ENHANCING STREAM REASONING BY MODELING THE IMPORTANCE OF THE STREAMING DATA

By

Rui Yan

A Dissertation Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: COMPUTER SCIENCE

Examining Committee:

_____

Deborah L. McGuinness, Dissertation Adviser, Committee Chair

_____

Peter Fox, Member

_____

James Hendler, Member

_____

Emanuele Della Valle, Member

_____

Mark T. Greaves, Member

Rensselaer Polytechnic Institute
Troy, New York

Spring 2018
(For Graduation Spring 2018)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

This journey has been long.

ABSTRACT

The requirement to extract the hidden information out of the data stream is rising, however, traditional stream processing systems cannot meet this requirement as they are not designed to do so. This gives birth to the new research domain of stream reasoning that aims to bring semantic reasoning into stream processing. An example is to predict highway traffic jam, given the explicit sensor data streams of cars' number and speed. It is very easy for humans to observe the traffic then forecast a traffic congestion. This is because humans know that a bigger car number and slower car speed can usually lead to a traffic jam. Unfortunately, machines do not. What they can "see" is probably a sequence of numerical numbers that are separated by commas.

Streaming data is boundless, enormous, and heterogeneous, which adds extra dimensions to the challenges of realizing the vision of stream reasoning, in addition to temporal constraints. A widely-adopted way to process the streams is via leveraging a window that isolates the latest streaming portion. This snapshot, mostly managed by the first in first out (FIFO) strategy under a popular silent assumption that the latest data is the most important, is all that a window can know about the stream. This inevitably provides only limited information during the processing. However, modeling the importance of the data is not necessarily based on pure arrival timestamps. If the latest data does not convey the necessary information to answer the query, there is surely no need to do anything other than evicting it.

Streaming data intrinsically has many different orderings, such as temporarily, precision, provenance, and trust, etc. If diverse data orderings can be utilized to model the data importance, stream reasoning can be benefited by being data-discriminative. It is able to understand the concept of importance so as to identify, and leverage more important data that are crucial to the query answering, which can improve the system performance. The notion that models the data importance is named as semantic importance. It is an umbrella-like concept with multiple branches, such that each branch models one aspect of currently included data or-

derings. The combinations of different branches describe the data importance, and enable various smart and flexible window management strategies that are previously dominated and limited by FIFO.

Generally speaking, this dissertation delivers a conceptual model, and a set of infrastructure that can facilitate its general application in stream reasoning. Specifically, the first contribution is an innovative notion of semantic importance. It is formalized in an ontology, represented in a priority vector, and works with carefully extended window semantics. The second contribution introduces a general sequential stream reasoning architecture, with the purpose of both showing how semantic importance can be used in stream reasoning systems, and providing pragmatic performance metrics to configure stream reasoning systems in different scale scenarios. Two exemplar real world use cases are implemented and evaluated based on this architecture and semantic importance. The third contribution proposes a generalization and benchmark framework for semantic importance. This part focuses on how to reuse and benchmark semantic importance in a generic and quantitative way. The semantic importance is generalized by connecting itself to the state of the art stream reasoning techniques. This framework also provides a benchmark interface compatible with a wide range of continuous queries, ontologies, data streams, and a set of built-in data-aware window management strategies enabled by semantic importance. The key performance indicators recorded for the benchmark includes precision, response time, memory consumption and throughput. The results are analyzed and visualized so as to facilitate decision-making on how to compose and deploy the suitable semantic importance in real use cases.

CHAPTER 1
INTRODUCTION

Streaming data is exploding. Applications such as Facebook and Twitter are constantly streaming the data that can be consumed by a wide range of streaming applications. A key task for almost all stream processing systems, from moment to moment, is to figure out which data to remember and which to forget, in a computationally efficient manner. This dissertation approaches this task with the concept of "data important". The core contributions include an extensible conceptual model called **semantic importance**, as well as a set of infrastructure for its general applications in stream reasoning contexts, as shown in Figure 1.1.

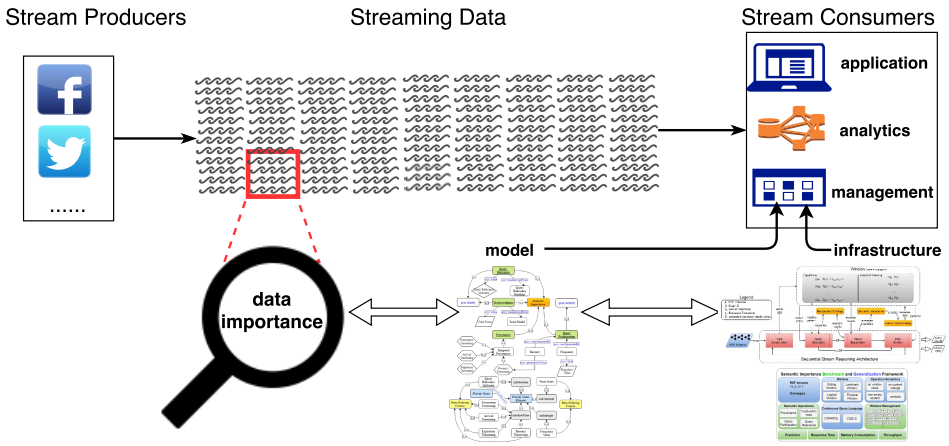


Figure 1.1: Dissertation Vision

Streaming data is boundless, enormous, and frequently updated [1]. It usually has intrinsic orders. For example, when a sequence of streaming data items arrive at the system, they are stamped with arrival timestamps. This arrival ordering is only scratching the surface, because there are many other data orderings. Data can be ordered by when it is expired, where it comes from, how much it can be trusted, how precise it is, how relevant it is to the query, etc [2].

Stream reasoning, defined as "logical reasoning in real time on gigantic and inevitably noisy data streams in order to support decision process of extremely large number of concurrent users" [3], is a newly identified research area that aims to

bridge semantic reasoning with stream processing, so as to extract hidden information out of the streams. Stream reasoning processes RDF streams [4], defined as "an infinite ordered sequence of data items" [5], that model data streams by extending RDF. The data items in RDF streams can be either triples or graphs, annotated by time-point timestamps or time-interval timestamps.

Stream reasoning has two key notions [3], the sliding window [6] and continuous processing [7]. A sliding window, or more generally a window, is a finite subset of the data stream, and manages the data. Continuous processing repeatedly evaluates the data in the window as the window proceeds. The sliding window has two parameters, a size and a step [8]. The size indicates how much data an active window can hold. The step describes how far a window can move at one time. According to the defined sliding window semantics [9], a sliding window's size and step should be both fixed over the same stream.

Windows have different types [4]: a logical or temporal window has its step and size defined in the time domain; a physical or tuple window has a tuple-based step and size. Windows can also be categorized based on the relationship between the size (l) and step (d): if $d < l$, it is a sliding window; if $d = l$, it is a tumbling window; if $d > l$, it is a sampling window [10]. The behavior of a window is modeled by window operational semantics [9] [11]. It includes four aspects: scope, content, report and tick. Scope not only determines the boundary of an active window, but also affects the starting time (t_0) of the continuous processing. Content describes the subset of the data stream that is currently in an active window. Report defines the condition to fire the query. Common policies include "on content change", "on window full", "non-empty content" and "periodic". Tick defines how and when to react to the input, which can be either tuple-driven or time-driven.

1.1 The Silent Temporal Assumption

Figure 1.2 shows a data stream being processed by two windows at t_1^{sys} and t_2^{sys} time respectively. Data is processed under a popular window management strategy called first in first out (FIFO), such that the window will have to evict the oldest data item A in order to consume the latest data item B at t_2^{sys}. FIFO is based on

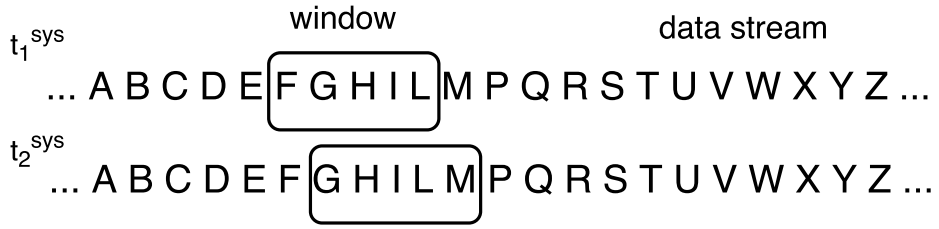


Figure 1.2: The Silent Temporal Assumption

the **silent temporal assumption** that "older information becomes irrelevant at some point" [3] [8]. A lot of stream reasoning work is based on this assumption. For example, "... new items are often more accurate and relevant than older items ..." [12]; "... recent items are typically more relevant than older ones ..." [13].

FIFO is computationally efficient, but ultimately problematic depending on the window size and streaming data characteristics. This dissertation will show the problems related to the silent temporal assumption, as well as the approaches to solve the problems by loosening this assumption.

1.2 A Running Example

In general, not all of the data items in streams can be used to answer the query [14]. There are situations where a single data item contains sufficient information (Situation 1), or multiple data items are required (Situation 2), so as to provide query answers. For the sake of convenience, this dissertation uses a term called **necessary data item** to refer to the data items that contain necessary information for question answering.

The FIFO window management strategy based on the silent temporal assumption can work well in Situation 1. For Situation 2, the answer will only be returned when all of the necessary data items exist in the window at one time-point. However, the distance among necessary data items can be arbitrarily apart, and if this distance is longer than the window size, false answers will be generated. In order to illustrate this problem, this dissertation uses a running example of real-life soccer offside offence detection.

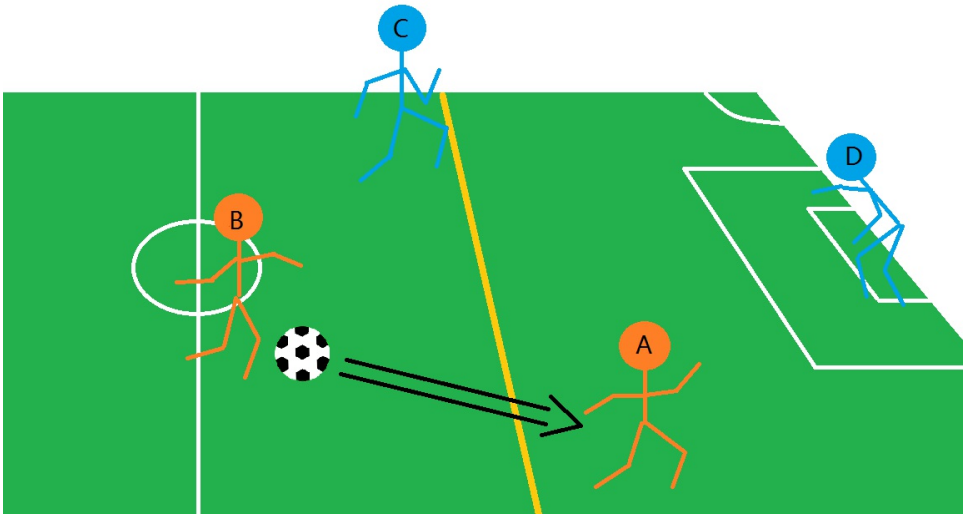


Figure 1.3: A Soccer Offside Example

Soccer offside offence is a common offence, which easily decides the direction where the game goes, thus can be very controversial sometimes. Fédération Internationale de Football Association (FIFA) defines soccer offside offence in a comprehensive way [15]. In this dissertation, a simplified definition will be used: a soccer offside offence is committed if an attacker who is at an offside position involves in an active play. So there are three atomic data items needed in order to determine an offside offence: an attacker who is a member of a team that controls the ball; this attacker is at an offside position that is nearer to the opponent's goal line than both the second last defender and the ball; and this attacker involves in an active play, meaning that he/she either touches the ball or challenges an opponent. Figure 1.3 shows a soccer offside offence example, where Player A and B are attacking. Player A is at an offside position indicated by the yellow line. The moment that Player A becomes the first person to touch the ball passed from B is when an offside offence is committed.

Reasons to use this running example are two-folds. First, situations in a soccer game can change all the time, thus a soccer offside offence can happen anytime in a very fast period. Second, the three atomic data items to determine a soccer offside offence will not arrive at the same time. This is because time will elapse as the ball

is passed from Player B to Player A, and some other data will be generated during that time, such as players' positional data.

Table 1.1: Soccer Streaming Data Items

icon	annotation	example	data sample
×	position	$\times_A$ $\times^{\circ}_A$	:PlayerA :hasPosition :PositionA :PlayerA :hasPosition :OffsideOffencePosition
○	active play involvement	$\bigcirc_A$	:PlayerA a :BallLastToucher
△	attacker	$\triangle_A$	:PlayerA a :Attacker
□	defender	$\square_A$	:PlayerA a :Defender

The data stream used for the running example throughout this dissertation contains four different types of data items in Table 1.1. × indicates a player's positional data; $\times^{\circ}$ denotes the offside position of a player; △ refers to an attacker; □ describes a defender; ○ represents that some player involves in an active play, such as touching the ball, challenging an opponent, etc.

Together with these denotations, and the logic and-operator ($\wedge$), that Player A commits an offside offence can be expressed as $\triangle_A \wedge \times^{\circ}_A \wedge \bigcirc_A$. Players on the field move from time to time, which will make offensive transitions happen at any moment. Any player can involve in an active play. Thus these four kinds of the data items can form a data stream that reflects the situations on the field. This example also assumes that a stream reasoning application consumes this data stream and constantly looks for players who commit offside offences.

1.3 Early Eviction

Consider the situation in Figure 1.4: at t_1 time, red team is attacking. Player B is passing the ball to Player A who is at an offside position indicated by the yellow line. At t_2 time, Player A is the first one to get the ball and about to take a shot. At this moment, the linesman flags to notify the referee that Player A commits an offside offence.[1]

[1]Conventionally, linesman usually flags if he/she thinks Player A will be the first one to touch the ball so as to help save some stamina.

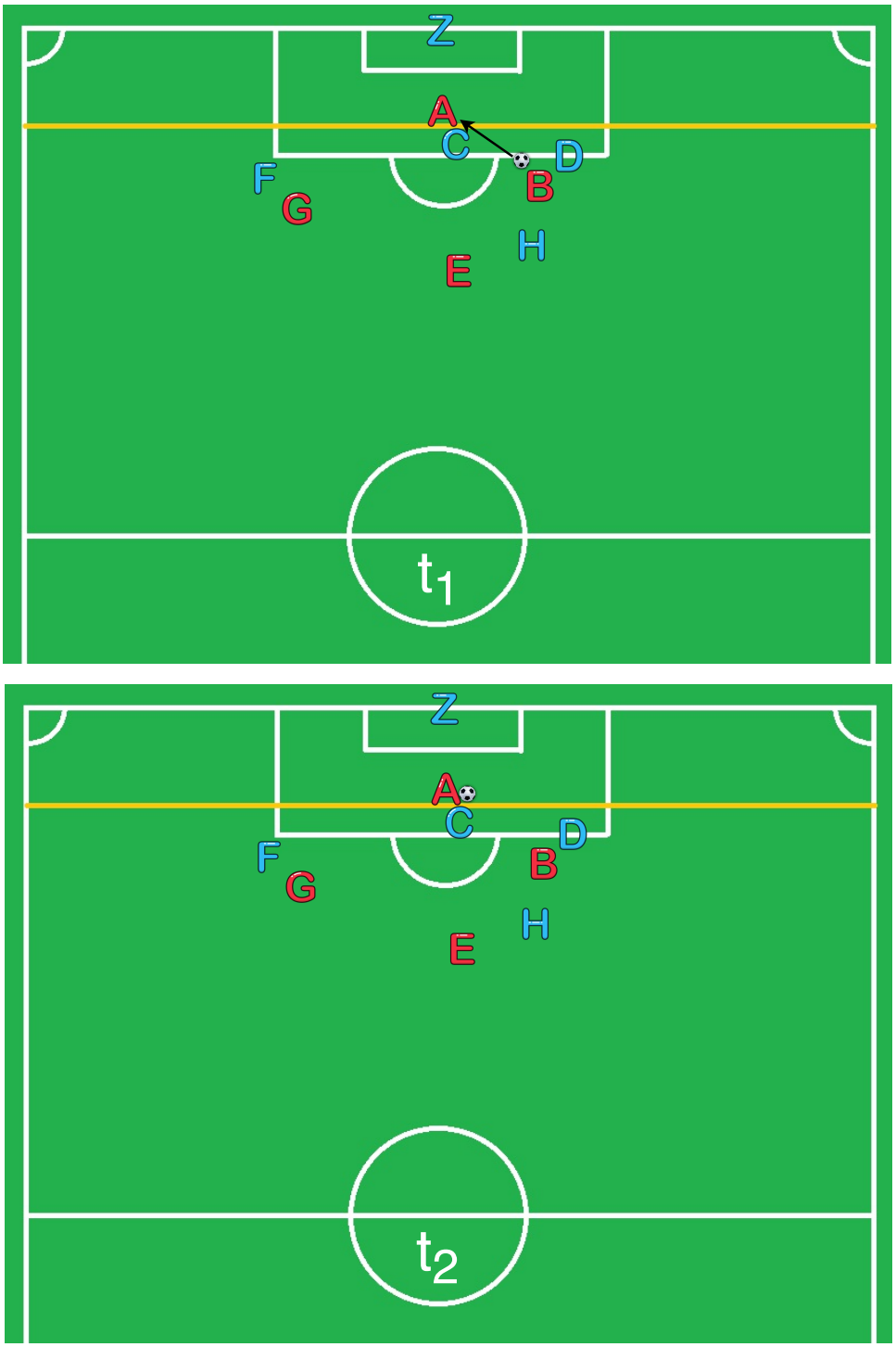


Figure 1.4: Soccer Offside Early Eviction

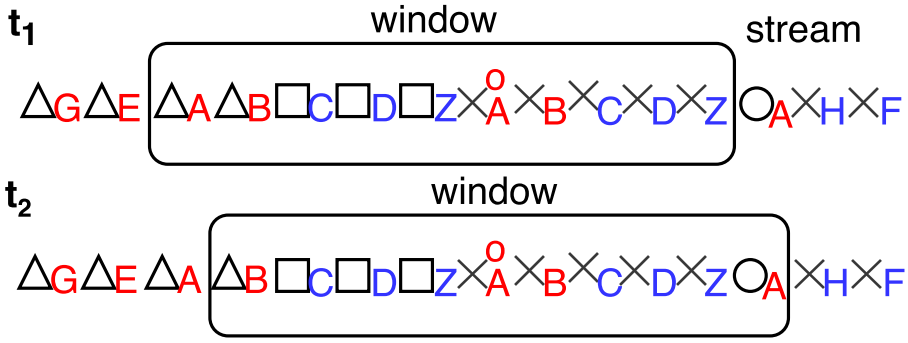


Figure 1.5: Early Eviction in Sliding Window with FIFO

Figure 1.5 shows what is happening in the window. At t_1, $\triangle_A$ and $\times^{\circ}_A$ are both in the window. $\bigcirc_A$ is about to enter the window. However, in order to consume $\bigcirc_A$, $\triangle_A$ has to be evicted because of the silent temporal assumption. Thus at t_2, only $\times^{\circ}_A$ and $\bigcirc_A$ exist in the window. In either time-point, there is no way to have all three of the necessary data items in the window, causing a false-negative answer. Namely, there should be an offside offence but the system fails to detect it. This problem is called **early eviction**: necessary data items are evicted too early before they can be used to answer queries.

There is a naive solution to the early eviction problem: to increase the window size such that it can hold $\triangle_A$, $\times^{\circ}_A$, and $\bigcirc_A$ in the same active window. Nonetheless, increasing the window size will possibly result in having more data within one window, which can drain more system memory and computation resources. Increasing window size based on the exact distance among all necessary data items is also not feasible, as in general data items can be arbitrarily apart, and one cannot foretell these distances.

1.4 Early Expiration

Figure 1.6 shows another possible situation during a soccer game. Again at t_1, red team is attacking, and Player B is passing the ball to Player A who is at the offside position. During t_1 to t_2, what happens is that Player C successfully steals the ball. Then Player A immediately challenges Player C in order to grab the ball back. At the moment when Player C steals the ball, there is an offensive transition

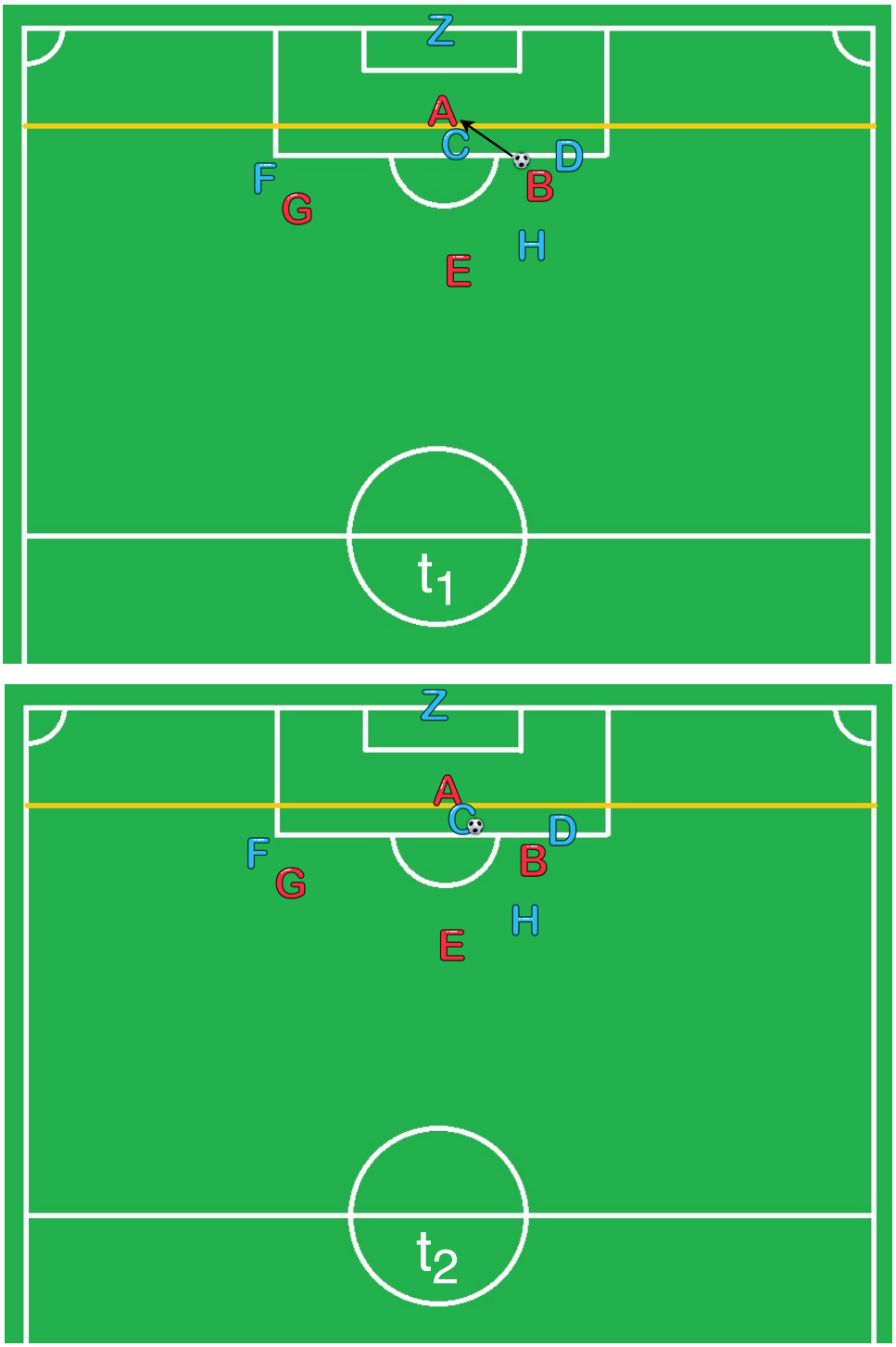


Figure 1.6: Soccer Offside Early Expiration

going on. The blue team is now controlling the ball, thus the red team becomes defensive. According to our simplified definition of soccer offside offence, a defender cannot commit an offside offence. Thus, the linesman doesn't take any action and let the match continue.

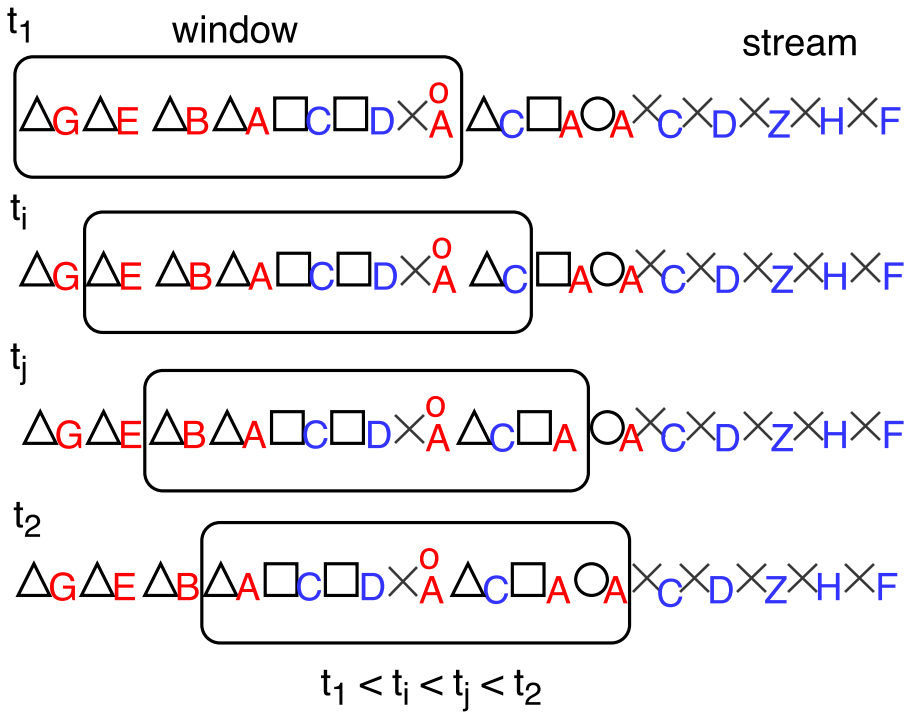


Figure 1.7: Early Expiration in Sliding Window with FIFO

Figure 1.7 shows how sliding window processes the soccer streaming data from Figure 1.6. As the sliding window moves, $\triangle_G$ and $\triangle_E$ are successively evicted to make room for $\triangle_C$ and $\square_A$. At t_i, there is already a contradictory: to the window, Player C is both a defender and attacker. Although $\triangle_C$ and $\square_C$ carry different arrival or generation timestamps, these timestamps are only annotations. For stream reasoning engines such as C-SPARQL [16], the continuous query will be parsed into a SPARQL query that is totally blind about the timestamps. At t_j, there is a same contradictory for Player A who is both a defender and an attacker. As time reaches t_2, $\bigcirc_A$ arrives. So all the necessary data items, $\triangle_A$, $\times^{\circ}_A$, and $\bigcirc_A$ are in the window, resulting a false-positive answer, i.e. there should not be an offside offence but the system reports positively. As a matter of fact, $\triangle_A$ should be

expired as $\square_A$ arrives. Because there is a transition already happened, the role of Player A is changed from attacker to defender. However, due to the silent temporal assumption, the FIFO strategy will not evict $\triangle_A$ until after t_2. This problem is called **early expiration**: the data item expires within the window, potentially causing inconsistency or leading to a wrong query answer depending on whether this data item is necessary to answer the query.

1.5 Research Hypotheses

This dissertation is built upon the research hypothesis (RH) that *semantically driven data orderings can effectively capture the data importance. By leveraging data importance modeled from various streaming data orderings, a window in the stream reasoning system can manage data in a smart and flexible way, by which the system precision, response time, memory consumption and throughput can be improved.* This hypothesis can be represented as a diagram shown in Figure 1.8. Specifically, this hypothesis can be decomposed into following four sub-hypotheses:

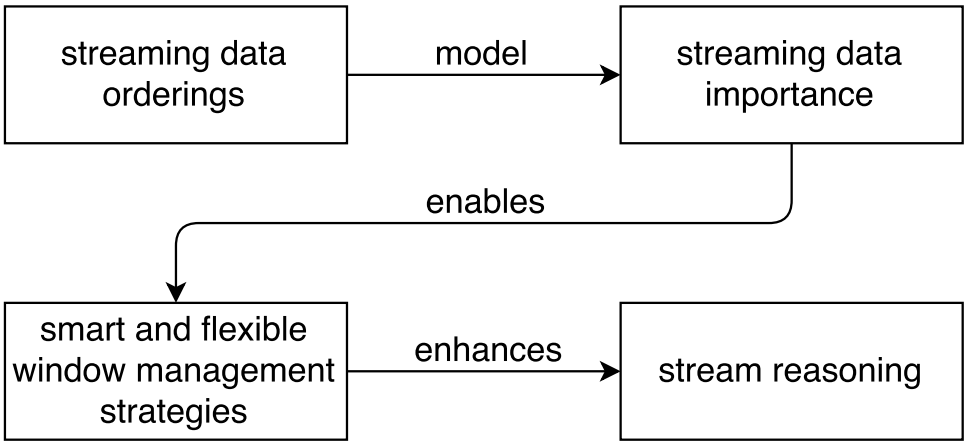


Figure 1.8: Research Hypothesis

RH1: *data orderings can model data importance.* Semantic importance is stated by observing the fact that streaming data usually has intrinsic orderings [2]. This not only includes temporal orderings such as arrival/expiration timestamps, but also precision ordering, trust ordering, geo-location ordering, relevance ordering, etc. This dissertation hypothesizes that all of these data orderings can be leveraged to

model the data importance. For example, a data item can be important if it arrives later, or has a bigger trust score, or both.

RH2: *data importance can enable window management strategies.* The model of data importance can be used as the window management strategies, which control how window consume, manipulate, query and evict the data. FIFO is a common strategy under the silent temporal assumption. A semantic importance enabled window management strategy possesses the ability to discriminate the data based on its importance. Consider if the data itself carries expiration timestamps, the window should pay attention to the data expiration within the window to guarantee valid results. In this setting, "not expired" should be seen as more important. An expiration-priority window consumes the most recent data, then evicts all the expired data, then executes the query on it. It should also take care of the data expiration during the query execution process.

RH3: *window management strategies can improve stream reasoning system performance.* Stream reasoning systems are born with the time constraints, which hinders real-time and correct outputs. A smaller-sized window holds less data to process, and possibly faster response time, but can potentially provide worse results due to being incapable to hold enough data, especially under the FIFO strategy. However, if the window can distinguish the right data that can contribute to the query results according to the data importance, the window size can be set to only hold that amount of necessary data. In this way, less but enough data for correct results needs to be stored, less computations need to be executed, and more response time is saved, thus the system is able to process more data items within a unit time.

RH4: *different window management strategies can produce different results.* Window management strategies are different, which causes the data in the window to be different, even for the same stream. The query will be executed on this data, thus the results will be different. This hypothesis emphasizes the suitability between the use case requirements and the window management strategies. It is critical to deploy suitable strategies in different use cases to maximize the benefits.

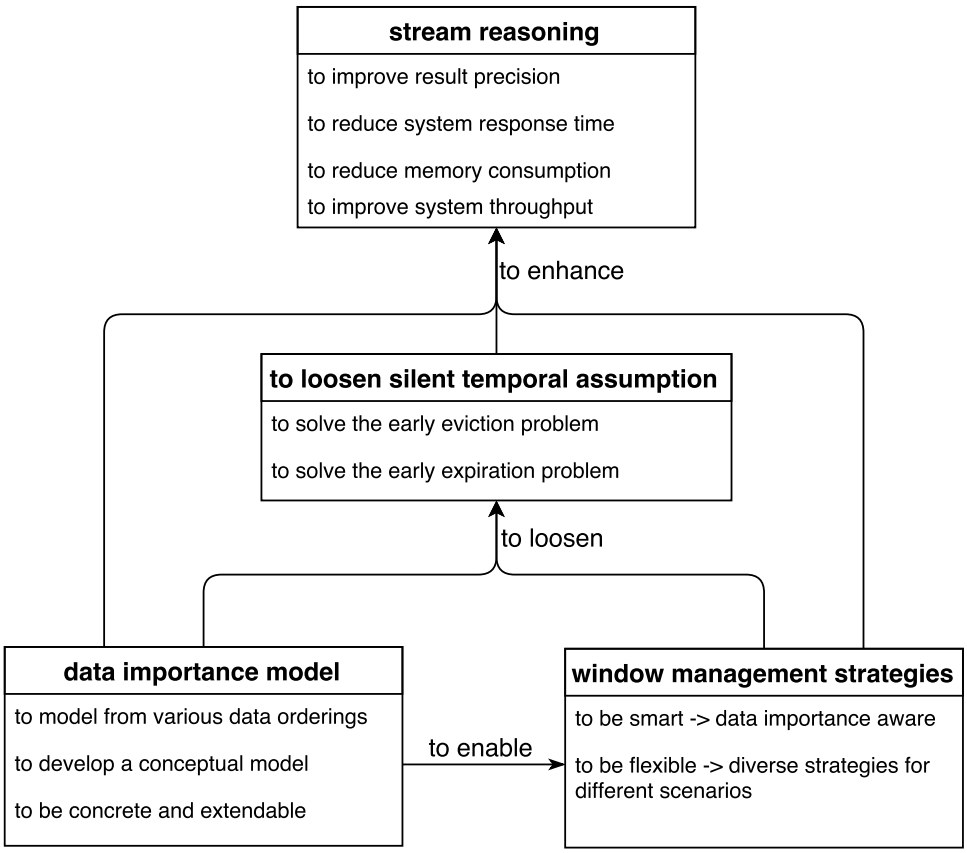


Figure 1.9: Research Goal

1.6 Research Goal

This dissertation aims to achieve the goal in Figure 1.9. This dissertation targets to enhance stream reasoning by loosening the silent assumption via explicitly modeling the importance of the streaming data to the query at hand. In order to do so, the dissertation will first propose a data importance model developed from various streaming data orderings. With this data importance model, smarter and more flexible window management strategies will be enabled. The silent temporal assumption can then be loosened by both the model and strategies. Together with the loosened silent temporal assumption, the data importance model and window management strategies, stream reasoning can be enhanced.

Specifically, this work will enhance stream reasoning from four perspectives, to improve system precision, to reduce response time, to reduce memory consumption, and to improve system throughput. "By loosening the silent assumption" refers

to solving the early eviction and early expiration problems. In order to explicitly model the streaming data importance, this dissertation proposes a concrete and expendable semantic conceptual model of data importance and enables smart and flexible window management.

1.7 Research Questions

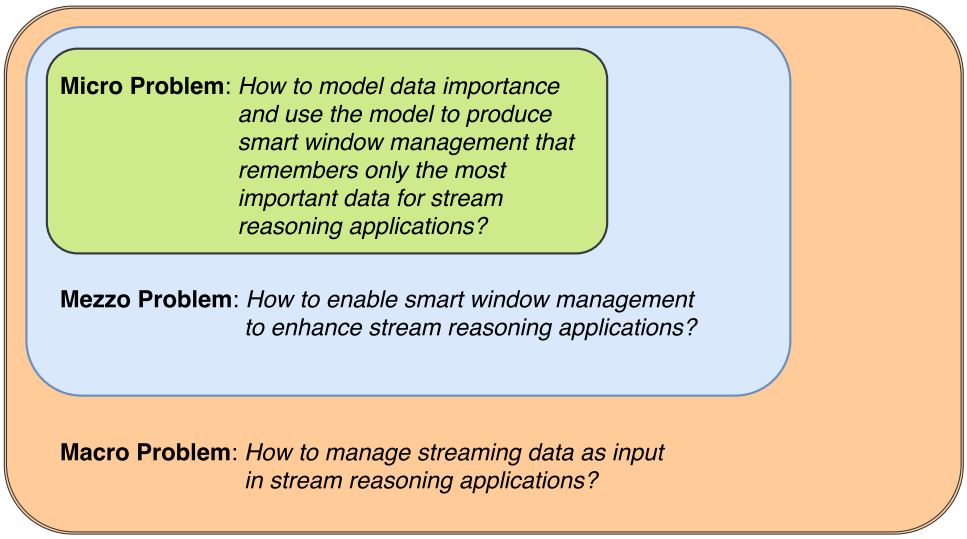


Figure 1.10: Research Questions

In general, this dissertation targets on streaming data management for stream reasoning. As mentioned in previous sections, the silent temporal assumption and FIFO management strategy are neither adequate to deal with problems of early eviction and early expiration. After all, these two problems influence the performance of the stream reasoning systems. So, a more concrete question is how to enable windows to manage streaming data in a both smart and efficient way, such that the stream reasoning systems can be enhanced. In order to answer this question, this work needs to define what a smart window management refers to, then how to enable this smart management. Figure 1.10 displays the three research questions in a micro-mezzo-macro [17] fashion.

The micro problem itself has defined what a smart window management is: to remember only the most important data. In order to enable this smartness, the system has to know what the data importance is about, and how to rank the data

based on its importance. Thus the problem boils down to model data importance that can be used during window managing streaming data.

The mezzo problem is a level more general than the micro problem. What it asks for is that if there is a solution for the micro problem, how to use this solution to enhance stream reasoning applications, and how to prove that this solution is general in a way such that it can be adapted into different scenarios and use cases. The window should know which data is more important than others. By only keeping the important data, window size can be shrank while the ability to answer the query is not influenced. Keeping a smaller window size means smaller amount of data, which saves memory and reduces query execution time.

The macro problem is how to manage the torrent, heterogeneous and large-scale streaming data for processing and reasoning. This problem is about the general vision of stream reasoning, including aspects of data model, query model, window operator, data processing etc, thus can be used to provide future research directions of this dissertation.

1.8 Dissertation Contributions

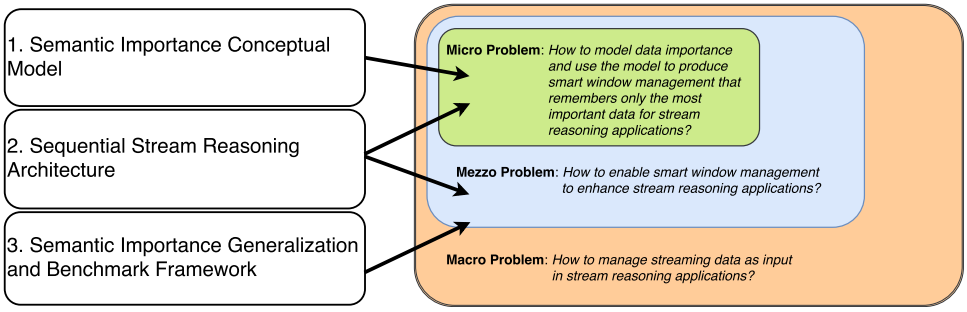


Figure 1.11: Dissertation Contributions

This dissertation claims three contributions. Figure 1.11 shows each of them and how they can be mapped to help answer the research questions.

Semantic Importance Conceptual Model The notion of "semantic importance" is proposed to describe the data importance from various streaming data orderings. Semantic importance currently includes four aspects,"provenance", "query participation", "trustworthiness", and "query relevance". These four aspects pro-

vide a preliminary set of metrics according to which different order-aware window management strategies can be constructed. Each data item is associated with a semantic importance indicator that is embodied as a mathematical priority vector. Priority vectors are comparable thus data can be ranked. With this ranking, more important data can be preserved while less important data will be evicted, no matter how different orderings are chosen as semantic importance to be deployed in the system.

Sequential Stream Reasoning Architecture Semantic importance is a new concept. Most existing stream reasoning applications are not compatible with semantic importance. However, in order to show how to deploy it and how many benefits it can bring, the sequential stream reasoning architecture has been designed and implemented. This architecture has one window design, and four components arranged in a sequential work-flow: (1) a data consumption component that consumes streaming data and sends it to the streaming window; (2) a query execution component that evaluates a standard SPARQL query against the data in the triple-store and background ontology; (3) a result explanation component that uses the query and reasoning process to explain how the result is generated; (4) a data eviction component that evicts selected data from the window and then returns to the data consumption component for continuous processing. All the four sequential components are interacting with the window from time to time. Leveraging this architecture and semantic importance, two use cases were implemented from different domains. The first use case explores soccer offside detection, where query contribution, provenance and query relevance are selected and evaluated. The second use case detects data exfiltration event in the insider threat domain, where trustworthiness and provenance are leveraged. The results have shown that semantic importance is able to improve precision, response time, memory consumption, and throughput for stream reasoning.

Semantic Importance Generalization and Benchmark Framework In order to provide the empirical support for the generalization of semantic importance, an evaluation framework is designed and implemented. This framework generalizes semantic importance by connecting it with the state-of-the-art stream reasoning

techniques, such as continuous query languages, window semantics, operational semantics etc. It also provides a comprehensive benchmark test bed, with various benchmark configuration parameters. It can work with either user provided data, or a default data generator that is evolved from LUBM data generator [18]. In this dissertation, the detailed experimental results are visualized and analyzed to illustrate the superiority of the semantic importance.

CHAPTER 2
RELATED WORK

Streaming data is so ubiquitous nowadays, such that it becomes a lot more critical to process data streams with efficient methods. Considering that streaming data is boundless, its size could be infinite. Merely storing then querying it within a database is obviously not a feasible solution. Stream processing [19] has been researched for a good time. The data stream management systems (DSMS) [20], such as STREAM [6] and Aurora [21], are able to process data streams at large volume and high update frequency, which lays the foundation to enable real-time process and query-answering. However, the requirements of processing data streams are limited to only superficial manipulations. Researchers start to look for methods that can gain insights as well as hidden knowledge from the data streams.

Both statistical and logical models can provide reasoning abilities. Statistical models extract data features to tune parameters in their math models for classification. Logical models encode human knowledge into a formal logic so that a logical reasoner can perform reasoning tasks[2]. An example of the logical models are semantic technologies such as ontology [22], OWL [23], and RDF [24]. However, these technologies are initially designed for large scale static data. One of their intentions is to structure and link the data on the web, and make it machine readable [25], such as Open Linked Data [26].

Before stream reasoning [27] is proposed, few work has been done to combine semantic reasoning with stream processing. Stream reasoning aims to "tame the

* Portions of this chapter previously appeared as: Rui Yan, Mark T. Greaves, William P. Smith, Deborah L. McGuinness 2016. "Remembering the Important Things: Semantic Importance in Stream Reasoning." Stream Reasoning 2016 Workshop, International Semantic Web Conference 2016. Rui Yan, Brenda Praggastis, William P. Smith, Deborah L. McGuinness 2016. "Towards A Cache-Enabled, Order-Aware, Ontology-Based Stream Reasoning Framework." Linked Data on the Web Workshop 2016, International World Wide Web Conference 2016. Rui Yan, Brenda Praggastis, William P. Smith, Deborah L. McGuinness 2015. "Towards Smart Cache Management for Ontology Based, History-Aware Stream Reasoning." Linked Science Workshop 2015, International Semantic Web Conference 2015.

[2] `http://www.obitko.com/tutorials/ontologies-semantic-web/reasoning.html`

velocity, veracity and volume of the data streams with both stream processing and semantic reasoning."[3]

2.1 Streaming Data

Streaming data is heterogeneous in physical formats, models, and semantics. The contents are also varied in images, videos, texts or urls etc. The data items have intrinsic orders. Temporarily is a popular data ordering. Others include precision, provenance, and trustworthiness [2]. Apparently, a specific schema can be only used to deal with one format of streaming data, let alone other types. For the sake of convenient processing, a unified data model is required. A popular data model called Resource Description Framework (RDF) is widely used to endow semantics to the static data. RDF streams [28] are proposed by extending RDF to model data streams. It is envisioned in two alternative formats: the RDF molecules stream, and the RDF statements stream. The RDF molecules stream is an unbounded set of tuples that contain RDF molecules [29] and timestamps. A timestamp denotes the temporal logic of each molecule's arrival order. A RDF statement is a special case of a RDF molecule. This format is also an unbounded set of tuples containing a RDF statement and a timestamp. RDF stream can be expressed as $< \rho, \tau >$, where ρ denotes either a RDF molecule or a RDF statement, τ denotes a timestamp. A RDF molecule contains the RDF data that cannot be further decomposed. The reason is because of the usage of blank nodes. An RDF molecule is also referred as a data item. Normally, a data item can be either a triple or a graph, annotated by a timestamp that can either be a time-point or a timer-interval [5].

Linked stream data, inspired by Linked Open Data, applies Linked Data principles [30] to stream data so that it becomes a part of the Web of Linked Data. [31] applies these principles to sensor data, in what they call "Linked Stream Data" (LSD). [32] proposes the same idea in a different way: the authors leverage C-SPARQL [33] to publish linked data stream.

The research on modeling streaming data with semantic methods has been conducted by different communities, which means the expressiveness and semantics

[3] `http://emanueledellavalle.org/Biography/Short-Biography.html`

of the streaming data is given informally [27]. In order to fill this gap, both [34] and [35] provide a theoretical framework to allow the exact description and comparison for not only streaming data, but also stream reasoning systems.

2.2 Continuous Processing

Stream reasoning, defined as "logical reasoning in real time on gigantic and inevitably noisy data streams in order to support the decision process of extremely large numbers of concurrent users" [8], aims to bridge stream processing with semantic reasoning. Stream reasoning can provide many benefits in areas that demand generation and analysis of data streams, and can be applied in a variety of domains. In smart cities scenarios [36] [37], sensor and video data are constantly streamed from different locations. The requirements include efficient integration of heterogeneous data, which is also a key foundation upon which advanced services are based. Social network analysis [38] keep up with the latest trends from the communities on the web. Facebook feeds and Twitter tweets have already been the objects of research on streaming data. Financial market [39] produces tons of streaming trade data, and the decision requires to be made within milliseconds. The vision of the Internet of Things [40] even stresses the challenges to stream reasoning: imagine all the objects that are connected with each other can be seen as stream sources, the streaming data are just enormous.

C-SPARQL [33], among the initial RDF stream processing (RSP)[4] languages, is a continuous extension of the standard SPARQL. It is tailored to semantically process data streams and facilitate reasoning. A C-SPARQL query is registered in a form of either a stream or a query, prior to the arrival of data streams. Its execution model is inherited from CQL [41], including operators of stream-to-relation, relation-to-relation and relation-to-stream. Its built-in translator will translate a C-SPARQL query into static and dynamic parts, and execute them separately. The C-SPARQL engine can be used as a linked data stream publisher [32].

Other works such as EP-SPARQL [42], TrOWL [43] and Stream SPARQL [44] are either extensions of SPARQL from different angles or are built from scratch to

[4] `https://www.w3.org/community/rsp/`

fulfill the purpose of continuously processing and reasoning on data streams. The IMaRS [45] algorithm has been proposed by the same authors of C-SPARQL, and is focused on managing (mainly statement deletions) inferred statements in a logical sliding window. This algorithm assigns an expiration timestamp for each RDF statement entering into the window, labels and updates all the related inferences with this expiration timestamp. The expiration timestamp is the time when the explicit data exit the window, and is calculated by adding the data arrival timestamp and the window size. A deletion is triggered when the original explicitly stated data exits the window and both explicit and inferred statements will be deleted.

C-SPARQL engine processes data in a sequential way, which limits the processing performance. Both [46] and [47] aim to leverage distributed systems to boost the performance.

2.3 Window

Window is defined as a finite subset of the streaming data [48]. It isolates a portion of the data stream and provides a working area for continuous processing. There are different window types. Depending on different measurement units, a window can be either a logical (time-based) window or a physical (tuple-based) window. Depending on overlapping successive windows, a window can be either a sliding, tumbling or sampling window. A window has an upper bound and a lower bound. A sliding window will have both bounds move at the same time and pace; while a landmark window will fix one bound and move the other bound. A window has two parameters, the window step and window size. The size defines how much data one active window can hold. The step defines how much the window can move at one time. For all the definitions and semantics for different kinds of windows, please refer to [48].

Albeit a lot of work in stream reasoning shares the same goal, the implementations are inevitably different. [11] has investigated the operational semantics of different stream reasoning work, with the conclusion that different window operational semantics will lead to different results, which are proven to be all correct after careful analysis. The paper urges that stream reasoning systems should be

implemented with its operational semantics documented. [9] and [49] propose a model called SECRET to formally describe different window operational semantics. It includes four aspects: scope, content, report and tick. Scope not only determines the boundary of an active window, but also affects the starting time (t_0) of the continuous processing. Content describes the subset of the data stream that is currently in an active window. Report defines the condition to fire a query, common policies include "on content change", "on window full", "non-empty content" and "periodic". Tick defines how and when to react to the input, which can be either tuple-driven or time-driven.

Window management strategies enable window to manage the data in different ways. Generally speaking, the way that a window manages the data also belongs to the realm of window operational semantics, which is not completely captured in the SECRET model. A window manages data by consuming, querying, reporting, evicting the data. A common management strategy is first in first out, which consumes the most recent data and evicts the oldest data in the window. Other window management strategies such as IMaRS [45] that deletes all the expired data and its related entailment to guarantee valid results. StreamRule [14] manages the data according to the assumption that not all of the raw streaming input will contribute to the query result. It introduces a set of user-specified rules to do the filtering. Delete and re-derive window management algorithm [50] maintains the data in a rather expensive way. It deletes all the useless data, then perform a reasoning process to re-derive the entailment in the window. [51] focuses on data eviction in semantic information flow systems. It proposes a strategy for load shredding based on the probability of the data future contribution. [52] leverages a circular buffer to manage the streaming data in a window.

2.4 Summary

I have plotted all the above-mentioned related work in Figure 2.1 with three dimensions of **continuous processing**, **window** and **streaming data**. On the continuous processing axis: Point C1 denotes *continuous processing engine*; Point C2 denotes *continuous query language*. On the window axis: Point W1 denotes

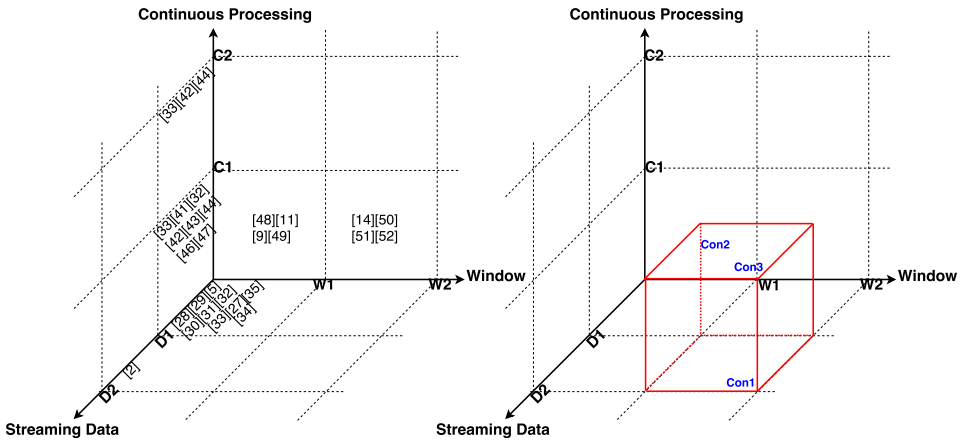


Figure 2.1: Contributions in Related Work Space

window semantics and window operational semantics; Point W2 denotes *window management strategies*. On the streaming data axis: Point D1 denotes *streaming data semantic modeling (RDF stream)*; Point D2 denotes *streaming data orderings*.

It turns out that this figure is very helpful to illustrate and understand where the related work is. For example, work located in the plane of C1-D1 is about continuous processing engines that process RDF streams. There are some empty areas, which doesn't necessarily mean there is no work. The related work is definitely not an exhaust list, but are more related to this dissertation.

Figure 2.1 also shows where my contributions can fit into this 3D space, which is the red cube. Contribution 1 models the data importance with the data ordering. Contribution 2 focuses on continuous processing architecture for RDF streams, and leverages some amount of data orderings and window management strategies. Contribution 3 focuses on not only continuous processing for RDF streams, generalizing the semantic importance, but also enabling to take a wide range of streaming data, ontologies, queries, and window management strategies.

CHAPTER 3
SEMANTIC IMPORTANCE

Timely producing query results from processing infinite streams requires efficient methods. Usually, the answers to a query are hidden in the heterogeneous streams where not all of the data items are used [14]. An ideal scenario is where all the data items contain the necessary information for answering the query. For example, registering the query "select ?s ?p ?o where {?s ?p ?o}" in the stream yields all the streaming data. Speaking from a practical perspective, this ideal scenario is not common, thus it becomes very critical to employ some efficient algorithms to produce timely and correct outputs from stream reasoning applications.

Streaming data is processed with a window, whose view of the entire data stream is limited by the window size. Even though some systems maintain a synopses of the overall stream, the synopses will grow in size as time goes by, which can increasingly consume the storage and computing resources. Thus, there is a need for smart and flexible window management strategies that allow identifying, preserving the necessary data items as well as evicting unnecessary data items in the window. The strategies also need to be computationally cheap and easy to implement.

This chapter introduces the notion of semantic importance, which models the importance of the streaming data from its various orderings. It also covers how window management strategies can be created with semantic importance, as well as how system performance is positively affected by semantic importance.

3.1 Semantic Importance

For the concept of semantic importance to be relevant, it needs to be the case that not all of the data in the window will contribute equally to the query result [14]. Given this, the ability to specifically identify the most important data and utilize

* Portions of this chapter previously appeared as: Rui Yan, Mark T Greaves, William P. Smith, Deborah L. McGuinness 2016. "Remembering the Important Things: Semantic Importance in Stream Reasoning." Stream Reasoning 2016 Workshop, International Semantic Web Conference 2016

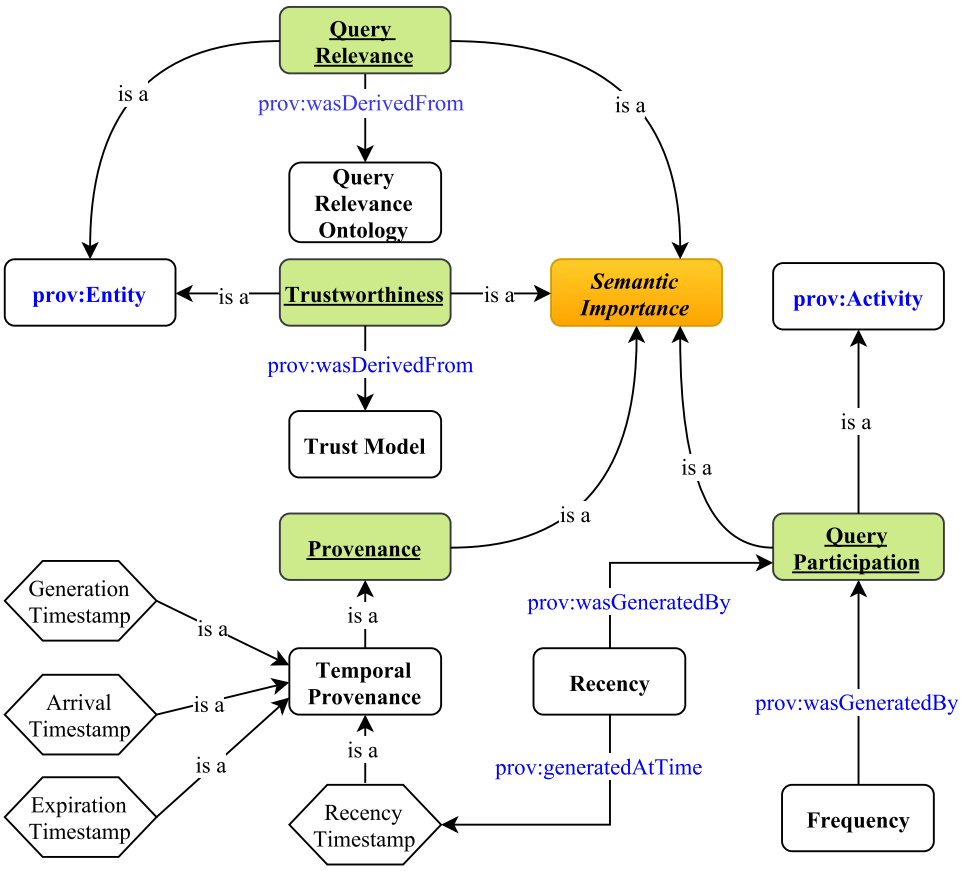


Figure 3.1: Semantic Importance Conceptual Model

it effectively is critical for improving stream reasoning system performance. So, the question is, can we differentiate and identify effective metrics for use in enabling this task? Specifically, in Figure 1.5, in order to get the correct answer at t_2, we need to find a principled and general way to keep $\triangle_A$, $\times^{\circ}_A$ and wait for $\bigcirc_A$ to arrive. Determining that $\triangle_A$, $\times^{\circ}_A$ and $\bigcirc_A$ are more important than other data items is an intuitive observation for a human to make. Our goal is to formalize certain parts of this task of judging the relative importance of different streaming data items and use those as the foundation of an ordering relation that can be used for window management. To do this, I present semantic importance.

Semantic importance is defined as an extensible conceptual model that describes the importance of the streaming data by leveraging various streaming data orderings.

Such data orderings are not limited to logical orderings, but also query participation, provenance, and trustworthiness, etc. Semantic importance is represented in a priority vector [53], with elements ordered according to a preference function. For example, priority vector $V_p = [x, y, z]$ has three elements x, y, and z. x is preferred to y, and x, y is each preferred to z. The lefter the element is placed, the more preferential the element is. There is no restriction on the number of vector elements. The traits of the priority vector allow us to consider multiple semantic importance metrics simultaneously while preserving the ability to prioritize some selecting metrics.

Figure 3.1 shows the conceptual model of semantic importance. This conceptual model leverages some concepts from Prov-O [54] to describe the inter-class relationships. It is modeled by data orderings, which currently include four aspects: provenance, query participation, trustworthiness, and query relevance. Each aspect will be introduced in details below.

3.1.1 Provenance

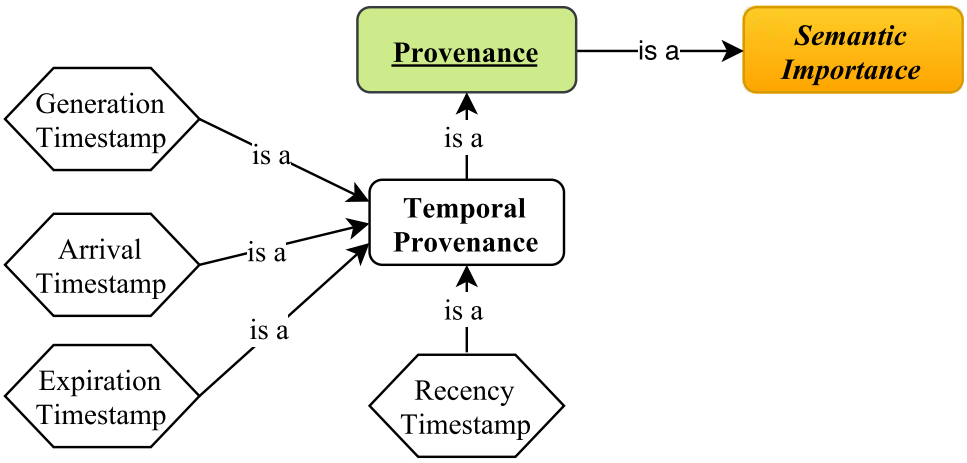


Figure 3.2: Semantic Importance - Provenance

Provenance is a factor in judging the importance of a streaming data item [55]. This dissertation takes its definition from a foundation paper [56] in data provenance, where **"provenance is defined as a set of n-tuples: what, when, where, how, who, which, and why."** There are several alternate definitions, but this one will suit the dissertation goal since it is a representative definition that

includes the extensibility with other definitions. It provides the directions to extend provenance aspect with other perspectives such as "What", "Where" or "How". In the current model, logical provenance is included and is corresponding to "When" aspect in the definition. This model can certainly go beyond that in the dissertation future work but "When" is really what should be started from, because I want to show it for the silent logical assumption.

There are four branches following the logical provenance. Generation timestamp (τ_g) is assigned by the streaming source, and describes when the data is generated. Arrival timestamp (τ_a) is assigned by the processing system, and describes when the data arrives at the system. Expiration timestamp (τ_e) is assigned by either the processing system or the streaming source, and describes when the data expires. Recency timestamp (τ_{qp}) is assigned by the system, and is associated with the query participation recency. For example, priority vector $[\tau_a]$ contains τ_a as the only aspect to describe the data importance, and this is the silent logical assumption. $[\tau_e, \tau_{qp}, \tau_a]$ uses three aspects to describe the data importance.

3.1.2 Query Participation

Query participation is defined as: data items participate in a query if they contain necessary information used by the query engine to return non-empty answers. Two query participation aspects are frequency and recency. Frequency (f_{qp}) is an integer value that describes how many times a data item participates in the query, and recency (τ_{qp}) is a timestamp that describes the most recent time-point for a data item participating in the query.

To further explain the definition of query participation, let us consider the example in Figure 3.4. We will continue to use the data stream of the soccer offside running example and Table 1.1. As mentioned in the previous section, Player A commits an offside offence if he/she is an attacker ($\triangle_A$), is at offside position ($\times^{\circ}_A$) and involves in an active play ($\bigcirc_A$) at some time point t. At this point, since all of the necessary data items are present in the window, the query "who commits an offside offence" will be answered. This example illustrates that not all of the data items in the window will convey the necessary information that can be used to

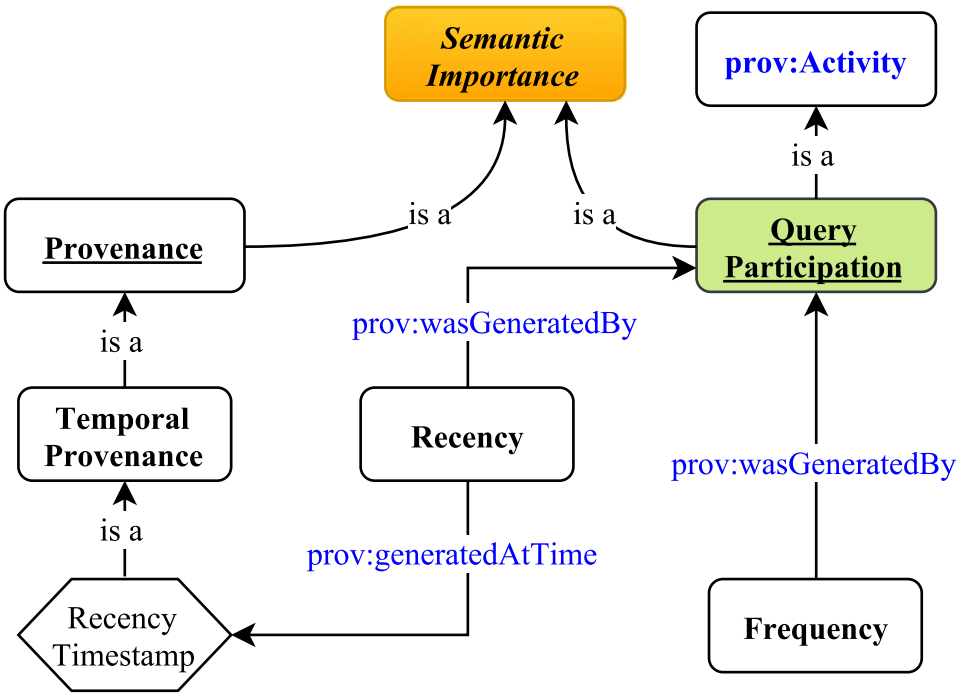


Figure 3.3: Semantic Importance - Query Participation

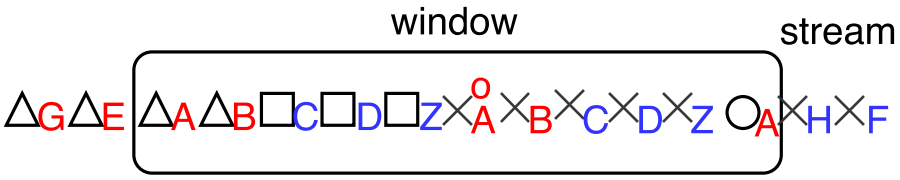


Figure 3.4: Query Participation Example

answer the query. The reason why the answer for the query is non-empty is because of the complete presence of $\triangle_A$, $\times^{\circ}_A$ and $\bigcirc_A$ at the current active window. Thus, we say that $\triangle_A$, $\times^{\circ}_A$ and $\bigcirc_A$ participate in the query, while others (such as $\triangle_B$, $\square_D$ or $\times_B$) do not.

Table 3.1: Updated Query Participation Frequency and Recency

data item	$\triangle_A$	$\triangle_B$	$\square_C$	$\square_D$	$\square_Z$	$\times^{\circ}_A$	$\times_B$	$\times_C$	$\times_D$	$\times_Z$	$\bigcirc_A$
f_{qp}	1	0	0	0	0	1	0	0	0	0	1
τ_{qp}	τ_{qp1}	-	-	-	-	τ_{qp1}	-	-	-	-	τ_{qp1}

Assume that it is the first time to execute the query, thus all data items should have $f_{qp} = 0$, and no τ_{qp}. When the query result is returned, data items' statistics

will be updated. Table 3.1 shows that $\triangle_A$, $\times^{\circ}_A$ and $\bigcirc_A$'s f_{qp} increases by 1, and τ_{qp} becomes τ_{qp1}. They have the same τ_{qp} because they participate in the query at the time $t = \tau_{qp1}$. Other data items' f_{qp} and τ_{qp} remain unchanged.

3.1.3 Trustworthiness

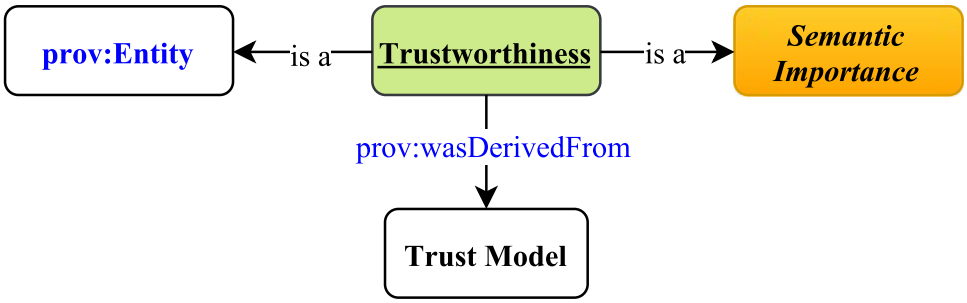


Figure 3.5: Semantic Importance - Trustworthiness

Different work models trustworthiness in different ways. For example, in [57], trustworthiness is modeled as "the probability of one data item to be correct". Other work models data trustworthiness from perspectives of data quality [58] [59] [60] [61], semantic integrity [62] and reputation techniques [63] [64]. The concept of trustworthiness can be very broad, which makes it very suitable to be included in the semantic importance conceptual model. In this dissertation, **trustworthiness is defined as "worthy of confidence"**.[5] However, in order to let the concept of trustworthiness to be useful and comparable, this work relies on the related literatures on trustworthiness modeling.

Semantic importance doesn't emphasize on any specific trust models. But for the sake of convenient processing, data items should be stamped with a numerical value as a trust score (t_s^{tm}) by a selected trust model (tm).

3.1.4 Query Relevance

Query relevance ties specific knowledge about queries and data into window management using formal semantics. It is **defined as the description of data items' potential to answer the query**. Query relevance shares the similar assumption from [14]: "not all raw data from the input stream might be relevant for

[5] https://www.merriam-webster.com/dictionary/trustworthy

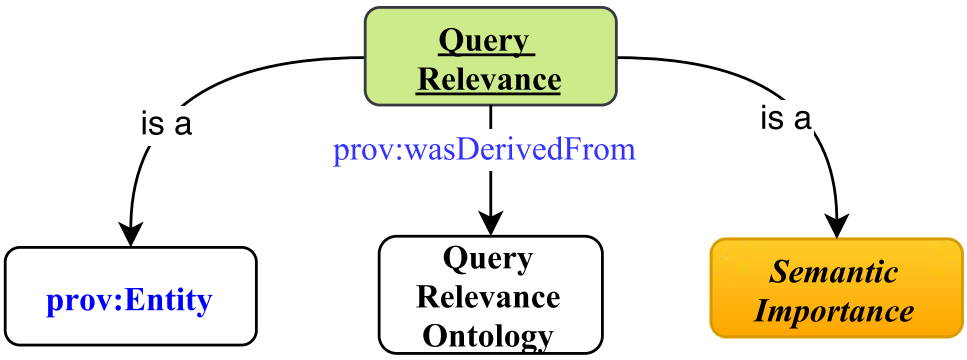


Figure 3.6: Semantic Importance - Query Relevance

complex reasoning". This description is query-informed, and encoded in an ontology called "query relevance ontology". The query relevance ontology is provided by the users, and enables query relevance as shown in Figure 3.6. For the concepts in the query relevance ontology to be useful, it is usually constructed with existing concepts in the background domain ontology, which is shown in Figure 3.7.

In Figure 3.7, all the red boxes are concepts encoded in the background domain ontology of soccer, which contains the general knowledge about the overall soccer domain. The background ontology can not only answer the query of "who commits an offside offence", but also "who are attackers", "which team is controlling the ball" and "who is challenged by whom", etc. the only concept in the query relevance ontology is marked in green, which is **OffsideIrrelevantPosition**. But the query relevance ontology only provides details of query relevance for one specific query, which in this case is "who commits an offside offence".

In the Query Participation Section, we have already mentioned that not all of the data will be participating in the query. The idea of query relevance is to encode the knowledge of human literacy about the query. In soccer offside detection, humans know that the positions of the offensive players at their own half will not participate in the query according to the soccer offside definition. It is easy to ignore such irrelevant data for linesmen during officiating the game since they are aware of this knowledge. However, in order to let the machines acquire this knowledge, users should provide a formalized representation and load it into the system. As what will be shown in Chapter 6, the benefits to deploy query relevance include

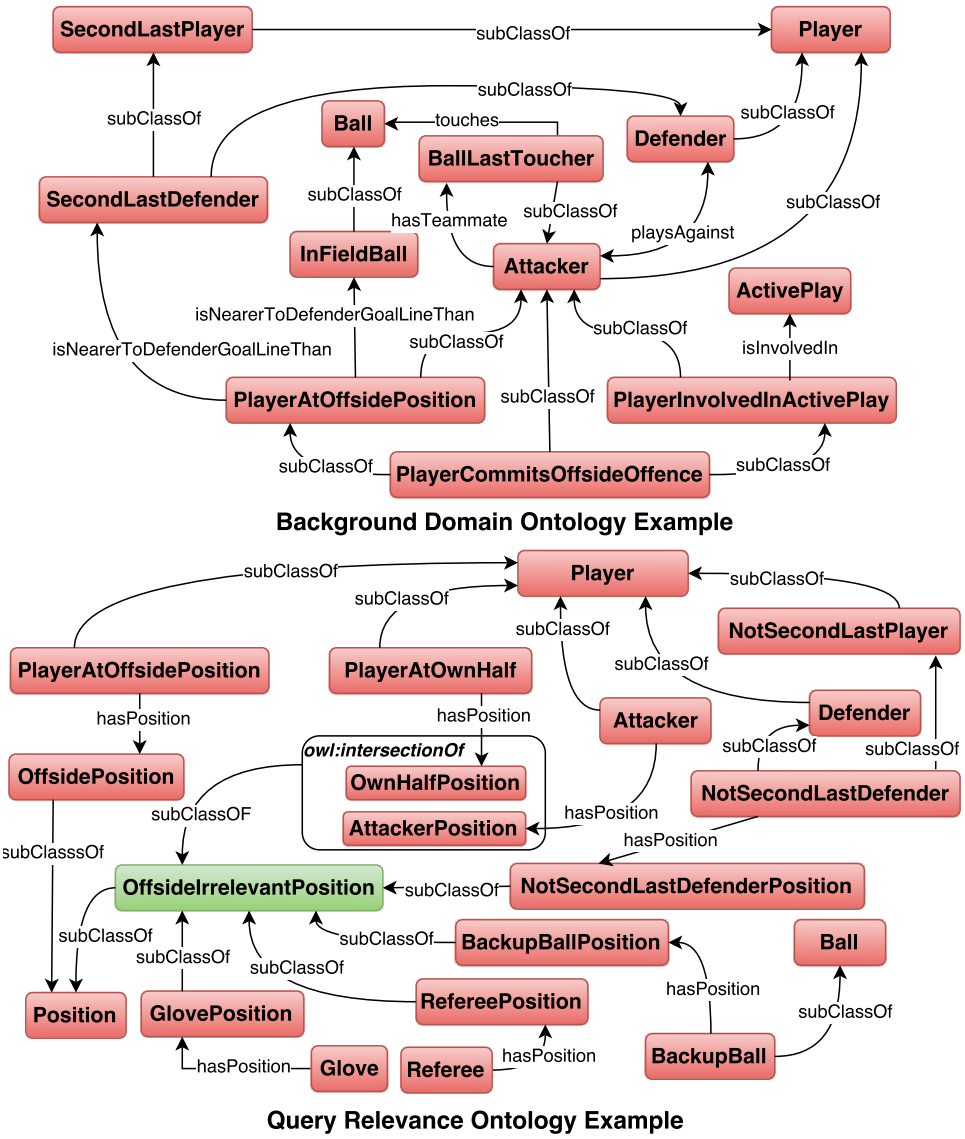


Figure 3.7: Query Relevance Ontology

reducing memory consumption and response time, as well as improving precision and throughput. The reason is because it allows the system to concentrate on a smaller portion of the original data, which still contains enough necessary information for query answering. Thus, for query relevance ontology of "who commits an offside offence", the concept of **OffsideIrrelevantPosition**, which is in green, is constructed in Figure 3.7. This **OffsideIrrelevantPosition** contains the positions of gloves, referees, defenders other than the second last defender, offensive players at their own half, etc. With this query relevance ontology, instances of **OffsideIrrele-**

vantPosition (such as $\times_B$, $\times_C$ and $\times_D$ in Figure 3.8) can be captured and filtered by the system via executing a filtering SPARQL query.

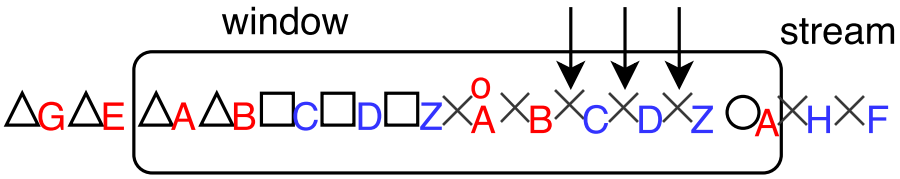


Figure 3.8: Query Relevance Example

The system can be totally blind or literate about streaming data query relevance, depending on whether or not a query relevance query/ontology is deployed, as well as how comprehensive the knowledge of the streaming data is encoded. For example, in Figure 3.8, $\square_D$ is irrelevant as a defender can never commit an offside offence. If such knowledge is also included in the query relevance ontology, the query relevance ontology becomes more comprehensive, and more irrelevant data can be filtered out. As the consequences of the query relevance ontology, data items pointed by arrows are not relevant to the current query and thus can be filtered out of the window.

3.2 Comparison Rule

The semantic importance model encodes the aspects to provide a multi dimensional view on importance. For example, $[\tau_e, \tau_a]$ uses expiration timestamp (τ_e) and arrival timestamp (τ_a) to describe data importance; while $[t_s^{tm}, f_{qp}]$ uses trustworthiness and query participation frequency to describe data importance. However, in order to enable order-awareness, there should be a comparable rule to be used to rank the data according to its importance. This rule, named semantic importance comparison rule, evaluates the data importance by comparing all dimensions of a priority vector for each data item. The comparison rule uses comparable operator "<", ">" and "==" to compare priority vector elements. The semantics of these comparable operators are defined in Table 3.2.

In Table 3.2, τ represents timestamps, τ^{sys} denotes current system time, t_s^{tm} refers to trust score by a trust model tm, qrf represents query relevance filter. When

Table 3.2: Comparable Operator Semantics

SI	<	>	==
τ_g	if τ_g^1 is before than τ_g^2 $\Rightarrow \tau_g^1 < \tau_g^2$	if τ_g^1 is after than τ_g^2 $\Rightarrow \tau_g^1 > \tau_g^2$	τ_g^1 is equal to τ_g^2 $\Rightarrow \tau_g^1 == \tau_g^2$
τ_a	if τ_a^1 is before than τ_a^2 $\Rightarrow \tau_a^1 < \tau_a^2$	if τ_a^1 is after than τ_a^2 $\Rightarrow \tau_a^1 > \tau_a^2$	if τ_a^1 is equal to τ_a^2 $\Rightarrow \tau_a^1 == \tau_a^2$
τ_e	if τ_e^1 is before than τ^{sys} if τ_e^2 is before than τ^{sys} $\Rightarrow \tau_e^1 < \tau_e^2$	if τ_e^1 is after than τ^{sys} if τ_e^2 is before than τ^{sys} $\Rightarrow \tau_e^1 > \tau_e^2$	if τ_e^1 is before than τ^{sys} if τ_e^2 is before than τ^{sys} $\Rightarrow \tau_e^1 == \tau_e^2$ if τ_e^1 is after than τ^{sys} if τ_e^2 is after than τ^{sys} $\Rightarrow \tau_e^1 == \tau_e^2$
τ_{qp}	if τ_{qp}^1 is before than τ_{qp}^2 $\Rightarrow \tau_{qp}^1 < \tau_{qp}^2$	if τ_{qp}^1 is after than τ_{qp}^2 $\Rightarrow \tau_{qp}^1 > \tau_{qp}^2$	if τ_{qp}^1 is equal to τ_{qp}^2 $\Rightarrow \tau_{qp}^1 == \tau_{qp}^2$
f_{qp}	$f_{qp}^1 < f_{qp}^2$	$f_{qp}^1 > f_{qp}^2$	$f_{qp}^1 == f_{qp}^2$
qrf	false < true	true > false	true == true false == false
t_s^{tm}	$t_{s1}^{tm} < t_{s2}^{tm}$	$t_{s1}^{tm} > t_{s2}^{tm}$	$t_{s1}^{tm} == t_{s2}^{tm}$

query relevance is included in the importance description, a data filtering SPARQL query will be executed.

When comparing priority vectors[6], the leftmost element will be compared first. If there is a tie for this element, the second element then will be compared, and so on. This is because the priority vector requires its elements to be ordered according to a preference function, as a consequence, the more preferred elements are placed on more lefter side. For example, if a data item is associated with an explicit expiration timestamp (τ_e), the priority vector $[\tau_e]$ will consider unexpired data to be important. If the data frequency of the query participation (f_{qp}) is included, the priority vector $[\tau_e, f_{qp}]$ emphasizes τ_e over f_{qp} to guarantee that one unexpired data item is more important than another. For all unexpired data, what is more important is which participates more frequently in the query. Expired data becomes less important no matter how frequent its query participation is. By explicitly characterizing the importance for each data item in the stream, order-awareness can be realized by ranking the semantic importance priority vectors via the following algorithm:

[6]Only priority vectors with same amount of elements can be compared.

Algorithm 1: Semantic Importance Comparison Algorithm

```
1  priority vectors v1 = [x1, ..., xn], v2 = [y1, ..., yn] ;
2  int i = 0 ;
3  while i < n do
4  |  if xi > yi then
5  |  |  return v1 > v2 ;
6  |  else if xi < yi then
7  |  |  return v1 < v2 ;
8  |  else
9  |  |  i++ ;
10 if i == n then
11 |  return v1 == v2 ;
```

3.3 Window Management Strategies

A sliding window is defined as a finite subset of the stream [9]. It has two parameters, a window size (width) that determines how much data a window can contain; a window step (slide) that is the distance between two consecutive windows. A window has different types [4] [48]. For all details about windows and operational semantics, please refer to Chapter 2.

As mentioned in the Introduction Section, the silent logical assumption that older information becomes irrelevant at some point [3] [8] is very popular in stream reasoning [12] [13]. Based on this silent logical assumption, the dominant window management strategy is first in first out (FIFO). This strategy consumes the most recent data based on the window step. In order to guarantee identical window size during processing, the amount of data to be evicted should be strictly identical with that of data to be consumed. The data to be evicted is usually the oldest.

In order to illustrate how semantic importance enabled window management strategies can manage the data items in the window, we will use a running example with a physical sliding window of a size of four tuples, and a step of one tuple. Six strategies, including FIFO, will be shown in detail about the way and differences they manage streaming data in the window.

FIFO: First In First Out (FIFO) describes data importance with a single arrival timestamp (τ_a) from the logical provenance aspect. Its priority vector is $[\tau_a]$.

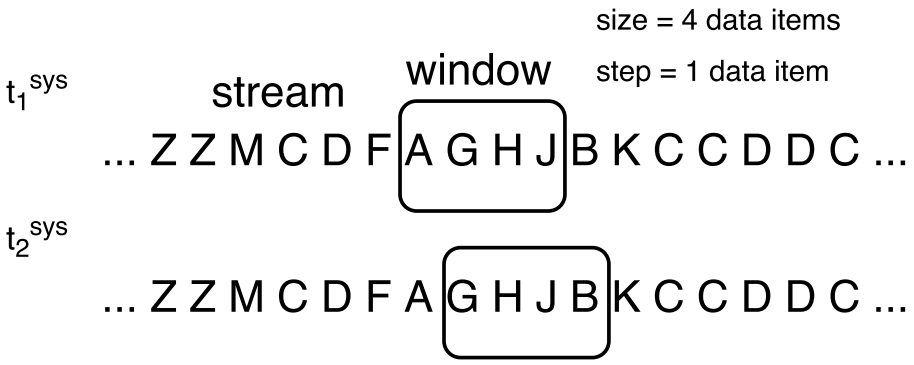


Figure 3.9: FIFO Management Strategy

In Figure 3.9, data **arrives** in a logical order such that $\tau_a^A < \tau_a^G < \tau_a^H < \tau_a^J < \tau_a^B$. At t_1^{sys}, items A, G, H, and J are in the window. At t_2^{sys}, as window proceeds, A is evicted so as to consume B. Given the arrival order, the data rankings at t_1^{sys} and t_2^{sys} are shown in Table 3.3. Data gets re-ranked as window moves, and data ranked at the bottom will be evicted.

Table 3.3: Data Ranking Under FIFO

t_1^{sys}			t_2^{sys}		
rank	data	τ_a	rank	data	τ_a
1	J	τ_a^J	1	B	τ_a^B
2	H	τ_a^H	2	J	τ_a^J
3	G	τ_a^G	3	H	τ_a^H
4	A	τ_a^A	4	G	τ_a^G

FEFO: First Expired First Out (FEFO) describes data importance with a single expiration timestamp (τ_e) from the logical provenance aspect. Its priority vector is $[\tau_e]$.

In Figure 3.10, at t_1^{sys}, data **arrives** in a logical order such that $\tau_a^A < \tau_a^G < \tau_a^H < \tau_a^J < \tau_a^B$, **expires**[7] in a logical order such that $\tau_e^G < t_1^{sys} < \tau_e^B < t_2^{sys} < \tau_e^A == \tau_e^J == \tau_e^H$. At t_1^{sys}, items A, G, H, and J are in the window, G is expired. At t_2^{sys}, G is evicted to consume B, which will be evicted next. Note that even though G arrives later than A, G is evicted because of FEFO management strategy, and even

[7]We use the "==" semantics from Table 3.2. Note that the value of τ_e^A, τ_e^J and τ_e^H are not necessarily numerically equal.

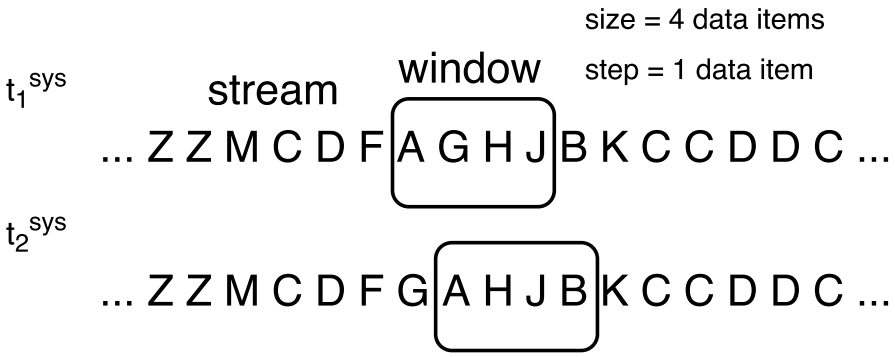


Figure 3.10: FEFO Management Strategy

though B is the latest data item, it is also an expired data item, thus needs to be evicted. Data rankings at t_1^{sys} and t_2^{sys} are listed in Table 3.4. The data item ranks at the bottom will be evicted.

Table 3.4: Data Ranking Under FEFO

t_1^{sys}			t_2^{sys}		
rank	data	τ_e	rank	data	τ_e
1	H, J, A	τ_e^H, τ_e^J, τ_e^A	1	H, J, A	τ_e^H, τ_e^J, τ_e^A
2	G	τ_e^G	2	B	τ_e^B

LFU: Least Frequently Used (LFU) is a classic caching algorithm[8], but not commonly used in the window management. LFU describes the data importance with a single query participation frequency (f_{qp}) from the query participation aspect. Its priority vector is $[f_{qp}]$.

In Figure 3.11, data **arrives** in a logical order such that $\tau_a^A < \tau_a^G < \tau_a^H < \tau_a^J < \tau_a^B$. As soon as the data arrives in the window, its f_{qp} is initialized to be 0. f_{qp}^X will increase by 1 if the data item X participates in the query once. At t_1^{sys}, assuming that f_{qp}^A = 3, f_{qp}^G = 2, f_{qp}^H = 0, f_{qp}^J = 0. Since $t_{qp}^H = t_{qp}^J = 0$, both H and J rank at the bottom in Table 3.5. Either H or J will be evicted in an undecided way because of the window step is 1 tuple. In Figure 3.11, let's tentatively evict J to consume B. After query is executed, assuming that B participates in the query thus f_{qp}^B becomes 1. Then, data items in the window get re-ranked at t_2^{sys}. Since f_{qp}^H is 0 and the smallest, data item H will be ranked at the bottom and evicted next.

[8] https://en.wikipedia.org/wiki/Cache_replacement_policies

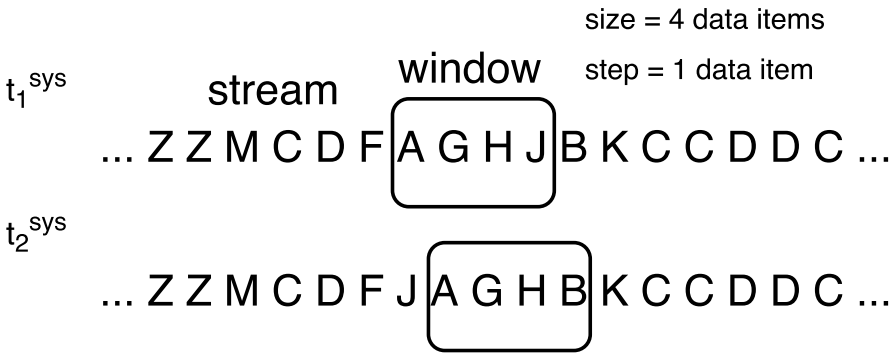


Figure 3.11: LFU Management Strategy

Table 3.5: Data Ranking Under LFU

t_1^{sys}			t_2^{sys}		
rank	data	f_{qp}	rank	data	f_{qp}
1	A	3	1	A	4
2	G	2	2	G	3
3	H, J	0	3	B	1
4	-	-	4	H	0

FE-LFU-FO: First Expired Least Frequently Used First Out (FE-LFU-FO) is a more complex window management strategy. It describes the data importance with the expiration timestamp (τ_e) and the query participation frequency (f_{qp}). Its priority vector is $[\tau_e, f_{qp}]$, which emphasizes τ_e over f_{qp}.

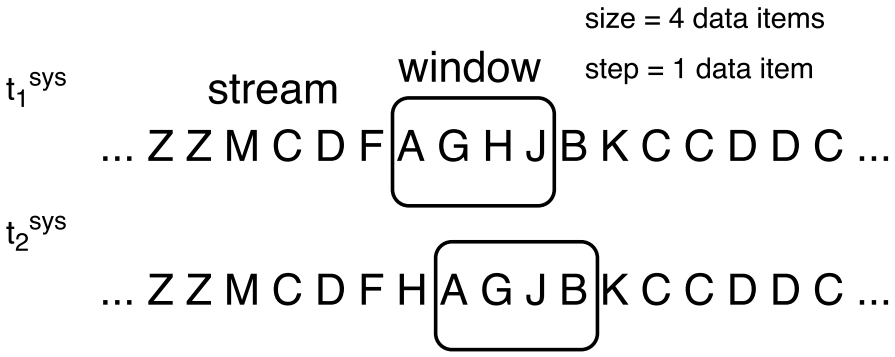


Figure 3.12: FE-LFU-FO Management Strategy

In Figure 3.12, assuming that data arrives in a logical order such that $\tau_a^A < \tau_a^G < \tau_a^H < \tau_a^J < \tau_a^B$, and expires in an order such that $\tau_e^H < t_1^{sys} < t_2^{sys} < \tau_e^G ==$

$\tau_e^J == \tau_e^A == \tau_e^B$. At t_1^{sys}, we can compare data items priority vector according to the comparison rule. Since $[\tau_e^H, 4] < [t_1^{sys}, -] < [\tau_e^G, 0] < [\tau_e^J, 2] < [\tau_e^A, 3]$, H ranks at the bottom thus evicted even though it participated most in the query. At t_2^{sys}, $[t_2^{sys}, -] < [\tau_e^G, 0] == [\tau_e^B, 0] < [\tau_e^J, 3] < [\tau_e^A, 4]$. Table 3.6 shows the data rankings. At t_2^{sys}, both B and G are ranked at the bottom, thus either will be evicted due to the window step.

Table 3.6: Data Ranking Under FE-LFU-FO

t_1^{sys}				t_2^{sys}			
rank	data	τ_e	f_{qp}	rank	data	τ_e	f_{qp}
1	A	τ_e^A	3	1	A	τ_e^A	4
2	J	τ_e^J	2	2	J	τ_e^J	3
3	G	τ_e^G	0	3	B,G	τ_e^B, τ_e^G	0
4	H	τ_e^H	4	4	-	-	-

LFU-FE-FO: Least Frequently Used First Expired First Out (LFU-FE-FO), just like FE-LFU-FO, describes the data importance with τ_e and f_{qp}. Its priority vector $[f_{qp}, \tau_e]$, albeit has the same elements as FE-LFU-FO's $[\tau_e, f_{qp}]$, it ranks data items in a totally different way.

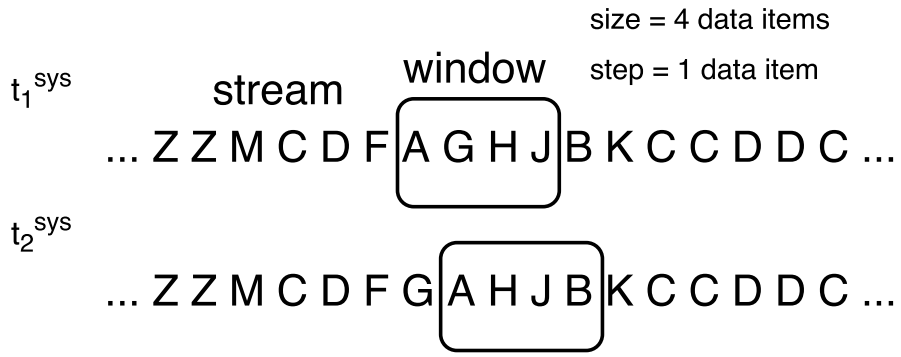


Figure 3.13: LFU-FE-FO Management Strategy

In Figure 3.13, assuming that data arrives in a logical order such that $\tau_a^A < \tau_a^G < \tau_a^H < \tau_a^J < \tau_a^B$, and expires in a logical order such that $\tau_e^H < t_1^{sys} < t_2^{sys} < \tau_e^G == \tau_e^J == \tau_e^A == \tau_e^B$. Together with Table 3.7, at t_1^{sys}, $[-, t_1^{sys}] < [0, \tau_e^G] < [2, \tau_e^J] < [3, \tau_e^A] < [4, \tau_e^H]$. G is ranked at the bottom thus will be evicted. However, H is ranked at the top even though H is expired. What is even worse is that H has the

most query participation frequency, that almost guarantees that H will be at the top forever, resulting in invalid answers. At t_2^{sys}, $[-, t_2^{sys}] < [0, \tau_e^B] < [3, \tau_e^J] < [4, \tau_e^A] < [5, \tau_e^H]$. H, A and J's query participation frequencies increase by 1, while B doesn't participate in the query. Thus B is ranked at the bottom and will be evicted.

From this example, we can see that even though the window management strategies formed by combining different semantic importance aspects can be various, what's crucial behind it is to careful design the combination so that the strategy can manage data in a legit way.

Table 3.7: Data Ranking Under LFU-FE-FO

t_1^{sys}				t_2^{sys}			
rank	data	f_{qp}	τ_e	rank	data	f_{qp}	τ_e
1	H	4	τ_e^H	1	H	5	τ_e^H
2	A	3	τ_e^A	2	A	4	τ_e^A
3	J	2	τ_e^J	3	J	3	τ_e^J
4	G	0	τ_e^G	4	B	0	τ_e^B

FE-LFU-FI-FO: First Expired Least Frequently Used First In First Out (FE-LFU-FI-FO) uses the expiration timestamp (τ_e), query participation frequency (f_{qp}) and arrival timestamp (τ_a) to describe the data importance. Its priority vector is $[\tau_e, f_{qp}, \tau_a]$, which contains three elements.

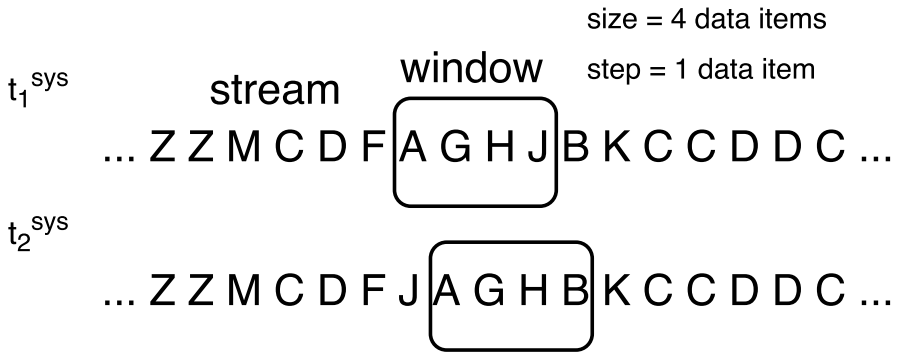


Figure 3.14: FE-LFU-FI-FO Management Strategy

In Figure 3.14, assuming that data arrives in a logical order such that $\tau_a^A < \tau_a^G < \tau_a^H < \tau_a^J < \tau_a^B$, and expires in a logical order such that $\tau_e^J < t_1^{sys} < \tau_e^G < t_2^{sys} < \tau_e^H == \tau_e^A == \tau_e^B$. At t_1^{sys}, $[\tau_e^J, 6, \tau_a^J] < [t_1^{sys}, -, -] < [\tau_e^G, 0, \tau_a^G] < [\tau_e^H, 1, \tau_a^H] <$

$[\tau_e^A, 4, \tau_a^A]$. At t_2^{sys}, $[\tau_e^G, 0, \tau_a^J] < [t_2^{sys}, -, -] < [\tau_e^B, 0, \tau_a^B] < [\tau_e^H, 2, \tau_a^H] < [\tau_e^A, 5, \tau_a^A]$. With these orderings, data is ranked as in Table 3.8. Data ranked at the bottom will be evicted.

Table 3.8: Data Ranking Under FE-LFU-FI-FO

t_1^{sys}					t_2^{sys}				
rank	data	τ_e	f_{qp}	τ_a	rank	data	τ_e	f_{qp}	τ_a
1	A	τ_e^A	4	τ_a^A	1	A	τ_e^A	5	τ_a^A
2	H	τ_e^H	1	τ_a^H	2	H	τ_e^H	2	τ_a^H
3	G	τ_e^G	0	τ_a^G	3	B	τ_e^B	0	τ_a^A
4	J	τ_e^J	6	τ_a^J	4	G	τ_e^G	0	τ_a^G

The above six strategies are certainly not an exhaustive list of all the window management strategies that semantic importance can enable. The point is provide both examples and evidence that semantic importance can not only enable classic FIFO strategy, but also smarter and more flexible strategies. FE-LFU-FO and LFU-FE-FO strategies manage data differently even though their priority vectors have same elements. Other possible strategies with priority vectors such as $[t_s^{tm1}, \tau_e, \tau_a]$, $[qrf, t_s^{tm1}, t_s^{tm2}, \tau_e, \tau_a]$ are also very similar to implement.

3.4 Window Semantics

Semantic importance is an interesting idea to pursuit because it brings a better concept of what to forget, so as to shrink the window size during processing the gigantic data streams. However, forgetting based on FIFO is only logical. The silent logical assumption is probably a good reflection of logically-oriented-only importance. The soccer example shows that the "importance" is a deeper notion, a more domain specific notion, which cannot be adequately modeled merely by time alone. One motivation of this dissertation is to take the notion of importance into semantics by bringing the domain into the computation with an efficient way. This requires not only to model the importance of the data, but also to extend the current sliding window semantics that is related to the silent logical assumption.

What will be shown later illustrates why it is important to extend the window semantics in order to realize the flexible and smart window management strategies.

The method is to re-define the window size and step, as well as to extend the sliding logical window semantics with the lower-bounded landmark window whose lower bound is fixed permanently, and upper bound proceeds. This extended window semantics can work well with the semantic importance.

3.4.1 Problems in Existing Winidow Semantics

"**A window W over a stream S is a finite subset of S**" [49]. A sliding window has two parameters, size and step, both have been mentioned and defined in related work [35] [49] [65] [9] [41]. [49] requires that all windows defined over a stream must have the same size and step, with a restriction that the step could be any value that must be no bigger than the size. [48] indicates that physical window step is 1 with the statement that "... window states are determined only at single-tuple units ...". [10] mentions that "... if the step is larger than the range then the windows sample the stream ...", indicating that the step can be larger than the size. As you can see, there are already some differences on how researchers understand and define window semantics. Window semantics is important to define a window and managing data.

In this section, I argue that none of the existing window semantics can be adapted when equipping window with management strategies other than FIFO. As a result, existing window semantics need to be extended.

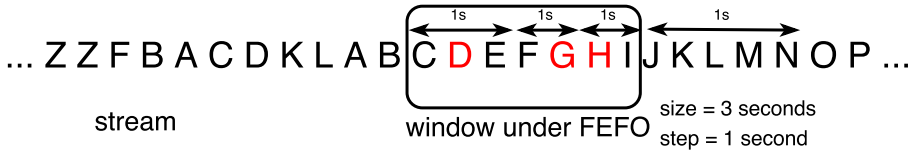


Figure 3.15: Logical Sliding Window Under FEFO

Consider the example in Figure 3.15, which is a logical sliding window under FEFO management strategy, with a size of 3 seconds and a step of 1 second. Assume that data items D, G, and H are expired (marked in red), thus ranked at the bottom. At this point, window size is still 3 seconds as $t_a^I - t_a^C = 3s$, even if D, G, and H are evicted. As a consequence, the sliding window will not move as it is still full according to its semantics, so that the continuous processing will be halted.

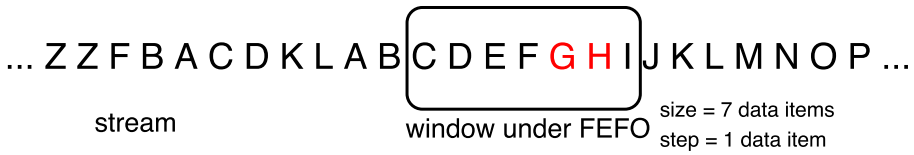


Figure 3.16: Physical Sliding Window Under FEFO

In Figure 3.16, a physical (tuple-based) sliding window manages data under FEFO. At this moment, data items G and H are expired (marked in red), thus ranked at the bottom. However, since the current window semantics defines its step to be 1 and fixed, it can only evict one data item. No matter which data item is evicted, a worse case is that the kept expired data item participated in the query, causing the early expiration problem and resulting invalid query answers.

The two problems above have shown that existing window semantics cannot work well with semantic importance enabled window management strategies. The primary reason is that these strategies manage data in a "data item" granularity, rather than a simple "logical" granularity.

3.4.2 Extended Logical Window Semantics

This dissertation will use the definitions of lower bound (τ_l), upper bound (τ_u), data stream (S), current stream contents ($S(\tau)$), current stream instance (s_τ), time domain (T) and lower-bounded landmark window (W^{lbl}) from [48]. A logical lower-bounded landmark window (W_τ^{lbl}) is employed as an extended window for a logical sliding window. For W_τ^{lbl}, its τ_l is fixed permanently, while τ_u proceeds as time goes by. Figure 3.17 shows an example of W_τ^{lbl}.

Rather than defining window size, this dissertation uses the definition from [66]: **"the instantaneous window size (n_τ) at time τ is defined as the number of tuples in the current active window."** With this definition, W_τ^{lbl} window size can vary. We define a logical window step for W_τ^{lbl}: **window step (n_β) is defined as the number of tuples within next β measurement units [48] (such as seconds, minutes, etc)**.

Figure 3.18 shows how the extended logical window semantics can work with semantic importance enabled window management strategies (such as FEFO) when

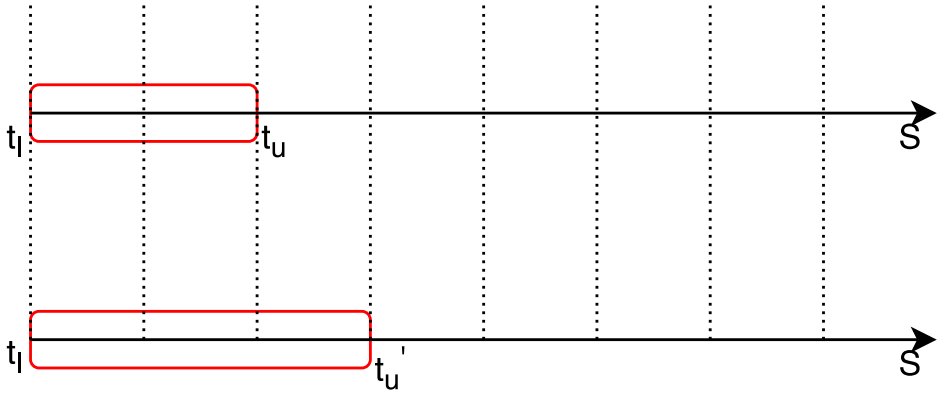


Figure 3.17: Logical Lower-bounded Landmark Window Example

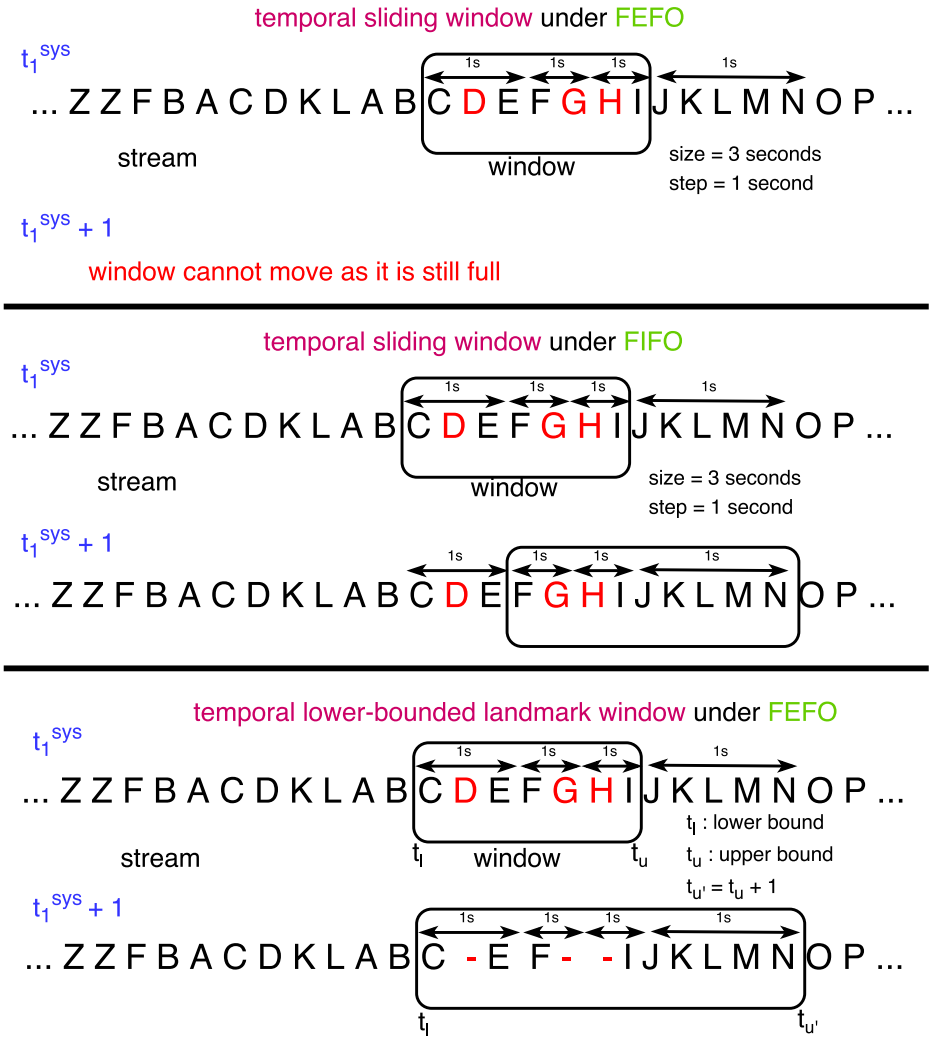


Figure 3.18: Extended Logical Window Semantics Example

managing data streams. By leveraging the extended logical window semantics from the landmark window, the logical lower-bounded landmark window can continue to proceed after evicting all the expired data. This is much flexible than a logical sliding window that will have to remove the oldest data under FIFO in order to proceed, which results in removing data items that are still valid (such as C and E in Figure 3.15), and keeping data items that are expired (such as G and H in Figure 3.15).

Table 3.9: Window Semantics Example

	landmark window under FEFO	sliding window under FIFO
window size	n_{t_u} = 7 $n_{t_{u'}}$ = 9	2s (7 tuples) 2s (9 tuples)
query: do both C and N exist in the stream? (early eviction) ground truth: yes	answer: yes	answer: no
query: do both G and N exist in the stream? (early expiration) ground truth: no	answer: no	answer: yes

Table 3.9 shows the different results of different window semantics. We have used the physical window size definition for the landmark window. At t_u and $t_{u'}$ time in Figure 3.18, its size is 7 and 9 respectively. For the sliding window under FIFO, its size is logical, which is 2 seconds. At t_1^{sys} and t_1^{sys} + 1 time in Figure 3.18, the number of tuples in the sliding window is 7 and 9 respectively. Thus, even though the landmark window seems to have "infinite" growth as its lower bound fixed permanently and its upper bound proceeds, the number of tuples in the window does not necessarily become "infinite", thanks to the extended window size definition. In this example, the span of the landmark window is 4 seconds, which is larger than the sliding window size of 2 seconds, but the number of tuples in both windows is equal.

The landmark window semantics can also help solve the early eviction and early expiration problem. The query "do both C and N exist in the stream" will have a false-negative answer from the sliding window under FIFO, as C has to be evicted in order to get N. This is an early eviction problem. In a landmark window

with FEFO, C stays as it is not expired, thus can provide the correct answer. For the query "do both G and N exist in the stream", a false-positive answer will be given by the sliding window under FIFO, as G is expired within the window but cannot be evicted. This is an early expiration problem. The landmark window with FEFO strategy can provide correct result as all the expired data can be evicted without affecting the continuous data consumption and processing.

3.4.3 Extended Physical Window Semantics

Different from the logical window, this dissertation will use a sliding physical (tuple-based) window ($W_{\#}$). From the window size definition [66], a sliding physical window size n_τ is always a fixed value. The only part to be extended from the sliding physical window semantics is the step. **The step (n_β) is defined as a variable with its range as $n_\beta = \{\ n_\beta \mid 0 \leq n_\beta \leq n_\tau\}$.**

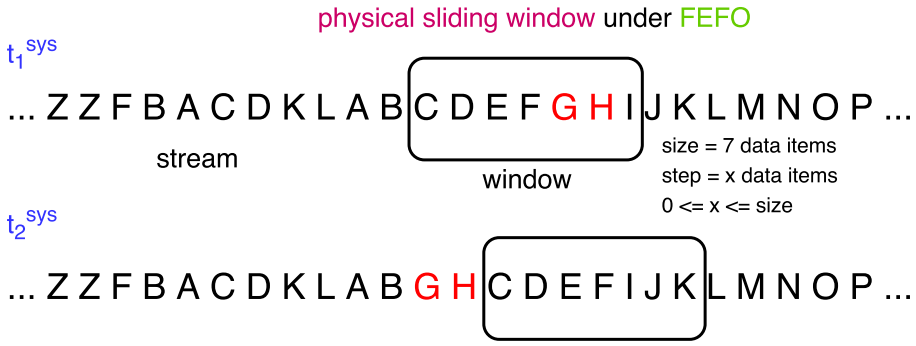


Figure 3.19: Extended Physical Window Semantics Under FEFO

Figure 3.19 shows how data can be managed with FEFO, and the extended physical window semantics. Since the window step is now a variable no bigger than the window size, evicting data becomes much more flexible: if two data items are expired, both of them will be evicted. Thus the window can continue to consume next two new data items. If all of the data are expired in the window, then evict all of them to consume next n_τ data items. This flexible window step guarantees that all of the unimportant data items are evicted, so as to solve the problems such as the early expiration. One thing to note is that, with FEFO, data has to be expired to be evicted. If all of the data are not expired, the window will not proceed but wait till at least one of them is expired. This indicates that the window management

strategies manage data differently, and it is crucial to choose a proper strategy to keep the desired data for query.

3.4.4 Discussion

Table 3.10: Window Semantics Comparison

	Sliding Window Semantics	Extended Window Semantics
logical window (W_τ)	sliding window	lower-bounded landmark window
W_τ size	fixed. defined with logical measurement units (e.g. 2s)	can vary. defined as n_τ, number of tuples at an instantaneous time-point (e.g. 23)
W_τ step	fixed. defined with logical measurement units (e.g. 1s)	fixed. defined as n_β, number of tuples within next β measurement units.(e.g. 15)
physical window ($W_\#$)	sliding window	sliding window
$W_\#$ size	fixed. defined as n_τ, number of tuples at an instantaneous time-point (e.g. 23)	fixed. defined as n_τ, number of tuples at an instantaneous time-point (e.g. 23)
$W_\#$ step	fixed. defined as n_β, number of tuples within next β measurement units.(e.g. 15)	variable. $n_\beta = \{n_\beta \mid 0 \leq n_\beta \leq n_\tau\}$

We discuss why the extended semantics is more general. Table 3.10 shows the window semantics comparison between the sliding window and extended window. For logical window (W_τ), the extended window type becomes the lower-bounded landmark window. W_τ size and step are both re-defined with the number of tuples, as opposed to logical measurement units. For the physical window, although the same sliding window type is used, its step is extended to be a variable that is no bigger than its window size, as opposed to a previous fixed step.

The extended window semantics is general, which is reflected by not only the examples shown in Figure 3.18 and Figure 3.19, but also the fact that sliding

logical window can be emulated with the extended window semantics under a proper semantic importance enabled window management strategy.

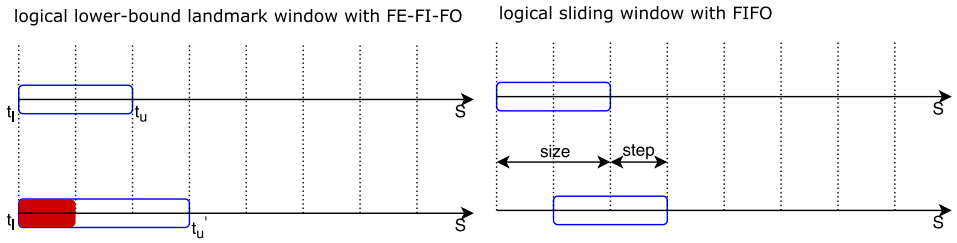


Figure 3.20: Landmark Window Emulating Sliding Window

Figure 3.20 illustrates how the extended window semantics can emulate a logical sliding window. A logical lower-bounded landmark window is managed with FE-FI-FO strategy. FE-FI-FO is first expired, first in, first out, with a priority vector as $[\tau_e, \tau_a]$. A landmark window simulates a logical sliding window via assigning an expiration timestamp for each data item by adding the arrival timestamp with window initialization length. This is very similar as how IMaRS [45] manages the data. When the time reaches $t_{u'}$, the data portion marked in red will be expired in the landmark window, thus evicted. The content in the landmark window is the same as the sliding logical window.

[SIZE 2s STEP 1s] is an example argument to define a logical sliding window in C-SPARQL query language [33]. Because this definition is only designed for the sliding window, in order to let the extended window semantics to work with the existing continuous query language, there is also a need to align the extended window semantics with the query language window definition. This portion of work will be covered in Chapter 6.

3.5 Summary

This chapter has introduced the notion of semantic importance, including the definitions, the conceptual model, the comparison rule, window management strategies and window semantics. The aspects in the semantic importance conceptual model are not exhaustive, and we encourage people to reuse and extend it. In order

to make it convenient to reuse semantic importance, I have created an ontology and ground it with real world soccer examples. All the details are in Appendix I.

Currently the model is consisted of two major aspects, the domain agnostic aspects and domain literate aspects. Domain agnostic aspects include query participation and provenance. This is because queries are often executed independently, regardless of data, backgrounds or domains. Provenance information is often carried when data is generated. Domain literate aspects include trustworthiness and query relevance. Often the trust models are dependent on the domain. For example, you might trust a Japanese friend to be your tour guide in Kobe but probably won't trust this friend to be the guide in New York. There are lots of trust models that have been proposed and work well in different scenarios. Query relevance, in our model, is realized by applying query relevance ontology and query in the system. This ontology is different from the background ontology that is usually deployed before the system starts. The background ontology encodes the essential knowledge of the domain, and is specifically focused on capturing domain knowledge that impacts querying and reasoning to answer questions. Query relevance ontology is focused on picking the query informed data for query execution. This requires a filtering SPARQL query to be executed before the target query is executed. This filtering query matches all the irrelevant data that can be classified by some concepts encoded in the query relevance ontology. So that the data pool for query execution is smaller to make the target query run faster. If data comes with trust scores, it is generally recommended to perform trust filter before the target query runs. Again, this is to guarantee that all the data enters into the query phase is trusted.

Using priority vector to encode semantic importance guarantees an efficient and multi-fold ranking. The comparison rule allows the ranking to consider different aspects comprehensively.

Different window management strategies can be enabled by semantic importance. Work such as [11] considers the operation semantics in stream reasoning systems. Window operational semantics usually refers to the way how the query is executed in the window, how the data is consumed and evicted. For example, C-

SPARQL executes the query when the window is full, while CQELS executes when the window receives the new data.

Table 3.10 provides a comparison between the sliding window semantics and extended window semantics. From what has been shown before, the extended window semantics works seamlessly with the semantic importance enabled window management strategies.

CHAPTER 4
SEQUENTIAL STREAM REASONING ARCHITECTURE

Most existing stream reasoning systems employ the silent temporal assumption when "forgetting" the data, and are not designed to leverage the semantic importance model. In this chapter, a sequential stream reasoning architecture (SSRA) is proposed to provide an infrastructure to deploy the semantic importance model in stream reasoning systems.

4.1 Assumptions

SSRA shares the common assumptions on ontologies and streaming data, under which existing systems work [33] [67] [16] [45] [42].

4.1.1 Ontology

SSRA follows assumptions for background ontologies: (1) the stream reasoning system leverages ontologies encoded in OWL [68] to support the symbolic reasoning; (2) the background ontology is usable, correct, consistent, and provided by the domain experts; (3) the background ontology is preloaded into the system before the processing starts; (4) the background ontology doesn't change during the processing.

A stream reasoning system doesn't necessarily have to employ ontologies to provide reasoning foundations. Existing systems [69] [70] leverage answer set programming (ASP) [71] to support reasoning. [14] encodes customized rules to enable reasoning. However, SSRA chooses to use ontologies for reasoning because they provide a flexible and expressive way to encode knowledge. Domain experts are key to composing the ontologies. The complexity of the ontology should be determined by the use case requirements. The correctness of the ontology is very crucial to determine the query results, as the knowledge for executing the query is solely pro-

*Portions of this chapter previously appeared as: Rui Yan, Brenda Praggastis, William P. Smith, Deborah L. McGuinness. 2016 "Towards A Cache-Enabled, Order-Aware, Ontology-Based Stream Reasoning Framework." Linked Data on the Web Workshop 2016, 25th World Wide Web Conference 2016

vided by the background ontology. The domain knowledge in the ontologies usually doesn't change (frequently). Previous work including [72] [73] considers the change of the background knowledge. Their method is to update the background ontology over time. Between two adjacent updates, the background ontology remains static. Overall, the assumptions on the background ontologies are reasonable and can be loosened with minimum efforts.

4.1.2 Streaming Data

SSRA follows assumptions for the streaming data: (1) the streaming sources encapsulate the streaming data items as RDF stream format, and in unique named graphs; (2) the streaming sources assign expiration timestamps to streaming data items; (3) SSRA assigns arrival timestamps for the streaming data.

RDF streams [28] are RDF data annotated with timestamps that are either time-points or time-intervals. The RDF stream format can be represented as $< \rho, \tau >$, where ρ is an RDF molecule [29], and τ is a timestamp denoting the arrival time of ρ. An RDF stream is defined as an ordered sequence of pairs [32] [33], each of which consists of an RDF triple and a monotonically non-decreasing timestamp τ, identified by a unique IRI that is a streaming source locater, and published in a named graph [74]. The above work justifies the assumption of leveraging named graphs in the RDF streams.

SSRA extends the RDF stream format by adding another timestamp and a unique graph ID. Thus, an RDF stream is represented by $< \rho, \tau_a, \tau_e, G >$, where ρ denotes the RDF molecule/statement, τ_a denotes its arrival timestamp, τ_e denotes its expiration timestamp and G denotes its unique graph ID.

Under the silent temporal assumption, the expiration timestamps are assigned by the system via adding arrival timestamps with the logical window size [45]. Given the information of the window and the system as a priori, the streaming source can assign expiration timestamps as exactly as what the system can do. Thus, the assumption on assigning expiration timestamps also cover the situation where the data itself carries the expiration timestamps, or data expires due to the new data arrival at some random time.

4.2 Architecture

SSRA consists of a window and four sequential components: *data consumption*, *query execution*, *result explanation* and *data eviction*. These four components are executed in a sequential order. The execution flow forms a loop to process data streams and to produce query results in a continuous fashion, as it is shown in Figure 4.1.

4.2.1 Window

The window is designed with a triple-store that stores the data physically, and a graph ID manager that indexes the data graphs. The background ontology is preloaded in the window triple-store. The window receives and keeps the data from the data consumption. The query is executed in the window with the triple-store query engine, background ontology and the data. The triple-store also provides the ability to explain the results. The graph ID manager helps index the data, manages data semantic importance priority vectors and ranks the data. Coupled with the window management strategies, the graph ID manager can provide the list of data to be evicted from the triple-store.

4.2.2 Data Consumption

RDF streams enter SSRA via data consumption component. Its behavior is modeled and controlled by both Tick and Report operational semantics from the SECRET model [9]. It consumes data in either a time-driven or tuple-driven way, depending on how the window is defined. It can realize different reporting policies as well. For example, “on window close” policy allows the streaming data enters the window till full before executing the query; while “on content change” policy allows to execute the query once the content is changed in the window, regardless of window fullness. In either policy, it will send the query execution request to the next component.

4.2.3 Query Execution

The query in the stream reasoning scenario should be continuous in order to provide proactive answers. This requires the query to be preregistered in the system

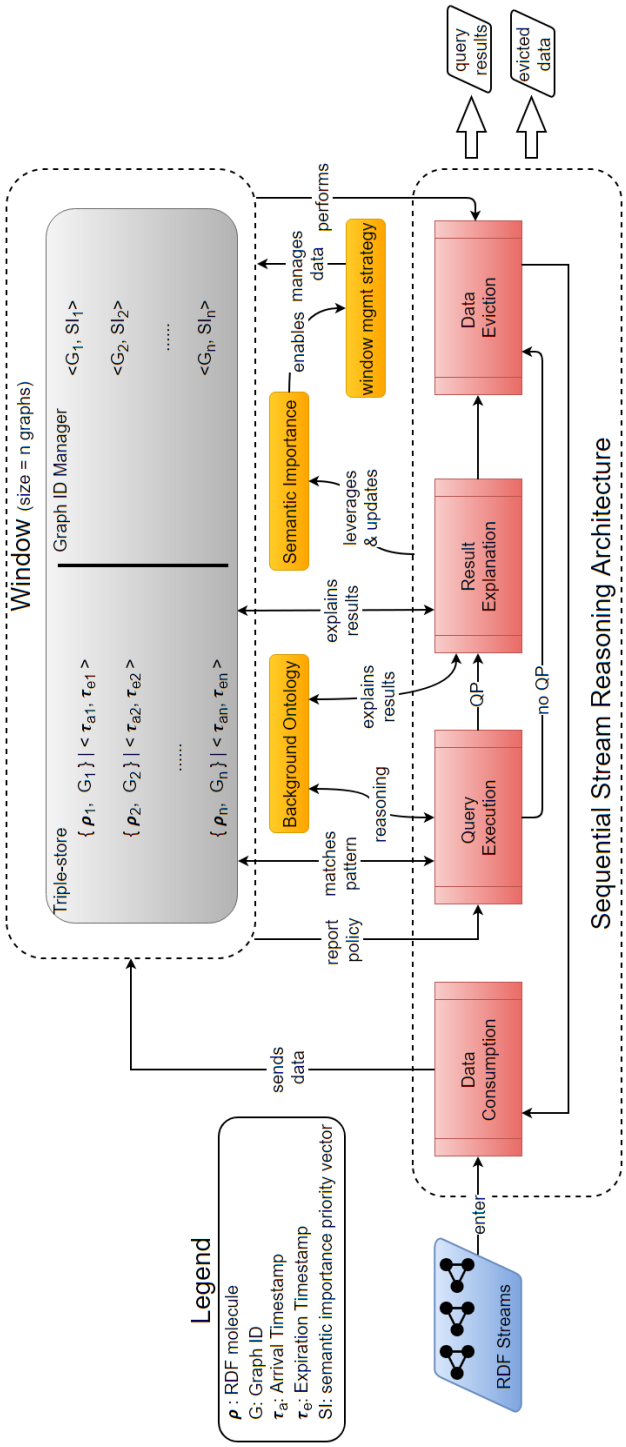


Figure 4.1: Sequential Stream Reasoning Architecture.

before the data arrival. SSRA, unlike what existing work does, employs standard SPARQL queries. This is for two reasons: (1) C-SPARQL [33], EP-SPARQL [42] and other extended SPARQL queries require different execution models and syntax that are not compatible. Even though RSP-QL [75] was proposed later with the goal to unifying the continuous queries, it currently is meant to be a reference model to explain the heterogeneity of stream reasoning engines. The execution engine is very challenging to develop as it will have to cover all the C-SPARQL/CQELS/other operational semantics which possibly will lose the responsive performance. (2) the requirement of "continuity" can be easily realized by forming a loop in SSRA, rather than employing some continuous query syntax or execution models. In fact, even the continuous SPARQL queries, such as C-SPARQL, need to be translated into the continuous query part (which includes the window and stream information), and the static query part (which is basically the SPARQL query). Thus, the mechanism of executing a continuous SPARQL query is essentially to execute a translated SPARQL query with associated window and stream definitions. This is the same as what SSRA does.

In SSRA, reasoning happens during the query time. This provides an advantage that only the necessary entailments for the answer will be computed. However, it is worth pointing out that reasoning does not have to happen at the same time as the query does. One example is materialization [45], which is performed iteratively: the materialized snapshot of the database is always updated as long as the new data arrives.

4.2.4 Result Explanation

If query participation (QP) is specified, the window needs to know the query participation status of every named graph after the query result is delivered. Result Explanation Component works in a way to trace back to the provenance of the query result; that is to find which named graphs participated in the query process. The result is explained by proof trees[9].

[9]For an example, please refer to `http://docs.stardog.com/#_proof_trees`

A(n) (inferred) result can be obtained by both explicit triples and intermediate inferences. Only the query participation statistics of those graph IDs belonging to explicit triples will be recorded.

In each run iteration, semantic importance has to be updated so as to re-rank the data. Based on different window management strategies, semantic importance is updated differently. For example, in FEFO strategy, the window is constantly looking for expired data. The important data is those unexpired. If the data expires, it will be moved to the deletion pool for eviction. For strategies such as FE-LFU-FO, the window updates semantic importance based on the expiration timestamp and query participation frequency. If the data is expired, it will be moved to the deletion pool for eviction regardless how frequently it used to participate in the query. If the data is not expired, it will be ranked based on the frequency. Other window management strategies will follow the same pattern to update the semantic importance. One thing to notice is that, rules to update semantic importance is only determined by window management strategies, without human-in-the-loop required.

4.2.5 Data Eviction

This component evicts the named graphs ranked at the bottom in the window. Depending on different window management strategies, data is evicted differently. For example, FEFO collects one statistic – the expired data count (E_d). E_d increments by one if a named graph is expired. FE-LRU-FO collects two statistics – E_d and the most recent query participation timestamps (τ_{qp}), for every named graph. E_d increases by one if a named graph is expired. Every valid named graph's old τ_{qp} is replaced with the corresponding new τ_{qp}. FE-LFU-FO collects two statistics – E_d and every named graph's query participation frequency counts (δ_{qp}), in the latest cycle. E_d increments by one if a named graph is expired, and every valid named graph's total query participation frequency counter (f_{qp}), adds the corresponding δ_{qp}. The window re-ranks the data immediately after semantic importance is updated.

4.3 Implementation

In SSRA, the most important part is the window. All of the components are constantly interacting with the window. SSRA implements and configures the window as follows.

Triple-store: SSAR has been implemented using two off-the-shelf triple-stores: AllegroGraph v5.0.2[10] and Stardog v4.0 RC3[11]. Both provide Java APIs and step-by-step tutorials. AllegroGraph is an efficient modern graph database by Franz Inc[12]. It supports up to OWL2 RL reasoning and full SPARQL 1.1. Stardog is a graph database by Complexible Inc[13]. It supports up to OWL2 DL & rule-based reasoning and SPARQL 1.1. Stardog supports reasoning explanation via proof trees, but AllegroGraph does not. Free versions of both products are used.

The SPARQL drop argument is leveraged to fully remove named graphs from the window. We avoid using the SPARQL delete argument because it only removes the statements in the graphs, not the graph ids. This will pollute the window's graph ID manager in Figure 4.1. Stardog supports both memory & disk-based databases, while AllegroGraph only supports disk-based databases. We have implemented three prototypes, focusing on Stardog memory & disk-based and AllegroGraph disk-based window stream reasoning system. As mentioned above, the reasoning abilities of these two triple-stores are different. In order to perform a fair comparison, we use a query that only requires RDFS reasoning.

Window Type: In this implementation, we employ the sliding physical window with a fixed size l. This is because, according to our extended window semantics, a physical window's size is a fixed value and thus controllable. In our extended semantics, a physical window step is a variable that is less than the window size. Thus, in order to control the window step (d), it is set to be 25%, 50%, 75% and 100% of l to see if the amount of data to be evicted has any influence on the query results.

[10] http://franz.com/agraph/allegrograph/
[11] http://stardog.com/
[12] http://franz.com/
[13] http://complexible.com/

4.4 Evaluation

The stream reasoning community has not yet come to consensus on the best method to evaluate stream reasoning applications. SRbench[76] was proposed as a general benchmark system designed to test streaming RDF engines. LSBench[77] was proposed to focus on assessing different Linked Stream Data (LSD) applications' capabilities. CSRBench[11] was proposed with a special emphasis on the effects of operation semantics for stream reasoning applications. More recently CityBench[78] was proposed to target smart city applications. Other work includes Heaven [79] and YABench [80].

Just like Heaven benchmark, LUBM benchmark[18] data is used in this work. Even though LUBM is not designed for streaming environment, data-set is carefully converted into a RDF stream format. LUBM provides a well-constructed ontology describing the relations among universities, professors and students etc. It also features a data generator which accepts customized parameters to generate arbitrary ABox data.

LUBM data generator produced 6,031,109 ABox triples. A data source generates streaming data by disk-reading this generated data line-by-line from a static n-triples file. The streaming data is configured as follows: (1) the streaming source packs either 1, 10 or 100 triples per named graph (T_{pg}); (2) each graph is assigned a unique graph id and expiration timestamp by the streaming source[14]; (3) the system assigns an arrival timestamp to each arriving named graph in a monotonically non-decreasing order.

The streaming data is then streamed to our three prototypes, where the window is configured as follows: (1) l is either 10, 100 or 1000 graphs; (2) d is either 25%, 50%, 75% and 100% of l; (3) the preregistered query, as shown in Listing 4.1, requires RDFS reasoning; (4) the background ontology is preloaded into the system.

Listing 4.1: SPARQL Query

```
PREFIX rdf:<http://www.w3.org/1888/02/22-rdf-syntax-ns#>
PREFIX lubm:<http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
```

[14] `http://streamreasoning.org/slides/2015/10/sr4ld2015-02-rsp-extensions.pdf`, Slide 7, from "Streaming Reasoning for Linked Data 2015" by J-P Calbimonte, D. Dell'Aglio, E. Della Valle, M. I. Ali and A. Mileo.

```
SELECT DISTINCT ?s
WHERE { ?s rdf:type lubm:Professor. }
```

Our evaluation platform specifications include 14.04 64bits Ubuntu LTS operating system, Intel(R) Xeon(R) CPU E5-2620 v2 @2.10GHz, 2040MB memory, and 16GB HDD.

Together with the generated ABox data and provided LUBM ontology, a ground truth of 27,192 results are obtained, and 252 experiments are conducted. Key performance indicators of each implementation are recorded, including memory consumption, quey runtime, result explanation and data eviction under different configuration in combination of streaming data and window. Result explanation time is not recorded for all the FEFO strategy and other strategies with the d = 100% of l, because the FEFO does not need result explanation, and it does not make sense to explain result as the whole window will be dumped. Using these results we ask the following questions:

1. What are the effects of different windows from different producers (AllegroGraph v.s. Stardog), configurations (disk-based v.s. memory-based) and semantics (different combinations of l and s)?

2. How do various strategies perform under different combined configurations of the streaming data and window[15]?

3. What are the trade-offs to consider when deploying SSRA in different scenarios?

We show two figures of the total 46 visualizations[16] generated from the results to answer the first two questions. The third one will be discussed shortly. For the sake of convenience, the following abbreviation is used: prototypes are abbreviated such that SM denotes Stardog memory-based window, SD denotes Stardog disk-based window and AD denotes AllegroGraph disk-based window. Each test case is labeled as <prototype abbreviation>_<window management strategy>_<streaming

[15]Currently we are using F-measure to evaluate the performance.

[16]Please refer to our github repository for all visualizations: `https://github.com/raymondino/SequentialStreamReasoningArchitecture`

data configuration>. For example, SD_FE-FI-FO_1 denotes a Stardog disk-based window with the FE-FI-FO strategy to process RDF streams with 1 triple per graph.

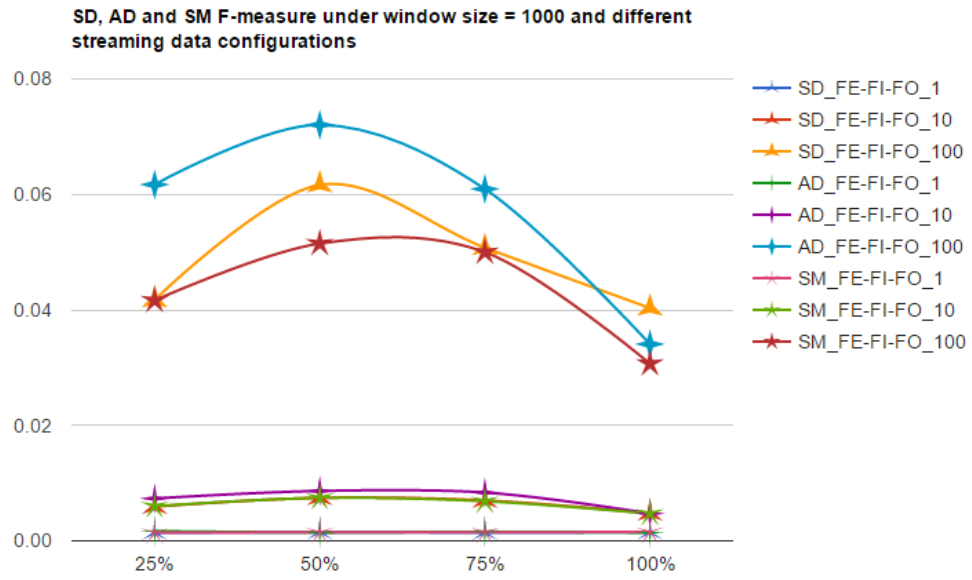


Figure 4.2: F-measure of Different Windows with FEFO

Figure 4.2 shows the F-measure performances brought by different windows. Some easy observations are: (1) the F-measure increases as the streaming throughput increases; (2) the F-measure at d = 50% l is always best, and 100% is always worst for all cases; (3) AD_FE-FI-FO performs similarly as others do when streaming throughput is 1 triple per graph, but outperforms as the streaming configuration increases; (4) SD_FE-FI-FO and SM_FE-FI-FO compete with each other in each test run.

The above observations can partially answer the first question, with the points that different triple-store producers do affect the F-measure performance, but different window configurations do not have a significant influence. The greater the streaming input, the more influences are made on F-measure. However, in order to give a thorough answer we need to look at other metrics before assessing the overall performance. For example, AD_FE-FI-FO gives the best F-measure, but does it take more time to execute the query and data eviction?

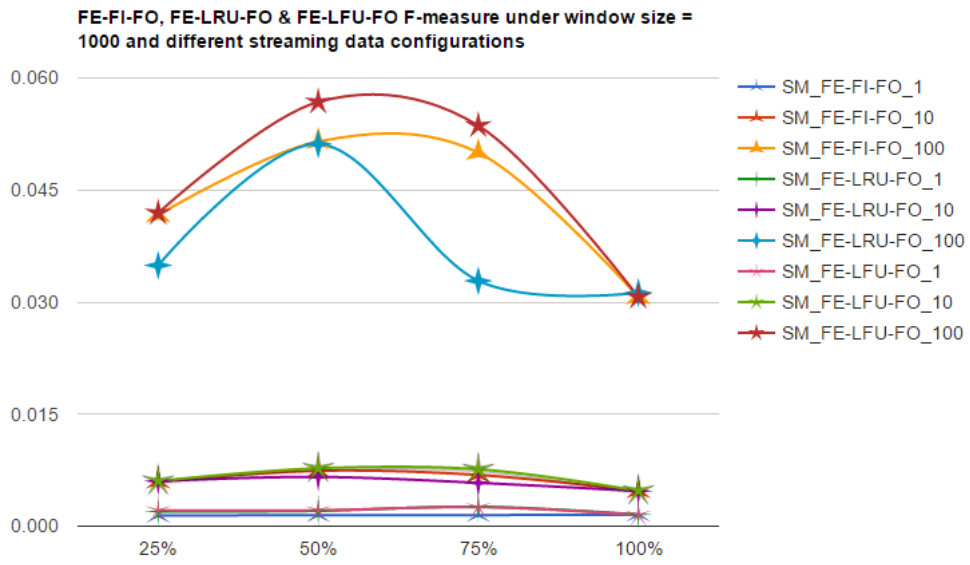


Figure 4.3: F-measure of SM Window with Different Strategies

Figure 4.3 shows the F-measure performance by different strategies for the SM windows with different streaming data configurations. The observations are: (1) F-measure increases as the streaming throughput increases; (2) d = 50% l has the biggest F-measure score, 100% has the smallest score for all the cases; (3) for the same streaming data configuration, FE-LFU-FO always performs best, followed by FE-LRU-FO and then FE-FI-FO.

These observations can answer the second question from the perspective of big window size. Nevertheless, does window size affect these strategies' performances? Does it take longer time to explain all of the inferences? If yes, is it worthwhile to sacrifice system responsiveness for a better F-measure?

Additional observations are as follows: (1) F-measure score: our raw experimental results have shown that F-measure increases as the window size increases. The F-measure score is also affected by different triple-stores. We believe this is because AlllegroGraph's and Stardog's inner implementation engines and mechanisms are different[17]. (2) Memory consumption: when l = 10, cases of T_{pg} = 1 require 3 times on average the memory of bigger streaming configured cases. When l = 100, the overall memory consumption decreases as the eviction amount (window step) in-

[17]Exploration and explanation of triple-stores' inner implementations are out of the scope of this dissertation

creases. When l = 1000, the overall memory consumption increases as the eviction amount (window step) increases. However, within this evaluation, no significant differences are observed between memory-based window and disk-based window. Small window and streaming throughput cases usually requires more memory. Average FE-FI-FO strategy memory consumed is 41.93MB, FE-LRU-FO is 40.4MB and FE-LFU-FO is 39.59MB. (3) Query time: AD requires the most query time for all cases. SD requires less time than AD but more than SM. The query time increases as window size and streaming input increases. One potential reason for this is because a disk-based window needs some IO time when executing a query, which takes more time than a memory-based window. Average query time for all FE-FI-FO cases is 18ms, FE-LRU-FO is 15ms, FE-LFU-FO is 13ms. (4) Explanation time: result explanation time increases approximately linearly as window and streaming input increases. In most cases, FE-LFU-FO takes longer time to explain. Average FE-LFU-FO explanation time is 90371ms, FE-LRU-FO is 86100ms. (5) Eviction time: eviction time increases as eviction amount, window size and streaming input increases. AD eviction time is very fast, 30ms on average. SD on average is 2879ms. SM on average is 4042ms.

According to the window and streaming data configurations, there can be either 10, 100, 1,000, 10,000 or 100,000 triples in the window during one processing loop. We identify a small case where 10 or 100 triples are processed, a medium case where 1,000 or 10,000 triples are processed, and a large case where 100,000 triples are processed. Together with Table 4.1, we present a thorough comparison among triple-stores, window types and data management strategies under these scenarios, as well as to help answer the third question above-mentioned, i.e., what are the trade-offs to consider when deploying our architecture in different scenarios?

Each value in the table is calculated from raw experimental results. Please refer to our github repository for these raw results. In the 10 triples/window scenario, for example, disk-based window's F-measure score is averaged from AD_FE-FI-FO, SD_FE-FI-FO, SD_FE-LRU-FO & SD_FE-LFU-FO test cases, while memory-based window's score is averaged from SM_FE-FI-FO, SM_FE-LRU-FO & SM_FE-LFU-

Table 4.1: The Trade-off Table

Scenario	Tuple # in window	window Config.	SPARQL (ms)	F-measure	Explain (ms)	Eviction (ms)	Memory (MB)
Small	10	AllegroGraph	20.25	4.35E-05	-	69.25	22.00
		Stardog	5.13	3.93E-05	-	11.00	56.75
		Disk	8.69	5.43E-05	1970.25	26.94	57.00
		Memory	5.00	5.40E-05	1790.25	9.50	56.25
		FE-FI-FO	10.17	4.07E-05	0	30.42	45.17
		FE-LRU-FO	5.00	5.68E-05	1571.00	11.50	61.38
		FE-LFU-FO	4.63	7.19E-05	2189.50	11.50	74.75
	100	AllegroGraph	11.63	2.97E-04	-	73.38	58.38
		Stardog	5.00	3.05E-04	-	20.81	42.81
		Disk	6.75	3.70E-04	8058.81	34.94	41.88
		Memory	4.92	3.84E-04	8178.63	20.17	41.08
		FE-FI-FO	5.41	2.27E-04	0	28.75	36.00
		FE-LRU-FO	5.41	4.17E-04	7846.44	21.38	37.19
		FE-LFU-FO	5.00	4.44E-04	8391.00	21.25	36.19
Medium	1,000	AllegroGraph	12.17	1.53E-03	-	89.42	59.08
		Stardog	6.46	1.55E-03	-	164.96	36.21
		Disk	8.35	1.68E-03	25372.54	129.35	40.52
		Memory	6.56	1.74E-03	25265.42	169.14	36.97
		FE-FI-FO	8.36	1.54E-03	0	139.78	43.53
		FE-LRU-FO	6.96	1.82E-03	24355.21	151.33	37.54
		FE-LFU-FO	7.04	1.85E-03	26282.75	151.42	33.21
	10,000	AllegroGraph	15.63	7.25E-03	-	90.00	33.38
		Stardog	9.56	6.49E-03	-	1575.38	37.93
		Disk	11.34	6.77E-03	122612.38	1020.72	39.41
		Memory	9.22	6.77E-03	144213.08	1150.50	36.28
		FE-FI-FO	11.58	6.75E-03	0	1080.25	36.42
		FE-LRU-FO	9.50	6.18E-03	113897.38	1556.94	40.81
		FE-LFU-FO	9.81	6.89E-03	113782.88	1004.19	32.13
Large	100,000	AllegroGraph	127.75	5.72E-02	-	187.25	23.50
		Stardog	76.13	4.60E-02	-	27949.00	36.13
		Disk	134.31	4.64E-02	261564.75	17122.81	31.13
		Memory	15.17	4.22E-02	271454.38	32248.08	36.67
		FE-FI-FO	93.33	4.97E-02	0	18695.08	31.92
		FE-LRU-FO	84.88	3.57E-02	263054.63	27104.50	33.63
		FE-LFU-FO	66.50	4.58E-02	269964.50	27470.63	35.75

FO test cases. We would like to highlight an empirical comparison of the two window types, the disk-based window (DC) and the memory-based window (MC).

The detailed comparisons of all aspects for different scenarios are provided as follows: (1) SPARQL query time: for every scenario, DC is slower than MC. As the scenario increases, DC grows much faster than MC. The ratio between 100,000 triples/window and 10 triples/window for DC is 15.45, whilst MC is 3.34. Though DC is less restricted by storage space, its query time slows down the system response time, and this effect will become worse as the scenario size grows. However, this is expected, as accessing disk when executing query is always slower than accessing the memory. (2) F-measure: for every scenario, DC's and MC's F-measure scores are very similar. This means both types are able to provide same level correctness. F-measure increases as scenario increases, this is because the more data in the window, the more correct results can be calculated, which raises the F-measure. (3)

Reasoning Explanation Time: in most scenarios, MC's explanation time is slower than DC's. The difference increases significantly as the scenario size increases. Reasoning explanation is very time-consuming. Strategies (such as FE-LFU-FO) requiring reasoning-explanation provide better F-measure when the scenario is small and explanation time is quick, but provide similar F-measure when the scenario is large and explanation time is slow. (4) Eviction Time: in small scenarios, DC is slower; in medium and large scenarios, MC is slower. The difference between DC and MC increases significantly as the scenario size increases. This indicates a bias towards DC for large RDF streams, and MC for small RDF streams. (5) Memory Consumption: Both DC and MC consume similar memory for each scenario. This could be because Stardog triple-stores are implemented very efficiently, but again, explaining this requires the knowledge of inner mechanisms of Stardog, which is out of the scope of this work.

Under the small case, AllegroGraph's query and eviction time is several times that of Stardog. Disk and memory performs equally on F-measure, though disk requires more time to query, evict and explain. FE-LFU-FO performs best in F-measure, followed by FE-LRU-FO then FEFO. Though FEFO needs more time to query and evict, the explanation time required by FE-LRU-FO and FE-LFU-FO is significantly greater. The trade-offs in deploying our framework under the small scenario is dependent on the use case. If system responsiveness is the first class citizen, a FEFO strategy will be chosen since it does not require explanation and provides a fine F-measure. Stardog memory window can be chosen since it provides faster execution time and better F-measure. If F-measure is most important, FE-LFU-FO is the right strategy. Stardog memory window is the best as memory window provides less explanation time. It is also noticed that d = 25% or 50% of l provides better F-measure.

For medium cases, Stardog's eviction time increases significantly. Though Stardog's query time is better than AllegroGraph's, the difference is very small when compared with the eviction time. Disk window provides less eviction time; its other metrics are similar as memory windows. FE-LFU-FO is the best at F-measure scores, but is traded for longer explanation time. Actually FEFO provides decent F-

measure without explanation time. Overall, AllegroGraph disk window with FEFO is most suitable for this scenario. d = 50% or 75% of l provides best F-measure as well.

For large cases, AllegroGraph's eviction time, F-measure and memory performs better, though query is slower. Disk window query time is 9 times greater than the memory dependent graph database, but provides better F-measure, explanation , eviction time and memory. FEFO performs best in F-measure but it spends more time on query, with smallest eviction time and memory consumption. Hence, AllegroGraph disk window with FEFO is most suitable for this experiment's use case. It is also recommended to use d = 50% of l to provide the best F-measure.

4.5 Summary

This chapter introduces the sequential stream reasoning architecture, and shows how semantic importance model can be used in stream reasoning applications. The window is implemented with two off-the-shelf triple-stores (Stardog or AllegroGraph), with different configurations (memory-based or disk-based). The benchmark leverages LUBM dataset by streaming it, and constantly looking for the answers of the query "who is a professor?", which requires RDFS reasoning. This can be seen to look for professors on a time line by streaming the LUBM dataset. In order to illustrate how to use semantic importance in stream reasoning with SSRA, four aspects τ_a, τ_e, f_{qp} and τ_{qp} are chosen to form three window management strategies: FE-FI-FO with its priority vector $[\tau_a, \tau_e]$, FE-LFU-FO with $[\tau_e, f_{qp}, \tau_a]$, and FE-LRU-FO with $[\tau_e, \tau_{qp}, \tau_a]$. The SSRA implementations have demonstrated a physical sliding window, but the temporal window is easy to implement as well according to the extended window semantics. The architecture operational semantics is very flexible, as the data consumption component is able to realize different operational semantics. In all three implementations, the report policy is "on window close". The continuous processing is enabled by the inner loop of the architecture. The query is a pure SPARQL query, which is different from most of the existing work. The window is implemented in the triple-store, with a graph ID manager that indexes graph IDs, and updates data semantic importance priority vectors.

The benchmark results show that the architecture is scalable, but for strategies involved with query participation aspects, as the window size increases, the result explanation time increases significantly. Thus, in order to use query participation aspect, the window size has either to be kept relatively small, which is one of the incentives why semantic importance is proposed, or to use some filtering functionality such as query relevance aspect in the semantic importance model in order to reduce the data in the window. In the next chapter, this dissertation will show how to keep the window size small with the architecture and semantic importance, as well as show how the proposed model and infrastructure can be leveraged for real-life use cases.

CHAPTER 5
USE CASES

In this chapter, two use cases are implemented based on semantic importance and the sequential stream reasoning architecture. The use cases leverage different combinations of semantic importance aspects which support different window management strategies. This chapter aims to show how semantic importance is determined, deployed in stream reasoning use cases, as well as how semantic importance impacts the system performance.

5.1 Soccer Offside Detection

This use case detects soccer offside offence with a particular focus on the accuracy, error and running performance of window management strategies based on the semantic importance.

5.1.1 Background

Soccer (a.k.a. association football) is one of the most influential and competitive sports across the globe. In soccer games, one common offence among others is called "offside offence". FIFA[18] defines it in a comprehensive way. For details, please refer to Rule 11 in FIFA laws of the game [15]. However, it is worth to mention a key point: that a player is at an offside position is determined at the moment when the ball is played by this player's teammate; once identified, this status will be retained until the ball is touched again. For example, when Player A2 passes the ball to his teammate Player A1 who is at an offside position, even if Player A1 relocates himself to touch the ball in an onside position, it is still considered as an offside offence.

In order to give accurate offside decisions, a linesman (assistant referee) in his/her assigned half field needs to keep close track of the second-last defender, the ball-passing moment and any attackers at the offside position. However, the fast

[18] `https://www.fifa.com/`

tempo and heated atmosphere of the soccer game, together with linesmen' limited visual field and potential tiredness, can contribute to erroneous offside judgments, which may allow illegal attacking opportunities or disallow legal ones, and consequently affect the outcome of the game.

A number of studies have focused on the prevalence of offside mistakes by linesmen. [81] investigates the accuracy of offside judgments in the English Premier League, with a reported error rate of 17.5%. This is possibly because the English linesmen tend not to signal in doubtful situations, so there is an overall bias towards non-flag errors (773 non-flag errors vs 95 flag errors). Another study [82] analyzes 337 offside judgments in all 64 games during the 2002 FIFA World Cup, and finds the error percentage to be 26.2%. [81] also examines the accuracy of offside judgments during the 2006 FIFA World Cup, and finds an error rate of 7.6%, significantly lower than that of [82]. This study argues that the reduction in error rate might be due to different approaches the authors take to count errors. The study also reveals higher rate of non-flag errors (9.6%) than flag errors (6.9%), within the overall error rate of 7.6%.

A 2013 report [83] analyzes several offside detection systems, and proposes a number of evaluation criterion such as accuracy, fairness, system delays, feasibility, cost and intrusiveness of the system development and deployment. Offside event detection methods may be based on various technologies, such as image processing & computer vision [84], and stereo vision tracking [85]. In addition to vision-based tools, sensor-enabled equipment is also employed in this scenario [86] [87]. An example is the sports vest provided by GPsports[19], which has been already used in the professional soccer. The GPsports vest is equipped with sensors that track volume, intensity and work rate of matches played or training by recording and analyzing real-time data such as player position, velocity and heart rate. According to its official website, this system has been successfully adopted by 150 clients of 10 sports, including prestigious soccer teams such as Real Madrid C.F. and Chelsea F.C.

[19] `http://gpsports.com/`

Overall, sensor-based systems have great advantages by providing real-time, high-frequency, high-speed motion data of all available participants, but they lack the ability to determine the precise positions of players with reference to relevant boundaries required for offside calculations. Vision-based technologies are getting more precise for player boundary proximity detection, but output credibility and cost of computation have been major concerns.

The fact that gears like the GPsports vest have become more pervasive provides both opportunities and challenges for soccer. The opportunities lie in analysis of the real-time streaming data for players' kinetic and biological statistics such as location, velocity, acceleration, heart rate, breathing, and so forth. The challenges also come with the data: these systems require technologies to efficiently process constantly-updating, high volume and heterogeneous real-time streaming data in order to better analyze player performance for coaches, provide timely input of relevant officiating content for referees, and help spectators to enjoy the match. While contemporary stream processing systems are able to store and query low-level data at high speed, they do not typically have a framework to also furnish higher-level information that is implicit in the data. This is because this implicit information (e.g. player tiredness, opponent strategy, etc.) is not directly revealed by a stream processing system, which can only process explicit data (e.g. player position, velocity) and support window-based queries. In a soccer scenario, for example, a pure stream processing system would not be able to tell the coach that his left fullback is becoming tired and could be exploited by the opponent's fast right winger, or make other tactical suggestions.

5.1.2 Approach

Timestamp metadata, such as arrival timestamp, has each been investigated as a key feature for window management. But it only scratches the surface of possible ways to establish a specific window management strategy. Overall, a good window management strategy implementation will look for the right combination of heuristic guidelines that best capture the important information for the queries at hand. In considering these heuristics, there are certain domain-independent contributors (e.g.

time and use frequency), but there are also domain-dependent contributors that stem from the requirements of the particular use case, such as the goal of the use case, the specific answers expected from the data, the background knowledge and constraints, specific space and time limitations, etc. Knowing these requirements, humans can often determine efficient policies about which data will contribute to the query. For example, because we know that only an attacker can commit an offside offence, we can conclude that positional data for the defenders will never contribute to the query, and so is unimportant and can be flushed from the window.

5.1.3 Date Stream

The data was collected during a practice game played on a half-sized soccer field. It includes information for one referee, two teams, eight players (including one goalkeeper) per team, and four balls (one ball is in play while the other three are backup balls). Sensors, installed in the center of the balls, and on the shin-guards and gloves of the players and referees, constantly stream kinetic data at high frequencies, such as position, velocity, acceleration, and directional vector. The ball sensor is 2000Hz, the player/referee sensor is 200Hz. For more details, please refer to [88].

The game duration is two halves of thirty minutes each. In addition to the streaming data, a full video of the game is provided. This use case hires a soccer referee certified by United States Soccer Association and National Intercollegiate Soccer Officials Association as a domain expert. He reviewed the game video and provided a list of suggested officiating decisions for a total of twenty relevant scenarios, where decisions are given as "offside offence", "no offence", or "unsure". His decisions are used as a ground truth to evaluate system precision.

5.1.4 Ontology

For this use case, the overall ontologies are modularized into three sub-ontologies: (1) the individual ontology that encodes the specific match information such as teams and players in each team; (2) the offside offence ontology that concentrates on the offside offence detection from the streaming data; (3) the query relevance ontology that focuses on filtering out offside-irrelevant data from the streaming data.

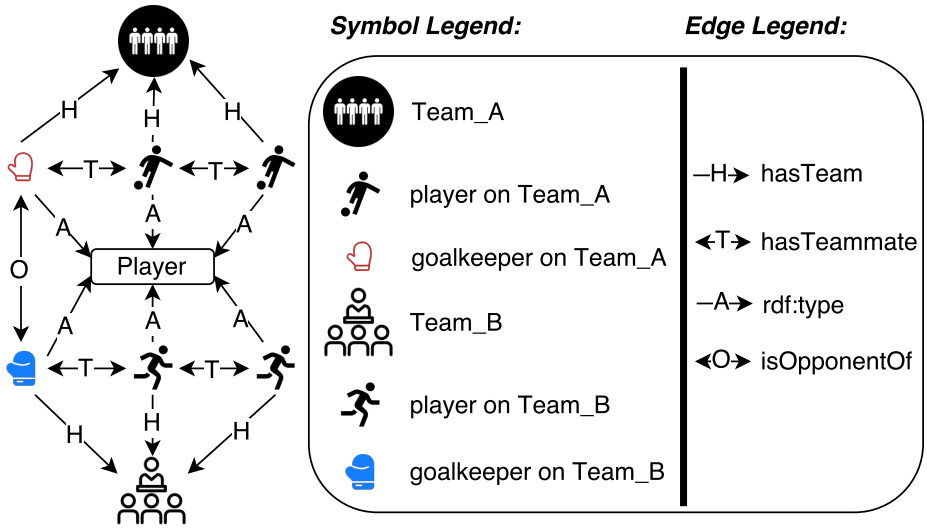


Figure 5.1: Individual Ontology

For the purpose of illustration, three players are drawn from each team in Figure 5.1. All of the player individuals have a type of **Player**, which is a concept in the offside offence ontology. *hasTeam* is a functional property with a domain of **Player** and a range of **Team**. Both *hasTeammate* and *isOpponentOf* are symmetric properties with **Player** as their domain and range. *hasTeammate* is transitive, meaning that a player can be his own teammate.[20] *isOpponentOf* is encoded only between two goalkeepers.

Figure 5.2 shows the major classes and relations in order to describe the concept of the offside offence. When developing this ontology, FIFA laws of the game [15] is used as the text corpus to extract a list of the offside relevant terms. The term list was then carefully examined and curated to create a conceptual model, which was refined and extended to create the offside offence ontology[21]. In a soccer game, usually the player who is touching the ball is considered to be an attacker, along with his teammates. The concept to describe a player touching the ball is modeled as **BallLastToucher**, and the attackers as **Attacker**. An **Attacker** instance can either be the type of **BallLastToucher** or (*hasTeammate some* **BallLastToucher**). **Defender** is defined as (**Player** *and* (*playsAgainst some* **At-**

[20] This is a modeling choice, not an error.

[21] For more details, please refer to `https://tw.rpi.edu/web/Courses/Ontologies/2016/projects/soccer`

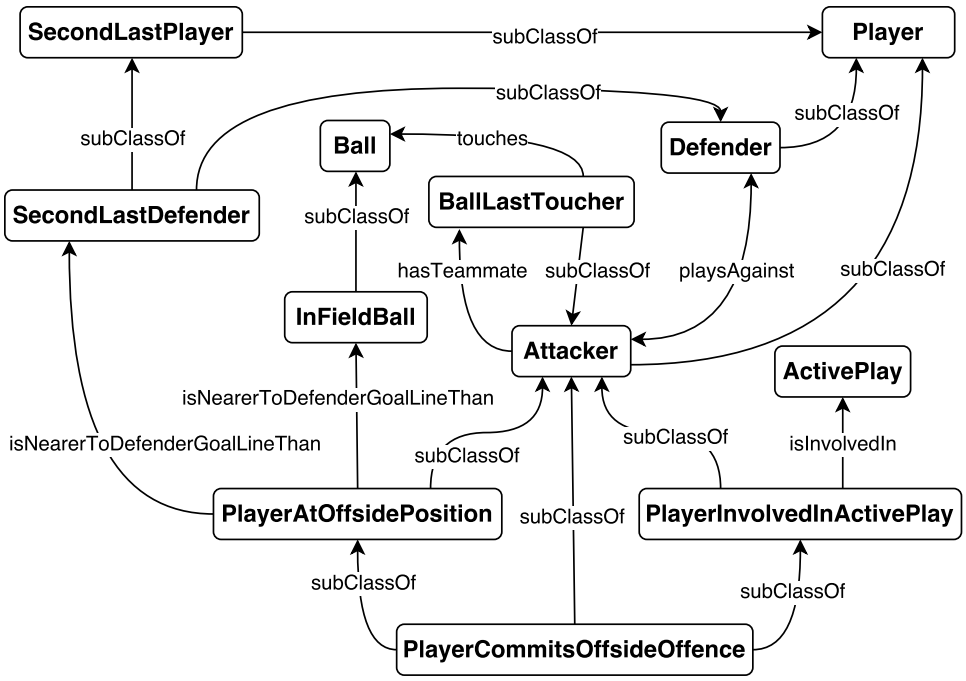


Figure 5.2: Offside Offence Background Domain Ontology

tacker)), where *playsAgainst* is a symmetric property chain as (*isOpponentOf o hasTeammate*). Instead of manually labeling play-against relationship for each pair of opponent players, leveraging this property chain and a reasoner, *playsAgainst* is automatically established. This saves lots of tedious work, keeps a compact ontology, and shows the value of the semantics. **SecondLastDefender** is one of the key concepts to define the soccer offside. It is the subclass of both **SecondLastPlayer** and **Defender**. We directly annotate a player as **SecondLastPlayer** for each team by selecting the player who is the second-nearest to his own goal line among his team. This is to reflect the real world scenario that the linesman should always keep up with the second last player of his assigned half. The ball being played is an **InFieldBall**, whereas the other balls are **BackupBall**. When a player is classified as an **Attacker**, and *isNearerToDefenderGoalLineThan* both **SecondLastDefender** and **InFieldBall**, then he/she is further classified as **PlayerInOffsidePosition**. If a player touches the ball or challenges an opponent, he/she is classified as **PlayerInvolvedInActivePlay**. According to the symplified definition of the soccer offside offence in Chapter 1, when a player is classified as both **PlayerInOffside-**

Position and **PlayerInvolvedInActivePlay**, he/she, who is therefore classified as **PlayerCommitsOffsideOffence**, commits an offside offence.

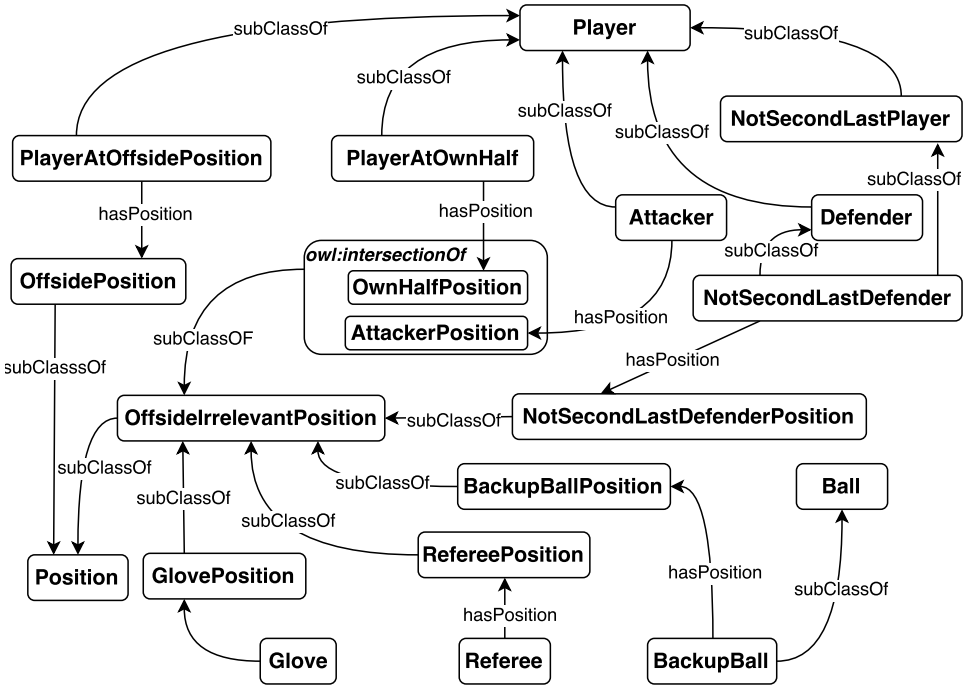


Figure 5.3: Soccer Offence Query Relevance Ontology

Figure 5.3 illustrates the query relevance ontology. This use case is scoped to answer the question of who commits an offside offence. If a data item does not contain offside offence information, there is no need to execute the query on it. The purpose of this ontology module is to identify and eliminate the irrelevant data by leveraging semantics to ask another question (an example is in Listing 5.2) of what the offside irrelevant positions are prior to executing the offside query. For the data set of this use case, all ball positions, referee positions, goalkeeper glove positions and player's positions are present. In this ontology, glove's position **GlovePosition**, referee's position **RefereePosition**, back up balls' positions **BackupBallPosition**, positions of defenders who are not the second last player **NotSecondLastDefenderPosition**, positions of attackers who are in their own half (*owl:intersectionOf* (**AttackerPosition** *and* **OwnHalfPosition**)) are all offside irrelevant positions **OffsideIrrelevantPosition**, which means that these types of data will not provide

necessary information to the query at hand, thus can be filtered out before the target query execution.

5.1.5 Data Annotation

The original streaming data comes in comma-delimited format with the schema as $(sid, ts, x, y, z, |v|, |a|, v_x, v_y, v_z, a_x, a_y, a_z)$, where sid denotes the sensor ID, ts denotes the generation timestamp in pico-seconds, (x, y, z) denotes a sensor coordinate in 3D space, and $(|v|, |a|, v_x, v_y, v_z, a_x, a_y, a_z)$ denotes sensor velocity, acceleration and their directions. In this use case, (sid, ts, x, y, z) will be used.

Each shin-guard of a player is installed with a sensor. A player's position is obtained by averaging his sensor coordinates. With the coordinates of the field markings, and player/ball positions, position coordinates can be calculated to get information, such as which balls are on or off the field or which players are in or not in their own half, etc. Euclidean distance is used to determine if there is an active play of either a ball-touch or an opponent-challenge. All players are ranked according to their distances from the ball in play. The Player-ball distance is not based on the averaged player coordinate, but the actual left and right shinguard sensor to ball sensor. This increases the annotation accuracy. That a player touches the ball is determined when he has the nearest distance to the ball that is less than 0.6 meter. An opponent-challenge scenario is also determined when opponent-opponent distance is less than 0.6 meter. Once a ball-touch is detected, all attacking players' distance from the defender goal line and the ball will be calculated. Each player's distance from his own goal line will also be calculated so as to determine who is the second-last player of each team. The data is then annotated as in Table 5.1.

5.1.6 System Implementation

The detection system is implemented with SSRA, where a SPARQL query in Listing 5.1 is deployed. This query is simple, but requires Description Logic (DL) reasoning.

Listing 5.1: Offside Offence Detection Query

Table 5.1: Data Annotation Table

data	annotation
player annotation	
playerX position	*:playerX :hasPosition :playerPosition_ts*
playerX is in own half	*:playerX a :PlayerInOwnHalf*
playerX is not in own half	*:playerX a PlayerNotInOwnHalf*
playerX is a second-last player	*:playerX a SecondLastPlayer*
playerX is not a second-last player	*:playerX a NotSecondLastPlayer*
playerX touches the ball	*:playerX :isInvolvedIn :ball_touch*
if *playerX* challenges *playerY*	*:playerX :isInvolvedIn :opponent_challenge* *:playerY :isInvolvedIn :opponent_challenge*
ball annotation	
ballA position	*:ballA :hasPosition :ballPosition_ts*
ballA is in the field	*:ballA a :InFieldBall*
ballA is off the field	*:ballA a :BackupBall*
referee & glove annotation	
referee position	*:referee :hasPosition :refereePosition_ts*
glove position	*:glove :hasPosition :glovePosition_ts*

```
PREFIX so:<http://tw.rpi.edu/Ontologies/2016/Soccer_Offside/>
SELECT ?player
WHERE { ?player a so:PlayerCommitsOffsideOffence.}
```

The data is sampled so that the balls' data update frequency is 20Hz, and the players' data update frequency is 10Hz. Sampling will not affect the result as both balls and players move only a short distance every 0.05 and 0.1 second.

A physical sliding window is leveraged. According to our annotation strategy, the maximum number of unique triples is 70, which includes the "*hasPosition*" triples from 1 referee, 16 players, 4 balls and 4 gloves, as well as a certain number of ball status triples, such as "*ball4 a InFieldBall(BackupBall)*", and player status triples, such as "*playerA4 a SecondLastPlayer(NotSecondLastPlayer, PlayerInOwnHalf, PlayerNotInOwnHalf)*", "*playerA4 isInvolvedIn ball_touch(opponent_challenge)*", etc. Thus the maximum physical window size (l_{max}) is set to be 70.

For this use case, semantic importance aspects including τ_a, τ_e, qrf, f_{qp}, and τ_{qp} are chosen to form the strategies shown in Table 5.2. All strategies labeled with QR perform query relevance filtering (Listing 5.2) before executing the offside

Table 5.2: Strategy Table

strategy	semantic importance priority vector
FE-FI-FO	$[\tau_e, \tau_a]$
FE-FI-FO-25%	$[\tau_e, \tau_a]$
FE-FI-FO-50%	$[\tau_e, \tau_a]$
FE-FI-FO-75%	$[\tau_e, \tau_a]$
QR-FE-FI-FO	$[qrf, \tau_e, \tau_a]$
QR-FE-FI-FO-25%	$[qrf, \tau_e, \tau_a]$
QR-FE-FI-FO-50%	$[qrf, \tau_e, \tau_a]$
QR-FE-FI-FO-75%	$[qrf, \tau_e, \tau_a]$
FE-LFU-FO	$[\tau_e, f_{qp}]$
FE-LFU-FO-25%	$[\tau_e, f_{qp}]$
FE-LFU-FO-50%	$[\tau_e, f_{qp}]$
FE-LFU-FO-75%	$[\tau_e, f_{qp}]$
QR-FE-LFU-FO	$[qrf, \tau_e, f_{qp}]$
QR-FE-LFU-FO-25%	$[qrf, \tau_e, f_{qp}]$
QR-FE-LFU-FO-50%	$[qrf, \tau_e, f_{qp}]$
QR-FE-LFU-FO-75%	$[qrf, \tau_e, f_{qp}]$
FE-LRU-FO	$[\tau_e, \tau_{qp}]$
FE-LRU-FO-25%	$[\tau_e, \tau_{qp}]$
FE-LRU-FO-50%	$[\tau_e, \tau_{qp}]$
FE-LRU-FO-75%	$[\tau_e, \tau_{qp}]$
QR-FE-LRU-FO	$[qrf, \tau_e, f_{qp}]$
QR-FE-LRU-FO-25%	$[qrf, \tau_e, f_{qp}]$
QR-FE-LRU-FO-50%	$[qrf, \tau_e, f_{qp}]$
QR-FE-LRU-FO-75%	$[qrf, \tau_e, f_{qp}]$

query in Listing 5.1. The percentage after each strategy indicates the percentage of l_{max}. For example, FE-LFU-FO-25% means FE-LFU-FO strategy is performed in a window with 25% of l_{max}, which is $70 \times 0.25 = 17$ data items.

This use case uses a different report policy that is not included in the SECRET model [9]. It is named as "on user-specified condition". Specifically, the offside query is fired whenever a different player touches the ball. It is important for this use case to execute the query based on ball touches, as all the status of the players will likely be changed upon each ball touch. Executing query based on time will potentially allow inconsistent data items, such as a player is both an attacker and defender.

The strategies executed are shown by the paths through the graph in Figure 5.4. Depending on the priority vector, Phase 1 is the selection of a base strategy

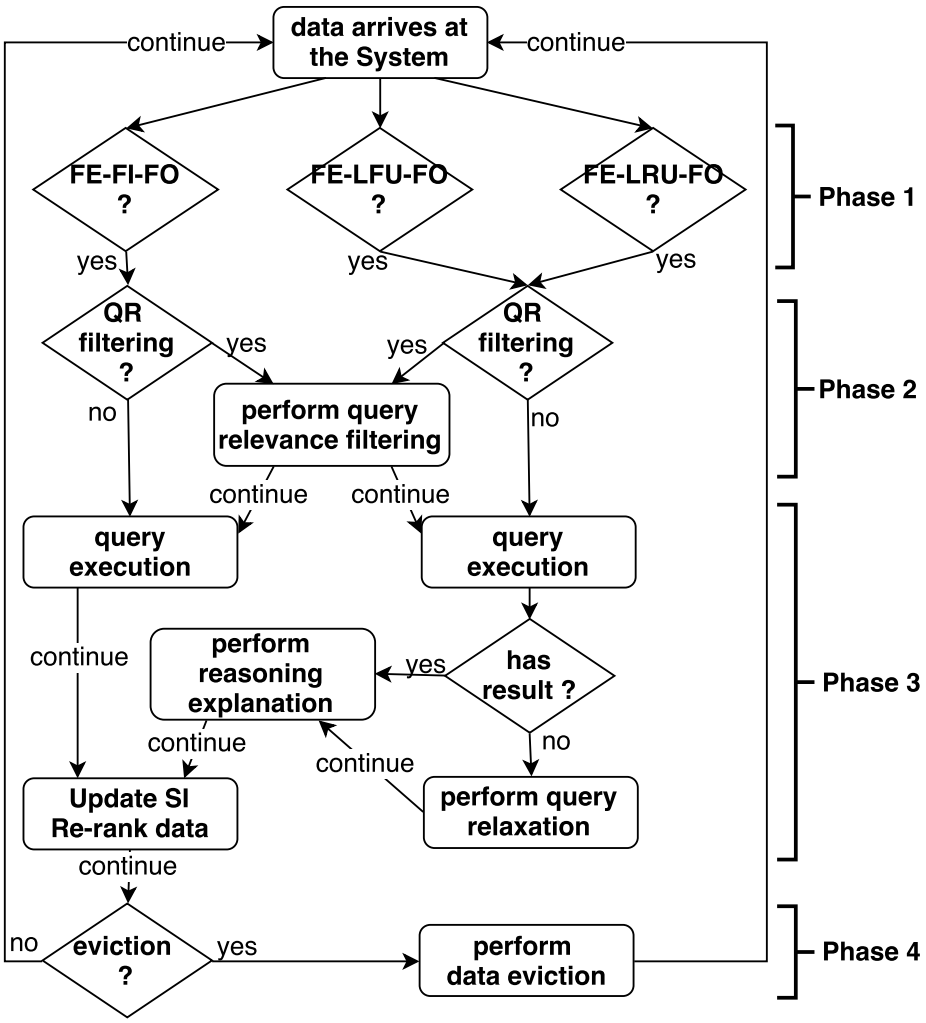


Figure 5.4: Strategy Execution Flowchart

family to run. If the SI priority vector has the first element of "qrf", the QR filtering is performed by executing the query in Listing 5.2.

Listing 5.2: Query Relevance Filtering Query

```
PREFIX so:<http://tw.rpi.edu/Ontologies/2016/Soccer_Offside/>
SELECT ?player
WHERE { ?position a so:OffsideIrrevelantPosition. }
```

Phase 3 executes the offside query when a new player touches the ball. If current strategy is in the *-FI-FO family, the system will go ahead to update the semantic importance and re-rank the data. If the current strategy is in the *-LRU/LFU-FO family, the system will need to check if the offside query returns any

results. If there are results, reasoning explanation is performed to track all the triples that contribute to the query. If no result, schema-aware query relaxation [89] will be performed to increase the scope of the query. For example, if the query returns no answer, the system continues to ask who touches the ball, who is the second-last player, who is nearer to the opponents' goal line than whom, etc. Listing 5.3 shows a relaxed query.

Listing 5.3: A Relaxed Query Example

```
PREFIX so:<http://tw.rpi.edu/Ontologies/2016/Soccer_Offside/>
SELECT ?s ?p ?o
WHERE {
        ?s a so:SecondLastPlayer.
        bind(IRI(rdf:type) as ?p).
        bind(IRI(so:SecondLastPlayer) as ?o)
}
```

Being able to preserve some partial necessary data and wait until the others arrive, and then answer the query, is a key feature for *-LRU/LFU-FO families enabled by τ_{qp} and f_{qp}. It also explains why they have higher accuracy – the offside offence judgement requires the window to preserve earlier information (player in offside position at the moment his teammate passes the ball) until later information (the player involved in an active play) arrives. After query relaxation, semantic importance priority vectors will be updated, and the window data will be re-ranked. Next, the system checks the percentage of l_{max} which the strategy comes with, and performs data eviction accordingly. Obviously, the use case expects that FE-FI-FO, QR-FE-FI-FO, FE-LFU-FO, QR-FE-LFU-FO, FE-LRU-FO and QR-FE-LRU-FO with l_{max} will perform equally well in accuracy.

5.1.7 Evaluation

Table 5.3 presents both the referee and system offside judging results. Table 5.4 and 5.5 show the averaged running performance and accuracy for each strategy family, respectively. Accuracy is computed by counting a correct answer only when referee gives ● and system gives ●, or when referee gives × and system gives ×. For

example, we calculate FE-FI-FO-25% accuracy as $11/20 = 0.55$ because it gives 11 correct results out of the total 20 cases.

From these tables, it can be observed that among all the strategies: (1) the *-FI-FO family intakes and evicts data the fastest, but has the slowest execution time and is more error-prone, especially when the window size is smaller; (2) *-LFU-FO and *-LRU-FO families both have slower throughput and eviction than *-FI-FO family, and require extra time to explain the inference, but maintain a good accuracy even if the window size shrinks; (3) the QR-* family's query and execution time is the fastest; (4) the results of FE-FI-FO, QR-FE-FI-FO, FE-LFU-FO, QR-FE-LFU-FO, FE-LRU-FO and QR-FE-LRU-FO at l_{max} are all identical, which validates our expectations above; (5) the use case situations at 21:11 and 54:46 makes all strategies fail; (6) no consensus is reached by all strategies for the unsure case at 58:11.

The reason for *-FI-FO family accuracy performance is that, when equipped with l_{max}, FE-FI-FO and QR-FE-FI-FO evict only expired/domain irrelevant data, so all unexpired/relevant data are preserved. However, when l shrinks, decisions to delete valid data from the window have to be made. According to the semantic importance priority vector $[\tau_e, \tau_a]$, if a data item is not expired, the one with the earliest arrival timestamp is ranked at the bottom, thus evicted. If the time interval/triple counts between the data of offside position and data of active play is larger than the deployed window size, it is impossible to have both of them in the window at the same time, therefore a false-negative (×) result will be given. This situation also causes a false-positive (●) or wrong foundation (○) judgment.

Figure 5.5.(a) shows that at the moment Player A2 touches the ball, Player B4 is the second-last defender; Player A7 is not in offside position. So Player A7 does not commit an offside offence when he touches the ball. According to our data annotation process, since Player A7 uses his left back heel to touch the ball in Figure 5.5.(b), his left shin-guard sensor is nearer to the defender's goal line than both the ball and Player B1, thus Player A7 is annotated as "*playerA7 isNearerToDefenderGoalLineThan ball*"; "*playerA7 isNearerToDefenderGoalLineThan playerB1*". Although it is only one touch, this ball-touch is detected 361 times based on the dis-

Table 5.3: Offside Detection Results

Time	2:53	4:26	7:04	12:21	12:44	15:28	20:40	21:11	22:07	23:04
Referee	×	●	●	×	×	●	●	●	×	×
FE-FI-FO	×	●	●	×	×	●	●	×	×	×
FE-FI-FO-25%	×	○	×	●	×	○	×	○	×	×
FE-FI-FO-50%	×	×	×	●	×	×	×	×	×	×
FE-FI-FO-75%	×	×	×	×	×	×	×	×	×	×
QR-FE-FI-FO	×	●	●	×	×	●	●	×	×	×
QR-FE-FI-FO-25%	×	○	×	●	×	○	×	○	×	×
QR-FE-FI-FO-50%	×	×	×	●	×	×	×	×	×	×
QR-FE-FI-FO-75%	×	×	×	×	×	×	×	×	×	×
FE-LFU-FO	×	●	●	×	×	●	●	×	×	×
FE-LFU-FO-25%	×	●	●	×	×	●	●	○	×	×
FE-LFU-FO-50%	×	●	●	×	×	●	●	○	×	×
FE-LFU-FO-75%	×	●	●	×	×	●	●	×	×	×
QR-FE-LFU-FO	×	●	●	×	×	●	●	×	×	×
QR-FE-LFU-FO-25%	×	●	●	×	×	●	●	○	×	×
QR-FE-LFU-FO-50%	×	●	●	×	×	●	●	○	×	×
QR-FE-LFU-FO-75%	×	●	●	×	×	●	●	○	×	×
FE-LRU-FO	×	●	●	×	×	●	●	×	×	×
FE-LRU-FO-25%	×	●	●	×	×	●	●	○	×	×
FE-LRU-FO-50%	×	●	●	×	×	●	●	○	×	×
FE-LRU-FO-75%	×	●	●	×	×	●	●	○	×	×
QR-FE-LRU-FO	×	●	●	×	×	●	●	×	×	×
QR-FE-LRU-FO-25%	×	●	●	×	×	●	●	○	×	×
QR-FE-LRU-FO-50%	×	●	●	×	×	●	●	○	×	×
QR-FE-LRU-FO-75%	×	●	●	×	×	●	●	○	×	×

Time	30:52	40:30	40:45	42:16	47:38	50:25	51:43	54:46	58:11	58:24
Referee	×	×	●	⊗	⊗	⊗	⊗	●	⊗	×
FE-FI-FO	×	×	●	×	×	×	×	×	●	×
FE-FI-FO-25%	×	×	×	×	×	×	×	○	×	●
FE-FI-FO-50%	×	×	×	×	×	×	×	○	×	●
FE-FI-FO-75%	×	×	×	×	×	×	×	×	×	×
QR-FE-FI-FO	×	×	●	×	×	×	×	×	●	×
QR-FE-FI-FO-25%	×	×	×	×	×	×	×	○	×	●
QR-FE-FI-FO-50%	×	×	×	×	×	×	×	×	×	●
QR-FE-FI-FO-75%	×	×	×	×	×	×	×	×	×	×
FE-LFU-FO	×	×	●	×	×	×	×	×	●	×
FE-LFU-FO-25%	×	×	●	×	×	×	×	×	●	×
FE-LFU-FO-50%	×	×	●	×	×	×	×	×	●	×
FE-LFU-FO-75%	×	×	●	×	×	×	×	×	●	×
QR-FE-LFU-FO	×	×	●	×	×	×	×	×	●	×
QR-FE-LFU-FO-25%	×	×	●	×	×	×	×	×	●	×
QR-FE-LFU-FO-50%	×	×	●	×	×	×	×	×	●	×
QR-FE-LFU-FO-75%	×	×	●	×	×	×	×	×	●	×
FE-LRU-FO	×	×	●	×	×	×	×	×	●	×
FE-LRU-FO-25%	×	×	●	×	×	×	×	×	●	×
FE-LRU-FO-50%	×	×	●	×	×	×	×	×	●	×
FE-LRU-FO-75%	×	×	●	×	×	×	×	×	●	×
QR-FE-LRU-FO	×	×	●	×	×	×	×	×	●	×
QR-FE-LRU-FO-25%	×	×	●	×	×	×	×	×	●	×
QR-FE-LRU-FO-50%	×	×	●	×	×	×	×	×	●	×
QR-FE-LRU-FO-75%	×	×	●	×	×	×	×	×	●	×

●:referee confirmed offside offence
×:referee confirmed no offside offence
⊗:referee unsure so no flag
●:system true-positive judgment
×:system true-negative judgment
●:system false-positive judgment
×:system false-negative judgment
○:system confirmed judgment but with wrong judging foundation

tance between Player A7's shin-guard sensor and ball sensor. For each ball-touch detection, the annotation strategy will detect if "*isNearerToDefenderGoalLineThan*" information exits and output it if it exists. Between two ball-touch triples, there are usually some triples as shown in Listing 5.4.

Listing 5.4: Ball Touch Annotation

```
1. playerA7 isInvolvedIn ball_touch
2. playerB1 a SecondLastPlayer
3. playerA7 isNearerToDefenderGoalLineThan playerB1
4. playerA7 isNearerToDefenderGoalLineThan ball8
   ...
5. ball8 a InFieldBall
6. playerA7 isInvolvedIn ball_touch
```

Triple #1 in Listing 5.4 is easily evicted by FE-FI-FO with a small l, which loses the ball-toucher information. When Triple #6 arrives, the system thinks this is a new ball toucher, thus the query is executed. Because all necessary data(Triple #2, 3, 4, 5 & 6) is present, a false-positive result is returned. The reason why FE-FI-FO, QR-FE-FI-FO, and QR-FE-FI-FO-75% return correct judgments is that their window size is large enough to hold the triples between two adjacent ball-toucher triples, thus the in-window ball-toucher triple only needs to update its generation timestamp, rather than being evicted. This example also illustrates the importance to fire query upon ball-touch events, not time.

However, for *-LFU/LRU-FO families, the mechanism is different. They can use query participation aspect to collect more important data via either query execution or query relaxation, thus Triple #1 will always be preserved to avoid executing the offside query again and again, till a different ball-toucher arrives. For ○ result, its cause is the same as the false negative judgment – the *-FI-FO family with small l cannot preserve the offside position status of Player A6 in Figure 5.5.(c), but uses the offside position status in Figure 5.5.(d), which is based on a wrong judging foundation. As a matter of fact, what happens during 21:11 to 21:13 is that, ball is passed from Player A1 to Player A6, while Player B1 tries really hard to steal the ball, but fails to change the ball's trajectory. Since Player B1 and the ball distance

is very close (smaller than 0.6m) when he reaches out his feet, the system thinks there is an offensive transition. Thus all the yellow teammates become defenders. However, as Player A6 touches the ball, the system thinks there is another offensive transition happens. Thus at this moment, which is shown in Figure 5.5.(d), Player A6 is at the offside position, is an attacker, touches the ball. So the system reports that this is an offside offence. Some strategies produce ○ because of the window size is small, as well as two offensive transitions that can expire the players' status data, so that the window cannot hold the correct offside judging foundation status happening at Figure 5.5.(c). In sum, that Player A6 commits an offside offence is true, but this judgment should be made upon the offside position in Figure 5.5.(c), not 5.5.(d)

In addition to the results following the process described above, we reviewed the data for the 54:48 case more closely. In this case, the nearest player to ball is always Player B4 from 54:44 to 54:46. However, the distance between Player B4 to ball increases from 59.9 cm to 140 cm, and the video shows that Player B4 touches the ball during that time. Therefore, we believe that Player B4's sensor failed and lost his position during that time. This potential sensor failure is not mentioned in DEBS 2013 grand challenge [88].

Table 5.4: Strategy Running Performance

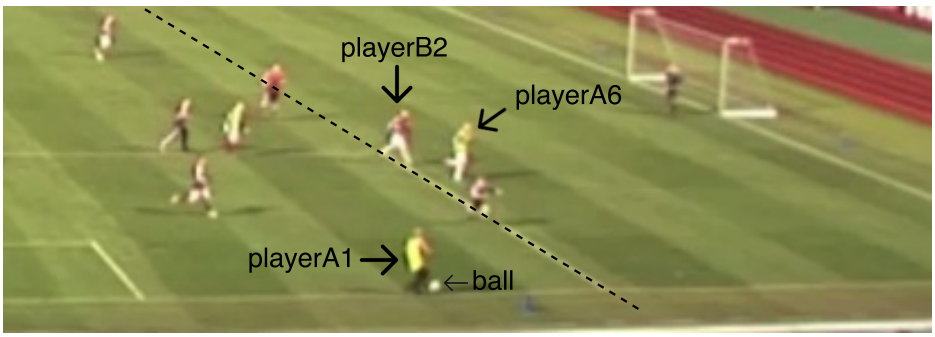
	*-FI-FO	*-LFU-FO	*-LRU-FO	QR-*
Throughput (#/s)	497.48	306.48	328.21	381.81
Eviction Rate (#/s)	130.55	120.32	114.63	123.31
Query Time (s)	0.34	0.30	0.33	0.18
Explanation Time (s)	-	0.03	0.03	0.03
QR filtering Time (s)	0.63	0.54	0.61	0.60
Total Execution Time (s)	0.97	0.87	0.97	0.81

Table 5.5: Strategy Accuracy Analysis

window size l	*-FI-FO	*-LFU-FO	*-LRU-FO	QR-*
25% l_{max}	0.55	0.90	0.90	0.80
50% l_{max}	0.55	0.90	0.90	0.80
75% l_{max}	0.65	0.90	0.90	0.83
l_{max}	0.90	0.90	0.90	0.90

(a) playerA7 is not at offside position at time 58m 24.4658s

(b) playerA7 touches the ball at time 58m 28.4318s



(c) playerA6 is at offside position at time 21m 12.3274s

(d) playerA6 touches the ball at time 21m 13.6443s

Figure 5.5: False-positive & Wrong Foundation Judging Analysis

Table 5.6 summarizes the occurrence ratios of flag errors and non-flag errors when different strategies are in use. In the soccer offside terminology, "flag error" refers to a wrong offside decision (linesman raises the flag) when there is in fact no offside offence. Conversely, "non-flag" error means that no offside offence is indicated when in fact one occurs. The statistical counterparts of the terms "flag error" and "non-flag error" are our system's false-positive and false-negative judgments, respectively. It is worth mentioning that, out of the total 20 cases, there are five judgments given by the referee as "unsure" and thus cannot be used as ground-truth.

Table 5.6: Strategy Error Analysis

	*-FI-FO	*-LFU-FO	*-LRU-FO	QR-*	Sys.
flag err.	0.07	0.00	0.00	0.02	0.02
non-flag err.	0.38	0.09	0.08	0.18	0.19

Finally, we consider the different scenarios where each type of semantic importance can be applied. Generation and expiration timestamps can be applied in a system where recent data is more important, the query is simple, and does not require a history look-up. The throughput speed in FIFO benefits the system's real-time response. However, if the query is complex and requires history look-ups (as in soccer offside query answering), reasoning participation frequency and recency is better, because they can preserve historical data in the window until other complementary data arrives. The trade-off is a relatively lower throughput as a result of the overhead of reasoning and query explanation. Query relevance filtering has the fastest execution time and also has the most potential to scale.

5.2 Data Exfiltration Detection

According to Wikipedia, "an insider threat is a malicious threat to an organization that comes from people within the organization"[22]. Compared to the outside threat, the organization tends to be more vulnerable when facing insider threat. This is because the insiders have access to the internal infrastructure, data, or network. It is very easy for them to steal the organizational intellectual properties.

[22] `https://en.wikipedia.org/wiki/Insider_threat`

What makes it even worse is that, the insider threat is much more harder to detect than the outside threat. A malicious insider is defined as "a current or former employee, contractor, or business partner who has or had authorized access to an organization's network, system or data or has intentionally exceeded or intentionally used that access in a manner that negatively affected the confidentiality, integrity, or availability of the organization's information or information systems" [90] [91]. This use case aims to detect data exfiltration[23], a sub-category of insider threat.

5.2.1 Background

This use case will focus on one type of insider threat attacks, the data exfiltration. According to [92], the data exfiltration means that a malicious user employs a removable media or the Internet to remove confidential data from an organization. An example of the data exfiltration is that the employee who finishes work usually around 5:30 PM comes back to the office very late at the night, and uses a removable disk to copy some files or upload the files into wikileaks.org.

There has already been related work that builds methods for data exfiltration. [93] proposes a framework called SSID that is based on statistical and signal processing techniques to detect sensitive information distribution and stolen over the Internet. [94] points out that the key for detecting data exfiltration is to identify the distinction between legal and illegal information communication. It also provides a detailed list and discussion of most known data exfiltration methods, which is very valuable to model data exfiltration. [95] takes a different way with several novel methods of enabling data exfiltration using web browsers and the JavaScript engine, with the purpose to raise the awareness of this kind of malicious intent.

5.2.2 Approach

Different from the above related work, this use case will be implemented with core notions of stream reasoning, including window and continuous processing, together with the semantic importance model and the sequential stream reasoning architecture. A stream reasoning system will be realized to identify and preserve

[23]For more details, please refer to the github repository: `https://github.com/raymondino/InsiderThreat-StreamReasoningUseCase`.

all previous suspicious actions as evidence to determine data exfiltration. Since the malicious actions can happen across arbitrary interval of time, a logical window will be leveraged. This poses a challenge in such a way that the bigger the window size is, the more data will be in the window to negatively affect the system responsiveness. In order to enable reasoning, the plan is to leverage and extend the existing insider threat ontology developed by CMU CERT[24]. Semantic importance aspects in provenance and trustworthiness will be selected to support two window management strategies, FIFO and QR-T (query relevance - trustworthiness). In order to enable trustworthiness, a simple trust model is established, which will be elaborated in the following section.

5.2.3 Data Stream

Real insider threat datasets are usually very organizational-sensitive and confidential, which makes it not easy to be published. However, [96] provides a synthesized streaming data set that contains four data exfiltration scenarios and provides timestamped employee actions of types including HTTP, device, login, log-off, file and email, which is shown in Figure 5.6.

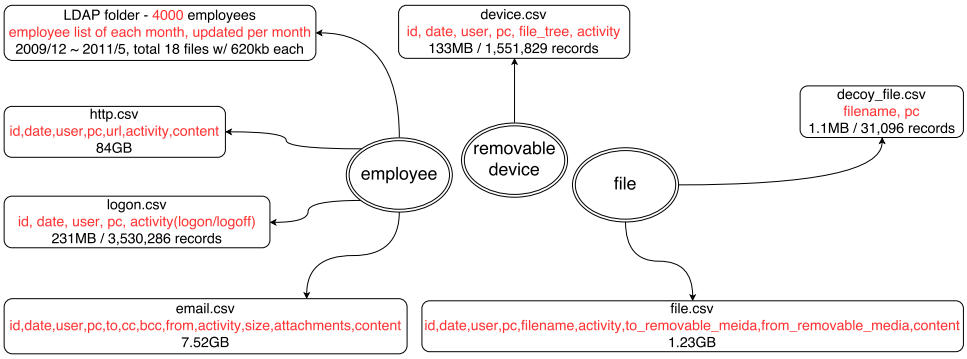


Figure 5.6: Synthesized Data for Insider Threat Detection

This figure indicates that the dataset is big in size and rich in content. It contains 3 categories, the employee data, the removable device data, and the file data. An employee can be seen as a sensor that constantly streams action data over a time line, such as HTTP actions including browsing, downloading, and uploading; log-on,

[24] http://www.cert.org/

log-off actions on the organization computers; as well as email actions. The removable device data includes the actions of using a certain removable device (such as a flash drive) to transfer data. The file data includes the employee actions performed on the file, such as the contents of the file copied. There is also some background information provided in the dataset, such as LDAP data in the employee category and the decoy-file data in the file category. The LDAP data contains the full list of employees, with their roles, office location and working relationships. This list updates weekly. The decoy-file data contains the files that are deliberately placed in the employees' working computers, and works like a bait. Each dataset has its own data-schema, and is encoded in the comma separated value format.

This dataset provides 4000 users' actions data, in which 5 users are malicious insiders. From the given brief scenarios descriptions in the data set, 4 out of 5 malicious insiders are data exfiltration insiders. For more details about the dataset, please refer to the [96] and the website: `https://www.cert.org/insider-threat/tools/`.

5.2.4 Ontology

CMU CERT has developed an insider threat indicator ontology[25] [97]. This ontology primarily contains the concepts and relationships that are extracted from their real insider threat use case events database. This ontology is leveraged to describe the insider threat scenarios instead of natural language for the purpose of providing structured data for machines to process. The ontology doesn't contain any specific concepts to model data exfiltration. It is either inconsistent: the concept **TradeSecretInformation** is classified to be both **Asset** and **Information**, which are disjoint classes. In order to extend the CERT ontology, the inconsistency problem needs to be addressed[26]. Currently, it is solved by removing the disjointness. The ontology of this use case is modeled as the figures below.

Figure 5.7 extends the concept **Actor** by introducing 5 subclasses. **Unemployee** and **SuspiciousEmployee** are both subclasses of **Person**. **Unemployee** describes previous employees who left the job (either resignation, or laid-off). **Sus-**

[25] `http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=454613`

[26] I have contacted CMU CERT and provided this feedback to them.

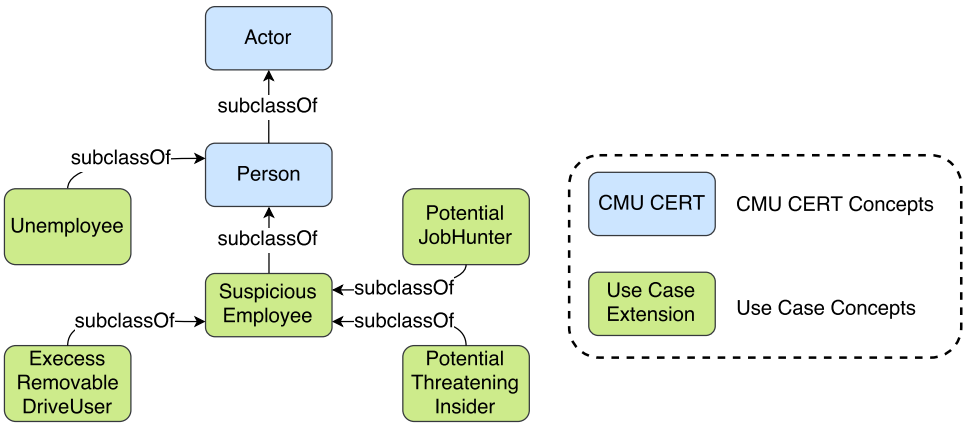


Figure 5.7: Data Exfiltration Background Ontology: Actor

piciousEmployee has three subclasses: **ExcessiveRemovableDriveUser** class describes a user who excessively uses the removable disk, compared to his/her normal daily removable disk usage. **PotentialJobHunter** class describes an employee who browsed some job hunting website. It's not necessary for an employee to be a jobhunter if he browses job-related website, but the system should be able to record it, then fire an alarm if further suspicious events are detected. This class is defined as (*browseWebsite some* **JobHuntingWebsite**). **PotentialThreateningInsider** class describes a potential threatening insider. The reason not to name it as something like "a threatening insider" is because humans are needed to make the final decision. Any users classified into this class will become the forensic target, and is what this use case is looking for. This class is defined as (*isInvolvedIn some* **DataExfiltrationEvent**).

Figure 5.8 shows the extended models for **Asset** class. The focus of using this class is to model the email addresses, files and computers. For example, it is important to classify the emails sent to an internal address or an external address. It is also crucial to know what kind of file is attached to the email, or copied to/from the flash drive. Being able to record which employee uses which computer is also necessary since malicious insiders can possibly get on other employees' computers to steal data.

In the CMU ontology, **TemporalThing** is the superclass of **Action** and **Event**, as both of them are time sensitive. **ActionModifier** describes the kinds

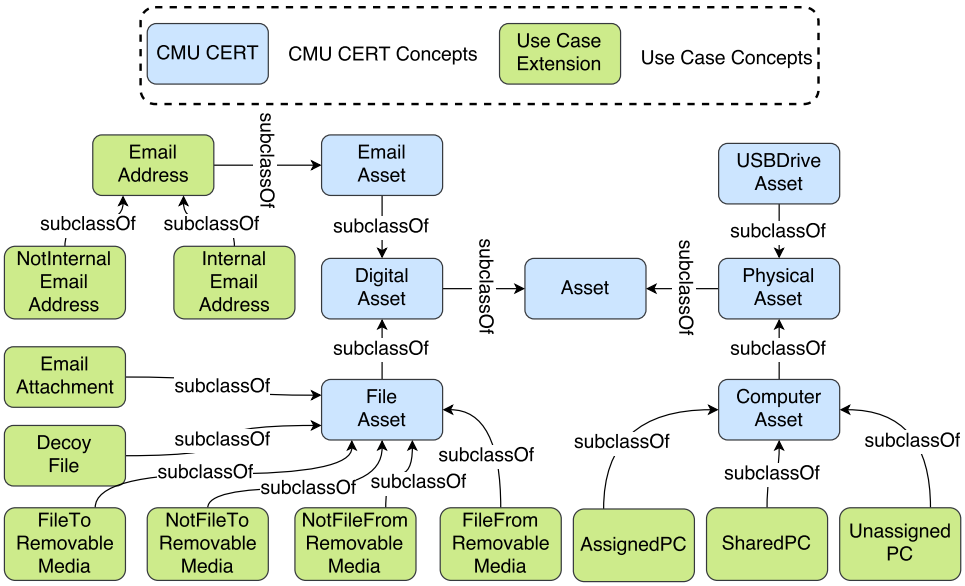


Figure 5.8: Data Exfiltration Background Ontology: Asset

of the action. For example, **AfterHourAction** describes an action that happens after hour. The **SuspiciousAction** is defined as (**JobHuntingAction** *or* **SuspiciousEmailSendAction** *or* **SuspiciousFileAction** *or* **SuspiciousLoginAction** *or* **SuspiciousRemovableDeviceUsageAction** *or* **SuspiciousWWWUploadAction**). This class classifies the defined suspicious actions. The reason to include the **JobHuntingAction** is because a potential job hunter should be taken care of. All of the subclasses of **SuspiciousAction** are defined, which provides the known data exfiltration patterns.

SuspiciousEmailSendAction is defined as (**EmailSendAction** *and* ((*bcc some* **NotInternalEmailAddress**) *or* (*cc some* **NotInternalEmailAddress**) *or* (*from some* **NotInternalEmailAddress**) *or* (*to some* **NotInternalEmailAddress**))) *or* (*hasEmailAttachment some* **DecoyFile**). It takes care of those emails sent to external email addresses.

SuspiciousFileAction is defined as (**AfterHourAction** *and* (**FileCopyAction** *or* **FileDeleteAction** *or* **FileOpenAction**)) *or* (**FileCopyAction** *and* (*hasFile some* **DecoyFile**)) *or* (((*isPerformedOnPC some* **UnassignedPC**) *and* **FileCopyAction** *or* **FileDeleteAction** *or* **FileOpenAction**)) *or* (*startsNoEarlierThanEndingOf some* **SuspiciousRemovableDeviceUsageAction**). Basically,

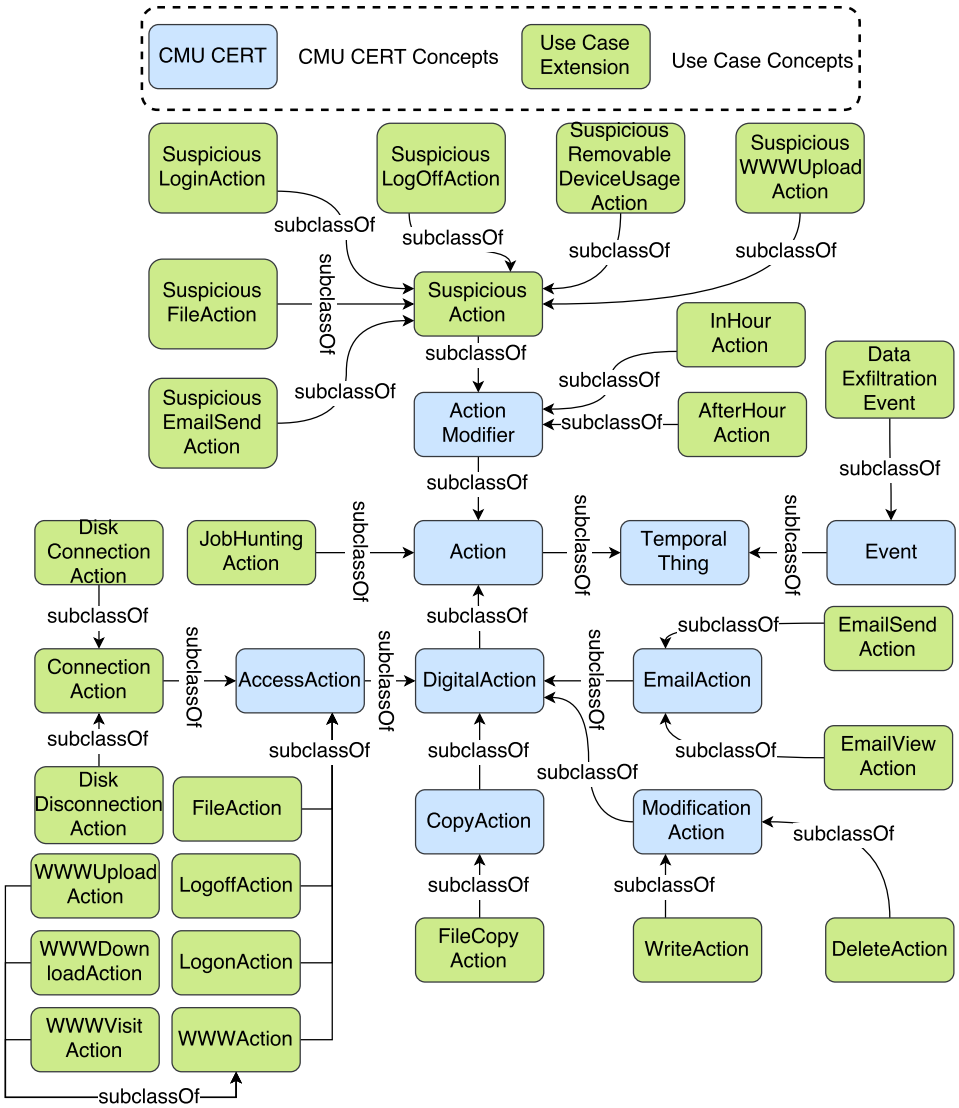


Figure 5.9: Data Exfiltration Background Ontology: Temporal Thing

a suspicious file action is an action that either is performed after hour by deleting, copying or opening a decoy file, or using a removable drive to delete, copy, or open files on an unassigned PC.

SuspiciousLoginAction is defined as (**AfterHourAction** *and* **LogonAction**) *or* (**LogonAction** *and* (*isPerformedOnPC some* **UnassignedPC**)). This means that any action that is performed after hours or on an unassigned PC will be classified as a suspicious login action. Similarly, **SuspiciousLogoffAction** is

defined as (**AfterHourAction** *and* **LogoffAction**) *or* (**LogoffAction** *and* (*isPerformedOnPC some* **UnassignedPC**)).

SuspiciousRemovableDeviceUsageAction is defined as (**AfterHourAction** *and* (**DiskConnectionAction** *or* **DiskDisconnectionAction**)) *or* ((**DiskConnectionAction** *or* **DiskDisconnectionAction**) *and* (*isPerformedOnUnassignedPC some* **UnassignedPC**)) *or* (*hasActor some* **ExcessiveRemovableDriveUser**). This class will classify the actions that are performed either after hours with a disk connection action, or from an excessive removable driver user.

SuspiciousWWWUploadAction is defined as (**AfterHourAction** *and* **WWWUploadAction**) *or* (**WWWUploadAction** *and* (*hasWebsite some* (**CloudStorageWebsite** *or* **HacktivistWebsite**))) *or* ((**WWWDownloadAction** *or* **WWWUploadAction** *or* **WWWVisitAction**) *and* (*isPerformedOnUnassignedPC some* **Unassigned-PC**)). It describes a kind of malicious action that a user uploads files to cloud storage or hactivist website after-hour or from an unsigned PC.

DataExfiltrationEvent class is the subclass of **Event**. It describes the data exfiltration event, and is defined as ((((*hasAction min 10* **SuspiciousEmailSendAction**) *and* (*hasAction max 10000* **SuspiciousEmailSendAction**)) *or* (*hasAction max 10000* **SuspiciousWWWUploadAction**)) *and* (*hasAction min 5* **SuspiciousWWWUploadAction**)) *or* ((*hasAction min 3* **SuspiciousFileAction**) *and* (*hasAction max 10000* **SuspiciousFileAction**)). In its definition, cardinality is used to quantify a threshold to determine a data exfiltration event.

Last but not the least, the ontology has a **Website** class that models different websites. It has four subclasses, **CloudStorageWebsite** that describes sites like dropbox.com; **HacktivistWebsite** that describes sites like wikileaks.org; **JobHuntingWebsite** that describes sites like indeed.com; and **NeutralWebsite** that describes sites other than the above three.

5.2.5 Data Annotation

Data is pre-processed before annotated by the data exfiltration background ontology. All employees' routine hours are calculated based on their first two week

working hours. This routine hours will be used as a basis of judging the after hour actions. Also, employees' routine log-on/log-off time, and daily removable disk usage situations are extracted for reference. In the LDAP folder of the synthesized dataset, a employee list is provided for every month. This list is updated monthly, thus employees who are not in current month LDAP list will be annotated as an instance of **Unemployee**.

Logon.csv file follows the schema of (id, date, user, pc, activity), where "id" is the unique action id, "date" is a xsd:dateTime timestamp indicating the generation timestamp of the action, "user" is the employee id, "pc" is where the action is performed, "activity" is a string of either logon or logoff. Thus, the data in Logon.csv file can be annotated in RDF stream as follows:

(1) :<user>-event :hasAction :<activity>_<id>, <date>.

(2) :<activity>_<id> rdf:type :Action, <date>.

(3) if <date> is not in normal working hours of <user>,

:<activity>_<id> rdf:type :AfterHourAction, <date>;

else, :<activity>_<id> rdf:type :InHourAction, <date>.

(4) :<activity>_<id> :hasActor :<user>, <date>.

(5) :<activity>_<id> :isPerformedOnPC :<pc>, <date>.

Device.csv file has the schema of (id, date, user, pc, file_tree, activity), where "id, date, user, pc" are the same as Logon.csv. "file_tree" is the directory content of the removable drive. "activity" describes the action taken by the user. The data is annotated as follows:

(1) :<user>-event :hasAction :device_<activity>_<id>, <date>.

(2) if current employee's daily removable device usage count is bigger than his/her routine usage, then annotate

:<user> rdf:type :ExcessiveRemovableDriveUser, <date>.

(3) :device_<activity>_<id> rdf:type :Action, <date>.

(4) if <date> is not in the normal working hours of <user>,

:device_<activity>_<id> rdf:type :AfterHourAction, <date>;

else, :device_<activity>_<id> rdf:type :InHourAction, <date>.

(5) :device_<activity>_<id> :hasActor :<user>, <date>.

(6) :device_<activity>_<id> :isPerformedOnPC :<pc>, <date>.
(7) if the activity is “connect”,
:device_<activity>_<id> :isPerformedWithRemovableDisk
:device_<activity>_<id>_disk, <date>.

Email.csv file encodes data with the schema of (id, date, user, pc, to, cc, bcc, from, activity, size, attachments, content), where “id, date, user, pc” are the same as above. “to, cc, bcc, from” respectively indicates the sender’s and receivers’ email addresses, “activity” is either send or view, “size” is the attachment’s size in bytes, “attachments” refer to the attached files in the email, “content” indicates the contents in the email. The data is annotated as follows:
(1) <user>-event :hasAction :email_<activity>_<id>, <date>.
(2) :email_<activity>_<id> rdf:type :EmailAction, <date>.
(3) if <date> is not in the normal working hours,
:email_<activity>_<id> rdf:type :AfterHourAction, <date>;
else, :email_<activity>_<id> rdf:type :InHourAction, <date>.
(4) :email_<activity>_<id> :hasActor <user>, <date>.
(5) :email_<activity>_<id> :isPerformedOnPC <pc>, <date>.
(6) :email_<activity>_<id> :to :<to>, <date>,
(7) :email_<activity>_<id> :cc :<cc>, <date>.
(8) :email_<activity>_<id> :bcc :<bcc>, <date<.
(9) :email_<activity>_<id> :from :<from>, <date>.
(10) if the email address (<email-address> includes <to>, <cc>, <bcc>, and <from>) is the organization address,
:<email-address> rdf:type :InternalEmailAddress;
else, :<email-address> rdf:type :ExternalEmailAddress.
(11) :email_<activity>_<id> :hasEmailAttachment :<attachments>, <date>.
(12) :<attachments> :hasSize :<size>, <date>.
(13) :email_<activity>_<id> :hasContent :<content>, <date>.

File.csv file contains the data schema of (id, date, user, pc, filename, activity, to_removable_media, from_removable_media, content), where “id, date, user, pc” are the same as the above. “filename” describes the name of the file, “activity” denotes

the action performed on the file, “to_removable_media” denotes if the file is performed to the removable media, “from_removable_media” is the opposite, “content” describes the file content. The data is annotated as follows:
(1) :<user>-event :hasAction :file_<activity>_<id>, <date>.
(2) :file_<activity>_<id> rdf:type :FileAction, <date>.
(3) if <date> is not within the normal working hours,
:file_<activity>_<id> rdf:type :AfterHourAction, <date>;
else, :file_<activity>_<id> rdf:type :InHourAction, <date>.
(4) :file_<activity>_<id> :hasActor :<user>, <date>.
(5) :file_<activity>_<id> :isPerformedOnPC :<pc>, <date>.
(6) :file_<activity>_<id> :hasFile :<filename>, <date>.
(7) if “to_removable_media” is true,
:<filename> rdf:type :FileToRemovableMedia, <date>;
else, <filename> rdf:type NotFileToRemovableMedia, <date>.
(8) if “from_removable_media” is true,
:<filename> rdf:type :FileFromRemovableMedia, <date>;
else, <filename> rdf:type :NotFileFromRemovableMedia, <date>.

Decoy.csv has the data schema of “filename, pc”, where “filename” is the name of the decoy file, “pc” is where the file is located. Decoy.csv file is used as the background information, thus no timestamps are associated. Data is annotated as:
(1) :<filename> rdf:type :DecoyFile.
(2) :<filename> :isIn <pc>.

LDAP folder contains 18 files, each of which is a list of employees during that month. They record from 2009/12 to 2011/05. The employee list is updated monthly. These files are used as the background information. Each file is with a data schema of (employee_name, user_id, email, role, projects, business_unit, functional_unit, department, team, supervisor). Thus the data is annotated as:
(1) :<user_id> :hasName :<employee_name>.
(2) :<user_id> :hasEmailAddress :<email>.
(3) :<user_id> :hasRole :<role>.
(4) :<user_id> :hasProjects :<projects>.

(5) :<user_id> :hasBusinessUnit :<business_unit>.

(6) :<user_id> :hasFunctionalUnit :<functional_unit>.

(7) :<user_id> :hasDepartment :<department>.

(8) :<user_id> :hasTeam :<team>.

(9) :<user_id> :hasSupervisor :<supervisor>.

Http.csv file has the schema of "id, date, user, pc, url, activity, content", where "id, date, user, pc" are the same as the above. "url" is the link that the user visits, "activity" is either view, upload, and download, "content" is the content of the web-page browsed. The data is annotated as follows:

(1) :<user>-event :hasAction :http_<activity>_<id>, <date>.

(2) :http_<activity>_<id> rdf:type :Action, <date>.

(3) if <date> is within the routine hours of the user,

:http_<activity>_<id> rdf:type :InHourAction, <date>;

else, :http_<activity>_<id> rdf:type :AfterHourAction, <date>.

(4) :http_<activity>_<id> :hasActor :<user>, <date>.

(5) :http_<activity>_<id> :isPerformedOnPC :<pc>, <date>.

(6) :http_<activity>_<id> :hasURL :<url>, <date>.

(7) URLs are annotated with **Website** class:

:<url> :whoseDomainNameIsA :cloudstoragewebsite

:<url> :whoseDomainNameIsA :hacktivistwebsite

:<url> :whoseDomainNameIsA :jobhuntingwebsite

:<url> :whoseDomainNameIsA :neutralwebsite.

5.2.6 System Implementation

The data exfiltration detection system is implemented on the basis of the sequential stream reasoning architecture. A lower-bounded landmark logical window is deployed in the system. The advantage of the landmark window is reflected incisively and vividly in this use case. For data exfiltration detection, the malicious insiders can do harm to the organization in any arbitrary time period, which makes determining the size for the conventional logical sliding window very challenging: a small size probably will not provide good precision, but a big size will result in more

data that hinders the system performance. A landmark window, on the other hand, can retain the useful data when evicting other data, and can work with various window management strategies.

The query is to look for suspicious actions, which is shown in Listing 5.5. This query involves the background data such as LDAP, the streaming data annotated by the background ontology as well as the background ontology itself. This query requires DL reasoning.

Listing 5.5: Suspicious Action Query

```
PREFIX de:<http://tw.rpi.edu/ontology/DataExfiltration/>
SELECT DISTINCT ?action
WHERE { ?action a de:SuspiciousAction.}
```

Employees who performed the detected suspicious actions will be recorded, together with the actions he/she performs. All of the records will be printed to the screen, written to the event log, and make it easier for humans to make a final judgment. However, detecting suspicious actions is not the ultimate goal of this use case. In the background ontology, the data exfiltration event is defined with cardinalities. If an employee performs suspicious actions several times, then this employee is a potential malicious insider, and should be paid special attention to. Thus, another query in Listing 5.6 examines the employees' performed suspicious actions and checks if he/she is a potential malicious insider. This query is parameterized, where <user> will be filled during the run time. If this query returns true, then the user will be reported. The query also requires DL reasoning.

Listing 5.6: Potential Malicious Insider Query

```
PREFIX de:<http://tw.rpi.edu/ontology/DataExfiltration/>
ASK WHERE { de:<user> a de:PotentialThreateningInsider.}
```

This use case has chosen provenance and trustworthiness from semantic importance to form two window management strategies, FIFO and QR-T. FIFO is first in first out, its priority vector is $[\tau_a]$; while QR-T is query relevance trustworthiness, its priority vector is $[qrf, t_s^{tm}]$ The reason to choose query relevance is because of the large amount of the streaming data. There is really a need to iden-

tify, and filter out the unnecessary data to improve system response. Thus, a query relevance ontology is provided. In the previous section, it is mentioned that this query relevance ontology is query-informed. However, for this use case, no dedicated query relevance ontology is provided, as the background domain ontology has already provided enough information to filter out the data. In this use case, the background domain ontology can be used as a query relevance ontology. For example, the query in Listing 5.5 really looks for the suspicious actions. As the action data keeps streaming into the system, each action data item will be examined by the query engine, and if that action data item is not classified as a suspicious action, it can be immediately dropped without having to enter into the window. Only those suspicious action instances will be entered into the window. One important thing to note is that, the reason this use case implementation can realize query relevance filtering with a background ontology is very use case specific, and should not be seen as a general solution. This use case can do this because the background ontology has already provide enough information, as well as the system working mechanism that each action data item will be examined before entering into the window.

A simple trust mode has been developed because the existing models are very heterogeneous, and not usually applicable for the scenario of this use case. The trust model works in this way: initially, each employee has a trust score of t_{s_i}. As the system runs, if an employee A performed a suspicious action, his/her trust score will reduce 1. There is a threshold Y, such that $0 < Y < t_{s_i}$. If A's $t_{s_i}^A$ is smaller than Y, then employee A becomes untrusted, thus all of his/her actions will be recorded and sent to the administrators to analyze. The QR-T strategy firstly filters out the unnecessary action data, then ranks the action data according to their actors' trust scores.

5.2.7 Evaluation

The window has been set to detect the period of 1 day, 1 week and 1 month for any suspicious actions and malicious users. Two strategies are deployed, and evaluated via recording the precision, recall, and the running time.

Table 5.7: Strategy Performance

	precision	recall
FIFO (1D)	0.18	0.93
QR-T (1D)	0.18	0.97
FIFO (1W)	0.26	0.87
QR-T (1W)	0.26	0.95
FIFO (1M)	0.26	0.87
QR-T (1M)	0.26	0.95

Table 5.7 shows the system precision and recall. The precision of FIFO and QR-T are equal for the same detection time period, and slightly increases as the time-period becomes bigger. QR-T's recall is always better than that of FIFO for the same detection time period. The precision, undoubtedly, is very low. There are reasons for that. Insider threat is by no means a simple problem that can be solved with only stream reasoning technologies, it should at least include natural language processing, machine learning and possibily some other technologies. The known pattern on data exfiltration is not very discriminative. That a person who uses a removable disk to copy some file and then upload it into dropbox could be either malicious or innocent. Thus the background ontology cannot do a good job in classifying data exfiltration actions.

Table 5.8: Strategy Running Time

	FIFO	QR-T
T_{total}	49.94 (ms)	29.63 (ms)

However, this use case implementation shows a positive example in Table 5.8, where clearly QR-T runs faster than FIFO. This is because FIFO strategy has more data in the window than QR-T, which slows down the running performance. The results have shown that semantic importance enabled window management strategy can improve the system response time.

5.3 Discussion

This section covers the two use case implementations using both semantic importance and sequential stream reasoning architecture. From the result tables, a

significant difference can be seen in the system outputs from different window management strategies enabled by different semantic importance aspects. The two use cases, albeit are from two different domains, share similar characters. The streaming data is big for both cases. The offside use case data updates frequently, while the data exfiltration data is massive due to frequent actions from many employees. Also, it requires complex DL reasoning in both cases. Streaming data alone, although provides the basic semantics from the ontology annotation, is not able to provide the desired results. From the results, we can see that semantic importance is able to improve the system precision, memory consumption, and response time.

However, one should also see that the implementations of both use cases are dependent upon very specific assumptions that are only applicable in specific scenarios. This limits the generalization of semantic importance. The topics of generalizing and benchmarking semantic importance will be covered in the next chapter.

CHAPTER 6
SEMANTIC IMPORTANCE GENERALIZATION AND BENCHMARK

The sequential stream reasoning architecture (SSRA) is an innovative architecture that is designed and implemented specifically for stream reasoning use cases aiming to mine the knowledge in the boundless and torrent streaming data. Based on SSRA, two use cases are implemented, which to some extent demonstrates the usefulness of the semantic importance in the stream reasoning context. However, SSRA implementations are evaluated out of commercial triple-store products, where only three strategies and a few of semantic importance aspects are included. In the use case implementations, specific case-by-case assumptions are applied. Even though the experimental results have shown some evidence to support the advancement of the semantic importance, they are still not sufficient to describe how semantic importance can bring values to the stream reasoning, as well as be generally applied to a wide range of stream reasoning applications.

This chapter provides a comprehensive view of the usefulness and reusability for the concept of semantic importance. The core contribution in this chapter is a generalization and benchmark framework called **SIGenBench**. It generalizes semantic importance by connecting it to the state-of-the-art stream reasoning techniques. It also features a flexible benchmark system that can be configured to adapt with different stream reasoning scenarios. This benchmark reports four system key performance metrics including memory consumption, response time, precision, and throughput, with which the impacts and benefits of semantic importance are discussed in details.

6.1 Introduction

Figure 6.1 shows the architecture of SIGenBench. SIGenBench is implemented based on SSRA, and provides two main functionalities: generalization (blue) and benchmark (green). Generalization refers to reusability, i.e. how semantic impor-

tance can be reused in different use cases. Benchmark refers to usefulness, i.e. how semantic importance can be useful in various stream reasoning settings.

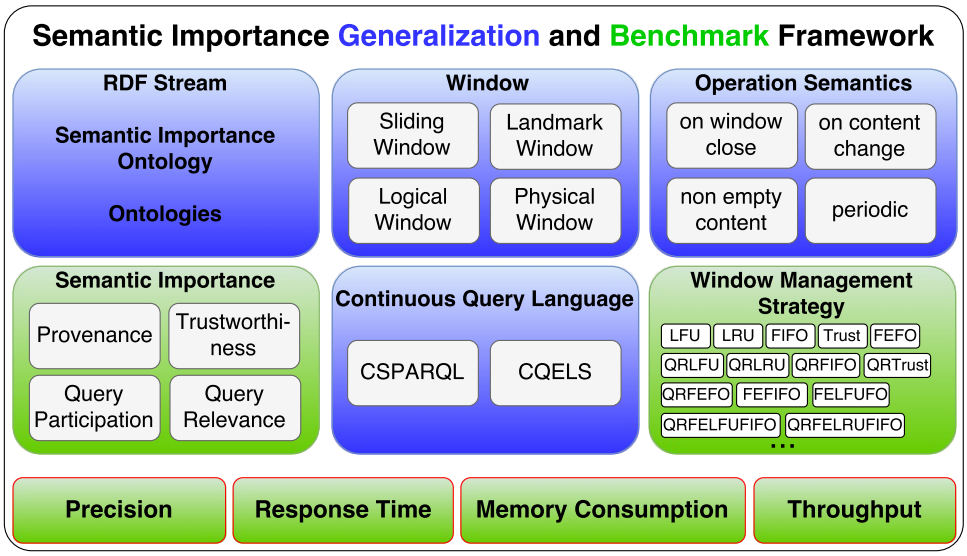


Figure 6.1: SIGenBench Architecture

In order to enable generalization, semantic importance ontology (SIO) is created. SIO is a descriptive ontology that extends the Prov-O ontology [54]. It encodes all the current aspects, and is grounded with real-world example instances, which makes the idea of semantic importance more easier to understand, reuse, and extend. SIGenBench also supports popular stream reasoning techniques, such as RDF stream [27], windows [48], report operational semantics [9], as well as continuous query languages [98] [33]. This enables SIGenBench to be adaptive and flexible when working with various stream reasoning systems.

In order to enable benchmark, SIGenBench records four performance metrics. Precision describes how precise the system can be under some window management strategy, the higher the better. Response time indicates how fast the system can respond under certain window management strategy, the faster the better. Memory consumption is measured in terms of the number of streaming data items in the window. It indicates how scale the system is under one window management strategy, the smaller the better. Throughput indicates how quickly the system can process the streaming data, which is measured by data items per second, the quicker the better. SIGenBench also features 26 built-in window management strategies that

are derived from the semantic importance model. This benchmark system supports both user-specified data or default data. The default data includes a RDF stream generator and a background ontology. It can be configured to simulate different types of streaming data thus can be used to test semantic importance in different settings.

6.2 Generalization

For the concept of semantic importance to be reusable, it must (1) provide a set of infrastructural tools so that the concept can be implemented, extended, and invoked in different applications, and (2) connect with the state-of-the-art stream reasoning work, such as RDF stream, continuous queries, window semantics and report operational semantics, etc.

6.2.1 Semantic Importance Ontology

Ontologies are portable, expandable, general and expressive. All the current semantic importance concepts have been implemented in an OWL-encoded ontology via extending PROV-O [54]. Figure 6.2 shows the classes and their relationships in the semantic importance ontology. The blue font indicates that classes are inherited from PROV-O ontology. Both **QueryRelevance** and **Trustworthiness** are **prov:Entity**, but **QueryParticipation** is a **prov:Activity**. For more details, please refer to Appendix A.1.

6.2.2 Connection to Stream Reasoning

SIGenBench is carefully designed in order to embrace both the state-of-the-art and popular stream reasoning techniques. For instance, SIGenBench is compatible with different continuous query languages and report operational semantics. As [11] mentions, the correctness of a stream reasoning engine is critically dependent upon the operational semantics. Since different stream reasoning systems can behave differently [9], the benchmark systems aiming to measure the stream reasoning performance should be able to consider the differences in various built-in operational semantics.

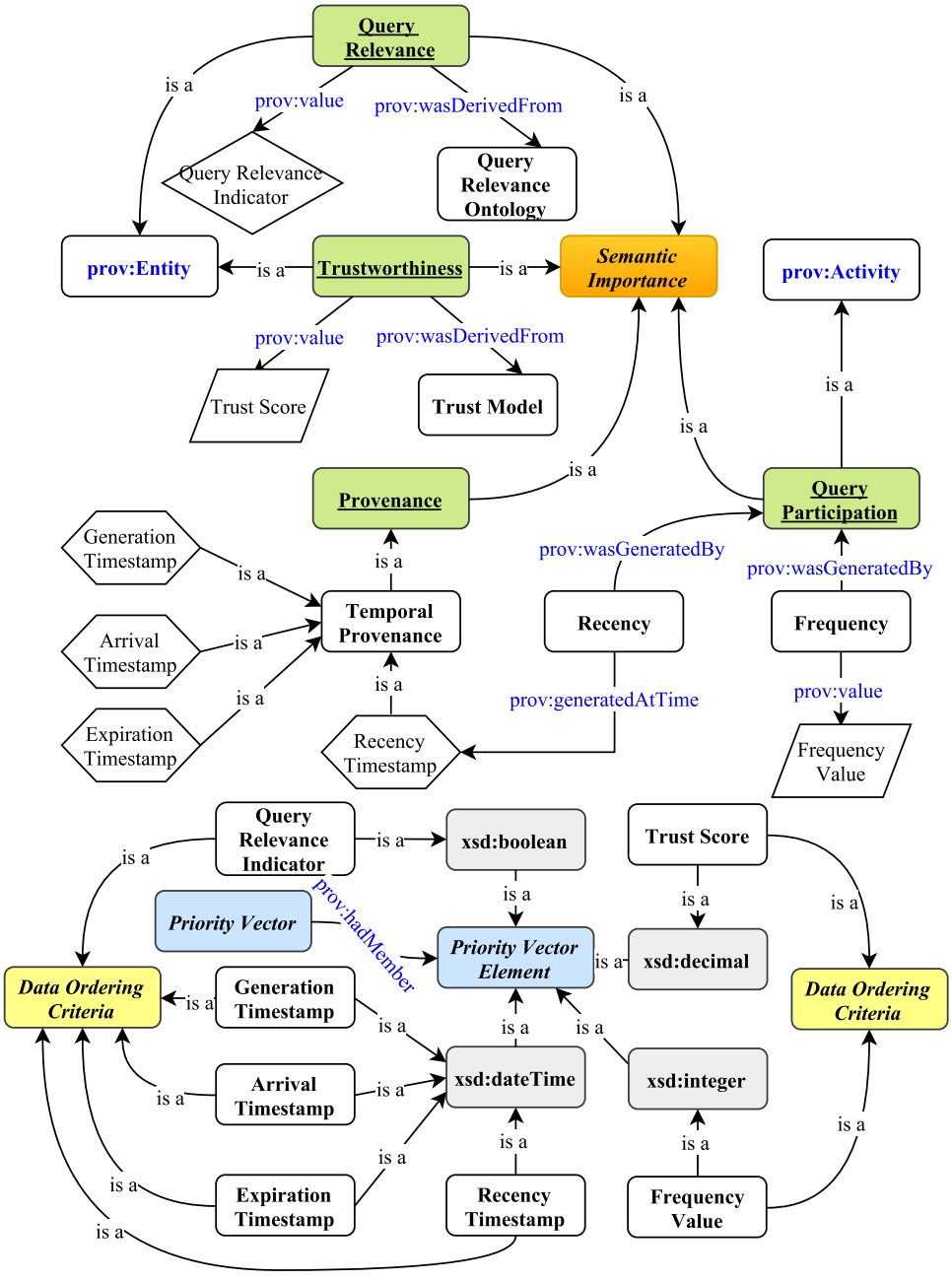


Figure 6.2: Semantic Importance Ontology

SIGenBench decouples the operational semantics from stream reasoning engines. For example, CSPARQL engine reports when the window is full, while a CQELS engine reports when the content in the window has changed (e.g. new data arrives). SIGenBench implements four different report policies, such that the users can specify which report policy and query language to use. This loosens the strict one-to-one-pair of the report policy and query engine in most stream reasoning systems. That is to say, a CSPARQL query can work with "on content change" report policy, which increases the flexibility for prototyping and testing.

SIGenBench can naturally consume RDF streams. RDF stream is also extended to carry more information such as explicit expiration timestamps and trust scores. More details will be covered in the Implementation Section.

SIGenBench works with two state of the art continuous query languages, namely CSPARQL and CQELS[27]. The reason to choose them is for their wide adoption in the stream reasoning applications [99] [100] [101] [102].

SIGenBench is compatible with mainstream window operators including sliding and landmark window. Chapter 3 has already covered the details of semantic extension for landmark windows. It also mentions that the window definition for both CSPARQL and CQELS are only for sliding window. SIGenBench aligns the these definitions for the landmark window. An exemplar logical window definition in CSPARQL is [RANGE 2s STEP 1s]. It defines the window size of 2 seconds, and step of 1 second. SIGenBench uses the same expression to initialize a logical lower-bounded landmark window as follows: the initial window size is 2 seconds worth of data items; its upper-bound moves per 1 second to consume the next 1 second worth of data items.

6.3 Benchmark

The primary goal of the benchmark system in SIGenBench is to provide detailed empirical evidence for the deployment of various window management strategies supported by semantic importance. SIGenBench is able to simulate a various of

[27]Both CSPARQL and CQELS provide a stream reasoning engine and a continuous query language.

stream reasoning scenarios, thanks to the default streaming data generator, which provides flexible test beds for window management strategies. The Evaluation Section will cover all the details. It is worth mentioning that SIGenBench does not focus on benchmarking stream reasoning applications as a whole. For such benchmark systems please refer to [11] [78] [79] and [80].

6.4 Implementation

SIGenBench is implemented based on SSRA, backed by Stardog triple-store. It is written in Java and accepts command line arguments to configure one experiment. It outputs the results in real time, provides summary logs and performance evaluation. This section highlights some core components of SIGenBench. For detailed description, please refer to Appendix A.2.

6.4.1 Window

SIGenBench has 4 different built-in windows, including sliding logical window, sliding physical window, landmark logical window, and landmark physical window. A logical window has a size and a step that are both in temporal units (such as second, minute etc). A physical window has a size and a step in terms of the number of data items. Sliding windows are commonly used windows in most stream reasoning systems. Landmark windows extend sliding windows by fixing the lower bound while allowing the upper bound proceeds. Four aforementioned window report operational semantics are supported. Please refer to Chapter 3 for all the details. Table 6.1 lists the parameters to configure a window in SIGenBench

6.4.2 Window Management Strategy

Semantic importance is the core contribution of this dissertation. It also enables various window management strategies. Currently, SIGenBench features 26 built-in strategies, as shown in Table 6.2. This table also includes the priority vectors, which can indicate the semantic importance aspects used in one strategy. For example, FIFO manages data like a queue, and data is ranked by the arrival timestamps (τ_a). A simple TRUST strategy is composed with a single trust score

Table 6.1: Window Configuration

parameter	option	annotation
wt	sliding landmark	sliding or landmark window
ut	logical physical	logical or physical window
qt	CSPARQL CQELS	CSPARQL or CQELS query language
opr	onWindowClose onContentChange nonEmptyContent periodic	report policy
stgy	strategies in Table 6.2	window management strategy

t_s^{tm}, which neglects all the temporal information associated with the data. There are certainly more compound strategies such as QR-Trust-FE-LRU-FIFO. A more compound strategy does not necessarily imply its better performance. What the strategies' performance is dependent upon involves a list of factors, such as use case requirements, streaming data traits, the knowledge encoded in the ontology, as well as the efficiency of the processing machines, etc. The performance can only be discussed after SIGenBench is deployed, is fed with streaming data and generates benchmark results.

6.4.3 Query

SIGenBench takes both CSPARQL and CQELS query languages. This is made possible because SIGenBench leverages CSPARQL and CQELS engines' query parsers. A parser splits a continuous query into two parts: a dynamic part with a stream and window definition, and a static part of a standard SPARQL query. Listing 6.1 shows an example of CSPARQL query, which defines the stream with "FROM STREAM" keyword, and the window with the argument of [RANGE 1s STEP 1s]. This indicates that the window size and step should be both 1 second. Listing 6.2 shows a CQELS query that contains the window definition as [RANGE 2s SLIDE 2s]. It defines both the window size and step as 2 seconds. Both queries can be parsed by their dedicated parsers into a standard SPARQL query shown in Listing 6.3.

Table 6.2: SIGenBench Built-in Window Management Strategies

#	Strategy Name	Semantic Importance Priority Vector
1	FIFO	$[\tau_a]$
2	QR-FI-FO	$[qrf, \tau_a]$
3	Trust-FI-FO	$[t_s^{tm}, \tau_a]$
4	FE-FO	$[\tau_e]$
5	QR-FE-FO	$[qrf, \tau_e]$
6	LFU-FO	$[f_{qp}]$
7	QR-LFU-FO	$[qrf, f_{qp}]$
8	LRU-FO	$[\tau_{qp}]$
9	QR-LRU-FO	$[qrf, \tau_{qp}]$
10	Trust-FO	$[t_s^{tm}]$
11	QR-Trust-FO	$[qrf, t_s^{tm}]$
12	QR-Trust-FI-FO	$[qrf, t_s^{tm}, \tau_a]$
13	FE-FI-FO	$[\tau_e, \tau_a]$
14	QR-FE-FI-FO	$[qrf, \tau_e, \tau_a]$
15	FE-LFU-FO	$[\tau_e, f_{qp}]$
16	QR-FE-LFU-FO	$[qrf, \tau_e, f_{qp}]$
17	FE-LRU-FO	$[\tau_e, \tau_{qp}]$
18	QR-FE-LRU-FO	$[qrf, \tau_e, \tau_{qp}]$
19	FE-LFU-FI-FO	$[\tau_e, f_{qp}, \tau_a]$
20	QR-FE-LFU-FI-FO	$[qrf, \tau_e, f_{qp}, \tau_a]$
21	FE-LRU-FI-FO	$[\tau_e, \tau_{qp}, \tau_a]$
22	QR-FE-LRU-FI-FO	$[qrf, \tau_e, \tau_{qp}, \tau_a]$
23	Trust-FE-LFU-FI-FO	$[t_s^{tm}, \tau_e, f_{qp}, \tau_a]$
24	QR-Trust-FE-LFU-FI-FO	$[qrf, t_s^{tm}, \tau_e, f_{qp}, \tau_a]$
25	Trust-FE-LRU-FI-FO	$[t_s^{tm}, \tau_e, \tau_{qp}, \tau_a]$
26	QR-Trust-FE-LRU-FI-FO	$[qrf, t_s^{tm}, \tau_e, \tau_{qp}, \tau_a]$

Listing 6.1: CSPARQL Query Example

```
REGISTER QUERY test AS
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?s
FROM STREAM <http://ex.org/streams/test> [RANGE 1s STEP 1s]
WHERE {
        ?s rdf:type ub:Chair.
}
```

Listing 6.2: CQELS Query Example

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?s
WHERE {
        STREAM <http://ex.org/streams/test> [RANGE 2s SLIDE 2s]
        {
        ?s rdf:type ub:Chair .
    }
}
```

Listing 6.3: Parsed Standard SPARQL Query

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?s
WHERE {
        ?s rdf:type ub:Chair.
}
```

6.4.4 Data Stream

SIGenBench is naturally compatible with RDF streams in basic or extended formats. A basic RDF stream format is $< \rho, \tau >$. In an extended format, for example, the RDF stream with graph IDs can be expressed as $< \rho, g, \tau_a >$, where g indicates the graph ID. This graph ID can contain information of the streaming source, or a dedicated ID for the specific data. τ_a represents the arrival timestamp. RDF stream format can be further extended as $< \rho, g, \tau_a, \tau_e, t_s^{tm} >$, where τ_e denotes the expiration timestamp, and t_s^{tm} denotes the trust-score. The expiration timestamp can be assigned by either the streaming source or the processing engine. The period between the arrival and expiration timestamp is the data's time-to-live. The trust score can be assigned either by the system or the streaming source, indicating the trust level of such data.

SIGenBench can accept either user specified data or a synthesized streaming data. The synthesized streaming data is derived from LUBM datasets [18]. Even though LUBM dataset is not designed for streaming settings, SIGenBench chooses it because of its easy usability and configuration. There are also some stream reasoning benchmark systems that use LUBM data [79]. A simulated streaming data is generated by considering 5 dimensions, including streaming mode, streaming rate, expiration timestamp, trust score, and the amount of data (lubm). Table 6.3 summarizes the parameters to configure the default streaming data.

Table 6.3: Streaming Data Generator Configuration

parameter	option	annotation
lubm	1: 100545 triples 2: 230063 triples 3: 337129 triples	LUBM source data
sm	c: constant mode f: fluctuating mode	stream mode
sr	10000, 30000, 50000 70000, 90000	constant rate in c mode, peak rate in f mode
ets	1: quick expiration (1s to 3s) 2: normal expiration (3s to 7s) 3: slow expiration (7s to 10s) 4: implicit expiration	how quickly the data expires
t	1: trust model 1 2: trust model 2 3: trust model 3 4: no trust model	choose a trust model, append a trust score to each data item

Parameter "lubm" controls the total amount of data items in the synthesized data stream. Normally streaming data is considered to boundless, however, in order to complete the experiment, the synthesized data is set to be finite. There are two streaming modes, the constant mode (c mode) and the fluctuating mode (f mode). C mode streams the data in a fixed data rate. For example, in the soccer use case, all sensors stream data with a fixed frequency. In c mode, the streaming rate indicates the actual data items streamed per second. F mode simulates bursts in streams with peaks and valleys. An example is the amount of traffic on a main road during the peak hours and off-peak hours. In f mode, the stream rate is actually the peak value. The probability of a peak value is 1/6, the probability of a valley (SIGenBench sets

it as 0) is 1/6, the probability of an in-between value is 2/3. The rationales to set probabilities like this are from the traffic model. During 24 hours, rush hours are usually 4 hours (usually 7am to 9am and 4pm to 6pm). From midnight to 4:00 am there are a few cars on the road; other time will usually have normal amount of traffic. Thus the peak probability is $4/24 = 1/6$, ditto for others. Streaming data can carry explicit or implicit data expiration stamps. Data can expire fast, normally or slowly. There three built-in trust models to assign trust scores for data items.

The data generator shuffles the LUBM data, so that the data is randomly distributed. Listing 6.4 shows an example of a streaming data item in $< \rho, g, \tau_a, \tau_e, t_s^{tm} >$ format. However, shuffling will possibly lead to no ground truth. Consider Examples in Listing 6.5 and 6.6. In Listing 6.5, graph32513 and graph47329 are both about `<http://www.Department5.University0.edu/FullProfessor4>`. They have an overlapping time period from 11:24:10.411 to 11:24:11.930. Thus they can generate the result that `<http://www.Department5.University0.edu/FullProfessor4>` is a **ub:Chair**. However, Listing 6.6 provides another data where graph47329 and graph56649 do not have an overlapping period, thus `<http://www.Department5.University0.edu/FullProfessor4>` will not be inferred as **ub:Chair**. These two examples are also corresponding to the early eviction and early expiration problems introduced in previous sections. If not enough ground truth is produced, the generator will run again till enough ground truth is produced.

Listing 6.4: Generated RDF Stream Example 1

```
<http://www.Department2.University0.edu/AssistantProfessor0/Publication7>
<http://swat.cse.lehigh.edu/onto/univ-bench.owl#publicationAuthor>
<http://www.Department2.University0.edu/GraduateStudent33>
<http://tw.rpi.pnnl.org/sigenbench/graph0>
11:24:05.679 11:24:08.679 0.7318180206732977
```

Listing 6.5: Generated RDF Stream Example 2

```
<http://www.Department5.University0.edu/FullProfessor4>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://swat.cse.lehigh.edu/onto/univ-bench.owl#FullProfessor>
<http://tw.rpi.pnnl.org/sigenbench/graph32513> 11:24:08.930 11:24:11.930 9.557774771623334

<http://www.Department5.University0.edu/FullProfessor4>
<http://swat.cse.lehigh.edu/onto/univ-bench.owl#headOf>
<http://www.Department5.University0.edu>
<http://tw.rpi.pnnl.org/sigenbench/graph47329> 11:24:10.411 11:24:12.411 9.5384934592904
```

Listing 6.6: Generated RDF Stream Example 3

```
<http://www.Department10.University0.edu/FullProfessor5>
<http://swat.cse.lehigh.edu/onto/univ-bench.owl#headOf>
<http://www.Department10.University0.edu>
<http://tw.rpi.pnnl.org/sigenbench/graph56649> 11:24:11.343 11:24:12.343 9.050994502744377

<http://www.Department10.University0.edu/FullProfessor5>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://swat.cse.lehigh.edu/onto/univ-bench.owl#FullProfessor>
<http://tw.rpi.pnnl.org/sigenbench/graph72474> 11:24:12.926 11:24:14.926 9.015205125311985
```

6.4.5 Ontology

SIGenBench can work with all ontologies encoded in OWL. Just like other benchmarks [11] [78], by default SIGenBench comes with a test ontology derived from LUBM [18]. This ontology itself contains detailed information about the updates that had been made and related rationales. However, it is worth mentioning this: a new class, the subclass of **Thing**, named **ub:DepartmentHead** is added, and defined as (*ub:headOf some* **ub:Department)**. It is added to facilitate the query relaxation on **ub:Chair**. Previously, **ub:Chair** is defined as (**ub:Person** *and* (*ub:headOf some* **ub:Department**)). There is an anonymous class that fails schema-based query relaxation due to blank nodes. In order to avoid this, **ub:DepartmentHead** is created and defined. The reason not to include it under **ub:Person** or **ub:Professor** is to avoid any possible RDFS hierarchical reasoning. *ub:headOf* is defined originally in LUBM without any domain or range, which means (*ub:headOf some* **ub:Department**) will not entail any type. This is also the reason not to include **ub:DepartmentHead** under **ub:Person** or **ub:Professor**. All the experiments will be conducted upon this default ontology in this dissertation.

6.4.6 Query Relaxation

Query relaxation is a general research topic that deals with relaxing a query when no results are returned,via loosening the constraints in the query [89] [103] [104]. In SIGenBench, a schema-based query relaxation method is used. The target query will be relaxed according to the window management strategy and system output. The classes in the target query will be replaced by their super classes in a relaxed query that will run again to collect any results. For instance, if the SPARQL query in Listing 6.3 cannot generate results, it can be relaxed according to

the schema in the background ontology **ub:Chair** ≡ (**ub:DepartmentHead** *and* **ub:Person**). Listing 6.7, and 6.8 show two relaxed queries.

The discussion of the performance of query relaxation in general is out of the realm of this dissertation. The purpose is to leverage some query relaxation techniques that can work well with SIGenBench. A future work would be to modularize the query relaxation component and allow users to specify they relaxation algorithms to use.

Listing 6.7: Relaxed Query 1

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?s
WHERE { ?s rdf:type ub:DepartmentHead.}
```

Listing 6.8: Relaxed Query 2

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?s
WHERE { ?s rdf:type ub:Person.}
```

6.5 Evaluation

This section provides detailed benchmark results. The analysis and discussions are focused on three main categories: semantic importance, window, and streaming data. All experiments are run on a Windows 7 machine with eight-core Inter(R) Core(TM) i7-3720QM CPU @ 2.6GHZ, 8GB RAM, and a SSD.

6.5.1 Semantic Importance

Each aspect of semantic importance has been benchmarked in multiple experiments so as to reveal its influence on the system performance.

6.5.1.1 Query Relevance

Query relevance is an important aspect, as it brings the domain into computation. It features a filter query and an optional query relevance ontology that encodes the knowledge of data-query relevance. The purpose to propose query relevance is to enable the system's identification of the relevant information to keep. By plugging a query relevance filter in front of the window, irrelevant data will be filtered out.

The hypothesis is that query relevance will help save system response time, which has been validated from the results at Table 6.5. The benefits brought by query relevance are augmented as system precision, memory consumption and throughput have all been improved. However, it is worth pointing out that the overall system performance under the query relevance aspect is dependent on the quality of the filter query, as well as the relevance ontology if provided. It is easy to imagine that a careless design and implementation of the filter query or ontology can adversely impact the results, as it has been shown by the below experimental results.

Three queries in different quality levels are included in Listing 6.9, 6.10, and 6.11. Listing 6.9 shows a good query relevance filtering query, where only the necessary data items are kept. To write good quality filter query, it is important to obtain enough knowledge about the streaming data and the target query. Being able to understand what data carries is a first step to a good quality filter query. This requires to do research on the streaming data. The second step is to understand the target query, which helps to determine what data to keep and what data to evict. Listing 6.10 shows an OK query relevance filtering query, albeit all the necessary data items are kept, it also introduces some unnecessary data items. Listing 6.11 shows a bad filter query that fails to capture all necessary data items.

Listing 6.9: Good Query Relevance Filtering Query

```
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?g
WHERE {
    GRAPH ?g {
        {?s rdf:type ub:FullProfessor}
```

```
        UNION
        {?s ub:headOf ?o}
    }
}
```

In order to determine if a person is a **ub:DepartmentHead**, according to the background ontology, two data items are needed: this person has to be both **ub:FullProfessor** and *ub:headOf* something. Technically, data items that do not have these two information are irrelevant, thus can be filtered out. Filter query in Listing 6.9 will only let the system to keep the relevant data. Typically, query relevance filter query can have two ways to filter the data: use some SPARQL Delete or Drop argument in the filter query itself to get rid of the irrelevant data, or use SPARQL graph pattern match to query the relevant data to keep. This dissertation adopts the second way, because the implementation is easier and faster.

Listing 6.10: OK Query Relevance Filtering Query

```
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?g
WHERE {
    GRAPH ?g {
        {?s rdf:type ?type}
        UNION
        {?s ub:headOf ?o}
    }
}
```

This OK query introduces some irrelevant data and sees it as relevant data, from the argument of "?s rdf:type ?type". This query will still be able to filter out some amount of the irrelevant data which can improve system response time. It is likely to use this type of filter query in real world use case though, because the knowledge about the streaming data and target query can be limited. However, the goal is to filter out as much irrelevant data as possible so that the system performance can be improved. As what will be shown later, streaming data rate

also influences the system response time. If the data rate is big, it will take longer time to process the filtering, which can hurt the overall usefulness of system.

Listing 6.11: Bad Query Relevance Filtering Query

```
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?g
WHERE {
    GRAPH ?g {
        {?s ub:headOf ?o}
    }
}
```

Listing 6.11 shows a badly designed filter query, where the relevant data of **ub:FullProfessor** is filtered out. Predictably, the target query will not provide any results. In general applications, one should avoid this type of query to be deployed in the system. Table 6.6 summarizes the impacts of the above three queries.

Table 6.4: Query Relevance Experiments Setup

experiment configuration	
window	logical lower-bounded landmark window initial window size = 1s, 2s and 3s worth of data items window step = 1s worth of data items report policy = onWindowClose all 26 strategies in Table 6.2
data stream	lubm = 1 sm = c (constant mode) sr = 10000 data items/second ets = 1 quick expiration (1s to 3s) t = 3 (trust model 3)
query	CSPARQL query
experiment variable	
QR filter quality	good as in Listing 6.9 OK as in Listing 6.10 bad as in Listing 6.11

Table 6.4 shows the experiments setup and variable. The reason to choose 1s to 3s for window sizes is because of the quick data expiration. Since the data will expire after 3 seconds, it is pointless to make the window size longer than 3 seconds

since data will expire within the window. In the following section, we will see that a bigger window size will impact the precision adversely when data quickly expires.

Table 6.5: Good Query Relevance Performance

strategy	memory	response time	precision	throughput
FIFO	11202.70	3363.00	0.08	2922.16
FE-FO	11748.26	3207.94	0.17	3039.05
LFU-FO	11265.96	2442.07	0.17	1657.71
LRU-FO	11265.96	2328.31	0.17	1647.76
FE-FI-FO	11791.24	2947.67	0.17	3282.85
FE-LRU-FO	11699.40	2937.96	0.17	3343.14
Trust-FO	1420.12	122.62	1	52088.41
FE-LFU-FO	11699.40	3184.02	0.17	3184.86
Trust-FI-FO	3371.94	178.03	1	41686.66
FE-LRU-FI-FO	11670.60	3207.79	0.17	3086.81
FE-LFU-FI-FO	11699.22	3033.25	0.17	3262.87
Trust-FE-LRU-FI-FO	2860.00	174.58	1	64967.53
Trust-FE-LFU-FI-FO	2128.82	162.76	1	69675.84
QR-FI-FO	18.98	372.07	1	31721.05
QR-FE-FO	27.35	330.27	1	33840.54
QR-LFU-FO	19.82	270.78	1	34113.49
QR-LRU-FO	19.82	263.19	1	34980.15
QR-FE-FI-FO	28.94	250.74	1	41555.22
QR-FE-LFU-FO	28.37	260.30	1	40068.32
QR-FE-LRU-FO	28.22	268.26	1	38881.49
QR-Trust-FO	82.13	255.51	1	29858.80
QR-Trust-FI-FO	82.13	252.80	1	30383.74
QR-FE-LFU-FI-FO	28.32	263.55	1	39400.00
QR-FE-LRU-FI-FO	28.29	263.65	1	39430.94
QR-Trust-FE-LFU-FI-FO	28.33	261.36	1	39796.71
QR-Trust-FE-LRU-FI-FO	27.94	277.98	1	37607.35

memory unit: number of data items
response time unit: ms
throughput unit: data items/second

Table 6.5 shows the system performance under all 26 built-in strategies with good query relevance filter query. It is obvious that QR aspect alone can improve system performance significantly. For example, for FIFO and QR-FI-FO, the memory consumption is 11202.70 vs 18.78, which is almost 600x improvement. Table 6.6 shows that compared with all non-QR strategies, on average good query rel-

Table 6.6: Filter Query Quality Experiment Results

quality	ave. memory	ave. response time	ave. precision	ave. throughput
none	8755.66	2099.23	0.42	19526.59
good	34.51	276.19	1.00	36279.83
OK	1882.19	739.97	0.92	19140.17
bad	5.62	250.33	0	41018.93

memory unit: number of data items
response time unit: ms
throughput unit: data items/second

evance can reduce memory consumption by 253.71 times, improve response time by 7.6 times, improve precision by 2.38 times, and improve throughput by 1.86 times. With a reduced memory consumption, the system is able to be deployed in a resource-constrained environment, or an environment that can allocate less resource to store streaming data. Since less data is in the window, the overall system will respond way more quickly, which can also increase the system throughput. In the good QR example, the system precision is improved because of only the necessary data is preserved.

For all non-QR strategies, FIFO has the worst precision, which shows that the temporal silent assumption generally does not work well. Trust related strategies show a superior performance in precision and throughput. In fact, non-QR trust related strategies have the best throughput of all. This is because of the fast execution of the trust comparison operation. Basically, all data will be ranked by their trust score first under Trust related strategies. Unlike query relevance where the filter query has to be executed, data filtered by trust score will only take linear time, which is way more faster. Another part of reason for non-QR trust related strategies' high precisions is that the data is generated with Trust Model 3. For details about the trust mode, please refer to the trustworthiness section below.

All QR related strategies have good precision of 1 and relatively high throughputs. Note that the streaming data has a constant rate of 10000 data items per second. All QR related strategies has a throughput bigger than 10000, which means that they are able to process that amount of data within one second. An eligible stream reasoning system should at least finish processing the data in the window

before window consumes next round of data so as to avoid data congestion or any delay. The average response time of all QR-related strategies with good QR filter is 276.19 ms and is smaller than window step, while non-QR strategies has 2099.23 ms. This means non-QR strategies will need more than 2 seconds to process 10000 data items, which makes the system not usable at all. The only exception in non-QR strategies is those non-QR Trust related strategies.

Table 6.6 shows the impacts from different filter query qualities. OK filter query impact is called "under filter impact"; bad filter query impact is called "over filter impact". In the under filter situation, more data are kept in the window thus system response time is slower than good QR's, but faster than the window step thus the precision is still good. Its throughput drops but still bigger than the stream rate thus capable of processing data stream. Nonetheless, in the over filter situation, not only all irrelevant data, but also some relevant data is filtered out, resulting in a very small memory consumption. The precision drops to 0, which means the system is barely useful. The experimental results illustrate that query relevance filter queries/ontology does have impacts on system performance, and it is critical to deploy good quality query relevance in the system for better performance.

6.5.1.2 Temporal Provenance

Temporal features are RDF streams' inherent nature. Stream reasoning applications should be able to keep a close track of the validity of all RDF stream items, and evict them if invalid within the window. Most stream reasoning applications use arrival or generation timestamps. Chapter 2 has already analyzed the impacts of these two. The recency timestamp will be evaluated in query participation section.

This section aims to show how system performance can be influenced by considering data expiration. The valid period or time-to-live of one data item can be obtained by subtracting its arrival timestamp from its expiration timestamp, assuming that data carries explicit expiration timestamps. If data only carries arrival timestamps, then its expiration timestamps will be assigned by window management strategies. If data expires quickly, the challenge of timely processing will be increas-

ing, as the delay tolerance will be decreasing. Table 6.7 shows the experiments setup.

Table 6.7: Data Expiration Experiments Setup

experiment configuration	
window	logical lower-bounded landmark window initial window size = 1s, 2s, and 3s worth of data items window step = 1s worth of data items report policy = onWindowClose
data stream	lubm = 1 sm = c (constant) sr = 10000 data items/second t = 3 (trust model 3)
query	CSPARQL target query good query relevance filter query
experiment variable	
data expire	ets = 1 quick expiration (1s to 3s) ets = 2 normal expiration (3s to 7s) ets = 3 slow expiration (7s to 10s)
strategy	FIFO, FE-FO, FE-FI-FO

Table 6.8: Data Expiration Experiment Results

expiration	strategy	memory	response time	precision	throughput
quick	FIFO	11202.70	3363.00	0.08	2922.16
	FE-FO	11748.26	3207.94	0.17	3039.05
	FE-FI-FO	11791.24	2947.67	0.17	3282.85
normal	FIFO	11202.70	2672.81	0.86	3819.41
	FE-FO	14415.96	2930.00	0.79	3335.24
	FE-FI-FO	14214.79	2850.27	0.79	3307.16
slow	FIFO	11202.70	2552.73	0.95	3908.22
	FE-FO	18308.39	2647.01	0.92	3099.00
	FE-FI-FO	18381.85	2607.76	0.92	3104.45

memory unit: number of data items
response time unit: ms
throughput unit: data items/second

In the default data generator, the expiration timestamps are randomly assigned but follow a uniform distribution. For example, in a quick expiration scenario, the probability of a data item to expire within 1s, 2s or 3s is equally 1/3. One data

items' expiration timestamp is its arrival timestamp plus its valid period. This is critical to explain the performance differences for three strategies in Table 6.8. In either data expiration scenario, all the strategies have similar response time and throughput. The obvious differences are precision and memory consumption.

In the quick expiration scenario, all three strategies have poor precision, and FIFO is the worst. All three strategies' response time is almost 3 seconds, which means none of them is able to keep up with data expiration. However, as data gradually expires slowly, the precision increases significantly. Response time has not changed much, but data time-to-live is prolonged, which allows finishing processing before data expiration and improves precision. However, no matter how good the precision looks, the fact that all three strategies are not able to finish processing before window moves makes them not applicable at all. In order to boost the response time, QR aspect is recommended. Table 6.5 shows that under quick expiration scenario, QR strategies show faster response time and higher precision than their non-QR counterparts.

Table 6.8 also shows that the memory consumption for FE-FO and FE-FI-FO increases as data expires slowly. This can be explained by the uniform distribution of data expiration as well. Consider the quick expiration scenario: FIFO is blind about the data expiration, thus each time it will evict the oldest 1 second worth of data items. However, generally only 1/3 of the oldest 1s worth of data will expire, thus the amount to evict in FE-FO and FE-FI-FO is always less than FIFO, resulting more data in the window. More data in the window means more time to process them, thus FIFO runs slightly faster than the other two as data expires slowly.

6.5.1.3 Query Participation

Query participation is essentially the statistics of data-query interaction. This includes query participation frequency (f_{qp}), and query participation recency (τ_{qp}). The idea is to collect either the frequency or the recency of the data that takes part in the query when results are given. Technically, in order to collect such statistics, reasoning explanation is required. Reasoning explanation is able to trace back to the query and reasoning process, and figure out the exact data items that contribute

to the results. Stardog provides such functionality. However, if the query result returns empty, query relaxation will be performed. Basically, if the query needs to be answered with multiple data items which can be arbitrarily apart, the system should be able not only to identify these partial data items, collect them and wait for all of them to arrive in the window, but also needs to keep an eye on the validity of the data. Table 6.9 shows the experiments setup.

Table 6.9: Query Participation Experiments Setup

experiment configuration	
window	logical lower-bounded landmark window initial window size = 1s, 2s, and 3s worth of data items window step = 1s worth of data items report policy = onWindowClose
data stream	lubm = 1 sm = c (constant) sr = 10000 data items/second t = 3 (trust model 3)
query	CSPARQL target query good query relevance filter query
experiment variable	
data expiration	ets = 1 quick expiration (1s to 3s) ets = 2 normal expiration (3s to 7s) ets = 3 slow expiration (7s to 10s) ets = 4 implicit
strategy	FIFO, LFU-FO, LRU-FO

Table 6.10 shows the experimental results. It can be seen that data expiration influences strategy precision most, which can be explained with the same rationales from temporal provenance. As data expires slowly, all strategies' precision increases. LFU-FO and LRU-FO has faster response time than FIFO, but smaller throughput. Generally speaking, query participation alone does not show an obvious improvements for overall system performance. The results indicate that query participation is most useful when data expires fast. Technically, none of the tested three strategies are applicable in stream reasoning settings due to their slow response time. However, if QR aspect is added as shown in Table 6.5, query participation related strategies are able to run faster. In summary, the purpose of proposing query participation

Table 6.10: Query Participation Experiment Results

expiration	strategy	memory	response time	precision	throughput
quick	FIFO	11202.70	3363.00	0.08	2922.16
	LRU-FO	11265.96	2328.31	0.17	1647.76
	LFU-FO	11265.96	2442.07	0.17	1657.71
normal	FIFO	11202.70	2421.43	0.88	4059.49
	LRU-FO	11232.80	2066.55	0.79	1968.92
	LFU-FO	11232.85	2084.09	0.79	1982.39
slow	FIFO	11202.70	2473.60	1.00	4070.88
	LRU-FO	11219.41	2140.02	1.00	2272.06
	LFU-FO	11219.41	2143.75	1.00	2252.67
none	FIFO	10832.36	2482.13	1.00	3789.58
	LRU-FO	11217.14	2148.26	1.00	2231.22
	LFU-FO	11217.14	2155.60	1.00	2230.48

memory unit: number of data items
response time unit: ms
throughput unit: data items/second

aspect is to make sure all the necessary data can be collected even though they are arbitrarily apart, with the premise that all data should be valid.

6.5.1.4 Trustworthiness

Trustworthiness of the streaming data is very helpful to enable data awareness. The trust model encodes the knowledge of whether and to what degree the data should be trusted. This dissertation assumes that data trustworthiness is reduced into a numerical value to facilitate data ranking. The trust score can be assigned by either the streaming source or the processing system using some trust models. Even though the trust models are not within the realm of this dissertation, it is worth pointing out that different trust models can affect the system performance.

The default streaming data of SIGenBench comes with 3 different trust models, which is shown in Table 6.11. The data items will be annotated with different trust scores by different models. Table 6.12 shows the experiment setup.

Table 6.13 obviously shows that different trust model can have different impacts for the system performance. The information needed to answer the target query is actually all about “FullProfessor”. This can be deducted from the target

Table 6.11: Trust Models Example

	trust model 1	trust model 2	trust model 3
related data	trust score range		
FullProfessor	(0, 1)	(5, 10)	(9, 10)
AssociateProfessor	(4, 7)	(5, 10)	(0, 1)
AssistantProfessor	(5, 9)	(5, 10)	(0, 1)
Publication	(5, 10)	(0, 10)	(0, 1)
ResearchGroup	(8, 10)	(0, 10)	(0, 1)
Student	(0, 2)	(0, 10)	(0, 1)
Course	(0, 1)	(0, 10)	(0, 1)
others	(0, 10)	(0, 10)	(0, 1)

Table 6.12: Trustworthiness Experiments Setup

experiment configuration	
window	logical lowerbounded landmark window initial window size = 1s, 2s, and 3s worth of data items window step = 1s worth of data items report policy = onWindowClose
data stream	lubm = 1 sm = c (constant) sr = 10000 data items/second ets = 1 quick expiration (1s to 3s)
query	CSPARQL target query good query relevance filter query
experiment variable	
trust model	t = 1 (trust model 1) t = 2 (trust model 2) t = 3 (trust model 3)
strategy	FIFO, Trust-FO

Table 6.13: Trustworthiness Performance

trust model	strategy	memory	response time	precision	throughput
Model 1	FIFO	11202.70	2517.42	0	3900.14
	Trust-FO	11544.25	500.62	0	24614.08
Model 2	FIFO	11202.70	2369.23	0.17	4211.35
	Trust-FO	22012.32	1448.05	0	6175.97
Model 3	FIFO	11202.70	3363.00	0.08	2922.16
	Trust-FO	1420.12	122.62	1.00	52088.41

memory unit: number of data items
response time unit: ms
throughput unit: data items/second

query in Listing 6.1 and RDF stream example in Listing 6.6. Model 1 assigns the smallest trust score for "FullProfessor", which means that the necessary data will be evicted quickly in the window by Trust-FO strategy. Model 2 Trusts information about professors but other data can also have high trust scores. Model 3 Trusts "FullProfessor" related data only. In order to explain the performance, both the target query and the window management strategies should be considered. In fact, Trust-FO only does one thing: to keep the data with the most trust scores. If the most trusted data contains all the necessary information to answer the query, it will get good result. If not, the precision will be bad. Model 3 consumes the least memory because the amount of data trusted in the stream is the smallest. Trust-FO always has a better response time and throughput over FIFO, but its precision performance is really dependent upon trust models.

6.5.2 RDF Stream

Streaming data is heterogeneous in many ways. SIGenBench models RDF stream in two dimensions, the streaming rate and the streaming mode. Streaming rate indicates how fast the streaming data flows. Streaming mode describes the behavior of the streaming data. A constant mode example is related to the frequencies of the sensor that streams the data. In the soccer offside detection use case, sensors are streaming data in high but constant frequencies. However, not all of the data streams are constant, there are situations where the data streams can be fluctuating. In other situations, for example, the traffic in the main street at rush hours will be significantly busier than non-rush hours, so the data streams to the traffic monitor systems will contain peaks and valleys.

SIGenBench results have shown that, the output will be affected by both stream rate and mode. For traditional window with FIFO strategy, this impact will be significant. However, for window management strategies supported by semantic importance, this impact can be reduced or even minimized.

6.5.2.1 Streaming Mode and Rate

Stream reasoning systems will often face high frequency data, which imposes the challenges to process the data in a timely and correct way. Window management

strategies are critical to help manage the data for this purpose. This section explores two features of the streaming data, which can be combined to stress test the system performance under different window management strategies. Table 6.14 shows the experiments setup.

Table 6.14: Streaming Mode and Rate Experiment Setup

experiment configuration	
window	logical lower-bounded landmark window initial window size = 1s worth of data items window step = 1s worth of data items report policy = onWindowClose
data stream	lubm = 1 ets = 1 quick expiration (1s to 3s) t = 3 (trust model 3)
query	CSPARQL target query good query relevance filter query
experiment variable	
stream mode	constant, fluctuating
stream rate	10000, 30000, 50000, 70000, 90000
strategy	FIFO, FE-FO, LFU-FO, Trust-FO QR-FI-FO, QR-Trust-FE-LFU-FI-FO

Table 6.15: Streaming Mode & Rate Experiment Results

rate/ peak	strategy	mode	memory	response time	precision	throughput
10000	FIFO	c	9090.09	2387.65	0	3479.44
		f	3689.74	1835.17	0	2488.33
	FE-FO	c	10889.80	2437.26	0	3594.77
		f	4314.85	1889.62	0	2359.58
	LFU-FO	c	10051.10	2080.50	0	2300.26
		f	5043.07	1732.40	0	1009.33
	Trust-FO	c	652.70	120.02	1	25171.51
		f	552.89	116.95	1	22709.22

Continued on next page

Table 6.15 – continued from previous page

rate/ peak	strategy	mode	memory	response time	precision	throughput
	QR-FI-FO	c	13.90	517.30	1	24171.51
		f	5.56	184.67	0	20474.84
	QR-Trust-FE-LFU-FI-FO	c	28.30	293.26	1	35803.06
		f	12.20	208.25	1	23005.49
30000	FIFO	c	29999.33	2841.18	0	7704.37
		f	12538.33	2309.36	1	6126.57
	FE-FO	c	32333.33	2979.89	0	7445.90
		f	12958.67	2274.00	1	6068.88
	LFU-FO	c	30009.00	2728.17	0	6070.64
		f	12560.83	2076.05	1	3685.24
	Trust-FO	c	679.00	115.72	1	137728.77
		f	618.83	112.45	1	92660.83
	QR-FI-FO	c	43.67	1150.80	0	28170.92
		f	18.33	562.06	1	37485.83
	QR-Trust-FE-LFU-FI-FO	c	58.00	638.88	1	48384.02
		f	28.83	336.41	1	52444.97
50000	FIFO	c	49999.50	3527.43	0	8161.62
		f	19756.00	2407.95	0.50	6062.71
	FE-FO	c	49999.50	3528.37	0	8159.63
		f	21394.60	2431.03	0.50	5884.92
	LFU-FO	c	50014.50	3466.14	0	6274.92
		f	19793.80	2306.43	0.50	4175.86
	Trust-FO	c	866.00	136.10	1	155880.62
		f	571.40	105.98	0.47	92661.75
	QR-FI-FO	c	46.33	2099.73	0	22068.26
		f	27.40	762.64	0	25789.82
	QR-Trust-FE-	c	78.50	959.14	1	47381.24

Continued on next page

Table 6.15 – continued from previous page

rate/ peak	strategy	mode	memory	response time	precision	throughput
	LFU-FI-FO	f	39.60	448.48	1	42709.43
70000	FIFO	c	70000.00	2915.00	0	13501.28
		f	19944.75	2286.47	0	8173.48
	FE-FO	c	70000.00	2781.25	0	14622.45
		f	22312.75	2375.16	0	7932.31
	LFU-FO	c	70000.00	2746.58	0	14479.26
		f	19976.75	2313.62	0	5725.95
	Trust-FO	c	816.00	140.43	1	389705.43
		f	616.67	111.17	1	179860.47
	QR-FI-FO	c	46.00	1670.50	0	49651.36
		f	36.67	718.50	1	45452.98
	QR-Trust-FE-LFU-FI-FO	c	46.00	1258.21	1	68397.28
		f	54.67	581.84	1	57191.13
90000	FIFO	c	90000.00	3621.67	0	10524.86
		f	18877.40	2432.07	0	5970.96
	FE-FO	c	90000.00	3560.00	0	10603.67
		f	20612.00	2516.05	0	5677.28
	LFU-FO	c	90000.00	3592.73	0	10478.79
		f	18917.00	2287.99	0	3967.91
	Trust-FO	c	1046.00	167.33	1	257805.13
		f	633.80	128.82	1	84771.50
	QR-FI-FO	c	61.50	2924.75	0	29737.95
		f	26.60	627.85	1	33434.99
	QR-Trust-FE-LFU-FI-FO	c	61.50	2034.42	1	41788.86
		f	41.20	439.10	1	44883.48

memory unit: number of data items

response time unit: ms

throughput unit: data items/second

Table 6.15 summarizes the streaming data rate and mode influences to the system under different strategies. In this table, mode "c" refers to constant mode, thus the streaming rate is constant; mode "f" refers to fluctuating mode, thus the streaming rate is a peak value. For all strategies, f mode streaming data will make strategies consume less memory consumption than what c mode does. This is expected, as in f mode, the data rate is fluctuating which can be possibly less than the constant rate. Because of less memory consumption, the response time is faster in f mode.

Generally speaking, as the data rate increases in c mode, FIFO strategy does not show any good performance, neither do FE-FO and LFU-FO. The reason is due to their slow processing speed. Trust-FO, QR-FI-FO, and QR-Trust-FE-LFU-FI-FO are able to keep a small memory and relatively fast response time, and good precision. However, both QR related strategies still failed to keep their response time within 1 second which is the window step. As it can be seen that, both QR-FI-FO and QR-Trust-FE-LFU-FI-FO's throughputs are gradually smaller than the data rate as the data rate increases, which also indicates a limit of these strategies as the data scale increases. Figure 6.3 shows the response time of all strategies in c mode and 90000 data rate. QR related strategies takes very long time to perform the query relevance filter. While Trust-FO is still able to keep a very fast response time. This indicates that query relevance filter aspect does have its limit in front of the big streaming rate. The reason for such good performance of Trust-FO is because trust Model 3 is deployed. If a different model is used, the performance will surely be different, as has been discussed in the above. Overall, Trust-FO performs the best.

In f mode though, things become different. Since now the streaming rate is the peak value with 1/6 appearance probability, the average data items arrived is surely less than that in c mode. Thus all strategies can perform slightly better than what they do in c mode.

Figure 6.4 shows how FIFO's response time fluctuates with the streaming rate. Stream reasoning system should be able to deal with stream bursts, which means

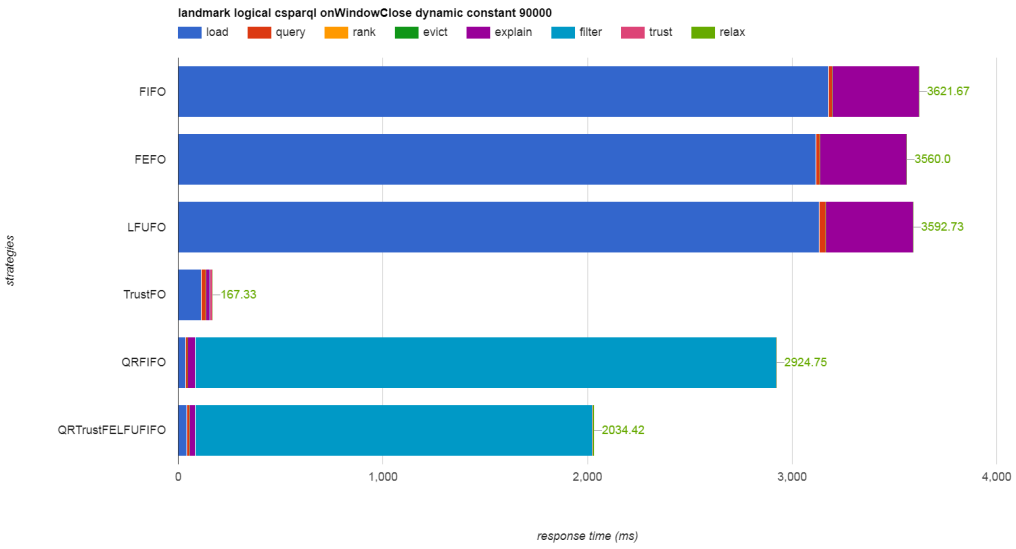


Figure 6.3: Constant Stream Mode 90k Rate Response Time

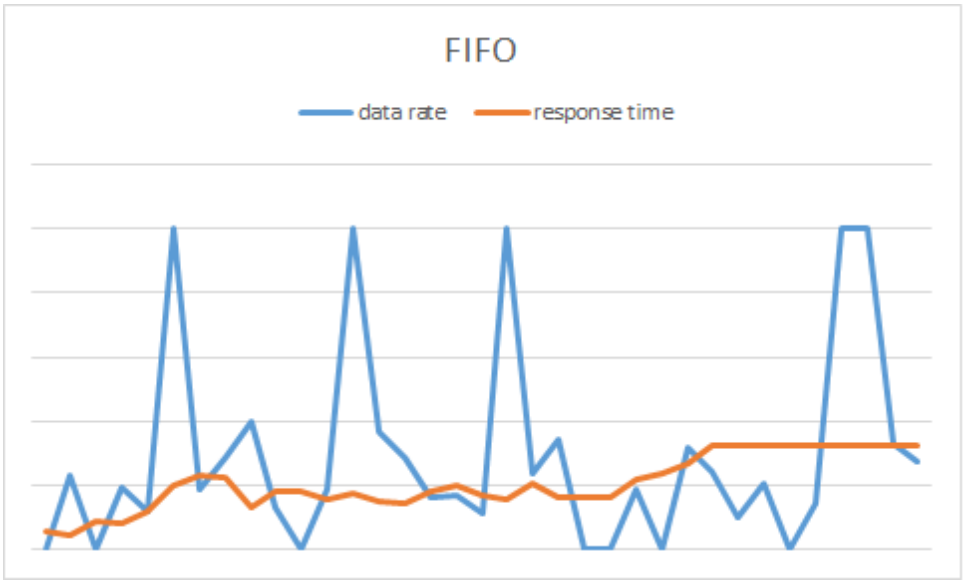


Figure 6.4: FIFO Response Time Fluctuates with Stream Rate

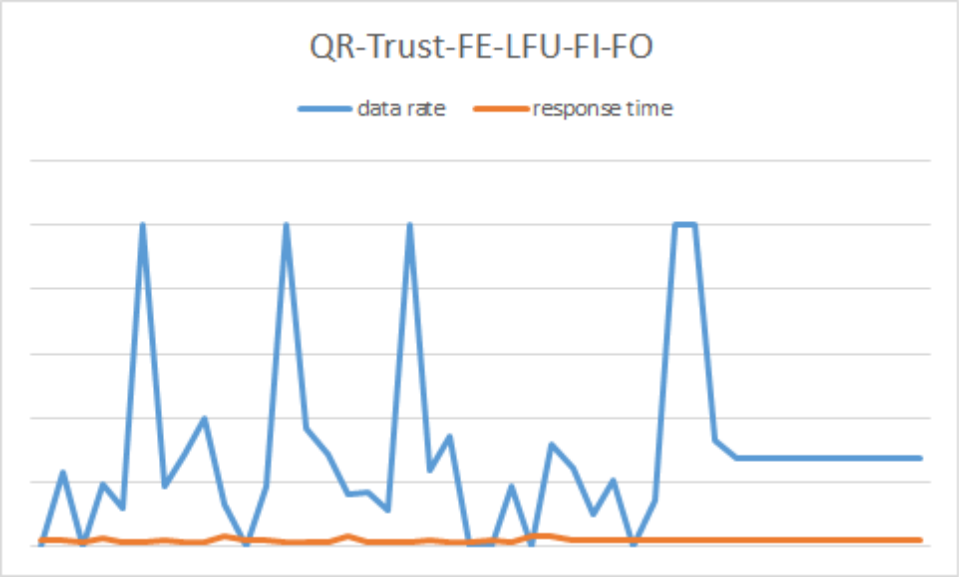


Figure 6.5: QR-Trust-FE-LFU-FI-FO Response Time is Stable

that the system should not lose response when data behaves with a sudden large rate increase. Figure 6.5 shows that semantic importance enabled window management strategy is able to keep a steady response time regardless of the streaming rate, which indicates that semantic importance is able to provide good system availability when facing versatile data streams.

6.5.3 Window

Window is an important part of stream reasoning systems. SIGenBench is flexible in configuring different types of windows. The results also show a significant relation between the window and the performance metrics. This section introduces some observations and brings up some discussions.

6.5.3.1 Window Size

Naively, the bigger the window size is, the more data it contains. This means that the window can "see" a bigger snapshot of the data, which makes it possible to provide better performance in precision. However, the evaluation will show that this is not generally true, especially when the data expires quickly. Table 6.16 shows the experiments setup.

Table 6.16: Window Experiment Setup

experiment configuration	
window	report policy = onWindowClose
data stream	lubm = 1 sm = c (constant) sr = 10000 data items/second t = 3 (trust model 3)
query	CSPARQL target query good query relevance filter query
experiment variable	
window	logical sliding window window size = 1s, 3s, 5s, 7s, 9s window step = 1s strategy: FIFO, QR-FI-FO
	logical lower-bounded landmark window initial window size = 1s, 3s, 5s, 7s, 9s worth of data items window step = 1s worth of data items strategy: FIFO, QR-FI-FO, FE-FO, LFU-FO, Trust-FO QR-Trust-FE-LFU-FI-FO
data expiration	ets = 1 quick expiration (1s to 3s) ets = 2 normal expiration (3s to 7s) ets = 3 slow expiration (7s to 10s) ets = 4 implicit

Table 6.17: Sliding Window Size Performance

expiration	strategy	window size	memory	response time	precision	through-put
quick	FIFO	1s	9999.10	2397.45	1	3816.67
		3s	12506.25	2713.23	0	3969.17
		5s	16691.67	2816.50	0	4635.94
		7s	24997.75	3342.02	0	5416.76
		9s	49995.50	5517.47	0	5691.2
	QR-FI-FO	1s	13.90	413.81	1	31280.34
		3s	27.00	292.25	1	37555.10
		5s	47.71	282.28	1	36745.25

Continued on next page

Table 6.17 – continued from previous page

expiration	strategy	window size	memory	response time	precision	through-put
		7s	69.80	305.25	1	33223.73
		9s	97.33	292.95	1	34335.72
normal	FIFO	1s	9999.10	2486.20	0.8	3578.65
		3s	12498.88	2732.78	1	3826.70
		5s	16665.17	3028.36	0	4254.73
		7s	24997.75	3230.50	0	5326.08
		9s	49995.50	4990.23	0	6035.96
	QR-FI-FO	1s	14.00	532.62	1	24508.78
		3s	29.62	271.65	1	39502.95
		5s	48.00	263.85	1	38786.65
		7s	75.00	281.03	1	35005.22
		9s	97.67	294.49	1	34277.19
slow	FIFO	1s	9999.1	2489.96	1	3820.44
		3s	12498.88	2716.76	1	3939.00
		5s	16665.17	2994.14	1	4356.12
		7s	24997.75	3283.32	1	5392.64
		9s	49995.50	4906.29	1	6294.06
	QR-FI-FO	1s	13.80	476.61	1	24921.91
		3s	28.50	272,70	1	39057.89
		5s	47.14	273.08	1	36624.77
		7s	70.20	269.89	1	36785.58
		9s	96.67	284.10	1	36531.61
none	FIFO	1s	9999.10	2307.75	1	3919.80
		3s	12498.88	2587.18	1	4106.65
		5s	16665.17	2928.87	1	4496.00
		7s	19998.20	2876.45	1	5260.03
		9s	49995.50	4704.67	1	6653.98

Continued on next page

Table 6.17 – continued from previous page

expiration	strategy	window size	memory	response time	precision	through-put
	QR-FI-FO	1s	13.90	683.52	1	18572.88
		3s	28.25	276.81	1	39690.09
		5s	49.57	274.90	1	39302.19
		7s	72.80	281.03	1	37639.46
		9s	97.00	283.03	1	36268.04

memory unit: number of data items

response time unit: ms

throughput unit: data items/second

Table 6.18: Landmark Window Size Performance

expiration	strategy	window size	memory	response time	precision	through-put
quick	FIFO	1s	9999.10	2430.96	0.50	3752.29
		3s	12498.88	2774.65	0	3795.21
		5s	16665.17	2818.66	0	4588.54
		7s	24997.75	3359.35	0	5153.53
		9s	49995.50	4847.37	0	6298.00
	QR-FI-FO	1s	14.00	284.13	1	38445.51
		3s	28.50	281.85	1	37866.29
		5s	51.67	281.55	1	37681.78
		7s	75.00	308.26	1	33850.17
		9s	101.33	305.02	1	36505.08
	FE-FO	1s	10756.00	2531.39	0.50	3541.09
		3s	12498.88	2691.70	1	3931.45
		5s	16665.17	2897.93	0	4484.57
		7s	24997.75	3336.05	0	5185.42

Continued on next page

Table 6.18 – continued from previous page

expiration	strategy	window size	memory	response time	precision	through-put
		9s	49995.50	4747.63	0	6456.97
	LFU-FO	1s	10045.40	2132.99	0.50	1938.02
		3s	12541.50	2279.74	0	2072.37
		5s	16717.17	2578.32	0	2390.56
		7s	25054.50	3175.83	0	3084.56
		9s	50044.50	4664.32	0	4838.07
	Trust-FO	1s	633.70	103.04	1	50698.44
		3s	3663.12	188.17	1	41681.18
		5s	1727.67	148.26	1	57547.22
		7s	2911.50	170.09	1	75533.43
		9s	4298.00	205.83	1	115557.47
	QR-Trust-FE-LFU-FI-FO	1s	23.80	280.54	1	36373.01
		3s	27.38	280.88	1	38139.23
		5s	30.29	283.78	1	38226.24
		7s	41.40	298.30	1	37966.39
		9s	53.00	289.23	1	41373.43
normal	FIFO	1s	9999.10	2419.96	0.80	3667.82
		3s	12498.88	2674.31	0.67	3837.80
		5s	16683.33	3000.33	0	4305.57
		7s	24997.75	3265.81	0	5274.38
		9s	49995.50	4582.60	0	6670.76
	QR-FI-FO	1s	14.00	266.50	1	40456.74
		3s	29.75	274.62	1	39103.46
		5s	48.14	271.19	1	37909.13
		7s	76.20	279.30	1	35815.82
		9s	97.00	279.83	1	37002.21
	FE-FO	1s	13623.80	2463.58	1	3373.77

Continued on next page

Table 6.18 – continued from previous page

expiration	strategy	window size	memory	response time	precision	through-put
		3s	15852.12	2736.27	1	3667.55
		5s	18404.50	3085.17	0	4192.63
		7s	24997.75	3264.10	0	5285.47
		9s	49995.50	4749.43	1	6437.54
	LFU-FO	1s	10048.00	2057.90	1	2093.87
		3s	12533.12	2313.39	1	2214.87
		5s	16707.50	2599.38	0	2497.70
		7s	25046.25	3091.38	0	3180.38
		9s	50035.50	4554.67	0	4853.48
	Trust-FO	1s	621.10	105.11	1	51929.24
		3s	3632.88	179.74	1	43540.49
		5s	1720.17	140.26	1	60236.67
		7s	2927.25	179.86	1	72851.45
		9s	4350.50	207.43	1	119400.24
	QR-Trust-FE-LFU-FI-FO	1s	49.20	276.39	1	34300.58
		3s	56.62	267.66	1	35337.43
		5s	58.86	277.59	1	35474.59
		7s	63.80	289.14	1	35905.36
		9s	73.33	280.02	1	39179.66
slow	FIFO	1s	9999.10	2495.20	1	3815.66
		3s	12498.88	2758.74	1	3789.91
		5s	16665.17	3033.92	1	4334.53
		7s	24997.75	3298.89	1	5377.64
		9s	49995.50	4734.86	1	6611.63
	QR-FI-FO	1s	13.80	263.60	1	41068.22
		3s	27.38	257.78	1	41135.43
		5s	46.71	266.54	1	37471.11

Continued on next page

Table 6.18 – continued from previous page

expiration	strategy	window size	memory	response time	precision	through-put
		7s	73.00	273.52	1	36718.41
		9s	96.67	265.43	1	39118.68
	FE-FO	1s	17786.00	2614.28	1	3063.04
		3s	20244.25	2907.29	1	3336.60
		5s	24809.00	3263.75	1	3616.50
		7s	29148.25	3436.78	1	5034.55
		9s	51284.00	4785.86	1	6618.06
	LFU-FO	1s	12256.30	2180.53	1	2118.27
		3s	12529.62	2297.83	1	2279.03
		5s	16701.83	2592.45	1	2591.91
		7s	25035.25	3127.57	1	3288.47
		9s	50024.00	4648.54	1	5091.41
	Trust-FO	1s	644.70	109.56	1	52009.83
		3s	3726.50	195.68	1	42545.49
		5s	1740.17	148.78	1	59877.90
		7s	2938.00	184.45	1	73760.09
		9s	4344.50	205.12	1	118976.33
	QR-Trust-FE-LFU-FI-FO	1s	64.00	268.07	1	31875.40
		3s	75.62	260.77	1	32973.11
		5s	80.14	274.95	1	32462.06
		7s	89.80	276.26	1	33907.25
		9s	99.33	268.83	1	38386.79
none	FIFO	1s	9999.10	2294.27	1	3948.59
		3s	12498.88	2602.75	1	3994.08
		5s	16665.17	3020.46	1	4367.48
		7s	24997.75	3235.59	1	5495.22
		9s	49995.50	4540.93	1	7001.04

Continued on next page

Table 6.18 – continued from previous page

expiration	strategy	window size	memory	response time	precision	through-put
	QR-FI-FO	1s	13.90	252.62	1	43018.83
		3s	28.62	270.16	1	40294.59
		5s	49.57	279.72	1	38342.87
		7s	73.00	267.42	1	39088.26
		9s	97.00	271.25	1	39286.83
	FE-FO[28]	1s	-	-	-	-
		3s	-	-	-	-
		5s	-	-	-	-
		7s	-	-	-	-
		9s	-	-	-	-
	LFU-FO	1s	10043.10	2088.77	1	1963.50
		3s	12533.25	2266.74	1	2091.60
		5s	16706.17	2557.57	1	2395.06
		7s	24043.50	3052.05	1	3083.90
		9s	50031.50	4666.90	1	4700.31
	Trust-FO	1s	638.20	108.65	1	52334.72
		3s	3705.00	199.62	1	42599.58
		5s	1727.83	147.46	1	61004.25
		7s	2929.50	164.79	1	77156.56
		9s	4337.00	222.00	1	111829.81
	QR-Trust-FE-LFU-FI-FO	1s	34.70	264.92	1	30697.71
		3s	39.25	258.48	1	31684.53
		5s	45.83	262.53	1	32078.81
		7s	42.60	271.61	1	32704.94
		9s	61.00	266.88	1	36624.77

memory unit: number of data items

[28]FE-FO can not run because of no data expiration.

response time unit: ms
throughput unit: data items/second

Table 6.17 shows sliding window size performance. Only two strategies FIFO and QR-FI-FO are employed so as to maintain the integrity of the sliding window semantics.

FIFO tends to consume much more memory than QR-FI-FO in all scenarios. Both strategies consume more memory as the window size increases. QR-FI-FO manages to keep the response time within 1 second, while FIFO's response time increases as the window size increases. When data expires quickly, FIFO's precision drops as window size increases. As data expires slowly, FIFO is able to keep a good precision. QR-FI-FO's precision is always 1, regardless how data expires or window size is. FIFO's throughput is way less than that of QR-FI-FO, in fact FIFO's throughput is less than the streaming rate.

Table 6.18 shows the performance of landmark window size. Lower-bounded landmark window will only have its upper bound proceeds. According to the extended window semantics, the window definition sets the initial window size of a lower-bounded window size. FIFO and QR-FI-FO's performance is quite similar as in Table 6.17. It can also be seen that FE-FO and LFU-FO performs similar as FIFO. Strategies with QR or trust aspect is capable to perform well even if window size increases. The compound strategy QR-Trust-FE-LFU-FI-FO shows a good performance as well.

The results show a strong correlation among the four metrics: the strategy will perform better as long as the memory consumption is small. With a small amount of data items in the window, response time will be fast, thus the system is able to keep up with the streaming data rate and expiration, thus the precision and throughput are improved. Data expiration also plays an important role for strategy performance. Strategies that can manage data in a fast speed can keep up with fast data expiration, which is the main reason to achieve the precision of 1. As the data expires slowly, all strategies can deliver good precision, because the data expiration is more tolerant with longer system processing time.

6.5.3.2 Report Policy

Report policies are part of window semantics. They determine the condition to fire the query and report the results. They are also part of the reasons to explain the different results reported by different stream reasoning engines [11]. SIGenBench supports four report policies. However, the influences of them on stream reasoning systems have already been discussed in the related work [11].

6.5.3.3 Strategy

Different window management strategies are supported by different combinations of semantic importance aspects, which views data importance in different ways, thus can provide different results and performances. However, the performance of any strategy is not only dependent on its semantic importance aspects, but also the environment it is deployed. For example, if the streaming data carries its own trust score, it is generally a good idea to add the trustworthiness in the strategy. In all ways, query relevance filtering is recommended because of its huge positive influence to the system performance. The reason to enable so many possible strategies from semantic importance is to allow the flexibility of choosing an appropriate one that can work well in a specific use case. This requires the users to be aware of the traits in their own use cases, as well as the performance impacts by different semantic importance aspect. The above experiments have already shown all the strategies' performances under different scenarios.

6.6 Summary

The above experiments have illustrated that SIGenBench is both powerful and flexible to generalize and benchmark semantic importance. Query relevance can significantly increase the strategy performance as in Table 6.5. A good query relevance can comprehensively improve stream reasoning systems. Even an OK or bad query relevance can still improve system response time. However, it can also be seen that as streaming rate increases, query relevance can reach the limitation where filtering time can take too long, which can negatively impact the response time and precision. Temporal provenance, especially the expiration timestamp, is suitable where

data carries explicit expiration timestamps. They are good at picking up expired data to avoid invalid query results. However, in order to guarantee system response time, the experimental results recommend to work with QR aspect. Query participation, although does not show an outstanding performance improvement, is still an important aspect. It is able to keep the partial necessary data items and wait for the other necessary data items to answer the query, which can improve system precision. From the experiment results, it is recommended to work with QR to improve system metrics, as well as temporal provenance if explicit expiration timestamps are carried. Trustworthiness is very straightforward. It relies trust models to assign numerical trust scores, based on which the data can be ranked. The experimental results have shown that different trust models can yield different system results. The algorithm to rank trust scores only takes O(n), where n is the number of data items in the window. This means trust aspect can run much faster, thus it is recommended to consider trustworthiness when data carries trust scores. Finally, a compound strategy like QR-Trust-FE-LFU-FI-FO considers multiple importance factors, and achieves far better performance than FIFO. This finding has already proves that semantic importance enabled window management strategies are able to improve the stream reasoning systems.

CHAPTER 7
CONCLUSION AND FUTURE WORK

Semantic importance (SI) is a powerful and flexible way to model the importance of the streaming data from various data orderings. The evaluation results have shown that SI is powerful at reducing system overhead and increasing system performance. The flexibility lies in the fact that SI supports a range of window management strategies that can efficiently manage the data in the window. SI is also expendable thanks to the SI ontology, which also helps understand what SI is really about by providing grounded instances. Implemented in OWL, SI ontology can be easily edited in ontology composing tools so that new aspects can be added. This chapter concludes this dissertation, and provides future insights on where SI can go from current standing point.

7.1 Conclusion

The core contribution of this dissertation is the notion of semantic importance, along with a set of infrastructure to enable its usage in the stream reasoning settings. Exemplar use case implementations have shown show how semantic importance can be leveraged in real-world scenarios. The comprehensive generalization and benchmark framework connects semantic importance to the state-of-the-art stream reasoning techniques, which allows testing system performances with multiple dimensional configuration parameters.

The motivation to propose semantic importance originally comes from the situations where the temporal silent assumption can fail. The temporal silent assumption, which regards the most recent data as the most important, only concentrates on one explicit data ordering – arrival order. This will cause two problems, early eviction and early expiration, as it has been shown in Chapter 1. It has been observed that these two problems are caused because the system is not able to distinguish the data based on its priority. Such priority, often implicit, leads to the efforts to make it visible for the processing system, which is formalized in the concept of

semantic importance. Semantic importance is derived from various data orderings, and currently provides four general aspects that can support a wide range of window management strategies. Like FIFO, these strategies help window to identify more important data with one or multiple data orderings; but unlike FIFO, these strategies provide more flexible options to choose under different stream reasoning scenarios, which can improve system performance.

Chapter 3 shows what semantic importance is, as well as its incompatibility with the existing window semantics that only works with FIFO. In order to deploy semantic importance in stream reasoning systems without breaking the integrity of window semantics, the landmark window is leveraged.

The semantics of landmark window provide a firm theoretical foundation for the application of semantic importance. Stream reasoning not only requires real-time stream processing, but also on-line reasoning. Most state-of-the-art work in stream reasoning performs logical reasoning, i.e., to provide a background ontology that contains the knowledge about the domain in the window. A reasoner also resides in the window, which can be used to infer hidden information together with the ontology and the streaming data. However, logical reasoning is relatively slow, and does not scale well with large volume of data. One method to minimize this problem is to reduce the data items in the window. This can be done by either shrinking the window size or filtering the data. When shrinking the window size, data items get more easily to exit the window under the silent assumption, but when filtering data, the system has to make sure to filter data correctly so that all the necessary data items are kept. Both of them require the system to be data discriminative, which is enabled by semantic importance.

Semantic importance currently includes four aspects, provenance, query participation, trustworthiness, and query relevance. It is designed to be flexible and extendable. It also comes along with an ontology that is grounded by real-life use case and instances. Semantic importance is embodied in a priority vector, which features a preference function that the most preferred element is placed leftist, as well as a comparison rule that enables ranking.

The semantics of sliding window, which works well with the temporal silent assumption, cannot be adopted when using semantic importance. This is because, other than FIFO, the strategies enabled by semantic importance will evict the data out of its arrival order. This can break the semantics of the sliding window. Thus, the landmark window is proposed to use, as its semantics works well under the semantic importance framework. Both time-based and tuple-based window semantics have been refined, which is not only compatible with the sliding window, but also opens up more window options to use in stream reasoning.

Chapter 4 introduces the sequential stream reasoning architecture (SSRA), with the purpose of providing the first architecture to deploy semantic importance for stream reasoning use cases. SSRA features five main component, other than the window implemented in off-the-shelf triple-stores, the data consumption component consumes the data by sending it into the window. Depending on different window report policies, the query execution component is fired and thus query results are generated. The reasoner inside of the triple-store can provide the ability to trace back to what happened during the reasoning and query process, such that the data items participated in the query can be tracked. At last, the data is evicted based on its semantic importance ranking. Usually, the lower-ranked data will be evicted first.

SSRA shows how semantic importance can be deployed in a stream reasoning application, with experimental evidence that shows SI efficacy. Since most state-of-the-art stream reasoning work is dependent on solely sliding window, which encapsulates the window and window strategies within the internal core, it is not feasible to implement semantic importance on the top of them. However, their architecture and design can be referenced, which leads to the sequential stream reasoning architecture. The core design is to let the window stand out of the processing architecture, so that it can be configured according to the users' need.

Chapter 5 shows two use cases implementations based on different window management strategies. Both of them are within the streaming context, where the data is either streamed in high frequency or large volume. They are real-world use

cases with different requirements. The results have shown that semantic importance is adaptable in different cases, and provide satisfactory system performance.

Chapter 6 generalizes semantic importance by connecting it to the state-of-the-art stream reasoning techniques, as well as provides a software that can benchmark the performance of stream reasoning applications. The experimental results have provided a systematical analysis on how different factors can affect the system performance, as well as how semantic importance can help minimize the adverse impacts and maximize the benefits. SIGenBench is carefully implemented to decouple the window semantics out of its processing engine, such as CSPARQL and CQELS. Currently, SIGenBench supports four kinds of window, four kinds of window report operational semantics, two kinds of continuous query language, and twenty-four different window management strategies. All of these open up a greater possibility to configure windows in many different ways. SIGenBench can also control stream rates and modes, so as to provide more simulation capabilities.

7.2 Future Work

Semantic importance currently only has four aspects, among which the provenance and trustworthiness aspects are both very broad. This dissertation only talks about some temporal provenance, and leaves other types of provenance untouched. For example, geolocation is an example of non-temporal provenance, and surely can be useful for some queries that involve geology. One future work is to explore and expand semantic importance both horizontally and vertically.

SIGenBench is a tool that helps test semantic importance. It is configured with parameters in the command lines. Users can feed their data, ontology, and choose their desired window and stream configurations before any experiments can run, from which the benchmark results will be generated. One restriction in this tool is that, all the strategies are currently built-in, which does not enable the user to choose their own desired strategies that are not listed within. The reason is because SIGenBench is implemented in Java, which is a static type language, and does not support explicitly creating undeclared classes during the run-time. There is really no solution to this in Java, thus what I did was to hard code all 26 strategies in

the explicit class files and call them during the run time. There are two directions in the future work to enhance it. First, to implement SIGenBench in a dynamic typed language, so that the strategies can be created during the run-time. Second, the strategies will be supported by loading the semantic importance ontology, where the users extend, specify and edit their extended aspects. SIGenBench should be able to create a list of possible strategies on the top of reading semantic importance ontology and let the user to specify what strategy they would like to test in their use cases.

The third future work is a more ambitious one. As time goes by, during which semantic importance is being used and extended, as well as different use cases are being developed, the data of both use case requirements, the chosen management strategies and semantic importance aspects will become more and more available quantitatively and in quality. By leveraging such data and supervised machine learning algorithms, a recommendation system should be feasible to implement, and able to provide some constructive suggestions on which semantic importance aspects and strategies to use for new users, based on the input of their use cases. The recommended results will be at least as a starting point from which the users will need to test their use case.

REFERENCES

[1] A. Rodriguez, R. McGrath, Y. Liu, and J. Myers, "Semantic management of streaming data," in *Proceedings of the 2nd International Conference on Semantic Sensor Networks-Volume 522*. CEUR-WS. org, 2009, pp. 80–95.

[2] E. Della Valle *et al.*, "Order matters! harnessing a world of orderings for reasoning over massive data," *Semantic Web*, vol. 4, no. 2, pp. 219–231, 2013.

[3] D. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "Stream reasoning: Where we got so far," in *NeFoRS 2010: 4th International Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic*, 2010.

[4] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "Querying rdf streams with c-sparql," *ACM SIGMOD Record*, vol. 39, no. 1, pp. 20–26, 2010.

[5] J.-P. Calbimonte, D. Dell'Aglio, E. Della Valle, M. I. Ali, and M. Alessandra, "Stream reasoning tutorial," 2015, http://streamreasoning.org/slides/2015/10/sr4ld2015-02-rsp-extensions.pdf, accessed 2016-02-01.

[6] A. Arasu *et al.*, "Stream: the stanford stream data manager (demonstration description)," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 665–665.

[7] S. Babu and J. Widom, "Continuous queries over data streams," *ACM Sigmod Record*, vol. 30, no. 3, pp. 109–120, 2001.

[8] H. Stuckenschmidt, S. Ceri, E. Della Valle, and F. Van Harmelen, "Towards expressive stream reasoning," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2010.

[9] I. Botan *et al.*, "Secret: a model for analysis of the execution semantics of stream processing systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 232–243, 2010.

[10] J.-P. Calbimonte, O. Corcho, and A. J. Gray, "Enabling ontology-based access to streaming data sources," in *International Semantic Web Conference*. Springer, 2010, pp. 96–111.

[11] D. Dell'Aglio, J.-P. Calbimonte, M. Balduini, O. Corcho, and E. Della Valle, "On correctness in rdf stream processor benchmarking," in *The Semantic Web–ISWC 2013*. Springer, 2013, pp. 326–342.

[12] L. Golab and M. T. Özsu, "Processing sliding window multi-joins in continuous queries over data streams," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 500–511.

[13] D. Barbieri *et al.*, "Deductive and inductive stream reasoning for semantic social media analytics," *IEEE Intelligent Systems*, vol. 25, no. 6, pp. 32–41, 2010.

[14] A. Mileo, A. Abdelrahman, S. Policarpio, and M. Hauswirth, "Streamrule: a nonmonotonic stream reasoning system for the semantic web," in *International Conference on Web Reasoning and Rule Systems*. Springer, 2013, pp. 247–252.

[15] F. I. de Football Association *et al.*, *Laws of the game*. FIFA, 1995.

[16] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus, "An execution environment for C-SPARQL queries," in *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 2010, pp. 441–452.

[17] J. R. Lacasse and E. Gambrill, "Making assessment decisions: Macro, mezzo, and micro perspectives," in *Critical Thinking in Clinical Assessment and Diagnosis*. Springer, 2015, pp. 69–84.

[18] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2, pp. 158–182, 2005.

[19] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[20] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[21] D. J. Abadi *et al.*, "Aurora: a new model and architecture for data stream management," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[22] N. F. Noy, D. L. McGuinness *et al.*, "Ontology development 101: A guide to creating your first ontology," 2001.

[23] S. Bechhofer, "Owl: Web ontology language," in *Encyclopedia of Database Systems*. Springer, 2009, pp. 2008–2009.

[24] O. Lassila and R. R. Swick, "Resource description framework (rdf) model and syntax specification," 1999.

[25] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, "The semantic web," *Scientific american*, vol. 284, no. 5, pp. 28–37, 2001.

[26] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data-the story so far," *Semantic services, interoperability and web applications: emerging concepts*, pp. 205–227, 2009.

[27] E. Della Valle, S. Ceri, F. Van Harmelen, and D. Fensel, "It's a streaming world! reasoning upon rapidly changing information," pp. 83–89, 2009.

[28] E. Della Valle, S. Ceri, D. F. Barbieri, D. Braga, and A. Campi, *A first step towards stream reasoning*. Springer, 2009.

[29] L. Ding, T. Finin, Y. Peng, P. P. Da Silva, and D. L. McGuinness, "Tracking rdf graph provenance using rdf molecules," in *Proc. of the 4th International Semantic Web Conference (Poster)*, 2005, p. 42.

[30] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data: Principles and state of the art," in *World wide web conference*, 2008, pp. 1–40.

[31] J. F. Sequeda and O. Corcho, "Linked stream data: A position paper," in *Proceedings of the 2nd International Conference on Semantic Sensor Networks-Volume 522*. CEUR-WS. org, 2009, pp. 148–157.

[32] D. F. Barbieri and E. Valle, "A proposal for publishing data streams as linked data," in *Linked Data on the Web Workshop*, 2010.

[33] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "C-sparql: Sparql for continuous querying," in *Proceedings of the 18th international conference on World wide web*. ACM, 2009, pp. 1061–1062.

[34] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink, "Towards ideal semantics for analyzing stream reasoning," *arXiv preprint arXiv:1505.05365*, 2015.

[35] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink, "Lars: A logic-based framework for analyzing reasoning over streams." in *AAAI*, 2015, pp. 1431–1438.

[36] S. Tallevi-Diotallevi, S. Kotoulas, L. Foschini, F. Lécué, and A. Corradi, "Real-time urban monitoring in dublin using semantic and stream technologies," in *The Semantic Web–ISWC 2013*. Springer, 2013, pp. 178–194.

[37] F. Lécué, S. Kotoulas, and P. Mac Aonghusa, "Capturing the pulse of cities: Opportunity and research challenges for robust stream data reasoning," in *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[38] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "Continuous queries and real-time analysis of social semantic data with c-sparql," in *Proceedings of Social Data on the Web Workshop at the 8th International Semantic Web Conference*, vol. 10, 2009.

[39] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[40] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[41] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal - The International Journal on Very Large Data Bases*, vol. 15, no. 2, pp. 121–142, 2006.

[42] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, "Ep-sparql: a unified language for event processing and stream reasoning," in *Proceedings of the 20th international conference on World wide web*. ACM, 2011, pp. 635–644.

[43] E. Thomas, J. Z. Pan, and Y. Ren, "TrOWL: Tractable OWL 2 reasoning infrastructure," in *The Semantic Web: Research and Applications*. Springer, 2010, pp. 431–435.

[44] A. Bolles, M. Grawunder, and J. Jacobi, *Streaming SPARQL-extending SPARQL to process data streams*. Springer, 2008.

[45] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "Incremental reasoning on streams and rich background knowledge," in *Extended Semantic Web Conference*. Springer, 2010, pp. 1–15.

[46] X. Ren and O. Curé, "Strider: A hybrid adaptive distributed RDF stream processing engine," *CoRR*, vol. abs/1705.05688, 2017.

[47] X. Ren, O. Curé, H. Khrouf, Z. Kazi-Aoul, and Y. Chabchoub, "Apache spark and apache kafka at the rescue of distributed RDF stream processing engines," in *Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016.*, 2016.

[48] K. Patroumpas and T. Sellis, "Window specification over data streams," in *International Conference on Extending Database Technology*. Springer, 2006, pp. 445–464.

[49] N. Dindar, N. Tatbul, R. J. Miller, L. M. Haas, and I. Botan, "Modeling the execution semantics of stream processing engines with secret," *The VLDB Journal*, vol. 22, no. 4, pp. 421–446, 2013.

[50] R. Volz, S. Staab, and B. Motik, "Incrementally maintaining materializations of ontologies stored in logic databases," in *Journal on Data Semantics II*. Springer, 2005, pp. 1–34.

[51] K. Nguyen, T. Scharrenbach, and A. Bernstein, "Eviction strategies for semantic flow processing," in *Proceedings of the 9th International Conference on Scalable Semantic Web Knowledge Base Systems-Volume 1046*. CEUR-WS. org, 2013, pp. 66–80.

[52] S. Gao, T. Scharrenbach, and A. Bernstein, "The clock data-aware eviction approach: Towards processing linked data streams with limited resources," in *The Semantic Web: Trends and Challenges*. Springer, 2014, pp. 6–20.

[53] T. L. Saaty, "Decision-making with the ahp: Why is the principal eigenvector necessary," *European journal of operational research*, vol. 145, no. 1, pp. 85–91, 2003.

[54] T. Lebo *et al.*, "Prov-o: The prov ontology," *W3C recommendation*, vol. 30, 2013.

[55] M. Gao, C.-Q. Jin, X.-L. Wang, X.-X. Tian, and A.-Y. Zhou, "A survey on management of data provenance," *Chinese Journal of Computers*, vol. 33, no. 3, pp. 373–389, 2010.

[56] S. Ram and J. Liu, "A new perspective on semantics of data provenance," in *Proceedings of the First International Conference on Semantic Web in Provenance Management-Volume 526*. CEUR-WS. org, 2009, pp. 35–40.

[57] E. Bertino, C. Dai, and M. Kantarcioglu, "The challenge of assuring data trustworthiness," in *International Conference on Database Systems for Advanced Applications*. Springer, 2009, pp. 22–33.

[58] J. Juran and A. B. Godfrey, "Quality handbook," *Republished McGraw-Hill*, 1999.

[59] B. K. Kahn, D. M. Strong, and R. Y. Wang, "Information quality benchmarks: product and service performance," *Communications of the ACM*, vol. 45, no. 4, pp. 184–192, 2002.

[60] R. Prince and G. G. Shanks, "A semiotic information quality framework," in *Proceedings of IFIP International Conference on Decision Support Systems (DSS2004): Decision Support in an Uncertain and Complex World*, 2004.

[61] Y. Wand and R. Y. Wang, "Anchoring data quality dimensions in ontological foundations," *Communications of the ACM*, vol. 39, no. 11, pp. 86–95, 1996.

[62] C. Date, "Database management systems," 2004.

[63] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, "The eigentrust algorithm for reputation management in p2p networks," in *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003, pp. 640–651.

[64] R. Levien, "Attack-resistant trust metrics," in *Computing with Social Trust*. Springer, 2009, pp. 121–132.

[65] E. PrudHommeaux, A. Seaborne *et al.*, "Sparql query language for rdf," *W3C recommendation*, vol. 15, 2008.

[66] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 702–713, 2015.

[67] D. F. Barbieri, D. Braga, S. Ceri, E. D. VALLE, and M. Grossniklaus, "C-sparql: a continuous query language for rdf data streams," *International Journal of Semantic Computing*, vol. 4, no. 01, pp. 3–25, 2010.

[68] D. L. McGuinness, F. Van Harmelen *et al.*, "OWL web ontology language overview," *W3C recommendation*, vol. 10, no. 10, p. 2004, 2004.

[69] T. M. Do, S. W. Loke, and F. Liu, "Answer set programming for stream reasoning," in *Canadian Conference on Artificial Intelligence*. Springer, 2011, pp. 104–109.

[70] M. Gebser *et al.*, "Stream reasoning with answer set programming: Preliminary report." *KR*, vol. 12, pp. 613–617, 2012.

[71] M. Gelfond and V. Lifschitz, "Classical negation in logic programs and disjunctive databases," *New generation computing*, vol. 9, no. 3-4, pp. 365–385, 1991.

[72] S. Gao *et al.*, "Planning ahead: Stream-driven linked-data access under update-budget constraints," in *International Semantic Web Conference*. Springer, 2016, pp. 252–270.

[73] Y. Ren, J. Z. Pan, and Y. Zhao, "Towards scalable reasoning on ontology streams via syntactic approximation," *Proc. of IWOD*, 2010.

[74] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler, "Named graphs, provenance and trust," in *Proceedings of the 14th international conference on World Wide Web*. ACM, 2005, pp. 613–622.

[75] D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte, and O. Corcho, "Rsp-ql semantics: a unifying query model to explain heterogeneity of rdf stream processing systems," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 10, no. 4, pp. 17–44, 2014.

[76] Y. Zhang, P. M. Duc, O. Corcho, and J.-P. Calbimonte, "Srbench: a streaming rdf/sparql benchmark," in *The Semantic Web–ISWC 2012*. Springer, 2012, pp. 641–657.

[77] D. Le-Phuoc *et al.*, "Linked stream data processing engines: Facts and figures," in *The Semantic Web–ISWC 2012*. Springer, 2012, pp. 300–312.

[78] M. I. Ali, F. Gao, and A. Mileo, "CityBench: A Configurable Benchmark to Evaluate RSP Engines Using Smart City Datasets," in *The Semantic Web-ISWC 2015*. Springer, 2015, pp. 374–389.

[79] R. Tommasini, E. Della Valle, M. Balduini, and D. Dell'Aglio, "Heaven test stand: Towards comparative research on rsp engines." in *SSN-TC/OrdRing@ ISWC*, 2015, pp. 87–92.

[80] M. Kolchin and P. Wetz, "Demo: Yabench benchmark, yet another rdf stream processing."

[81] P. Catteeuw *et al.*, "Offside decision making in the 2002 and 2006 fifa world cups," *Journal of sports sciences*, vol. 28, no. 10, pp. 1027–1032, 2010.

[82] W. Helsen, B. Gilis, and M. Weston, "Errors in judging "offside" in association football: Test of the optical error versus the perceptual flash-lag hypothesis," *Journal of sports sciences*, vol. 24, no. 05, pp. 521–528, 2006.

[83] S. Fowler, "How feasible is officiating technology in football?" 2010.

[84] W. C. Naidoo and J. R. Tapamo, "Soccer video analysis by ball, player and referee tracking," in *Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*. South African Institute for Computer Scientists and Information Technologists, 2006, pp. 51–60.

[85] J. Borg, "Detecting and tracking players in football using stereo vision," Ph.D. dissertation, PhD thesis, Linköpings Universitet, Institutionen för systemteknik, 2007.

[86] M. Regan and S. Holthouse, "Sports data collection and presentation," Aug. 30 2013, uS Patent App. 14/014,840.

[87] M. Garcia, A. Catalá, J. Lloret, and J. J. Rodrigues, "A wireless sensor network for soccer team monitoring," in *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*. IEEE, 2011, pp. 1–6.

[88] C. Mutschler, H. Ziekow, and Z. Jerzak, "The debs 2013 grand challenge," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 289–294.

[89] C. A. Hurtado, A. Poulovassilis, and P. T. Wood, "Query relaxation in rdf," in *Journal on data semantics X*. Springer, 2008, pp. 31–61.

[90] D. L. Costa *et al.*, "An insider threat indicator ontology," Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 2016, https://resources.sei.cmu.edu/asset_files/TechnicalReport/2016_005_001_454627.pdf, accessed 2016-08-07.

[91] G. J. Silowash *et al.*, "Common sense guide to mitigating insider threats," 2012.

[92] CERT, "Analytic approaches to detect insider threats," Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 2015, http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=451065.

[93] Y. Liu *et al.*, "Sidd: A framework for detecting sensitive data exfiltration by an insider attack," in *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*. IEEE, 2009, pp. 1–10.

[94] A. Giani, V. H. Berk, and G. V. Cybenko, "Data exfiltration and covert channels," in *Defense and Security Symposium*. International Society for Optics and Photonics, 2006, pp. 620 103–620 103.

[95] K. Born, "Browser-based covert data exfiltration," *arXiv preprint arXiv:1004.4357*, 2010.

[96] J. Glasser and B. Lindauer, "Bridging the gap: A pragmatic approach to generating insider threat data," in *Security and Privacy Workshops (SPW), 2013 IEEE*. IEEE, 2013, pp. 98–104.

[97] D. L. Costa *et al.*, "An ontology for insider threat indicators development and applications," DTIC Document, Tech. Rep., 2014.

[98] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth, "A native and adaptive approach for unified processing of linked streams and linked data," in *International Semantic Web Conference*. Springer, 2011, pp. 370–388.

[99] M. Dao-Tran and D. Le Phuoc, "Towards enriching cqels with complex event processing and path navigation." in *HiDeSt@ KI*, 2015, pp. 2–14.

[100] M. Kolchin, D. Mouromtsev, and S. A. Arustamov, "Web-based visualisation and monitoring of smart meters using cqels." in *TC/SSN@ ISWC*, 2014, pp. 109–112.

[101] A. Dejonghe, F. Ongenae, S. Verstichel, and F. De Turck, "C-geosparql: streaming geosparql support on c-sparql."

[102] A. S. Okure and J. Z. Pan, "Querying el ontology stream with c-sparql," in *Advanced Computer Science Applications and Technologies (ACSAT), 2013 International Conference on*. IEEE, 2013, pp. 337–340.

[103] B. Reddy and P. S. Kumar, "Efficient approximate sparql querying of web of linked data," in *Proceedings of the 6th International Conference on Uncertainty Reasoning for the Semantic Web-Volume 654*. CEUR-WS. org, 2010, pp. 37–48.

[104] A. Viswanathan, "Pragmatic querying in heterogeneous knowledge graphs," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

APPENDIX A
Semantic Importance Ontology

Semantic Importance Ontology (SIO) is an important intellectual contribution of this dissertation. SIO not only formalizes the concept of the semantic importance (SI), but also provides detailed exemplar grounding instances that greatly facilitate the understanding and reuse of SI. The OWL file resides in the github repository[29]. The instances are based on the soccer use case.

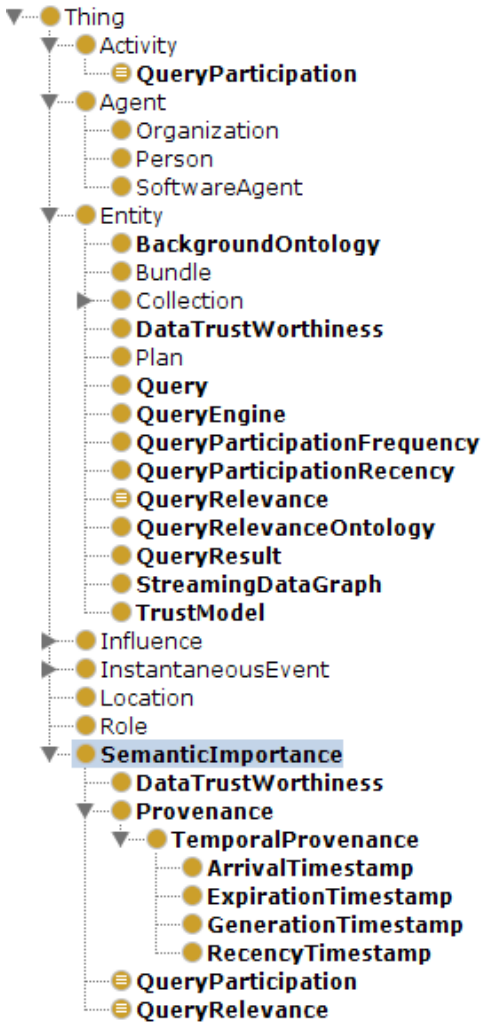


Figure A.1: Semantic Importance Ontology Class Hierarchy

[29] https://github.com/raymondino/SemanticImportanceOntology

A.1 SIO Class Definitions

Figure A.1 shows the class hierarchy in SIO. SIO imports Prov-O ontology, and its own classes are in bold. SIO is designed as a descriptive ontology.

Table A.1: SIO Class Definition

class name	definition
QueryParticipation	(*used some* **QueryEngine**) *and* (*used some* **StreamingDataGraph**) *and* (*used some* **BackgroundOntology**) *and* (*generated some* **QueryResult**)
QueryRelevance	*wasDerivedFrom some* **QueryRelevanceOntology**
other classes do not need to be defined with OWL since they are not compound concepts, and can be described in natural languages.	

Table A.1 shows the class OWL definitions. Both **QueryParticipation** and **QueryRelevance** are compound classes, thus needs to be defined in OWL with clear semantics. Other classes do not have to be defined in OWL as they can be described in natural languages.

A.2 SIO Class Instances

This section covers the grounded instances for each class in SIO. Table A.2 shows the all instances for each class. These instances are self-explanatory by their names. This section will only highlight some discussions.

DataTrustworthiness class has a concrete instance of *left_foot_position_is_trusted_right_foot_position_is_untrusted*. Chapter 5 has mentioned that Player B4's sensor is likely to fail. This observation leads to this simple data trustworthiness model, which essentially only trusts left foot position. Of course more sophisticated trust models are welcome, the point is to show an example of the trust model. Developing a trust model is out of the realm of this dissertation.

The instance of **Query** is *who_commits_an_offside_offence*. When deployed in the system, this instance is a physical continuous query file that can be loaded in the system, such as CSPARQL or CQELS queries. **QueryEngine** indicates what triple-store and SPARQL query engine that the system uses. In this case, Stardog

is used. Nonetheless, any other database will be the instance of **QueryEngine** if used.

QueryParticipationFrequency describes how many times the data participates in the query during its time-to-live. When deployed in the system, it is an integer value that can be used to rank the data.

StreamingDataGraph class describes the actual RDF streaming data. In this case, each triple is encapsulated in a unique graph, which makes it easier to manage the data. In fact, a graph can contain multiple streaming data items. An example is to use the streaming source as the graph ID and all the generated data items are within this graph.

Table A.2: SIO Class Instances

class name	instance example	annotation
Background Ontology	soccer_offside_ background_ontology	soccer offside background ontology provides necessary knowledge for defining & determining soccer offside.
DataTrust-worthiness	right_foot_position_ is_trusted_ left_foot_position_ is_untrusted	right foot position is trusted, left foot position is not trusted
Query	who_commits_an_ offside_offence	the query "who commits an offside offence" is the target query and registered in the system
QueryEngine	stardog_triplestore_ SPARQL_engine	in soccer offside use case, Stardog triplestore is used thus its SPARQL query engine is used.

Continued on next page

Table A.2 – continued from previous page

class name	instance example	annotation
Query Participation Frequency	graph1_query_ participation_ frequency_value	graph1 refers to "graph1_PlayerA_a_Attacker" (see StreamingDataGraph individuals for details)
Query Participation Recency	graph1_query_ participation_ recency_timestamp	the timestamp when graph1 participates in the query.
Query Relevance Ontology	soccer_offside_query_ relevance_ontology	the query relevance ontology to help filter irrelevant data out
QueryResult	PlayerA_commits_ an_offside_offence	the query result indicates that PlayerA commits an offside offence
Streaming DataGraph	graph1_PlayerA_a_ Attacker. graph2_PlayerA_a_ BallLastToucher. graph3_PlayerA_a_ PlayerAtOffsidePosition. graph4_PlayerB_a_ Attacker.	"graph1_PlayerA_a_Attacker" indicates that the triple "PlayerA a Attacker" is in graph1. In this example, every graph only has 1 triple. (It could have more, but we set it to only have 1)
TrustModel	left_foot_position_ has_trust_score_ of_1_right_has_0	trust model stamps trust scores to data
Arrival Timestamp	graph1_arrival_ timestamp	the arrival timestamp of graph1
Expiration Timestamp	graph1_expiration_ timestamp	the expiration timestamp of graph1

Continued on next page

Table A.2 – continued from previous page

class name	instance example	annotation
Generation Timestamp	graph1_generation_ timestamp	the generation timestamp of graph1
Recency timestamp	graph1_query_ participation_recency_ timestamp	the most recent query participation timestamp of graph1
Query Participation	graph1_graph2_graph3_ background_ontology_ stardog_query_engine_ target_query_ query_result	please refer to Table A.1
Query Relevance	graph1_is_relevant_ to_query graph7_is_irrelevant_ to_query	the relevance is determined by the query relevance ontology

A.3 Instance Property Assertions

Instances are inter-related. This relationship is established via properties.

Table A.3: SIO Instance Property Assertions

Class Name	**Instance Example**	**Property Assertion**
DataTrust-worthiness	left_foot_position_ is_trusted_ right_foot_position_ is_untrusted	object property assertion: wasDerivedFrom left_foot_position_ has_trust_score_ of_1_right_has_0 data property assertion: value 10
Query Participation Frequency	graph1_query_ participation_ frequency_value	data property assertion: value 3
Query Participation Recency	graph1_query_ participation_ recency_timestamp	data property assertion: value “2011-07-16T02:52:02Z” ^^dateTime
Arrival Timestamp	graph1_arrival_ timestamp	data property assertion: value “2011-08-16T02:52:02Z” ^^dateTime
Expiration Timestamp	graph1_expiration_ timestamp	data property assertion: value “2011-09-16T02:52:02Z” ^^dateTime
Generation Timestamp	graph1_generation_ timestamp	data property assertion: value “2011-10-16T02:52:02Z” ^^dateTime
Recency timestamp	graph1_query_ participation_recency_ timestamp	data property assertion: value “2011-11-16T02:52:02Z” ^^dateTime

Continued on next page

Table A.3 – continued from previous page

Class Name	Instance Example	Property Assertion
Query Participation	graph1_graph2_graph3_background_ontology_stardog_query_engine_target_query_query_result	object property assertion: generated PlayerA_commits_an_offside_offence; used graph1_PlayerA_a_attacker; used stardog_triplestore_sparql_engine; generated graph1_query_participation_frequency_value; generated graph1_query_participation_recency_timestamp; used soccer_offside_background_ontology; used graph2_PlayerA_a_BallLastToucher; used graph3_PlayerA_a_PlayerAtOffsidePosition
Query Relevance	graph1_is_relevant_to_query graph7_is_irrelevant_to_query	object property assertion: wasDerivedFrom soccer_offside_query_relevance_ontology

A.4 Described Examples

This section shows some concrete examples for selected classes and instances.

A.4.1 Query Participation Example

Data items participate in a query if they contain necessary information used by the query engine to return non-empty answers. When a data item particpates in a query, related meta-data such as the timestamp and frequency value can be collected. Consider a snapshot of streaming data as follows:

Listing A.1: Example Streaming Data

```
PlayerA a Attacker, graph1
PlayerA a BallLastToucher, graph2
PlayerA a PlayerAtOffsidePosition, graph3
PlayerB a Attacker, graph4
PlayerB hasPosition LeftFootPosition, graph5
PlayerB hasPosition RightFootPosition, graph6
PlayerC a Defender, graph7
```

In this use case, Stardog SPARQL engine is used as the **QueryEngine**, **BackgroundOntology** encodes that a player who commits an offside offence should be: (i) *player rdf:type* **Attacker**, (ii) *player rdf:type* **BallLastToucher**, and (iii) *player rdf:type* **PlayerAtOffsidePosition**. For the streaming data in Listing A.1; graph1 contains Triple i; graph2 contains Triple ii; graph3 contains Triple iii; graph4 contains Triple i; graph5 does not convey Triple i, ii, or iii; graph6 does not convey Triple i, ii, or iii; graph7 does not convey Triple i, ii, or iii. Thus, a **QueryResult** is given: Player A commits an offside offence. According to **QueryParticipation** class definition in Table A.1, the streaming data graphs used to generate the query result are said to have participated in the query. Thus: graph1, graph2, & graph3 have successfully participated in the query, while graph4, graph5, graph6, & graph7 have unsuccessfully participated in the query.

Graph1, graph2, & graph3's frequencies and recency will be updated. Graph4, graph5, graph6, & graph7's frequencies and recency will not be updated. If ranked by query participation, then graph1, graph2, graph3 will rank equally on the top, and graph4, graph5, graph6, graph7 will rank equally at the bottom.