1],D = BV[2]      //Boundary time differences
10.    minTR = -TEQ , maxTR = TEQ    //minimum and maximum  threshold
11.  //isBetween checks if the given parameter is between minTR and maxTR
12.  IF((isBetween(S))AND(isBetween(E))) Then
13.     WinRel= "Similar"
14.  Else IF((S<minTR AND isBetween(E))OR(isBetween(S) AND E>maxTR)
15.  OR(S<minTR AND E>maxTR)) Then
16.        WinRel= "Includes"
17.  Else IF((S>maxTRANDisBetween(E))OR(isBetween(S) AND E<minTR)OR
18.  (S>maxTRAND E<minTR))  Then
19.        WinRel= "Included-In"
20.  Else IF(D<minTR OR D = 0) Then
21.        WinRel= "Disjoint"
22.  Else IF(((S<minTR)AND(E<minTR))OR((S>maxTR)AND(E>maxTR))) Then
23.        WinRel = "Overlap"
24.  Else
```

Pseudo Code 4-1: WinRelation Identifier Algorithm

```
25.         WinRel= "Disjoint"
26.     End IF
27.     Else                    // Different feed sources thus Disjoint
28.         WinRel= "Disjoint"
29.     End IF
30.     Return(WinRel)
        End
```

Example 4-2: [Window Relationship Type]

Referring to queries in Table 1-1, the window relationship type between each query window definition is given in Table 4-3.

Table 4-3: Windows Relations for of queries identified using WinRelation algorithm

Query	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6
Q_1	$\equiv$	$\supset$	$\equiv$	$\rhd\lhd$	$\rhd\lhd$	$\rhd\lhd$
Q_2	$\subset$	$\equiv$	$\subset$	$\rhd\lhd$	$\rhd\lhd$	$\rhd\lhd$
Q_3	$\equiv$	$\supset$	$\equiv$	$\rhd\lhd$	$\rhd\lhd$	$\rhd\lhd$
Q_4	$\rhd\lhd$	$\rhd\lhd$	$\rhd\lhd$	$\equiv$	$\rhd\lhd$	$\Cap$
Q_5	$\rhd\lhd$	$\rhd\lhd$	$\rhd\lhd$	$\rhd\lhd$	$\equiv$	$\rhd\lhd$
Q_6	$\rhd\lhd$	$\rhd\lhd$	$\rhd\lhd$	$\Cap$	$\rhd\lhd$	$\equiv$

Once windows relationship types have been identified, the last step in MQO plan preparation is MQE base chain generation as presented in next subsection.

STEP 3. MQE Chain Generation

Before the actual execution of multiple queries, appropriate data structure should be used for efficient utilization of resources and faster execution of queries at run time. In this work, we used chaining to structure queries based on their window relations. A chain is a singly linked list of queries with window related with a given relationship type as detailed in Definition 4-4.

Definition 4-4: [Chain of X]

Given *n* queries, Q_1, Q_2, Q_3,…,Q_n, and windows relationship type X, Chain of X, is set of queries related with the relationship type X and are stored in a linked list data structure. Formally, it is denoted as:

$$C_X = \{Q_1, Q_2, Q_3, \ldots, Q_n\}$$

Where:

- X= *WinRelation*(W_i, W_j), *WinRelation* returns window relation type between W_i and W_j
- W_i , W_j are windows associated to $Q_i, Q_j \in \{Q_1, \ldots, Q_n\}$ respectively

Accordingly, queries in a given chain will be executed on shared windows according to the MQE rules given in Subsection 4.2.4.

The algorithm given in Pseudo Code 4-2 is devised to generate a MQE chain. It accepts a collection of windows relationship types identified using Pseudo Code 4-1 and list of queries as input. In line 5-19, the algorithm generates chains of Includes, Similar, Disjoint, Included-In and Overlap type represented as a base linked list based on Definition 4-4.

Pseudo Code 4-2: Chain Generator Algorithm

```
    Input:
1.  WinRel: Collection-Window       // set of Window relation type
2.  Query: Array                    // set of queries
    Intermediate:
3.  SimChain, IncChain, IncInChain, OverChain, DisChain: LinkedList
    Output:
4.  Chain: Array                     // set of chains
    Begin:                           // generation of chain
5.  Fori=0 To rowLength(WinRel)
6.     For j=i+1 To columnLength(WinRel)
7.         IF(WinRel[i][j]="Similar") Then
8.             addToChain(simChain, Query[i], Query[j])
9.                 //addToChain links queries to a given chain
10.            Else IF(WinRel[i][j]="Includes") Then
11.               addToChain(IncChain, Query[i], Query[j])
12.            Else IF(WinRel[i][j]= "Included-In") Then
13.               addToChain(IncInChain, Query[i], Query[j])
14.            Else IF(WinRel[i][j]="Overlap") Then
```

Pseudo Code 4-2: Chain Generator Algorithm

```
15.             addToChain(OverChain, Query[i], Query[j])
16.          Else
17.             addToChain(DisChain, Query[i], Query[j])
18.       End IF
19.    End For
20. End For
21. Return⟨SimChain, IncChain, IncInChain, OverChain, DisChain⟩
    End
```

4.2.1.2 Shed Key Extractor

One of the potential problems of evaluating queries over streaming sources arises when the incoming stream rate exceeds the processing capacity of the CPU. This is even worst when query evaluation done over a semantic information. In such cases, a technique called load shedding is used to gracefully decrease the data load to increase the performance of the system without affecting the relevancy of query result [6].

There are two types of load shedding techniques; random and semantic dropping as discussed in Chapter 2 of this paper. Based on the analysis of the loss/gain rate, random load shedding determines the amount of data to shed to guarantee the output rate. For semantic drop, it assumes that different data element values vary in terms of utility to the application. Among these, we have employed semantic load shedding technique as detailed in Subsection 4.2.3.2. Accordingly, our semantic load shedding technique uses keys retrieved from user query`s search term to drop non-relevant data elements when stream of data arrive at run time. The *Shed Key Extractor* subcomponent in the Static Query Optimizer is responsible to perform the task of key retrieval. It performs its task in two main steps: *Key Extraction* and *Key Expansion* as presented next.

STEP 1. Key Extraction

Key extraction is the process of filtering out important words or terms from a given set of queries` search term. A key is a word whose meaning helps in representing the notion of the query`s search term. Since nouns and verbs are successful to represent a search text [19], we use them as key. Key extraction process generates a *Key Set* given by Definition 4-5.

Definition 4-5: [Key Set]

Given a set of queries Q containing queries q_1, q_2, q_3,...,q_n and each query has a valid search term, a key set K is a set of keys extracted from each queries in Q containing a collection of keys k_1, k_2, k_3,..., k_m.

In the key extraction process we perform a pre-processing tasks on a given collections of search terms. This implies Tokenization and Part of Speech Tagging. Tokenization is the process of breaking a given text (search term) into words or other meaningful elements called tokens. The list of tokens becomes input for the part of speech tagging task, which is the process of marking up a word in a text as corresponding to a particular part of speech based on both its definition as well as its context. The part of speech tagging task is performed with Brill`s tagger algorithm [40]. Then after these preprocessing tasks, 'nouns and verbs are selected as key representing the whole search term. Example 4-2 shows the extraction process.

Example 4-2: [Key Extraction]

Referring to queries Q_1 and Q_2 in Table 1-1 with search terms "car crush killed two people in Iran" and "Dead in Tehran car accident ", the keys extracted after key extraction process are: "*car, crush, kill, people, Iran, dead, Tehran, accident*". These keys are used as a representative for search terms of the two queries.

STEP 2. Key Expansion

The second step in our semantic load shedding approach is *Key Expansion.* Since our QuickDrop algorithm (Pseudo Code 4-3) employs text matching to drop data elements from the incoming data stream, applying keys identified in the first step may lead to considering keys having same meaning as if different. The solution to such problem is to perform key expansion (i.e., enhancing key with concepts extracted from knowledge base). Key enhancement is given in Definition 4-6.

Definition 4-6: [Key Enhancement]

Given semantic Knowledge Base KB, k' is the semantically enhanced form of the key k, if k is rewritten with related concepts extracted from KB. It is formalized as:

$$k' = getConcepts(k, KB)$$

Where:

- *getConcepts*() returns similar words having related concepts from knowledge base with a knowledge depth of 2.

So the key expansion process enhances each search key in a given key set and as a result the search key space increases. And this provides enough search keys to the *quickDrop* algorithm and helps to reduce the probability of dropping relevant data elements.

Example 4-3: [Enhancing Keys]

For keys identified in Example 4-2 i.e. "*car, crush, kill, people, Iran, dead, Tehran, and accident*", the key enhancement task produces the following clusters of keys:

> "***car**, auto, vehicle", "**crush**, leather", "**kill**, terminate, deathblow", "**people**, group", "**Iran**, Middle East",* and *"**dead**, slain, accident, collision, and crash*"

Generally, the output of the Static SNF Query Optimization is a set of query execution plans containing the following:

- *MQE Chain:* as detailed in Section 4.2.1.1 the MQE chains (i.e. chain of Includes, Included-In, Overlap, Similar and Disjoint) is generated. Each chain contains a set of queries to be executed on shared window.
- *Key Set:* as explained in Section 4.2.1.3 before the actual load shedding is done at run time, keys are needed to filter out less relevant data from the incoming stream of data. Thus, a set of keys are generated to be used later to apply load shedding on shared window.

Finally, after the static query optimization, the next phase is runtime optimization performed at run time during query execution which is the issue of the next section.

4.2.2 Execution Manager

The *Execution Manager* subcomponent is responsible to trigger phase two of SNF query optimization i.e. run-time optimization process and multi-query execution. To trigger execution,

the execution manager reads user schedule, which is stored in profile database as part of user preferences. User schedule contains day of the week and specific time of a day of which the execution result of a particular query will be delivered. To do so, the execution manager employs daemon thread to actively inspect user schedule per query and maintain the list of queries scheduled for execution. Figure 4-5 shows sample list of queries together with associated execution schedule generated by *Execution Manager* for queries given in Table 1-1.

However, as multi-query execution adopted in this research, the initial schedule is synchronized i.e., query schedules are rearranged to fit the task of multi-query execution. Hence, the execution of one query may trigger the execution of many queries which are planned to be executed together by static query optimizer. Accordingly, when the time scheduled for a particular schedule is reached, the execution manager starts by reading MQE plan from the profile database generated by static query optimizer for a set of queries. Then it interacts with the run time optimizer components to get a shared window with optimal size. Finally, it provides the Multi-Query Executor components with MQE plan and the generated shared window.

QID	Execution Time (GMT)	Time Remaining(Min.)	Scheduled Time(GMT)
100	00:12:50	12	10:00:00
101	02:12:50	132	12:00:00
102	03:12:50	192	13:00:00
103	06:12:50	372	16:00:00
104	07:12:50	432	17:00:00
105	07:12:50	432	17:00:00

Figure 4-5: Sample Execution table/schedule generated by *Execution Manager*

4.2.3 Run Time Optimizer

The optimization tasks undertaken in the first phase are static and are performed before the actual stream of data arrives. However, phase two optimization of SNF query is performed just when the actual stream of feeds arrives at run time. To do so the *Run Time Optimizer* component in the query engine accepts MQE plan generated in phase one from the *Execution Manager and* generates a window with optimal size. The subcomponents in the Run Time Optimizer are *Load Shedder* and *Window Generator*. Each of these components is discussed in detail in the following sections.

4.2.3.1 Window Generator

The *Window Generator* is specifically responsible to generate window with optimal size using the information passed from *Run Time Optimizer*. As discussed in Chapter 2, a window is a mechanism to extract finite set of items from infinite data stream. Each news item in a given feed has a timestamp value which is added when the item is created or later upon transmission. The load manger takes the user preferences containing feed source and window boundary condition from the Execution Manager and interacts to the actual feed sources from the web and generates the desired window. Since feeds are dynamic and streamed continuously to the feed reader application, sliding window technique is used in our proposed approach.

Sliding Window

Instead of sampling the data stream randomly, we use the sliding window model to analyze stream data. The basic idea is that rather than running computations on all of the data seen so far, or on some sample, we can make decisions based only on *recent data*. More formally, at every time t, a new data element arrives. This element “expires” at time $t + w$, where w is the window “size” or length. The sliding window model is useful for applications where only recent events may be important. It also reduces memory requirements because only a small window of data is stored [20].

RSS news items are dynamic and in [3], Liu et al. stated that the update rate of RSS feeds is widely distributed and 55% of RSS feeds are updated hourly. Hence, in the proposed approach, the size of window to be used is one hour over a non-overlapping window. So a feed item will remain active for an hour and it will expire then after. The window generator requests for feeds in one hour interval shown in Figure 4-6.

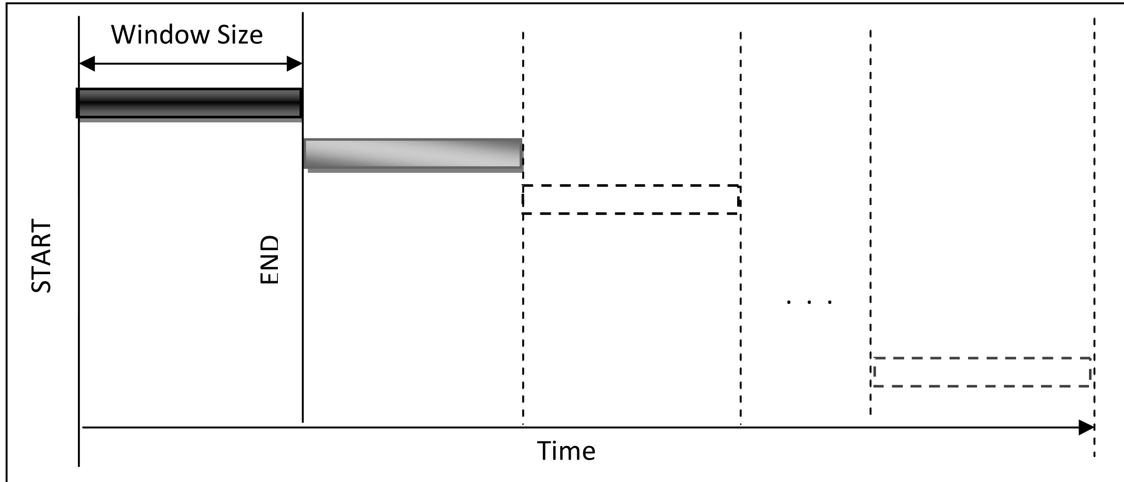

Figure 4-6: Non-overlapping sliding window with variable size

4.2.3.2 Load Shedder

Sometimes the arrival rate can significantly increase and exceed system capacity. So the immediate solution for such a case is to shed some of the load. In general terms, load shedding is the process of dropping excess load from the system [21]. The *Load Shedder* subcomponent employs a shedding operator called QuickDrop to drop non-relevant data elements based on text matching, which is much faster than semantic analysis based comparison as explained next.

The *QuickDrop* Operator

The operator `quickDrop(W, T, K, e)` takes four parameters; `W` denotes a shared window among a set of queries for Multi-Query execution, `T` denotes the attribute of the news feed for shedding like *title*, *description*, `K` denotes a set of enhanced keys and `e` denotes the level of precision to be used for sampling.

Our shedding algorithm is given in Pseudo Code 4-3. The *quickDrop* operator drop those news items having key sets extracted from attributes in T which do not share common key and estimated to have less contribution to the query result. To perform this, it first determines the optimal window size and estimates the error that might occur in the query processing due to shedding. It is known that, as the window size decreases, the error estimation will be large and degrades the query processing result. Thus, the operator dynamically adjusts the window size to keep the relevance of query result. After the determination of window size, the operator drops non-relevant data elements. Next the steps followed in shedding data load are presented in detail.

STEP 1. Determining Optimal Window Size

We say a window has optimal size when the total numbers of data elements it contain do not exceed data load processing capacity of the system. Since every data elements in a given window may not be relevant to the query processing result, dropping those non-relevant data elements helps to identify optimal window size. To determine optimal window size, relevancy analysis is made on individual data elements (news feeds). Relevancy analysis is the process of measuring the importance of a given data element based on its *key content* given in Definition 4-7.

Definition 4-7: [Key Content]

Given the set K containing key terms/concepts extracted from multiple queries and a data element(news feed) *e represented by* its attribute*(Title, Description)*, *key content* of e determines to what extent e shares keys with K and thus it measures how much information that a news feed e contains. Formally, it is denoted as:

$$KeyCon(e) = \frac{|e \cap \mathrm{K}|}{|K|} \in [0\,,1]$$

Where:-

- $|e \cap \mathrm{K}|$ returns number of matching keys in *e to K*,
- $|K|$ is the number of elements in *K*.

Key content is a value between 0 and 1. A value of 0 tells us that a data element contains no matching key to the keyset. Even though, the key content for a particular data element is 0, there is a probability to get semantically similar word(s) that match one of the keys in the given key set at a higher semantic knowledge depth. On the other hand, a value of 1 tells us that a given data element contains all the keys in a given key set. Finally, we classify the data elements in the window based on their key content into *relevant* and *less relevant* classes. We assume that relevant class has data elements with a key content greater than 0 and less relevant class has data elements with a key content of 0.

After the classification process, data elements in relevant class will be chosen as candidate elements to be included in the new optimal window. And to decrease the probability of missing those data elements having hidden semantic information relevant for query answer at a higher knowledge depth, all data elements in less relevant class may not be dropped. Thus, we apply statistical techniques to select some of the data elements at random and include them to the new

optimal window. Hence, among the total population N of data elements in less relevant class, a sample n is selected. A statistical sample size determination formula, $N/\left(1+Ne^2\right)$, in [22] is used to calculate n; where e is the level of precision[4].

Finally, if M is the number of elements in relevant class, then the new optimal window will have $M+n$ number of elements; where n is the sample taken from less relevant class.

STEP 2. Error Estimation

A good load shedding scheme drops data load with a graceful reduction in system performance. Load shedding increases the performance of the system at the cost of degrading the quality of query result. Hence, to keep QoS, error on the query result should be estimated; and we have quantified the error which is dependent on the level of sampling precision e. Thus, if the number of data elements in less relevant class is N and for sample size n, then the number of data elements to be dropped is given by $N-n$. Based on this we have estimated the error ε as a ratio of dropped data elements to number of data elements in less relevant class as given by Equation 4-2.

$$\varepsilon = \frac{N-n}{N} \in [0,1] \qquad 4\text{-}2$$

If the error is above a given threshold value, then the operator dynamically adjusts the sampling precision level. Equation 4-3 is the sampling precision level e adopted from the sample size determination formula in [22] and from Equation 4-2.

$$e = \sqrt{\frac{\varepsilon}{N(1-\varepsilon)}} \qquad 4\text{-}3$$

Finally, with the guidance of estimated error, the final stage is generating the new optimal window by dropping data elements at random from less relevant class, which is the issue of the third step.

[4] The level of precision, e, *is* calculated as 1-confidence interval, in the sample size determination formula used in [22] with a 95% confidence level and 50% degree of variability.

STEP 3. Dropping News Feeds

Having identified those news feeds in relevant class which are expected to yield query result and identified the sample size n to select from less relevant class, we drop $N - n$ number of data elements randomly from less relevant class which has a total population of N.

The three core steps in our *quickDrop* operator i.e. window size determination, error estimation and data element dropping are given by algorithm given by Pseudo Code 4-3.The algorithm accepts a window, feeds attribute on which to perform the load shedding process, enhanced key set as discussed in Section 4.2.1.3, sampling precision level *e,* and error threshold value. Then in line 12-21, the algorithm performs classifications of news items into relevant and non-relevant classes. In line number 22-27 the algorithm computes the sample size and estimates the error. If the error is above the given threshold, then it computes the precision level e and re-computes the sample size going back to line 22. And if the error is below threshold, then the algorithm selects the specified sample randomly in line 29. After all in line 31-33 the algorithm merges those selected sample together with the elements in relevant class into a new optimal window. As compared to the original window accepted by the algorithm, the final resulting window has optimal size and hence meets our objective. Having a detail on the QuickDrop operator, in the next section we deal with how to place it in a Multi-Query Execution plan.

Pseudo Code 4-3: Algorithm for *quickDrop* Operator

```
    Input:
1.  Window: Collection          // Shared window
2.  Attrb:  String              // data element attribute
3.  keySet: Array               // enhanced key set
4.  e:  Decimal                 // sampling precision level
5.  TError: Decimal             // error threshold
    Intermediate:
6.  lessRel: Collection         // less relevant class
7.  keyCon:  Decimal            // key content
8.  Error:   Decimal            // estimated error
9.  SamSize: Integer             // sample size
10. Smaple:  Collection
    Output:
11. opWindow: collection         // optimal window
```

Pseudo Code 4-3: Algorithm for *quickDrop* Operator

```
     Begin:
12.    For i=0 to sizeOf(Window)     // classification
13.     For j=0 to sizeOf(keySet)
14.        keyCon = count(keySet[j],Window[i],Attrb)/sizeOf(keySet)
15.          If keyCon> 0 Then
16.             opWindow.add(Window[i])      // relevant class
17.          Else
18.             lessRel.add(Window[i])       // non-relevant class
19.          End If
20.     End For
21.    End For
22.  START:
23.        SamSize = sizeOf(lessRel)/(1+(sizeOf(lessRel)*e*e)
24.        Error = (sizeOf(lessRel)-SamSize)/ sizeOf (lessRel)
25.     If Error>TError Then
26.        e = sqrt(Error/(sizeOf(lessRel)*(1-Error)))
27.        GOTO START
28.     Else
29.        Sample = getSample(lessRel,SamSize)
30.                     // getSample() selects random sample
31.        For q = 0 to sizeOf(Sample) //merging sample to opWindow
32.           opWindow.add(Sample[q])
33.        End For
34.     End If
35.  Return (opWindow)
     End
```

QuickDrop Operator Placement

In general, load reduction should be performed early in a query plan to avoid wastage of computation on non-relevant data. Therefore, given a Multi-Query Execution plan on a shared window, the *quickDrop* Operator is applied on the given shared window before the actual query operation. Figure 4-7 shows a Multi-Query Execution plan for three queries Q_1, Q_2 and Q_3 on a shared window *W*. Figure 4-7 (A) shows the original plan and (B) shows a plan with quickDrop operator β.

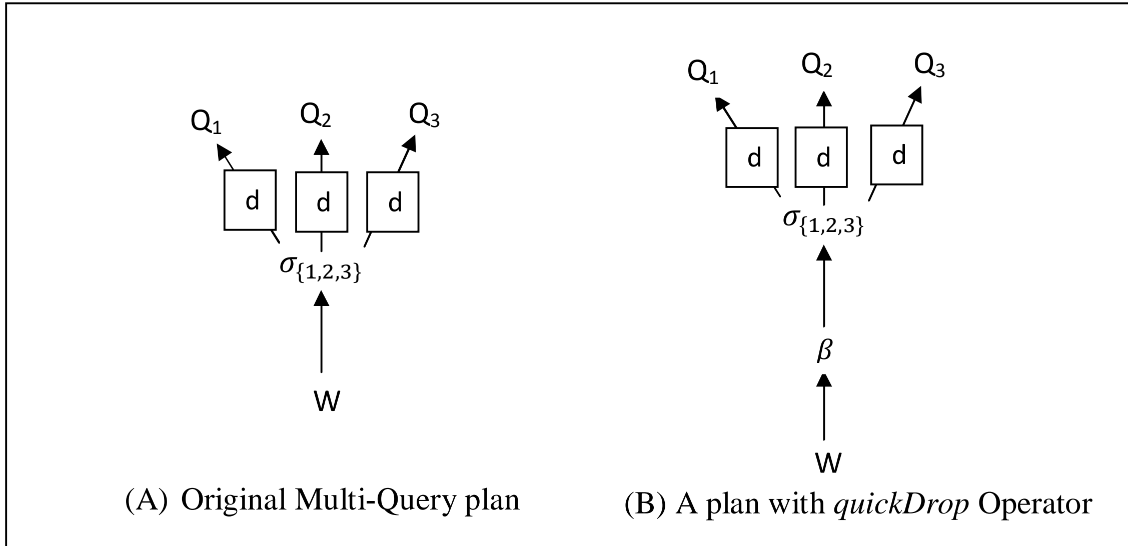

Figure 4-7: QuickDrop operator placement in Multi-Query Execution plan

As shown on the figure, the *quickDrop* operator is applied early in the query plan which helps to eradicate non-relevant data elements from query processing.

Finally, the run time SNF query optimization process produces sharable window with optimal size that can satisfy the need of all the queries in the current execution queue. Then after run time query optimization, the query engine passes the execution plan produced by static query optimizer with a shared window generated by run time optimizer to the next component in the query engine called *Multi-Query Executer* which is discussed in next section.

4.2.4 Multi-Query Executer

Query execution is the process of executing the plan chosen during query optimization. The objective is to execute the plan quickly by returning the answer to the user (or more often, the program run by the user) in the least amount of time. This is not the same as executing the plan with the fewest resources (CPU, I/O, and memory). For example, a parallel query almost always uses more resources than a nonparallel query, but it is often desirable because it returns the result more quickly. Hence, in order to execute multiple queries together, we proposed a rule based Multi-Query execution strategies supplemented with thread based execution. Accordingly we have identified the following five MQE rules:

MQE Rule 1: If queries Q_1 and Q_2 are semantically similar, then apply the execution result of one of the queries to the other.

MQE Rule 2: If a query Q is member of $C_{Disjoint}$ i.e., in disjoint chain, execute Q independently.

MQE Rule 3: If a query Q is member of $C_{Includes}$, $C_{Similar}$ or $C_{Included-In}$ i.e., in *Includes, Similar* or *Included-In* chain, then identify the largest chain containing *Q* and generate a window for the first element in the chain and apply MQE Rule 1. If there are more than one chain with equal size containing Q then apply MQE Rule 4 to execute Q.

MQE Rule 4: If there are multiple equal sized chains i.e. $C_{Includes}$, $C_{Included-In}$ or $C_{Similar}$ containing Q as their first element then execute Q on the union of the chains.

MQE Rule 5: If queries Q_1 and Q_2 are members of $C_{Overlap}$ with windows W_1 and W_2 respectively and if Q_1 is scheduled earlier than Q_2 then execute Q_1 on W_1 and Q_2 partially on the intersection of W_1 and W_2.

Accordingly, the *Multi-Query Query Executer* subcomponent in the query engine employs these MQE rules to execute a set of queries considering semantic information from the *Knowledge Base* database on a shared window generated by the *Load Manager* subcomponent.

4.3 Database Schema Design

In the proposed system, we have two databases: profile database and Knowledge base described in this section.

4.3.1 Profile Database

The logical database model shown in Figure 4-8 represents entities in the *Profile Database* and the interaction between entities. The following are the list of tables extracted from the logical database model:

- *User (UserID,FirstName, LastName, Email, Password):* contains the basic information about a user uniquely identified with UserID.
- *Feed_Source* (Source_ID, Source_URL, Source_Name): represents news feed providers information such as feed address (URL), source name.

- *User_Query* (QID, Search_Value, feed_attrubute, operator,Treshold,Semantic_Flag, ,Optimize_Flag, Item_Connector, K): represents user query detail such as search value, RSS operator, threshold value, semantic flag, item connector and K value for TopK RSS operator.

- *Optimized_Query* (QID, Query_Expression, MQE_With, Window_Rel_Type): represents the detail about a particular user query after optimization process. It includes the best operator execution order that minimize execution time, query (ies) with which to execute together and the window relationship type, the load shedding technique to apply to shed load.

- *Window* (QID, Feed_Src, Win_Begin, Win_End): represents the information to generate a window like source of news items, the start and end boundary conditions.

- *User_Schedule*(QID, Day_of_Week, Time_Of_Day): represents user query execution schedule information like day of the week and time on which the result of query execution is returned.

- *MQE_Chain* (CID, Chain): represents a base chain generated containing a set of queries to be executed together. It is used to generate a singly linked list at run time.

4.3.2 Knowledge Base

The knowledge base adopted in this research is WORDNET 2.0, English lexical database stored in MySql relational database. Thus, the logical database in Figure 4-9 represents entities referring to Concepts, Words, Semantic-Relation-Type, and Semantic-Relation and their interaction. The following are the list of tables extracted from the logical model:

- *Concept* (Synset_ID, description): represents concepts in Knowledge Base (i.e., collection of words having the same meanings). Synset_ID is a unique identifier that identifies a concept.

- *Words* (word_id, Synset_ID, word, ss_type): represents a word together with the concept or synset it belongs to. SS_type indicates the type of word (such as verb, noun).

- *Semantic-Relation-Type* (SemRelationID, Type, Description): represents the different semantic relationship discussed in Chapter 2. Each relationship is uniquely identified by SemRelationID.

- *Semantic-Relation* (Synset_Id_1, Synset_Id-2, SemRelationID): represents the semantic-relationship existing between two concepts identified with corresponding synset Id.

Figure 4-8: Logical database model of profile database for SNF query engine

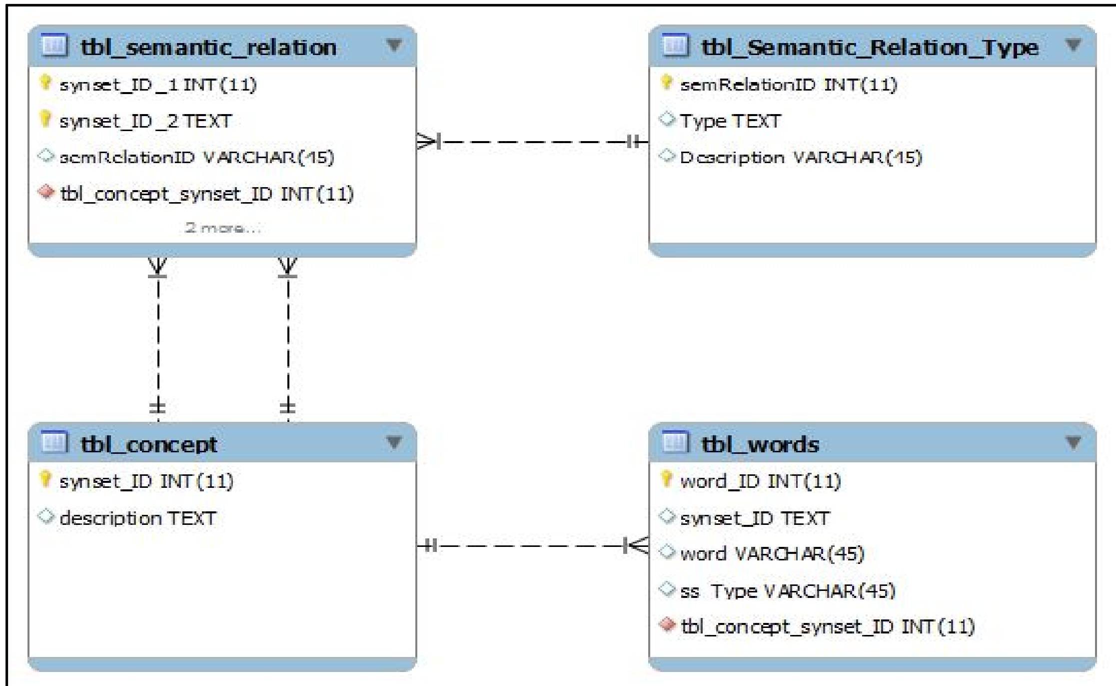

Figure 4-9: Logical database model of profile database for SNF query engine

4.4 Summary

In summary, the work presented in this chapter on query processing tasks in our SNF query engine focus on two main phases of optimization; Static SNF Query Optimization and Run-Time Optimization. The static optimization is performed just after the submission of user queries. It focus on MQE plan generation based on semantic similarity of queries plus windows relation and shed key filtering. After the static optimization tasks, the MQE plan will be registered to profile database to be used later at Run-Time optimization and query execution.

Then after the tasks in the first phase are accomplished, the run time optimization process starts just at the time of query execution. The main objective of run time optimization is to generate a shared window with optimal size. To meet that, it employs semantic load shedding scheme incorporated into our QuickDrop operator to gracefully decrease non-relevant data from a shared window to get a window with optimal size. Finally, the combined MQE plan of phase one and optimal sized window from phase two are used to execute multiple queries using thread based execution strategy guided by a set of MQE rules.

CHAPTER FIVE

5 IMPLEMENTATION AND EXPERIMENTAL RESULTS

5.1 Introduction

Developing a prototype to demonstrate the validity of the proposed semantic news feed query optimizer is one of the objectives of this research work. Hence, to realize the proposed approach, we have implemented a desktop based prototype using Microsoft C# programming language. The prototype allows users to:

1) Subscribe their preferences and define queries
2) Schedule queries and manage query schedule using query scheduler component and
3) Auto execution of queries according to the given user schedule.

In addition, the prototype is used to test the performance of the proposed system. In particular, three groups of experiments are conducted. These are:

1) Network access time (NAT) experiment to show how sharing a single window for multiple queries decreases the network access time.
2) Multi-Query execution (MQE) experiment to show how execution of multiple queries together increases the performance of the system.
3) Load Shedding experiment to show how load shedding increases the performance of the system considering quality of service issues.

All the experiments were conducted on personal computer with a single Intel Celeron CPU (2.2 GHz speed), 3 GB RAM and with a network connection speed of 54.0 Mbps. In addition, we have used WordNet 2.1 for the knowledge base.

The rest of this chapter is organized as follows. In Section 5-2 presents the user interfaces of the developed prototype. Then in Section 4-3 experimental results are presented.

5.2 User Interfaces

We have developed a set of Graphical User Interfaces to facilitate the interaction between a user and the proposed system. The interfaces include login, user preference subscription interface, user query definition, query schedule editor and query execution result displaying interface. Next these main interfaces are presented in turn.

5.2.1. User Preference Subscription Interface

The user preference subscription interface shown in Figure 5-1 is used to register preferred feed sources to the system. The system uses user`s preferences to get feed items to execute queries at run time.

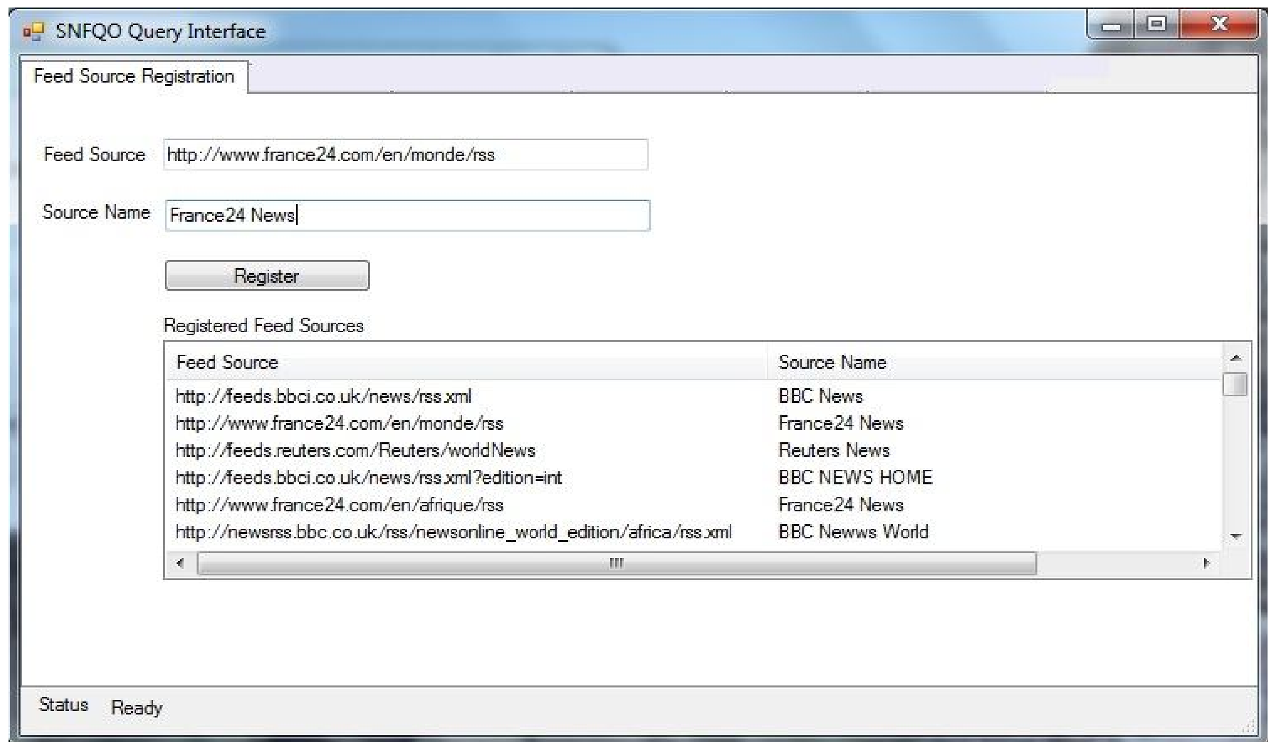

Figure 5-1: User preferences subscription interface

5.2.2. Query Definition Interface

The user query definition interface shown in Figure 5-2 is used to define user`s SNF query. To define an SNF query, first the feed source is selected from a given set of sources, and then the

attribute of the feed source, the RSS operator, search term and operator parameters are filled. The operator parameters include the equality threshold, semantic flag, K if the operator is TopK, and the item connector.

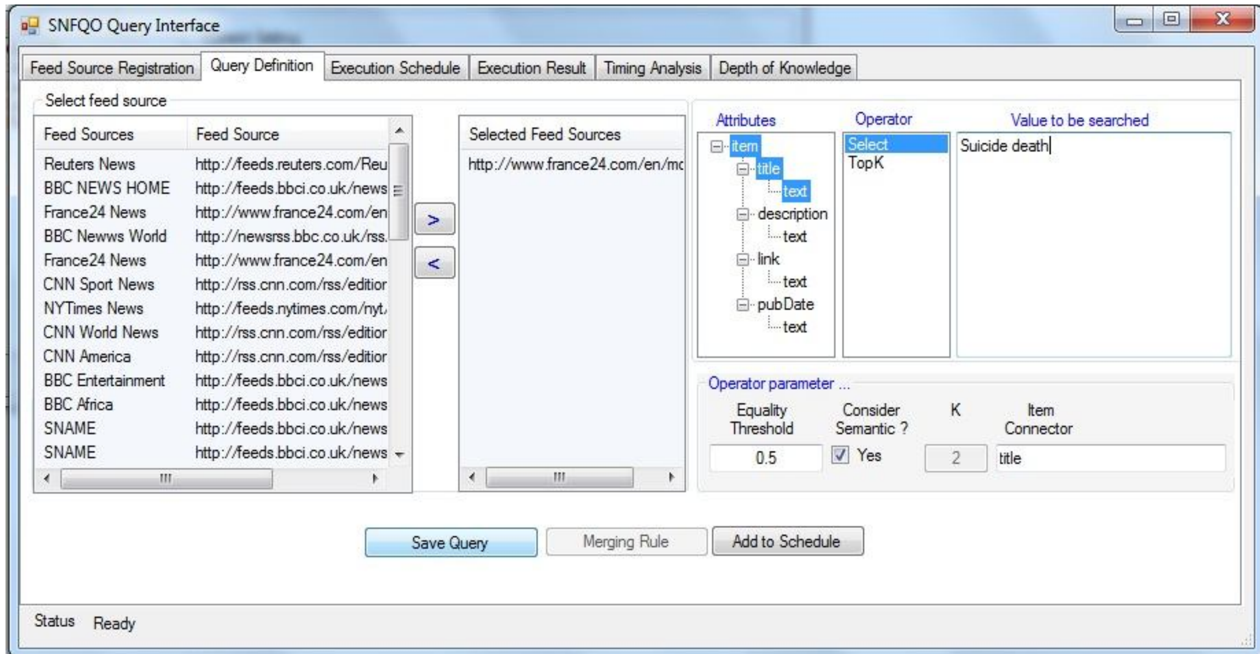

Figure 5-2: SNF query definition interface

After filling all the required information, the next step is to arrange a schedule for the given query to be executed. To do so, clicking the “*Add to Schedule”* button displays a small query scheduling window described in Section 5.2.3.

5.2.3. Query Scheduling Interface

The query scheduling interface shown in Figure 5-3 is used to schedule query. To make a schedule for the given query, the user highlights a particular square representing the day of the week and the time of day user wants the result. A given query can be scheduled more than one by highlighting more squares.

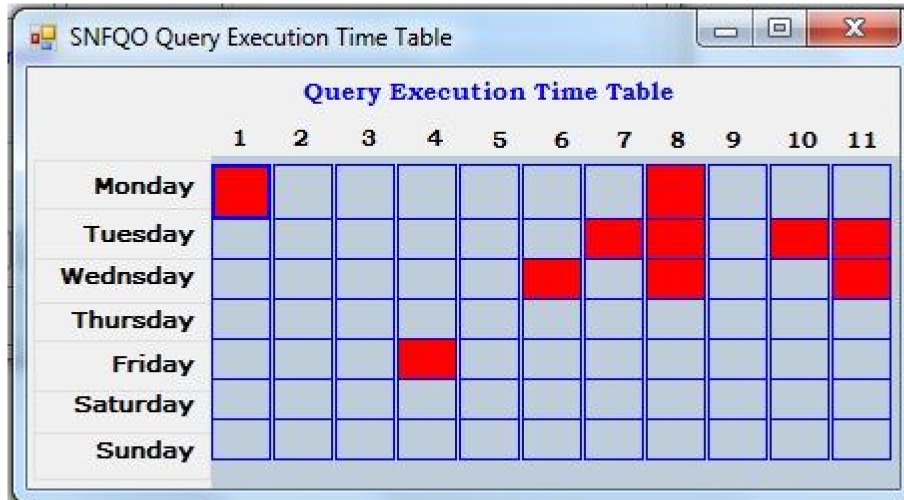

Figure 5-3: Query scheduling interface

5.2.4. Query Execution Status Displaying Interface

The interface shown in Figure 5-4 is used to view the query execution status and the weekly query execution time table. The time table displays the weekly schedule and when a particular square is clicked, it displays the query scheduled at that particular time and day of week. The user can also make a change on query schedule directly on the time table presented.

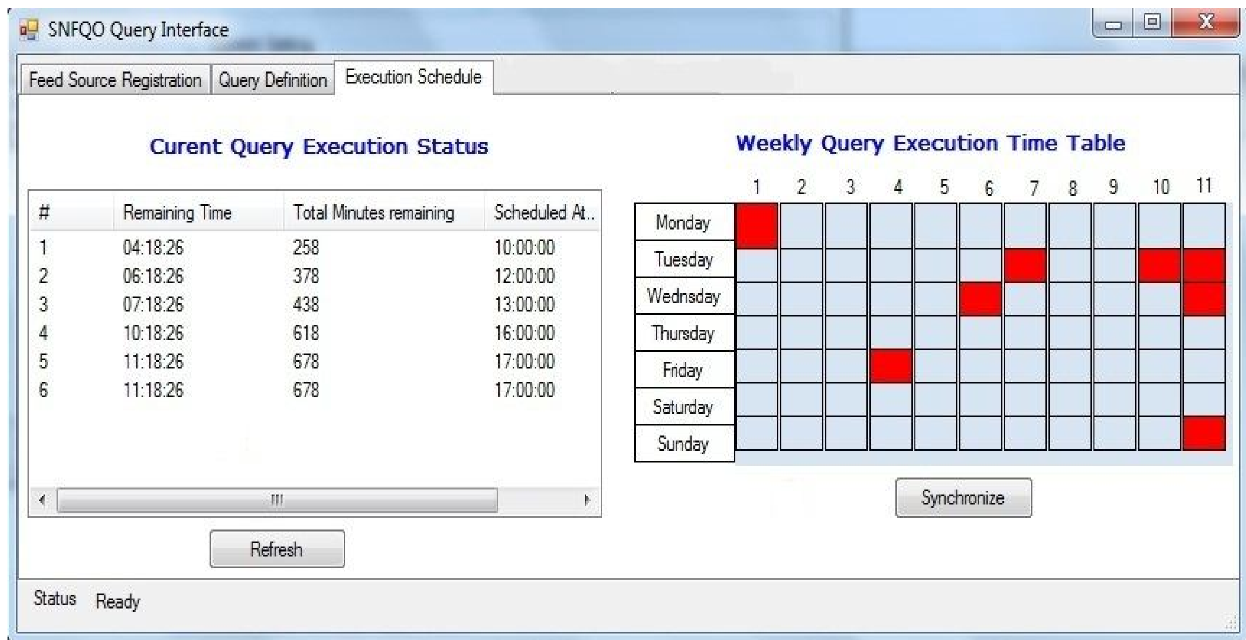

Figure 5-4: Query execution status displaying interface

5.2.5. Query Execution Result Display Interface

After the execution of a particular query, the interface shown in Figure 5-5 is used to display the query execution result.

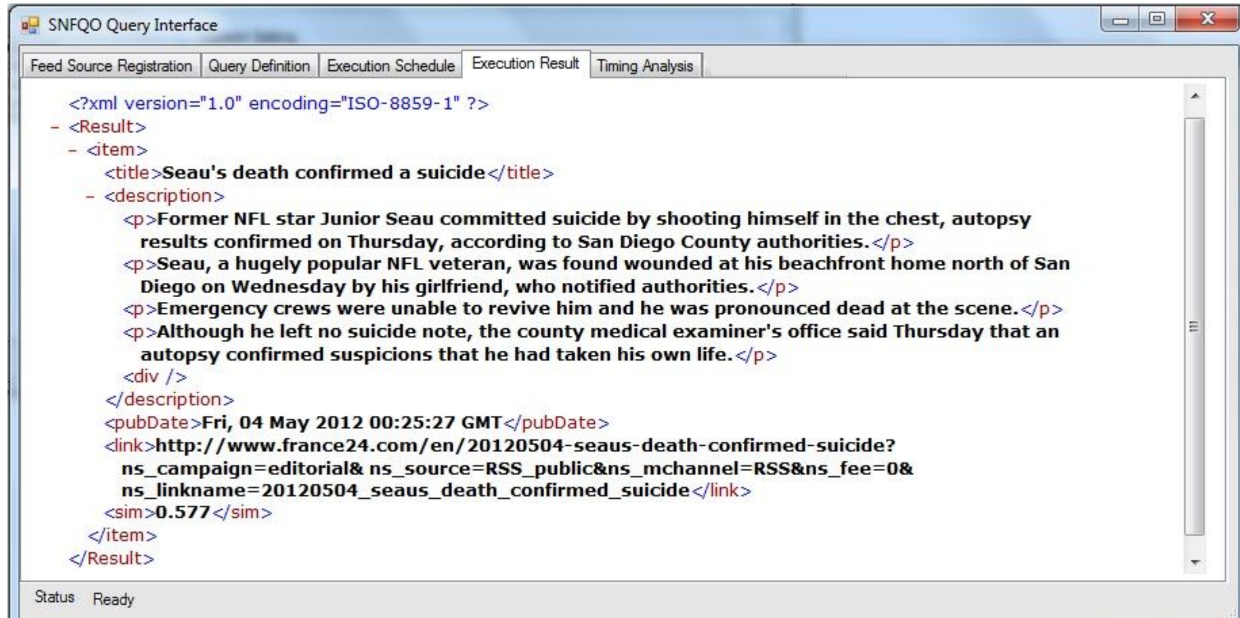

Figure 5-5: Interface for displaying query execution result

5.3 Experimental Results

The performance of the system has been demonstrated with three experiments. The first experiment is done to show how sharing a single window for multiple queries decreases the Network Access Time (presented in sub-section 5.3.1). The second experiment is done to show a performance increase of our proposed approach of MQE based on the window relationship type that exists between a set of queries (presented in sub-section 5.3.2.) and the third experiment deals on the effect of load shedding on the performance of the system(presented in sub-section 5.3.3).

5.3.1 Network Access Time (NAT)

This experiment is done to show how queries having same feed source can share a window to be executed on whether they are semantically similar or not. The experiment is performed on 50 queries and Figure 5-6 shows the network access time required to generate a window for

multiple queries (shared network access) in comparison with generation of windows for individual query (linear network access).

We made the system to process 50 queries at a time. And, the execution is on a pair [D, Y], where D denotes disjoint queries and Y denotes other relations i.e. Includes, Similar or Overlap. Accordingly, the black line shows the network access time required to generate windows of the 50 queries. The graph shows that the network access time is almost constant independent of the number of queries. However, in the case of our shared network access approach, the result shows a decrease in time as the number of disjoint queries decreases in the pair [D, Y]. Initially the pair of queries was [50, 0], in which all of the query set are disjoint; and the corresponding network access time required to generate windows is almost the same as the traditional linear approach. When the number of queries related with *Includes*, *Similar* or *Overlap* window relationship type increases and *Disjoint* queries decreases in the pair, the proposed approach utilizes semantically similar feed sources to generate a common window for queries with the same window relationship type.

Hence, the required time for shared network access decreases each time as shown in the red, blue, and green lines representing Overlap, Includes and Similar relationship type for Y respectively in the pair [D, Y].

5.3.2 MQE of SNF Queries

This experiment is done to show the performance of the system on executing multiple SNF queries together using our proposed approach and the traditional linear query execution approach. Figure 5-7 shows query execution time required to execute all the 50 queries. Using traditional approach, the required time is almost constant as shown by the pink line. However, in the case of our proposed approach to execute multiple SNF queries together, the time requirement decreases as the number of disjoint query in the pair [D, X] (D for disjoint and X for Similar, Includes or Overlap) decreases and queries with *Includes*, *Similar* or *Overlap* increases. As a result, the lines in red, green, and blue on the graph for pairs of disjoint with Overlap, Includes and Similar respectively approaches down to the origin showing a decrease in execution time and hence an increase in the performance of the system.

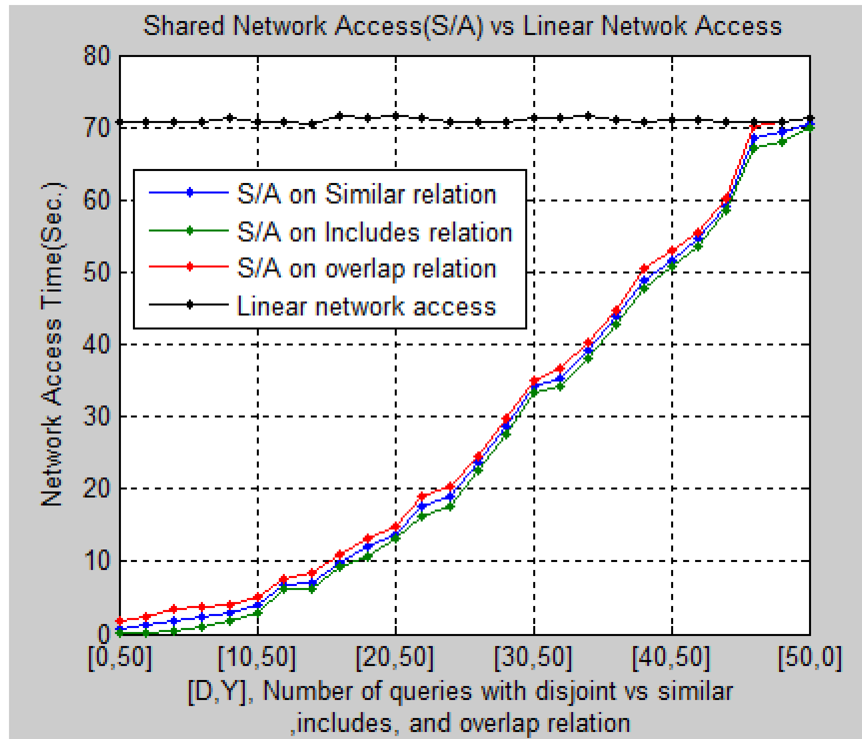

Figure 5-6: Graph for experimental result of NAT

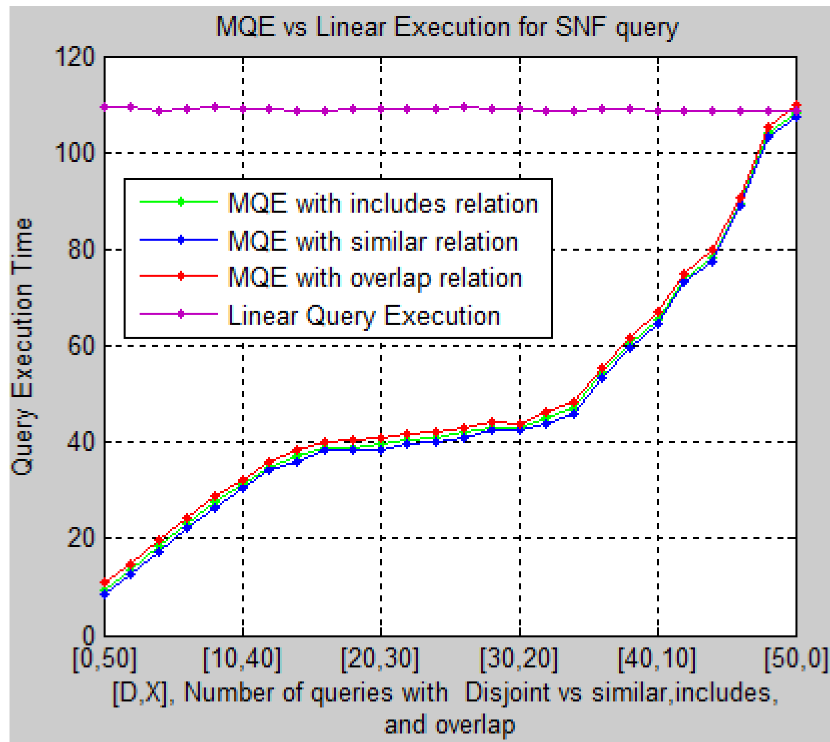

Figure 5-7: Graph for MQE timing of queries based on window relationship type

5.3.3 Load Shedding

This experiment is done to show the performance of the system before and after load shedding is applied on the incoming data. In addition, the experiment shows the performance of the system on different sampling precision level. For this experiment the 50 queries used for the above experiments were employed and with a sampling precision level (*e)* of 0.5. Figure 5-8 shows the experimental result of query execution timing before and after applying our *quickDrop* operator for load shedding. From the figure, the graph shown in blue is query execution timing before employing our *quickDrop* operator. And the graph shown in red is query execution timing after employing our *QuickDrop* operator.

As shown by the graph, initially the execution timing before using the *QuickDrop* operator was between 0.2-1.3, but after the employing our *QuickDrop* operator, the execution timing drop down to the range 0.04 - 0.3. This shows that the graceful decrease in data load by our *QuickDrop* operator decreased the execution timing and hence increased the performance of the system.

On the other hand as discussed in Section 4.2.2.3, when the precision level decreases the sample size increases according to [22] and hence the window size increases and as a result decreases the query processing performance of the system. The experimental result shown by the graph in Figure 5-9 shows the effect of decreasing the precision level from 0.5 to 0.2 on the performance of the system. Accordingly, the execution timing of queries before load shedding shown by the graph in blue remain the same and the execution timing after load shedding with a precision level of 0.2 shown in green increased and hence the performance of the system decreased as compared to the execution timing after load shedding with a precision level of 0.5. However, as the precision level increases, the error decreases and ensures the quality of query result. To keep the balance between precision level and the performance of the system, our *quickDrop* operator uses the given error threshold value. If the resulting error is above the given threshold the operator dynamically adjusts the sampling precision level to maintain the balance between the quality of query result and the performance of the system.

Finally, as shown by the experimental results, the proposed QuickDrop operator is effective to gracefully decrease the data load to increase the performance of the system while keeping the quality of query result. And by this, it meets the objective of phase two SNF query optimization.

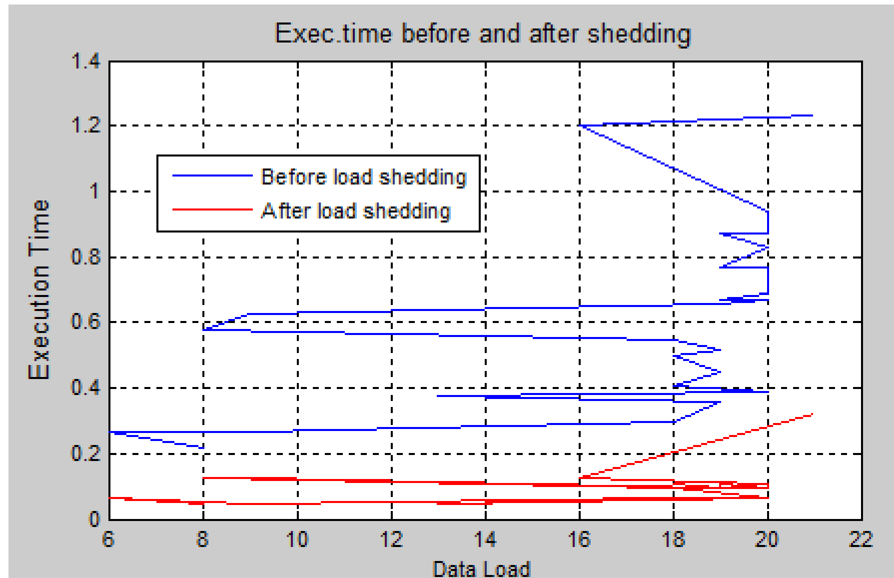

Figure 5-8: Experimental result for query execution before and after load shedding

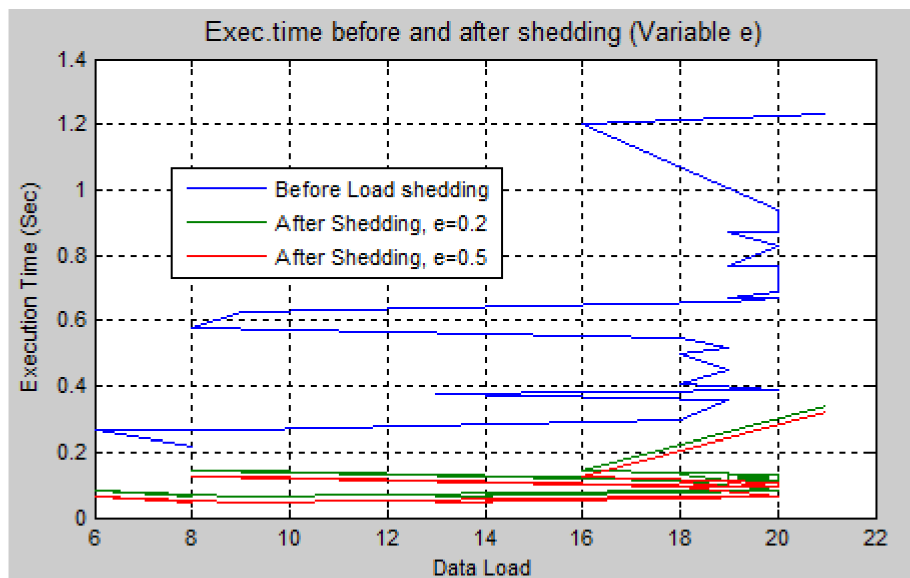

Figure 5-9: Experimental result for query execution with different precision level

CHAPTER SIX

6 CONCULUSION AND FUTURE WORKS

6.1 Conclusion

Optimization is the process of searching for best query execution plan which minimizes system resource utilization and maximizes the performance of the system among a set of alternative plans. Since the data in traditional relational databases are relatively permanent, cardinality based query optimization technique has been instrumental. But, in stream applications query optimization is a challenging task due to dynamic and time dependent nature of stream data.

SNF query processing system is one of the streaming applications which process SNF queries semantically on stream of RSS feeds. The use of semantic information in query processing increases the relevance of query result at the cost of degrading system performance. Hence, to benefit from SNF query processing on semantic information keeping the efficiency of system performance, in this research work we have proposed an optimization approach for SNF query.

The proposed query optimization approach performs the optimization in two main phases; static SNF query optimization and run time optimization. The first phase mainly performs Multi-Query optimization, and extraction of keys from a set of queries for load shedding. The proposed Multi-Query optimization strategy for semantic news feed query follows three steps to generate a MQE plan. First it evaluates the semantic similarity of a set of queries registered in profile database, and then identifies windows relationship type among queries accessing from same feed source and finally MQE chains are generated for the window relationship types identified. In addition, keys extraction is performed on a set of queries which are expected to be executed together and are used for load shedding at run time.

The second phase of SNF query optimization is run time optimization which is performed just at the time of query execution. Here the optimization is done through load shedding which drops non-relevant data from a given shared window generated by window generator component in the query engine. For load shedding, we have proposed an operator called *QuickDrop* operator. The operator `quickDrop (W, T, K, e)` takes four parameters; `W` denotes a shared window

among a set of queries for Multi-Query execution, T denotes the attribute of the news feed for shedding i.e. *title* or *description*, K denotes a set of enhanced keys and e denotes the level of precision to be used for sampling.

After the optimization phases, the Multi-Query execution is performed. Supplemented with thread based execution, the MQE is follows a set of MQE rules identified to manage the execution of a set of queries in a MQE plan.

To validate our proposal, we have developed a prototype which helps users to subscribe to their favorite feed sources and define queries, schedule queries and manage query schedule using query scheduler component, and auto execution of queries according to user`s schedule. Using the developed prototype, we have also conducted experiments to show the proposed approach`s performance increase on the system. The experiments were conducted to show how MQE helps to decrease the network access timing (NAT) required to execute multiple window sharing queries, how MQE on shared window increases the performance of the system, and the performance gain as a result of load shedding using the proposed *quickDrop* operator.

Finally, the contributions of this research work are:

- Applying semantic similarity measure as a tool to optimize multiple queries accessing XML stream
- Introducing semantic similarity measure for windows,
- Introducing the concept of window sharing technique into feed query processing systems
- Introducing the concept of chain based MQE strategy for XML stream query processing system
- Semantic load shedding scheme for stream systems accessing XML type of data.

6.2 Future Works

This research work opens up several avenues for future research directions especially in optimization of stream queries accessing XML type stream of data in data stream management system with a publish/subscribe architecture in general.

Hand-held devices have limited system resources and hence needs special attention in developing DSMS architecture which is costly. Thus, we will check to see if the proposed approach can be generalized for hand-held device based RSS readers. In addition, our approach needs to consider how to apply MQE for resource sharing for partially similar SNF queries. And this will be our next task to make our scheme full. Finally, we will enhance our load shedding scheme with the ability to decide when to shade data load.

7 REFERENCES

1. RSS Specifications, Retrieved May 12, 2012, from http://www.rss-specifications.com/What-is-rss.htm.

2. Fekade Getahun, Richard Chbeir, Semantic Aware RSS Query Algebra, In Proceedings of iiWAS'2010, Pages 291-298, 2010.

3. H. Liu, V. Ramasubramanian, and E. G. Sirer, Client Behavior and Feed Characteristics of RSS: a Publish/Subscribe System for Web Micro-News, In Proceedings of SIGCOMM`05, Pages 78-85, 2005.

4. Princeton University Cognitive Science Laboratory, WordNet: a Lexical Database for the English Language. http://wordnet.princeton.edu/.

5. Juan M., Vector based Approaches to Semantic Similarity Measures, A. Gelbukh (Ed.) Advances in Natural Language Processing and Applications Research in Computing Science 33, 2008, Pages 163-174, 2008.

6. Nauman A., Kevin S., Mahdi A., Stream Data Management, Springer, 2005.

7. F. Fabret, H.A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha., Filtering Algorithms and Implementation for very Fast Publish/Pubscribe, In Proc. SIGMOD, Pages 115–126, 2001.

8. M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid, Scheduling for Shared Window Joins over Data Streams, In Proceedings of VLDB, Pages 297–308, 2003.

9. S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman, Continuously Adaptive Continuous Queries over Streams, In Proc. SIGMOD, Pages 35-55, 2002.

10. S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson, The Case for Precision Sharing, In Proc. VLDB, Pages 972–986, 2004.

11. S. Krishnamurthy, C. Wu, and M. Franklin, On-the-fly Sharing for Streamed Aggregation, In Proc. of SIGMOD, Pages 623-634, 2006.

12. S. Viglas, J. F. Naughton, Rate-based query optimization for streaming Information Sources. SIGMOD Conference, Pages 37-48, 2002.

13. Arvind A., and Jennifer W, Resource Sharing in Continuous Sliding-Window Aggregates, Technical Report, Stabford, Pages 336-347, 2004.

14. Song W., Elke R., Samrat G., and Sudeept B., State Slice: New Paradigm of Multi-Query Optimization of Window Based Stream Queries, VLDB ‘06, 2006.

15. Mingsheng H., Alan D., and Johannes G., Massively Multi-Query Join Processing in Publish/Subscribe Systems, SIGMOD’07, Pages 147-160, 2007.

16. J. Li., David M., Kristin T., Vassilis P., Peter A. No Pane, No Gain: Efficient Evaluation of Sliding Window Aggregates over Data Streams, SIGMOD Rec., Pages 1-11, 2005.

17. Shenoda G., Mohamed A., Panos K., and Alexandros L., Optimized Processing of Multiple Aggregate Continuous queries, CIKM’11, Pages 1515-1524, 2011.

18. Moustafa A, Michael J, Walid G. and Ahmed K., Scheduling for Shared Window Joins over Data Streams, In Proceedings of VLDB '03, Pages 297-308 2003.

19. Allen and James, Natural Language Understanding, Benjamin/Cummings Publisher, New York, 1995.

20. Jiawei Han and Micheline Kamber, Data Mining: Concepts and Techniques, Second Edition, Diane Cerra, 2006

21. Dhananjay K., Chinya V., Mitch C., Real-Time, Load-Adaptive Processing of Continuous Queries over Data Streams, In Proceedings of the ACM DEBS’08, Pages 49-60. 2008.

22. T. Yamane, Statistics, an Introductory Analysis, 2nd Ed., New York: Harper and Row, 1967.

23. Fekade G., Joe T., Chbeir R., Marco V., Kokou Y., Relating RSS News/Item, In proc. of ACM ICWE '9, Pages 1644-1654, 2009.

24. Alasdair J. and Werner N., A Data Stream Publish/Subscribe Architecture with Self-Adapting Queries, In Proceeding of ACM OTM'05, Pages 38-49, 2005.

25. Ying L. and Beth P., Survey of Publish Subscribe Event Systems, Computer Science Department, Indiana University, Pages 27-38, 2003.

26. Nesime. T., Uger C., and Stan Z., Load Shedding on Data Streams, In Proceedings of VLDB`03, 2003.

27. BAI, Y., H. THAKKAR, H. WANG, A Data Stream Language and System Designed for Power and Extensibility, In proceedings of CIKM`06, Pages 336-347, 2006.

28. KORN F., B. PAGEL, and C. FALOUTSOS, On the 'Dimensionality Curse' and the 'Self-

Similarity Blessing', IEEE Trans. Knowledge. Data Eng, Pages 96-111, 2001.

29. Brian B., Mayur D., and Rajeev M., Load Shedding for Aggregation Queries over Data Streams, Proceedings of ICDE '04, Pages 232-239, 2004.

30. Rajeev M., Jennifer W., Arvind A., Brian B., Shivnath B., Mayur D., Gurmeet M., Chris O., Justin R., and Rohit V., Query Processing, Approximation, and Resource Management in a Data Stream Management System, Proceedings of the 1st Conference on Innovative Data Systems Research, 2003.

31. Sirish C. and Michael J., Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams, In proceedings of VLDB`04, Pages 439-459, 2004.

32. James F. Allen, Towards a General Theory of Action and Time, Artificial Intelligence, v.23 n.2, Pages 123-154, 1984.

33. Yun Chi, Haixun Wang, Philip S. Yu, and Richard R. Munt, Loadstar: A Load Shedding Scheme for Classifying Data Streams, In Procs. of SIAM Conf. on Data Mining, Pages 71-80, 2005.

34. N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, Load Shedding in a Data Stream Manager, In Proceedings of the 29th VLDB Conference, Pages 309-320, 2003.

35. H. Wang, W. Fan, P. S. Yu, and J. Han, Mining concept-drifting data streams using ensemble classifiers, In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, Pages 1113-1118, 2003.

36. W. Fan, Y-A Huang, H. Wang, and P. S. Yu, Active mining of data streams, In Proceedings of the Fourth SIAM International Conference on Data Mining, Pages 84-97, 2004.

37. A. Das, J. Gehrke, and M. Riedewald, Approximate join processing over data streams, ACM SIGMOD International Conference on Management of Data, San Diego, CA, Pages 40-51, 2003

38. J. Kang, J. F. Naughton, and S. D. Viglas, Evaluating window joins over unbounded streams, In Proceedings of the 19th International Conference on Data Engineering, India, Pages 341-352, 2003

39. A. M. Ayad and J. F. Naughton, Static optimization of conjunctive queries with sliding windows over infinite streams, In Proceedings of the ACM SIGMOD International Conference on Management of Data, France, Pages 419-430, 2004

40. Brill E., A Simple Rule Based Part of Speech Tagger, In Proceedings 3rd Conference on Applied Natural Language Processing (ACL), Pages 133-142, 1992.

41. M. Garofalakis and P. Gibbons, Approximate Query Processing: Taming the Megabytes, In VLDB, Rome, Italy, Pages 1-10, 2001.

42. J. Hellerstein, P. Haas, and H. Wang, Online Aggregation, In ACM SIGMOD, Tucson, AZ, Pages 171-182, 1997.

43. F. Reiss and J. Hellerstein, Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ, In IEEE ICDE Conference, Japan, Pages 81-92, 2005.

8 APPENDIX

Sample Microsoft C# Codes Taken From Prototype Development

I. Window Relationship Identifier Function

```
Public string WinRelationShip(ArrayList w1,ArrayList w2)
  {

    string w1s = w1[0].ToString();
    string w1e = w1[1].ToString();
    string w2s = w2[0].ToString();
    string w2e = w2[1].ToString();
    string relationship="";

    double s1 = DateTime.Parse(w1s).TimeOfDay.TotalMinutes;
    double s2 = DateTime.Parse(w2s).TimeOfDay.TotalMinutes;
    double e1 = DateTime.Parse(w1e).TimeOfDay.TotalMinutes;
    double e2 = DateTime.Parse(w2e).TimeOfDay.TotalMinutes;
    double TR = 0.5;
    double S = (s1 - s2) / 60;
    double E = (e1 - e2) / 60;
    double D = 0;
     if (s2 > e1){
             D = (e1 - s2) / 60;}
     elseif(s1>e2)
           {
             D = (e2-s1) / 60;}
     else{D=0}
     elseif ((S >= - TR && S <= TR) && (E >= - TR && E <= TR))
           {
             relationship ="Similar";
           }
elseif (((S < -TR) && (E >= -TR && E <= TR)) || ((S >= -TR && S <= TR)   &&
              (E > TR))||((S<-TR&&E>TR)||(S>TR&&E<-TR)))
{
            relationship="Includes";
}
elseif (((S > TR) && (E >= -TR && E <= TR)) || ((S >= -TR && S <= TR) && (E <
              -TR)))
      {
            relationship="Included_In";
      }
elseif (D < -TR||D==0)
      {
            relationship="Disjoint";
      }
elseif (((S < -TR) && (E < -TR)) || ((S >TR) && (E >TR))) {
            relationship="Overlap";
      }
return relationship;}
```

II. *quickDrop* Operator

```
publicArrayListquickDrop(ArrayList WNFeed,stringT, string[] K, double e)
        {
ArrayList newWindow = newArrayList();
ArrayList nonRelevant = newArrayList();
ArrayList text = WNFeed;
ArrayList searchTerm = newArrayList();
ArrayList result = newArrayList();
TFIDFMeasure.noSyn = 1;
TFIDFMeasure.depth = 0;
int error;

//Enhancing search key
for (int q = 0; q < pred.Count; q++)
    {
     result = tf.SemanticNeighborhoodAtDistance(pred[q].ToString());
     TFIDFMeasure.concept_D cd = newTFIDFMeasure.concept_D();
    for (int i = 0; i < result.Count; i++)
       {
        cd = (TFIDFMeasure.concept_D)result[i];
        searchTerm.Add(cd.concept.Replace(' ', ',').Replace('_', ' '));
       }
    }
//classification into relevant and non-relevant classes
  for (int i = 0; i < text.Count; i++)
     {
       string[] source = text[i].ToString().Split(newchar[] { '.', '?', '!',
        ' ', ';', ':', ',' }, StringSplitOptions.RemoveEmptyEntries);
      for (int j = 0; j < searchTerm.Count; j++)
          {
           string st = searchTerm[j].ToString();
           var matchQuery = from word in source
           where word.ToLowerInvariant() == st.ToLowerInvariant()
           select word;
           int wordCount = matchQuery.Count();
     if (wordCount > 0)
       {
        if (newWindow.Contains(Owind[i].ToString()) == false)
           {
            newWindow.Add(Owind[i]); // relevant class
           }
       }
     elseif(nonRelevant.Contains(Owind[i])==false)
           {
            nonRelevant.Add(Owind[i]);
           }
       }
     }
        // sample selection
        int lessCount = nonRelevant.Count;
        int sampSize;
        sampSize = (int)((lessCount) / (1 + (lessCount * e * e)));
        // sampling from less relavant class
        error = (lessCount - sampSize) / lessCount;
        ArrayList randList = newArrayList();
        int randNo;
```

```
        Random randObj= newRandom();
// generating random number and assign each element in less relevant class
     for (int k = 0; k < lessCount; k++)
     {
     randNo=randObj.Next(1,lessCount);
     randList.Add(randNo);
     }
//select the first top sampSize from randList
   ArrayList indxList=newArrayList();
   int coun=0;
   int indxx = 0;
    while (coun < sampSize)
      {
       for (int l = 0; l < randList.Count; l++)
          {
          if (indxList.Contains(l) == false)
             {
            for (int m = l + 1; m < randList.Count; m++)
                 {
                  if ((int)(randList[l]) >= (int)(randList[m]))
                      {
                         indxx = l;
                      }
                  else
                      {
                         indxx = m;
                      }
                 }
             indxList.Add(indxx);
              }
            }
          coun++;
       }
         // merge the sample with relevant class
        for (int j = 0; j < indxList.Count; j++)
            {
            if (newWindow.Contains(nonRelevant[(int)(indxList[j])]) == false)
               {
                newWindow.Add(nonRelevant[(int)(indxList[j])]);
               }
            }
  return newWindow;
 }
```

Publication

International Conference:

We have published our multi-query optimization approach in an international conference (ANDINET ASSEFA and FEKADE GETAHUN, Multi-Query Optimization for Semantic News Feed Query, In Proceedings of MEDES`12, Pages 150-157, 2012.)

International Journal:

We have also submitted to the international journal for a Special Issue on Advanced Data Stream Management and Processing of Continuous Queries in Transactions on Large-Scale Data and Knowledge Centered Systems (TLDKS) (ANDINET A. and FEKADE G., 2013).

Declaration

I, the undersigned, declare that this thesis is my original work and has not been presented for a degree in any other university, and that all source of materials used for the thesis have been duly acknowledged.

Declared By:

Andinet Assefa _______________ _____________

Student Signature Date

Approved By:

Fekade Getahun (Ph.D.) _______________ _____________

Advisor Signature Date

Place and date of submission: Addis Ababa, Ethiopia

April, 2013