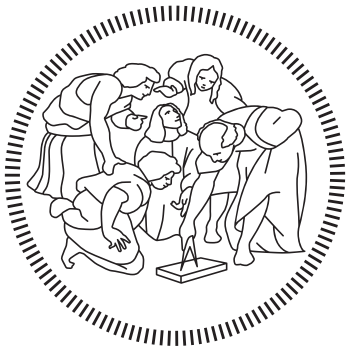
Politecnico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria



Scuola di Ingegneria Industriale e dell'Informazione

Leveraging Timely and Differential Dataflow for Efficient RDFS Reasoning

Advisor: Prof. Emanuele Della Valle
Co-Advisor: Riccardo Tommasini

Thesis Author:
Xhimi Hypi, Matr. 898382

Master of Science in Computer Science and Engineering
Academic Year 2020–2021

To the collective hard work, patience and abnegation of my family.

Abstract

In recent Big Data applications, Data Variety and Data Velocity are becoming progressively more important. Big Data is often associated with the Data Volume dimension that deals with the size of the data subject to analysis. As the size increases new challenges arise. Data Variety deals with the heterogeneity of the input data: data may come from different sources that generate information from different and complex domains using different structures. On the other hand, applications may require timely constraint and well-specified latency bounds in the output of the analysis. Data Velocity encompasses these type of requirements. Comprehensive approaches able to tackle challenges from all three dimensions are subject to research and they are often described as ideal models. In reality, approaches try to narrow down requirements to address separately, usually resulting in trade-offs. For example, Data Stream Management Systems (DSMS) and Complex Event Processors (CEP) successfully tackle part of the challenges in Data Volume and Data Velocity, but they proved to have limitations in other applications where Data Variety constraints are critical.

The purpose of this thesis work is to provide a new approach that collocates in the aforementioned scenario. In particular, we will start from high level requirements and progressively narrow down the scope of the problem in order to be able to formulate a more specific research problem. The aim of the thesis is to provide a working system able to perform Stream Reasoning from a Graph Stream Processing point of view. To achieve this we use Timely Dataflow, a framework written in Rust that proved great results when dealing with Graph Processing applications. Timely Dataflow introduces a new model of computation based on the Dataflow paradigm oriented towards efficient data-driven computation. We use DynamiTE as a reference system to approach aspects in this research field and implement its tasks with Timely Dataflow. In this thesis work we will provide the design and the implementation of the system. We will make performance evaluations and compare them to DynamiTE. The system provides highly-generic interfaces to favor extensibility. This, along with the performance evaluation results, will allow us to define directions to further improve performance and extend the system capabilities.

Sommario

Nelle più recenti applicazioni dei Big Data, la Varietà e la Velocità dei dati assumono un ruolo sempre più influente. Spesso i Big Data vengono associati al Volume dei dati soggetti ad analisi. Con l'aumentare del volume dei dati, si può incorrere ad una serie di nuovi problemi. La Varietà dei dati si occupa dell'eterogeneità dei dati in input: i dati possono provenire da sorgenti diverse che generano informazione appartenente a domini di natura diversa e complessa, strutturando i dati in maniera differente. La Velocità dei dati si occupa di quella serie di requisiti riguardanti il tempo. Alcune applicazioni possono richiedere requisiti di latenza ben definiti per funzionare correttamente. Approcci in grado di affrontare requisiti in tutte e tre le dimensioni sono tuttora soggetto di ricerca e vengono definiti in termini di modelli ideali. In approcci più realistici, viene considerato un sottoinsieme di requisiti, operando spesso in termini di trade-off. Ad esempio, i Data Stream Management Systems (DSMS) e i Complex Even Processors (CEP) sono in grado di fornire soluzioni a parte dei requisiti di Volume e Velocità dei dati, ma si mostrano limitati quando impiegati in applicazioni in cui i requisiti di Varietà sono centrali.

Questo lavoro di tesi si pone l'obiettivo di proporre un nuovo approccio nello scenario appena descritto. Partiremo definendo dei requisiti di alto livello per inquadrare la specifica collocazione di tale approccio, per poi restringere progressivamente i requisiti per poter formulare un problema di ricerca concreto. Questo approccio verrà supportato con lo sviluppo di un sistema in grado di eseguire operazioni di Stream Reasoning da un punto di vista di Graph Stream Processing. Utilizzeremo Timely Dataflow, un framework scritto in Rust che offre risultati promettenti quando applicato ad algoritmi di Graph Processing. Timely Dataflow è basato sul modello di computazione Dataflow per favorire l'efficienza della computazione data-driven. Useremo DynamiTE come riferimento per approcciare le sfide che tale area di ricerca pone, implementandone le funzionalita' principali. In questo lavoro di tesi descriveremo il design e l'implementazione del sistema, misurandone le performance, per poi comparare i risultati con DynamiTE. Il sistema offre un'interfaccia generica che ne favorisce l'estensibilità. Utilizzando questa caratteristica e i risultati forniti dalla misurazione delle performance saremo in grado di delineare le direzioni da intraprendere per migliorare le performance ed estendere le funzionalità del sistema.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Big Data applications introduce new aspects for the efficient execution of the functionalities. These aspects can be summarized with the renown "3 Vs" of Big Data [1]:

- Data Volume refers to the size of the data. Generally, the size of the data is defined through bytes number, usually terabytes or even petabytes.
- Data Variety refers to the large variety of sources, formats and structures. For example, in social media, networks have different data models and API.
- Data Velocity refers to the frequency at which data are produced and consumed. For example, sensor data in a network of sensors, or Social Networks data, fly into the systems in real time at high frequencies. This aspect leads to a relevant source of growth in Big Data. Moreover, new challenges arise when considering streams of data.

The Data Volume is addressed by distributing the computation and the input files over multiple computational units. One example of such approach is the MapReduce model [8], that is at the base of Apache Hadoop [9] and Apache Spark [10].

The Semantic Web Stack [11] offers solutions to Data Variety. These technologies are shown in Figure 1.2. The Semantic Web Stack defines the Resource Description Framework [12] that allows to formulate statements about resources that are uniquely identified by means of Uniform Resource Identifiers and and its extension to Internationalized Resource Identifiers. The framework enriches the data with semantics using metadata. The data model is not enough by itself for the full description of the data represented. Ontologies and Ontological Languages are introduced to give the definition of the terminology used by the matadata, defining constraints and logical properties between them. The Ontological Languages use the RDF data model. With these elements it is possible to

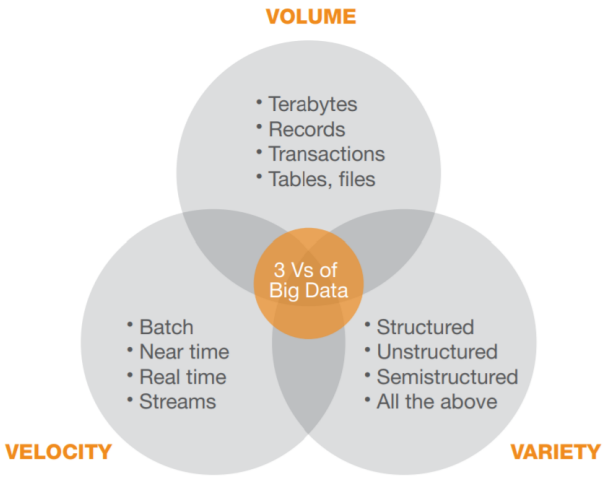


Figure 1.1: The three Vs of Big Data. Image from [1]

perform logical deduction to infer information from statements about Resources in the application by evaluating Logical Rules according to well defined formalisms.

Stream Processing is the research area that focuses on Data Velocity. It deals with managing highly dynamic data in a timely fashion. More and more often these challenges appear at the same time. In this context, Stream Reasoning is the area that tries to join the Data Velocity aspect with Data Variety taking into account Data Volume as well. In particular, it is aimed at performing reasoning tasks over Data Streams.

1.2 Research Questions

In this dissertation, we use the *Macro-Meso-Micro* framework [13] to outline three different level of analysis. The *Macro* level of analysis focuses on defining broad research question generally complex. At this level, we identify a set of requirements that narrow down the problem scope. The *Meso* level of analysis starts from these requirements and formulates more specific questions that belong to a medium scale application. The *Micro* level of analysis formalizes precise research problems that have clear success metrics, tackling specific parts of the more general context described at the higher level of analysis.
The Macro level of analysis of this dissertation aligns with the Stream Reasoning research question [14]: *can we make sense, in real-time or near real-time, of vast, heterogeneous data streams that originate from different complex domains, in order to be able to perform useful analysis?*. A promising approach to such challenge is the application of Graph Stream Processing based solutions. The semi-structured nature of the data is suitable for a Graph representation. This representation can be used by the components of the Semantic Web Stack to

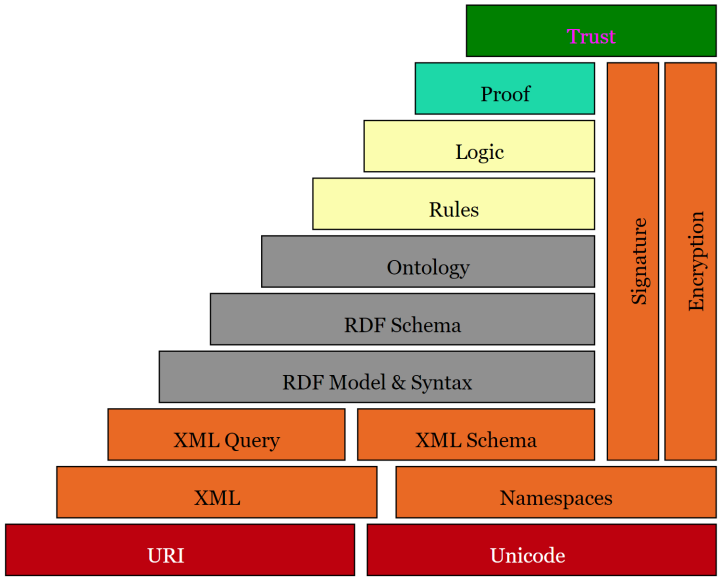


Figure 1.2: The Semantic Web Stack as in [2]

represent the meta-data keeping the graph representation. Thus, at this level, the focus narrows to using the Graph data model to deal with Data Variety in Stream Reasoning applications. Although Graph Stream Processing techniques exist, they were not employed, to the best of our knowledge, for Stream Reasoning. *Can we approach Stream Reasoning from a Graph Stream Processing standpoint?* To get to more specific research problems we need to narrow our focus even more. The Micro level of research question focuses on Timely Dataflow [15], i.e. a recent framework written in Rust developed for efficient Graph Stream Processing. Timely Dataflow has shown remarkable results when applied to common problems in Graph Stream Processing (e.g., connected components and node reachability of a large and dynamic graph such as the Twitter mention relation) and it provides a highly expressive dataflow language that can be used for Stream Reasoning applications. We want to explore if we can use this framework to perform efficient Stream Reasoning. To deal with this aspect we consider 3 Stream Reasoning approaches that can be ported on Timely Dataflow. In particular we select RDFox [16], IMaRS [17] and DynamiTE [7]. In this thesis, we focus on DynamiTE: we show how it is possible to use Timely Dataflow to outperform DynamiTE in its main functionalities.

1.3 Use Case

In this thesis work, we focus on performing full materialization and incrementally maintaining updates made of batches of triples. This is useful not only to explore Differential Dataflow technique to further extend it for timestamped updates (e.g. IMaRS [17]), but it can be applied for real world problems. We provide next one example of how maintaining batches of updates can be useful.

The *General Data Protection Regulation* (GDPR) is the European suite of laws that regulates how organizations are allowed to collect, process and delete data. One important topic faced by it is the *right to be forgotten*. As stated in the Article 17 [1] of the GDPR, a client of an organization has the right to request the the complete erasure of all the data pertaining him/her without undue delay. This problem can be interpreted as applying an update to the existing dataset of the organization. When the client requests his data to be deleted, the organization needs to apply a delete update on all the data that are related to the client. It might be a hard task for the organization to keep track of all the records that a client's information is included on. This can also be dangerous: in fact, if the organization forgets to delete some records, they would violate the *right to be forgotten*. For this reason, structuring the data so that inference procedures can detect all client's data and then delete them accordingly would be a solution to this problem. This is what ultimately we aim at doing in this thesis work: given a dataset and an inference procedure, the goal is to produce all the records that can be deduced from the existing dataset, with the ability of incrementally deleting (or inserting) batches of records and all the ones deriving from them.

1.4 Thesis Outline

This thesis work is divided in the following chapters:

- Chapter 2 offers a comprehensive view on all the knowledge used as a reference for a better understanding of the thesis: we describe the Semantic Web and how it copes with Data Variety, followed by the description of Stream Processing and how it deals with Data Velocity along with a brief description on Stream Reasoning, giving a general reference model describing the different components. We briefly present DynamiTE's approach that will be used as a reference in the thesis work.

- Chapter 3 uses the Macro-Meso-Micro level of analysis to define the Problem Statement this thesis revolves around.

- Chapter 4 deals with the design of the system provided and the design of the Dataflow computation that implements reasoning. We are going to state

[1] `https://gdpr.eu/article-17-right-to-be-forgotten/`

all the assumptions needed to support our work and formulate four goals that will lead the design and the implementation. We describe in details the design of the core library that provides functionalities to perform efficient materialization of RDF Data with incremental maintenance. After that, we describe the design of the dataflow computation we are going to use in out experiment.

- Chapter 5 shows the implementation of the main components of the library described in Chapter 4 and how these interoperate towards the goal of providing efficient reasoning support.

- Chapter 6 shows how the library defined in Chapter 4 and Chapter 5 is used to build a reasoner. We show how Timely Dataflow adapts to Stream Reasoning problems. At the end, we show the performance evaluation of the experiment we perform using the reasoner we built and compare our result to the system of reference. This will lead us to make meaningful observations on where the system can be improved and extended.

Chapter 2

Background

This section focuses on some of the main approaches and results in the context of Big Data challenges and attempts to gather most of the essential preliminary knowledge as a reference for a better understanding of the thesis work.

2.1 Data Variety: Semantic Web

Web Data do not always comply with the formal structures and specifications of the relational model, in fact along with Structured Data we can find:

- Semi-Structured data, such as XML data or RSS feed.
- Unstructured Data, such as natural language text, email, or even more advanced data such as audio an video files.

These types of data are increasingly important to society, so in order to support them, the definition of new data models and formalisms is required. The different structure of data suggests a graph representation, also referred to as graph data. The graph data models are also suitable to be extended to cover aspects of data semantics, giving rise to what is called Knowledge Graph. In these context the World Wide Web Consortium (W3C) has a rich suit of standards supporting this scenario.
The Semantic Web [11] is the extension of the World Wide Web that uses standards provided by W3C and it aims at making the data readable by the machine, by enriching the content of web data with metadata that provides a well defined interpretation of the content. It focuses on the interoperability of different data sources. To achieve this, the Semantic Web use the W3C's framework for graph data called Resource Description Framework.

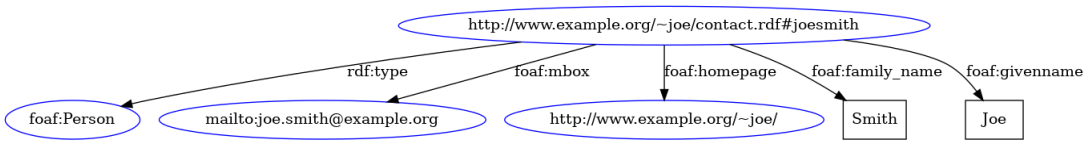


Figure 2.1: RDF Graph example.

2.1.1 Resource Description Framework - RDF

The Resource Description Framework [12] (RDF) is a W3C standard for describing resources. It supports features that facilitate data merging even in presence of different underlying schemas and data evolution over time. It uses three main concepts: *Resources* represent entities in the world, or portion of the world that the data is about. Determining what an entity is, is rather a deep question and is still subject of debate. *Properties* represent relations between resources and other resources or *Literals* that in turn represent constant values in the domain of the Resource. To address the ambiguity of resources coming from different domain, RDF identifies resources with Internationalized Resource Identifiers (IRI), that are a generalization of Uniform Resource Identifiers (URI) that provide a wider range of Unicode characters. In particular the set of URIs is a proper subset of the set of IRIs.
Resource and literals are the node of the graph and are visually represented as ellipses and rectangles respectively; Properties are the arcs in the graph. Each arc in the RDF model represents is called a Statement (it is also called a Triple) and it represent a fact about a resource. Each Statement has three parts, a *Subject*, *Predicate* and an *Object*. The Subject represents the Resource the Statement is about. Graphically, it is the node from which the arc originates; the Predicate is the property labeling the arc; the Object is a Resource or a Literal. It is the node pointed by the arc.

In these scenario, we move from tuples in a Database to triples in a RDF Graph, that we will consider from now on a set of statements. The RDF data model defines a formal semantics that is based on the intuition that tells that a statement makes a claim about the world. In other words, the world represents an *interpretation* that makes the statement true. From this perspective, the statements represent the constraints that the (possibly many) worlds need to fulfill in order to be an interpretation for that statement. The intepretations refer to a specific set of names, called vocabulary. In this context the formal definition of an Interpretations I of a vocabulary V is formalized by:

- *Domain* of I: a non empty set IR of resources
- Set of properties of I called IP
- A function $IEXT$ that maps IP into $\wp(IR \text{ x } IR)$, i.e. the set of of sets of

pairs of elements belonging to IR

- A function IS that maps URI values in V into $IR \cup IP$
- A function IL that maps typed literals in V into IR
- The set of all the plain literals in V, LV that is a subset of IR

This definition lays the foundation for the specification of further conditions that formally characterize the semantics of the data model. A complete description of this formalization is out of the scope of this document. Based on the definition of interpretation, we say that a set S of RDF graphs *entails* ($\models$) a graph E if every interpretation that satisfies every graph in S also satisfies E.

Different are the ways of serializing an RDF Graph to obtain an accessible machine representation[1]. *RDF/XML* uses the standard XML syntax for the description of the triples. Resources and predicates are represented as XML terms. It uses XML QNames to represent IRIs, which have a namespace name (the IRI) and a local name. Listing 2.1 shows the basic structure of a RDF/XML triple.

```
<Description about="subject">
  <predicate>object<predicate>
</Description>
```

Listing 2.1: RDF/XML basic triple structure

The serialization of the example graph in 2.1 is showed in 2.2

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns="http://www.example.org/~joe/contact.rdf#">
  <foaf:Person rdf:about=
        "http://www.example.org/~joe/contact.rdf#joesmith">
    <foaf:mbox rdf:resource="mailto:joe.smith@example.org"/>
    <foaf:homepage
            rdf:resource="http://www.example.org/~joe/"/>
    <foaf:family_name>Smith</foaf:family_name>
    <foaf:givenname>Joe</foaf:givenname>
  </foaf:Person>
</rdf:RDF>
```

Figure 2.2: RDF/XML serialization of the running example

The *NTriples* serialization technique is arguably the most straightforward representation: it consists in concatenating the IRIs related to the subject, predicate, and object. Listing 2.2 shows the basic structure of a Ntriples triple.

[1]The running example is taken from `https://www.obitko.com/tutorials/ontologies-semantic-web/rdf-graph-and-syntax.html`

```
<http://..subject> <http://..predicate> "value" .
```

Listing 2.2: NTriples basic triple structure

Listing 2.3 shows the Ntriples representation of our running example.

```
<http://../contact.rdf#joesmith> <http://..#type> <http://../foaf
   /0.1/Person> .
<http://../contact.rdf#joesmith> <http://..mbox> <
   mailto:js@example.org>  .
<http://../contact.rdf#joesmith> <http://..homepage> <http://../~
   joe/> .
<http://../contact.rdf#joesmith> <http://..family-name> "Smith" .
<http://../contact.rdf#joesmith> <http://..givenname> "Joe" .
```

Listing 2.3: NTriples serialization of running example

As NTriples does, *Turtle* concatenates the subject, the predicate and the object. But it aims at being more developer-friendly, so for this reason it shortens the concatenation of the different parts in the triple by declaring first the namespaces that are used and uses them as prefixes when listing the triples, as shown in Listing 2.4

```
@prefix sub: <http://..>
@prefix pred: <http://..>
sub:subject pred:predicate "value" .
```

Listing 2.4: Turtle basic triple structure

The Turtle serialization of our running example is in the next listing. Multiple facts related to the same subject can be listed using ';', without repeating the subject. Moreover, conventional properties like rdf:type are usually replaced with a more straightforward name, like in the example 'a' stands for rdf:type.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex-cont: <http://www.example.org/~joe/contact.rdf#>
ex-cont:joesmith
  a foaf:Person ;
  foaf:mbox <mailto:js@example.com> ;
  foaf:homepage <http://www.example.org/~joe/> ;
  foaf:family-name "Smith" ;
  foaf:givenname "Joe" .
```

Listing 2.5: Turtle serialization of the running example

2.1.2 SPARQL Query Language

Given this new type of data, new query languages were necessary to make use of them. SPARQL Protocol and RDF Query Language (SPARQL [18]) is the one designed in the context of the Semantic Web. It is both a query language and a protocol for accessing RDF. Its main goal it to retrieve semi-structured and structured data represented with the RDF data model. SPARQL gets in input the query and it produces a set of bindings or a RDF graph the result of query. It is important to observe that SPARQL does not perform any sort of reasoning on the data and so the result reflects the explicit data.
A SPARQL query is made up of different parts. These parts are shown in Figure 2.3.

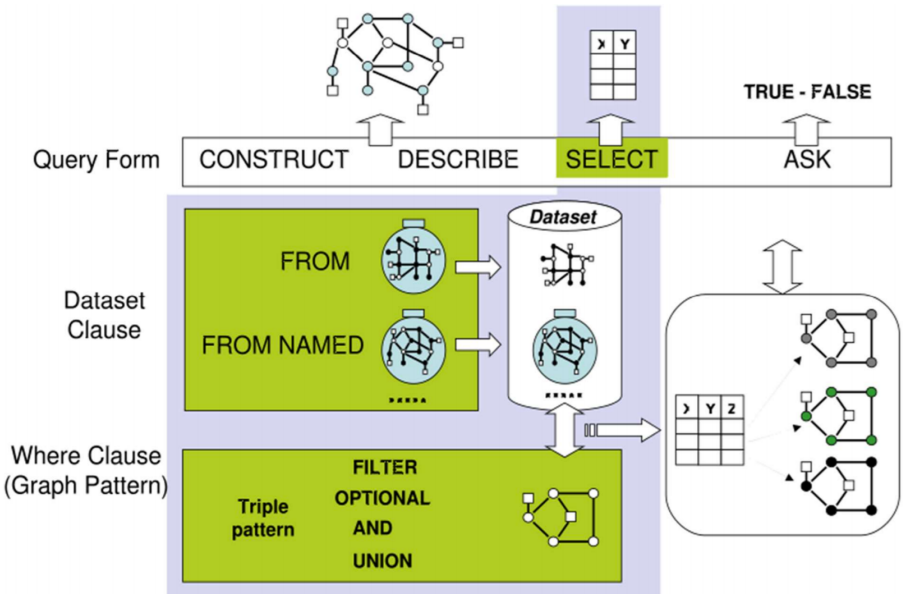


Figure 2.3: SPARQL Query Structure. Source: [3]

SPARQL retrieves results by graph pattern matching. This consists in finding a set of bindings between facts and patterns, so that the substitution of variables for those values creates a sub-RDF graph of the original RDF Graph.

2.1.3 Ontological Languages

With the RDF data model we just described we can represent knowledge. Knowledge representation is based on the definition of:

- Terminological Box (TBox)
- Assertion Box (ABox)

These two part make up the Knowledge Base of the part of the world the application is considering. The Tbox comprises Terminological Axioms of subsumption or equivalence, about atomic and compound concepts, e.g. *humans are biped mammals*. The ABox contains the factual knowledge with respect to individuals of a domain and represents the realization of the TBox given an interpretation, e.g *Alice is a woman* or *Bob is friend of Carl*. Ontological languages are the formal languages in this scenario used to express Knowledge Bases and to define their meaning (semantics) and they provide formalization through a suitable logic that for sake of simplicity we are not going to cover in this document. Ontological languages are described in terms of:

- A well-defined syntax: needed so that the machine is able to read this languages
- A well-defined semantics: that describes the meaning of the content of the knowledge, usually expressed as a set of constraints over the domain univocally determining the *models* of the statement, i.e. possible instances of the domain that satisfy the statement.
- Efficient reasoning support: defining models allows the definition of "deduction", i.e. being able to retrieve statement that are true in all the models of an ontology
- Expressive Power: the ability to express large classes of ontologies and knowledge.

Many Ontological Languages are available and they differ one from another by the language expressiveness and computational tractability. The Ontology Languages that we are going to introduce are: RDF-Schema [19] (RDFS) and Web Ontology Language 2 [20] (OWL 2). These formal languages are compatible with the RDF data model.

RDF-Schema - RDFS

The RDF-Schema [19] Ontology Language allows the user to:

- Define Class and Class membership: *rdf:type*.

```
:Professor rdf:type rdfs:Class
:Bob rdf:type :Professor
```

Listing 2.6: RDFS class and class membership

- Define and use Properties: *rdf:type*

```
:teacherOf rdf:type rdf:Property
:Bob :teacherOf :CS101
```

Listing 2.7: RDFS properties

- Define Class and Property hierarchies: *rdfs:subClassOf* and *rdfs:subPropertyOf*

```
:Professor rdfs:subClassOf :faculty
:memberOf rdfs:subPropertyOf :worksFor
```

Listing 2.8: RDFS class and property hierarchies

- Define Domain and Range of properties: *rdfs:domain* and *rdfs:range*

```
:teacherOf rdfs:domain :Professor
:teacherOf rdfs:range :Course
```

Listing 2.9: RDFS domain and range properties

These elements constitute the set of properties (a) of the RDFS language. RDFS provides additional supports:

(b) a set of classes, e.g. *rdfs:Class rdfs:Property*

(c) representation of abstract containers and container memebership properties, through words in the vocabulary like *rdfs:Container* and *rdfs:ContainerMembershipProperty* and their sub-classes and sub-properties.

(d) a set of utility words, e.g. *rdfs:seeAlso* or *rdfs:isDefinedBy*

RDFS main advantage is that it is a light Ontology Language that the definition of simple vocabularies, for this reason algorithm in this area are tractable. This thesis is going to focus on this Ontology Language, in particular on a fragment of it, called ρDF.

ρDF - RDFS Fragment

Theoretical studies on the RDFS physiognomy [21] led to observing the fact that features (b), (c), (d) offered by RDFS describe their internal function in the system of classes in RDFS. So this features do not refer to external properties. For this reason it is inconvenient to expose this features. Conversely, group (a) is the core of the language and it is the one used in most applications. From this observation, follows the fact that the features in group (a) form a meaningful fragment of RDFS. This fragment is called ρDF, and it is a subset of the RDFS vocabulary that uses only the words: *rdfs:subPropertyOf*, *rdfs:subClassOf*, *rdf:type*, *rdfs:domain* and *rdfs:range*.
ρDF has the important property of being the minimal subset of RDFS that retains the original semantic. In fact, as stated in [21]:

Theorem 1 *Let $\models$ be the RDFS entailment, $\models_{\rho DF}$ the ρDF entailment as defined in Section 2.1.1 and let G and H be RDF graphs that do not mention RDFS vocabulary outside ρDF. Then:*

$$G \models H \textit{ iff } G \models_{\rho DF} H$$

In other words, for statements that only use the ρDF vocabulary the inference G *entails* H does not need words in RDFS and not in ρ to hold. This shows how ρDF captures the essential semantic of RDFS.

Web Ontology Language 2 - OWL 2

Applications may require a higher expressiveness. For example, we might want to express cardinality constraints, e.g. a triangle has (exactly) 3 edges; Types of property: transitive, inverse or symmetric properties or classes as combination of other classes, such as union, intersection or negation. The Web Ontology Language [20] (OWL) aims at providing an enlarged expressive powers. This higher expressiveness does not come for free: the more the expressive power the less its computational tractability. For this reason OWL comes with several profiles each of which represents a differemt trade-off between expressiveness and tractability. To summarize, Figure 2.4 shows the relative inclusion of the different Ontology Languages we described so far.

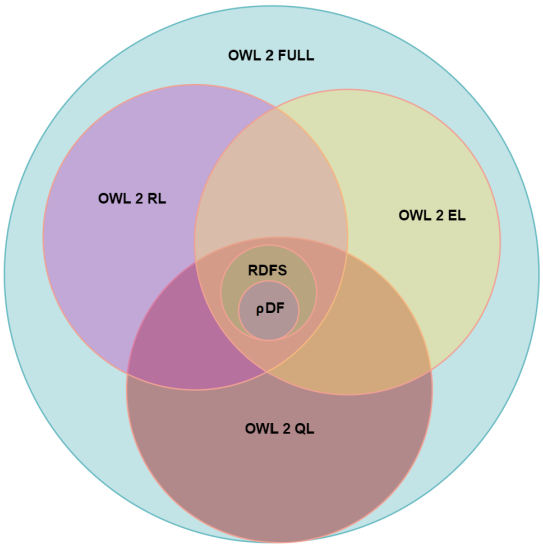


Figure 2.4: Ontology Languages. The figure represents the taxonomy that we described

2.2 Logical Reasoning

Semantically annotated data allows to perform reasoning services. Reasoning, in this context, means inferring logical consequences from an initial knowledge base.

These allows the derivation of implicit information from the existing explicit one. For example, if we assume we have the following simple Knowledge Base shown in Listing 2.10

```
:Teacher rdfs:subClassOf :Person
:Alice rdf:type :Teacher
```

Listing 2.10: Knowledge Base simple example

We want to be able to infer the fact that Alice is a Person so that the simple SPARQL query in Listing 2.11:

```
SELECT ?x
WHERE { ?x rdf:type :Person }
```

Listing 2.11: SPARQL Query against the example KB

will correctly return the expected binding in Listing 2.12.

```
?x -> :Alice
```

Listing 2.12: Inferred binding as a result of the SPARQL query

To allow this we need to determine reasoning procedures that would extend the initial Knowledge Base to contain the triple in Listing 2.13.

```
:Alice rdf:type :Person
```

Listing 2.13: Running example inferred triple

The extended Knowledge Base is referred to as the Materialization of the initial dataset. To express the inference rule, we need the specification of appropriate formalisms. The de-facto standard for expressing and querying knowledge is Datalog. A thorough description of the formalism can be found in Appendix B.

2.3 Data Velocity: Stream Processing

The Semantic Web is based on the *One-Time Semantic*, where reasoning is performed on static data . Clearly, this is a narrow view on the broad scope of data management and processing. In many applications Data Velocity is a fundamental characteristic of the data to be processed. In this case data is modeled as a Stream [14],[5]. For example, in financial applications, timely analysis on a continuous monitoring of stocks data is required to maintain up-to-data analytics. Sensors networks that communicate using the RDF data model can be used for environmental monitoring. The data in this scenario represents an observation of the world and, in order to detect anomalies a timely analysis of the data is key.

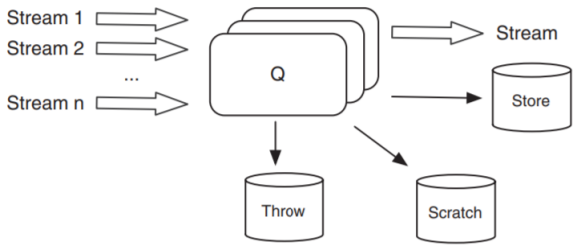


Figure 2.5: General DSMS model as in [4]

In general:

- Data enters the system as a flow of information that is usually timestamped. This timestamp must often be taken into consideration for a correct execution of the functionalities.
- Data is transient and does not need to be persistently stored. The system has to process the data on the fly, producing new information and/or updating the output. With that done, the input data can be discarded.
- Queries on these data are continuous, meaning that the answers to the queries have to be maintained over time as the data flows into the system, unlike DBMS systems, where queries are asynchronous with regard to changes in the input database.
- The propagation of the changes to the output has often tight latency constraints.

In literature, there are two main approaches to processing streams, that two complementary aspects of it, as described in [4]:

- *Data Stream Management Systems (DSMS)*: extend DBMSs for processing unbounded stream of data. Queries in these systems are installed and continue to produce results every time new data items arrive.
  DSMSs perform a sequence of transformation taken from SQL, such as selections and joins and other relational-algebra related operators. The general structure of a DSMS is shown in Figure 2.5. A DSMS is composed of a set of *standing* queries Q, possibly different input streams and four outputs. Let us call the answer of the queries A:
  - *Stream* is a data stream that contains the results to be appended to A and that never change.
  - *Store*: contains part of the answers that can are supposed to be in A but at some point the will no longer be in A. *Stream* and *Store* together define the current answer to Q.

  - *Scratch*: stores the data that is not to append to A but that may be useful for computing the answers.
  - *Throw*: represents all the data item that are not needed now nor later.

- *Complex Event Processors (CEP)*: this model of Stream Processing applications considers the data items as notifications of events happening in the external world. In this scenario, the focus is on comparing the input events with particular patterns to detect occurrences that represent compound events object of interest of the application. This model has its root in the publish-subscribe framework paradigm.

2.4 Stream Reasoning

What we described so far lays down the foundation for a new research trend: Stream Reasoning. As its name suggest, Stream Reasoning aims at bridging the gap between Semantic Reasoning and Stream Processing. In fact, it studies how to perform online logical reasoning on highly dynamic data.
In recent years, applications require the interoperability between heterogeneous data streams and large background knowledge bases [14]:

- in traffic monitoring applications, the locating of areas with a certain probability of a traffic jam, given the current weather and traffic conditions. In this case, the background knowledge base contains the map of the city/area in consideration, while the streaming data can represent the position of the cars in that particular area.

- Can I anticipate the failure of a electricity-producing turbine based on the (statistical) comparison between its sensor readings and the sensor readings of a turbine that had a failure in the past? In this case, we can consider the schematic of the component and the recorder readings as our knowledge base and the sensor data as the streaming data.

Reasoning services are usually non-trivial tasks and in some cases (e.g., OWL Full) are undecidable so no reasoning software is able to perform complete reasoning. On top of this, requirements like reactiveness that we find in Stream Processing introduce a more complex set of challenges.

As we can see in Figure 2.6, the generic Stream Reasoner model defined in [5] offers both DSMS and CEP-like query answering interfaces to the application and it is built to deal with streaming data. We can recognize two main components that have different levels of application:

- *Window Operators*: they control the access to the stream with time frames within which the reasoner performs reasoning. We have two possible interfaces:

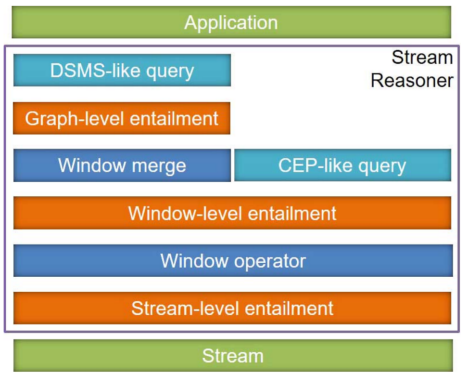


Figure 2.6: The Stream Reasoner reference model, taken from [5]

  - CEP-like query: CEP engines use the timestamps to determine if the temporal constraint in the event patterns are satidfied or not. Pattern matching is evaluated over a large portion of the stream when no time frame is specified.
  - DSMS-like query: time annotations are not needed once the windows have been computed. Aggregations and filters focus on the content of the stream. For this reason, the model defines the *window merge* operator that moves from temporal data to atemporal data, that is a batch of data items in the predefined window.

- *Logical entailment*: multiple are the moments where the inference process can be taken into account:

  - *Graph-level Entailment*: it happens after the window merge operator. This reduces this level of entailment to inference processes in the context of the evaluation of graph patterns over graphs. The downside of this approach is that only a portion of the information of the stream is used in the process.
  - *Window-level Entailment*: the inference process has access to the timestamped data items captured by the window operators, so in this case the process does not deal with a graph, but with a finite sequence of time annotated data items. This also represents its limitation. The process has a view on the most recent portion of the streams, and once the window has slid, the past information is lost.
  - *Stream-level Entailment*: The inference process has access to a larger portion of the stream (ideally the whole stream).

The decision on where to perform the different operation affects performances, the result and the behavior of the engine.

2.4.1 RDF Stream Processing

RDF Stream Processing extends the RDF data model to include timing annotation. This data model support the interoperability of complex and different domains at high speed. It deals with Data Variety and Data Velocity. Formally, a stream of data is an unbounded sequence of time-varying data elements. The basic data unit in this model is a RDF triple and a timestamp annotation. In this context, queries are registered on streams and use window to compute a continuous stream of answers. Some existing RDF Stream Processing systems are C-SPARQL engines, that comes with a RDF store and a stream processor. These are combined to provide continuous results to SPARQL like queries that are appropriately translated into a static component, that is run against the static knowledge base, and a streaming component run against the stream of data; CQELS [22] focuses on performances and it implements native algorithms to support continuous query answering. SPARQL$_{STREAM}$ [23] provides ontology based query answering, based on rewriting mappings that generate virtual RDF views, that are later fed to a DSMS/CEP-like engine to perform stream processing.

2.4.2 Incremental Reasoning

One fundamental challenge that the area of Stream Reasoning introduces in this context is how the reasoning process is maintained against changes in the input dataset, given by streaming nature of the data. This problem is also referred to as the *Incremental Reasoning* problem. The basic idea is to affect part of the materialization of a dataset when a change occurs, without re-materializing everything from scratch. These techniques have to identify the facts that have to be removed when facts are deleted and to be added when instead new facts occur. One of the most used incremental reasoning algorithms is the *Delete and Rederive (DRed)* algorithm [24].
The DRed algorithm builds two sets of axioms, one to be added and one to be removed and it is devided in three steps:

- compute the facts of the knowledge base that have to be marked as candidates for deletions from the set of axioms to be deleted. All the facts that derive by them should be marked to be deleted. Some of the facts produced in this step may have an alternative derivation in the existing dataset removed of the facts in the delete set of the input update. This results in an overestimation of the facts to be deleted.

- look for the facts to be deleted and remove them from the dataset.

- compute new derivations along with the alternative derivations of any deleted fact from the set of axioms to be added in the knowledge base.

The performance bottleneck of this algorithm is the overestimation performed in the first step: a significant amount of computation is spent to identifying the facts to be deleting to then re-compute them again. To cope with this aspect many alternative approaches to the DRed algorithm have been designed to identify early the set of facts to be deleted that has no alternative derivations.

2.4.3 Encoding

Stream Reasoning applications defined in this context work on RDF data. RDF Terms are encoded as strings (IRIs, literals, blank nodes). Performing complex operations on this datatype, e.g. join operations, is costly and methods to map RDF Terms as strings to a lighter datatype are required for an efficient computation. The basic approach to this problem is Dictionary Encoding. In this approach, the memory footprint of RDF dataset is reduced by building a dictionary like data structure that maps RDF Terms into numbers. The MapReduce framework [8], with systems such as Apache Spark [10], is used to perform this type of computation in a parallel and distributed fashion. Many techniques have been developed for the encoding of RDF datasets. Usually, the encoding method used is tailored to a specific application, to make the encoding efficient for a certain set of operations. For example, LiteMat [25] uses semantic encoding to ease the cost of materialization and query rewriting for RDFS reasoning. GStore [26], on the other hand, uses an encoding that optimizes graph pattern matching for SPARQL query answering. Other encoding techniques, such as the one presented in [27] take advantage of the regularities of the spatial RDF data for an optimized query evaluation. What characterizes these techniques is that the identifier associated with a RDF Term includes some information that makes some other operations more efficient:

- **gStore**'s [26] basic intuition is that answering a SPARQL query means finding a sub-graph over the RDF Graph corresponding to the dataset. The idea is that the neighbor structure of this sub-graph must match the structure of the SPARQL query graph in order to have pattern matching. With this in mind, the encoding consist in storing for each vertex of the RDF Graph information on each adjacent label and corresponding neighbor vertex as bit-strings. This is done through a system of signatures automatically computed that allows to reduce the pattern matching to a match of the corresponding signatures, therefore increasing the performances of query answering.

- **Encoding scheme for spatial RDF data** [27]. The identifiers are enriched with information regarding an approximation of the resource location and geometry. The space is partitioned using a grid and a *Hilbert* curve is generated to fill entirely this grid. This curve is needed to map a cell to a one-dimensional value. The identifier of the resource will then be split into

two parts, one containing the Hilbert order of the resource and the other containing a local identifier that distinguishes the resource from other resources in the same cell. For non-spatial RDF Terms a bit is used as a flag indicating whether the encoding needs to be interpreted as a spatial encoding or not. This encoding can then be used to optimize Filter conditions in a query evaluation plan and/or define spatial operators that apply on the approximate location/geometry indicated by the identifier.

LiteMat

LiteMat [25] is a scalable, cost-efficient inference encoding scheme for RDF graphs. This approach uses the distinction between a concept or predicate in the TBox and an instance in the ABox. In particular the two sets are encoded differently:

- TBox: The main idea is to assign to each entity an integer identifier that lies in an interval that can be easily computed from a super-entity identifier. In other words, considering two entities A and B such that B $\sqsubseteq$ A and *idA*, *idB* their identifiers, we have that: $idB \in]idA, idA + \epsilon[$, where ϵ depends on the number of direct sub-entities. First, the TBox is materialized using external reasoners, in order to capture entirely the hierarchy. After that, the hierarchy is navigated top-down and a previx value is given to the entites. After this phase it is guaranteed that given two entities A and B with B $\sqsubseteq$ A, the prefix of *idB* coincides with the *idA*. As we can see in Figure 2.7, the entity *owl:Thing* is encoded with the number 0. Since all entities are sub-entities of *owl:Thing*, they all share the same prefix 0 that corresponds to the super-entity identifier. Along the same line, the entity *Person* is encoded as 0 100 0..0. All sub-classes of *Person* (*TeachingAssistant, Student, AssociateProfessor*) start with 0 100, sharing the identifier.

- ABox: the ABox does not require an external reasoner. For this reason, it can be implemented in parallel using the Spark framework. No semantic is attributed to the encoding that therefore boils down to assigning a unique identifier to all the RDF Terms in the ABox that have not been already encoded in the TBox phase.

LiteMat aims at minimizing the amount of types that are associated with an instance in the dataset. The information stored in the encoding allows to retain only the most specific concept as the super-entities could be derived examining prefixes. To achieve this, the system sets up a MSC (Most Specific Concept) set. Starting from all explicit and implicit types involved in any of the individual's assertion (these concepts make up a set of candidates), the system inserts the candidates in the MSC set only if the concept to insert does not belong to the subsumption interval of any of the concepts in the MSC. The interval can be computed numerically with the identifiers attributed (See [25] for algorithm to

Encoding	Concept label
0 000 0000000000	Thing
0 001 0000000000	Schedule
0 010 0000000000	Organization
0 011 0000000000	Publication
0 100 0000000000	Person
0 100 0100000000	TeachingAssistant
0 100 1000000000	Student
...	
0 100 1110010011	AssociateProfessor
...	
0 101 0000000000	Work

Figure 2.7: Example of LiteMat encoding of the LUBM dataset

compute this interval). The MSC set contains the most specific concepts. This allows to avoid storing their super-entities, reducing the overall memory footprint. Furthermore, this approach simplifies the materialization logic. If we call *bound(p)* the function that returns the upper bound of the subsumption interval:

- A triple *(s, p,o)* produces a triple *(s,* p_1*, o)* if p is a sub-property p_1. This last condition is equivalent to: $p_1 \leq p < bound(p_1)$.
- A triple *(a, type, b)* produces a triple *(a, type, c)* if b is a sub-class of c. This last condition is equivalent to: $c \leq b < bound(c)$

The main limit of this approach is that it deals only with ρDF entailment and it does not support transitive properties.

2.4.4 DynamiTE

In this thesis document we focus on specific Stream Reasoner: DynamiTE. DynamiTE [7] aims at maintaining very dynamic and large materialized knowledge bases against set of input updates. It provides monotonic rule-based reasoning in a parallel and distributed environment. DynamiTE performs graph-level-entailment: in fact, the updates provided to the systems are removed of their time annotation, then they are processed as a batch of data. This makes the system suitable for DSMS-like queries, but inappropriate for complex event processing. Although limiting, this aspect allows us to focus on the design of reasoning processes setting aside for a moment the time management logic. DynamiTE takes as input the initial knowledge base and a set of Datalog rules that represent the reasoning task. The reasoning that is performed by DynamiTE is related to the ρDF ontology language, the minimal RDFS fragment that we introduced in Section 2.1.3. DynamiTE performs three main operations:

- Full Materialization of the input dataset.
- Incremental materialization when triples are added into the dataset.
- Incremental materialization when triples are removed from the dataset.

The dataset is compressed and B-Trees indices are created for faster access to the data when evaluating a query. The RDF terms are converted to numbers using the Dictionary Encoding technique. DynamiTE is able to perform reasoning according to the Datalog program given in input. A full description of the system and the incremental materialization techniques implemented is given in Appendix C.

2.5 Timely Dataflow

Timely Dataflow [15] is a framework written in Rust that requires understanding at different levels:

- The Timely Dataflow paradigm
- The Dataflow paradigm
- The Rust programming main principles

All these three different levels stand out as approaches that depart from the standard well-known programming conventions (e.g., imperative programming with Java, Python, C/C++, ..) in many ways, offering a challenging, yet fascinating, approach for data analysis. We briefly introduce Rust in Appendix A

2.5.1 Dataflow programming

Dataflow programming is based on the principle of describing a program as independent components, each one of these components is activated based on availability of input data. In this scenario, data drives the computation. This is in contrast with the imperative computation, for which the computation is specified by a sequence of instruction that describe what needs to happen next. In dataflow programming, we give the computer flexibility on how to perform the program instead of telling it what steps to execute in sequence. This separation of control of the computation among different independent operators allows an efficient and scalable data processing. Key feature of dataflow systems is the fact that each operator can be issued to a different worker, this allows for distributed and parallel data-driven computation.
Differential Dataflow can support timestamped data in scenarios that require them. Through these timestamps the system is able to communicate with the operators, telling them what time they can still receive data at. For the system

to know this information at a particular operator in the dataflow, it requires the upstream operators to make a statement about what is the last time they are done processing data at (e.g., input operator: "I'm done processing data at time T"). This information would flow downstream to any operator that was waiting on that time, enabling it to execute and advance is time accordingly. This design of the Dataflow programming has some limitations:

1. The operators are distributed over multiple independent nodes. Each operator uses the node resources differently. For example the *FILTER* and *MAP* operators use mainly the CPU while the *COUNT* operator uses memory. Specializing a computational node to one operator may lead to a poor resource management, as the node dedicated to a *FILTER* operator would not use its memory.

2. The operators have a narrow view over the whole dataflow: each operator does not know where his information is going to end up being once left its adjacent operators. This results in poor coordination of the system.

3. Dataflow programs are represented as Directed Acyclic Graphs (DAG), so for this reason the expressive power of the this paradigm is quite limited as it does not allow iterative computation. In fact, the narrow view of the operator combined with the fact that the system reports on the absence of the data doesn't allow the operator to realize when to exit the loop.

In this context, Timely Dataflow accounts for this limitations.

2.5.2 Timely Dataflow programming

Timely Dataflow extends the Dataflow programming and it revolves around three main characteristics.

1. The computation is distributed among *workers*, where a worker can represent a different thread, process or even computational node. Each o has a whole view on the dataflow. The dataflow program, in fact, is replicated in every worker so that a uniform and even resource management of the unit is allowed. In this way, every operator has responsibility over a fraction of all the operators, with the possibility to exchange data.

2. The operators interact with the whole system through different APIs: rather than reporting the absence of the data, they report the presence of the data (e.g., input operator: "I have data at time T"). The worker has that information and so it can report it to any operator that needs that information. This is also due to the fact that now, the workers have a complete view of the dataflow, so the only sort of coordination that is required is the one between all the workers that have to agree on what time the operators can still see data at.

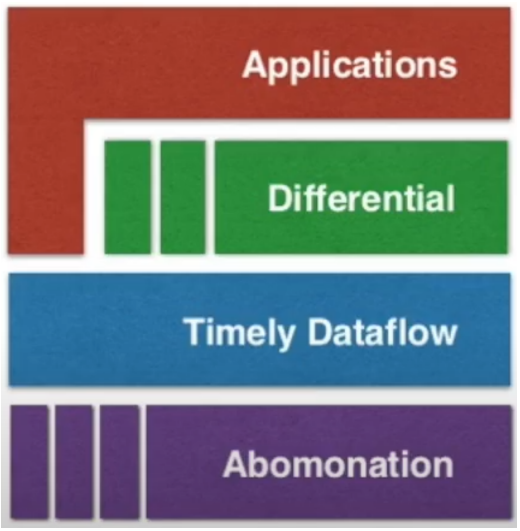


Figure 2.8: Timely and Differential Dataflow software stack

3. The previous points are also necessary to allow Timely Dataflow to enable iterative computation. Reporting on the presence of the data, along with the whole view of the dataflow of each worker, allows to realize when the loop is over.

The difference about the APIs between the two operators has a great impact on the expressive power and is subtle: In a timely dataflow, when data does not flow in the iterative loop, the inner loop does not have the information "I have data at iteration n", this fact is enough for the system to understand that the lack of the presence of that information represents that the iterative computation is done. In standard dataflow programming, the narrow view of each operator and the fact that the data is not available at a certain time does not necessarily mean that the computation is over, as it might mean that the computation is over for that iteration and so the operator is ready for the next iteration. Not being able to identify when to stop.

2.6 Differential Dataflow

Differenital Dataflow is a computational framework built on top of Timely Dataflow. It extends timely dataflow as it aims at performing computations of large datasets and perform output maintenance as data changes. This framework tries to abstract from all the low level implementation details that Timely Dataflow introduces, providing an interface that looks like many standard "big data" application interfaces, using idioms from SQL and MapReduce. Figure 2.8 shows the different layers that make up the application stack of a Timely/Differential Dataflow

application. We can define the following components:

- Abomonation is the serialization framework that provides data serialization at memory bandwidth, that it is used by Timely Dataflow to move data around the operators.
- Timely Dataflow provides core services for data-parallel computation. Serves as the operating system for this kind of application.
- Differential Dataflow represents a language for (real-time) scalable computation
- Differential Dataflow allows to re-export lower levels tools and make them available at the Application level in case applications needed.

The main extension with respect to Timely Dataflow is given by the fact that Differential Dataflow defines functions to modify the input data and propagates this changes to the output. This is the functionality the we ultimately aim at testing as a new approach to the problem of incremental reasoning. The technique used by Differential Dataflow to perform the incremental maintenance is called Differntial Computation and the formal explanation can be found in [28].

Chapter 3

Problem Statement

In this chapter, we formulate the problem we are going to investigate in the rest of the thesis work. First, we introduce relevant concepts in the literature of stream reasoning. Second, we present relevant work in graph (stream) processing. Third, we connect the dots across existing solutions, highlighting important gaps in the state of the art. Along the chapter, we formulate our research questions, progressively narrowing down the thesis scope to identify precise research problems.

3.1 Problem Formalization

In Big Data applications [1], Data Volume, Data Variety and Data Velocity represent the main challenges. Volume refers to the size of the input data. For example, in social media, the amount of data generated per minute is vast and needs to be processed to make useful analytics. Variety relates to the different nature of data sources. For example, Web click-streams, web logs or social media information. On the other hand, Velocity refers to the speed in which the input data is accessible and the speed in which analysis has to be provided. In practice, these challenges rarely appear in isolation. Vast and heterogeneous data sources produce data continuously. For example, on the web, we must deal with one without neglecting the others.
Stream Reasoning is a research area that focuses on solving these challenges at once in the attempt to answer the following research question [14]:

Can we make sense, in real-time or near real-time, of vast, heterogeneous data streams that originate from different complex domains, in order to be able to perform useful analysis?

This question is still subject of active research and most likely its scope is too broad to find a complete answer, but this allows us to formulate meaningful requirements to take into consideration in the research. We express the problem using the Macro-Meso-Micro framework [13] that defines three level of analysis to formulate the question:

- Macro: tackles the problem from a broader scope. Deals with a large-scale analysis that often results to questions with no realistic answer.
- Meso: the scope of the problem gets narrower, medium-scale analysis with small groups of entities involved.
- Micro: specifically outlines research problems. These problems can be faced and refer to specific parts of entities described at the higher levels.

The general question, main subject of the Stream Reasoning research area, leads to the specifications of some high level requirements:

R.1 The system must be scalable with respect to the Volume of the data.

R.2 The system must deal with heterogeneous data, to cope with data Variety.

R.3 The system must deal with Streaming Data.

R.4 The system must provide output in a timely fashion.

R.5 The system must support reasoning services to extract implicit information.

R.6 The system must deal with complex domain models.

Up to a decade ago, no system was able to fulfill all these requirements as addressing all of them simultaneously has proven to be a tough challenge. In the context of Stream Processing, Data Stream Management Systems (DSMS) and Complex Event Processors (CEP) were designed to target a part of the requirements (Section 2.3). DSMSs allow performing continuous analytical queries on streaming data, e.g. aggregations and statistics calculations. CEPs deal with complex event identification on event streams through event pattern matching. As summarized in Table 3.1, these systems effectively address the Data Volume, Data Velocity and timely output requirements (R1, R3, R4). However, these systems are limited when it comes to applications with heterogeneous data with a complex domain model.

Table 3.1: Relation between systems and requirements

	R1	R2	R3	R4	R5	R6
DSMS/CEP	yes	no	yes	yes	no	no
Semantic Web	yes	yes	no	no	yes	yes

Conversely, Semantic-Web based systems, tried to provide solutions to cover the complementary requirements (R1, R2, R5, R6). The Semantic Web provides the RDF data model (Section 2.1.1) that provides a graphical representation of the data and the corresponding query language SPARQL [18] (Section 2.1.2) in order

to deal with Data Variety (Section 2.1). Additionally, knowledge representation languages aim at annotating the data with its semantic to give a well defined interpretation of the data. Nevertheless, these systems focus on static data and lack support for reactive management of streaming data.
A new research area emerged from complementary needs of Stream Processing and Semantic Web: Stream Reasoning (SR). SR focuses on performing online logical reasoning over streams of data. Stream Reasoners are systems in this field that aim at fulfilling all the aforementioned requirements. In this context we will define our own system.

3.1.1 From SR to GSP and Back

Stream reasoning opens up to new techniques for merging Data Variety and Data Velocity.
It turns out that we can identify two main approaches that state-of-the-art systems make use of:

- Logic based
- Graph oriented

Logic-based solutions enhance the logic with time in order to comprise the timely nature of the data and extend rule-based languages to express reactive reasoning. They are able to deal with dynamic data and support incremental inference. For example to support changes to the input data, RDFox [16] implements a novel Datalog evaluation algorithm that extends the Backward and Forward algorithm. ETALIS [29] follows a completely deductive rule-based paradigm and it defines its own languages to express composition of events. Another significant example is LARS [30], which is a modal logic that represents the whole stream content as a formula and it takes into account time through the definition of specific time operators.
Graph-Oriented systems are based on the RDF data model and extend it to the notion of RDF streams and the continuous semantic. They typically extend SPARQL queries over RDF streams. These systems provide DSMS-like query answering over RDF Streaming Data. Figure 3.1 summarizes the taxonomy indicating some systems that we described in the background. Data Variety encompasses the three types of data: structured, unstructured and semi-structured data. For this reason, the Semantic Web research led to the definition of RDF, i.e, a graph data model that is suitable for data sharing and that fosters interoperability. On the other hand, Graph Processing is a well established area of Computer Science, with many systems and frameworks available to efficiently perform graph-related common problems (Google's Pregel [31], Giraph [32]). Graph Stream Processing is the evolution of Graph Processing that deals with dynamic graphs whose topology changes in time, where millions of edges are added or removed per second [33].

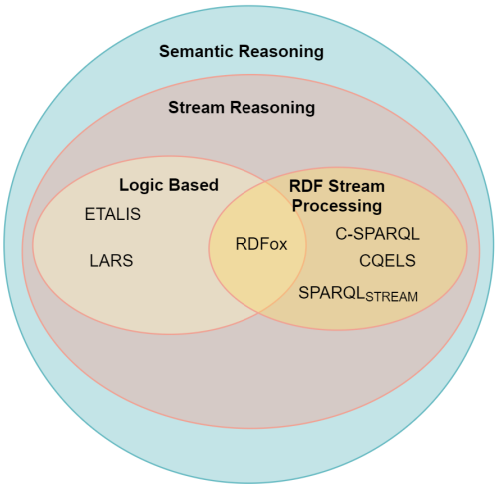


Figure 3.1: Summary of Semantic Reasoning areas that we considered

Many Graph Stream Processing systems have similar applications as systems in Stream Reasoning and in recent years concepts as Online or Temporal Graph Analysis or Update Maintenance on time-evolving graphs have been established. This allows to narrow the research question to the following:

Can Graph Stream Processing be used to address Stream Reasoning goals?

In this context, the requirements we defined in the previous section adapt as follows:

R.1 The system must be scalable with respect to the size of the input graph.

R.2 The Graph representation as defined in the RDF data model allows us to cope with Data Variety.

R.3 The system must support dynamic graph whose edges change in time, and enter the system as Streaming Data.

R.4 The system must provide output maintenance in a timely fashion. As with Stream reasoning, these systems must generate output updates within well-specified latency bounds.

R.5 The system must support languages and approaches expressive enough to perform analytic algorithms.

R.6 The Graph representation encompasses data coming from complex domains.

GraphTau [34] is an example of Graph Stream Processing framework. It is built on top of Apache Spark, a distributed dataflow system. It supports efficient

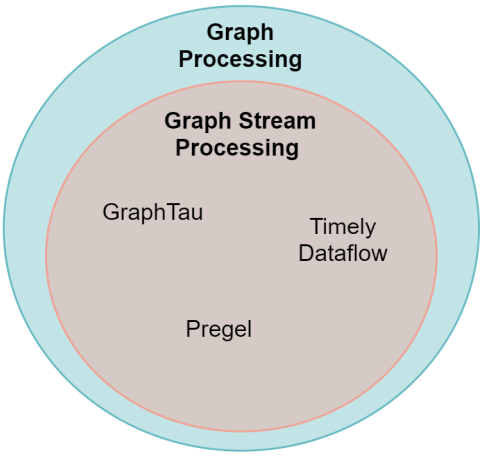


Figure 3.2: Summary of graph processing

streaming computation through the building of graph snapshots as new data occurs in the system. Another very interesting framework well suitable for Graph Stream Processing is Timely Dataflow. This framework is the main tool used in this thesis and it will be thoroughly explained in the next sections.

3.1.2 Micro Level Analysis

Figure 3.1 and Figure 3.2 show the relation of Semantic Reasoning and Graph Processing to the interest of this dissertation work. We showcase the direct relation between the RDF data model and Graph Processing by means of an example. The RDF data model (Section 2.1.1) represents facts and concepts using triples that can be interpreted as directed edges connecting nodes in a graph. Thus, RDF can be represented through a directed graph.
If we consider the examples depicted in Figure 3.3 and Figure 3.4, on the left we have a generic graph, on the right we have a RDF graph representing a hierarchy of Classes. The two graphs are isomorphic as they contain the same number of graph vertices connected in the same way.
The typical Semantic Web scenario of looking for the super-classes of the FullProfessor resource translates into a reachability problem of the corresponding graph. Figure 3.6 and Figure 3.5 show the correspondence between the materialization of the RDF dataset and the transitive closure on the generic graph.
At this level of analysis we look for systems that try to tackle the problem of relating Stream Reasoning to Graph Stream Processing. We then want to build upon this foundation to offer a new approach.
We focus our attention to three systems:

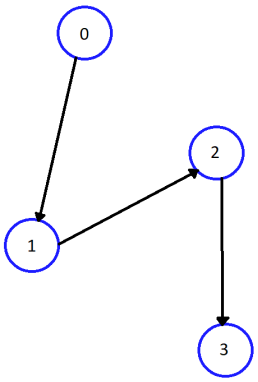


Figure 3.3: Example of a generic graph

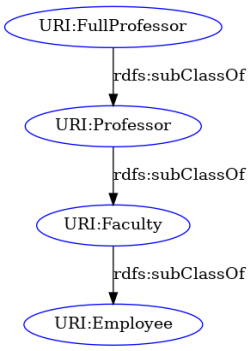


Figure 3.4: Example of a RDF graph

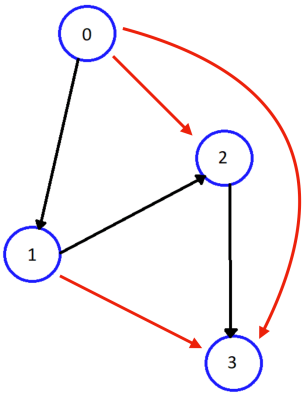


Figure 3.5: Transitive closure on the generic graph

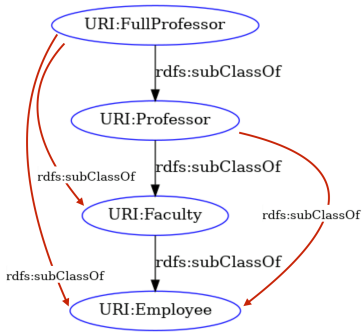


Figure 3.6: RDF graph materialization

- RDFox [16]: a highly scalable, centralized and in memory RDF store with parallel Datalog rule evaluation and SPARQL query answering
- DynamiTE [7] : a Parallel and Distributed System for Incremental Maintenance of Dynamic RDF data.
- IMaRS [17]: Incremental Materialization of RDF Streams

RDFox provides a highly scalable RDF store that supports datalog reasoning using novel datalog evaluation techniques. It implements incremental maintenance using the *backward/forward* algorithm. This algorithms supports early detection of facts that can be rederived using a combination of backward and forward chain implications. It provides a querying package for SPARQL [18] queries. DynamiTE aims at maintaining a dynamic knowledge base in the presence of frequent changes. It uses RDF dataset to store the data and it models changes in the input dataset

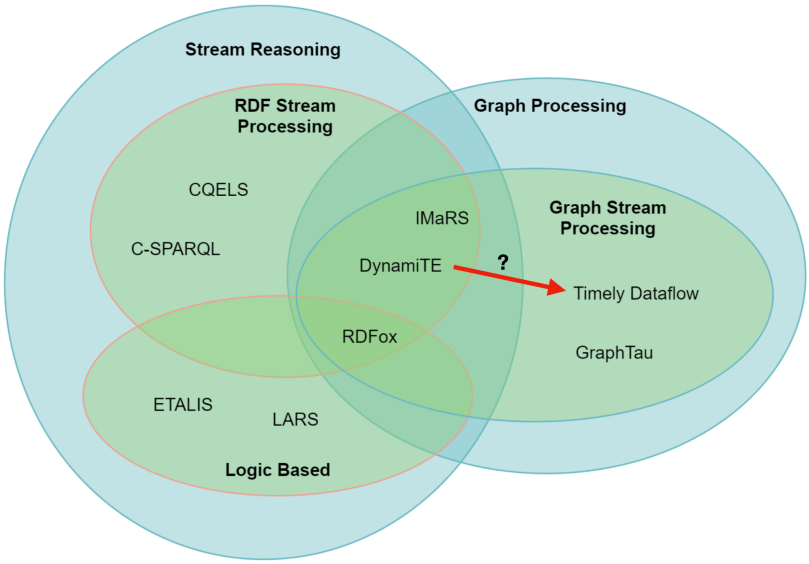


Figure 3.7: Micro Level Analysis deals with specific systems in the areas of interest of this thesis

as triple additions and triple removal. Incremental maintenance for triple addition requires the re-computation of the materialization to add new derivations. Data removal needs more care as it requires the deletion of both explicit and implicit knowledge: the implicit knowledge that is a candidate for deletion might have alternative derivations in the resulting version of the initial data set. DynamiTE [7] implements two incremental algorithms:

- Counting algorithm (Section C.0.3): based on the number of derivations of each RDF triple
- DRed algorithm [24]

The biggest limitation of this system is the fact that the dynamism of the knowledge bases and the assertion compononent is not really a stream of event in its true meaning. Updates are provided as batches of data with no real timestamp specifications.
IMaRS not only considers RDF Streaming data, but it makes use of it to identify early the data that has to be removed and that has no other possible derivation in the resulting data set. For this reason IMaRS is more suitable for a Streaming context. Indeed, IMaRS performs the materialization with respect to a time frame called window, specified by the user of the system. This window allows the computation of the expiration time of each triple entering it. The expiration time indicates when the fact should be safely removed from the materialization. The logic to maintain over time this expiration time, is pushed in the maintenance

algorithm favoring efficient incremental maintenance.
The reason why we are considering specifically these systems is that they offer different points of view of the problem:

- RDFox is aimed at performing parallel generic datalog reasoning and SPARQL query answering.
- DynamiTE focuses on the parallel and distributed computation of the Incremental Maintenance of RDF dataset in the presence of batches of random updates.
- IMaRS focuses more on the management of Streaming Data and how to make sense of their timestamp to optimize the Incremental Maintenance computation.

In this thesis work, we will focus on DynamiTE's approach. In particular, we will develop a system that will implement DynamiTE main functionalities and engineered in a way to be extensible, in order to possibly cover IMaRS and RDFox as a future work.
These three approaches will represent reference approaches in exploring possible answers to the question formulated in the section before: we are trying to understand if we can port these challenges in a purely Graph Stream Processing environment. The framework we are going to focus on is Timely Dataflow [1] [15] and its close relative, Differential Dataflow [35] [2].
Timely Dataflow was born as a low-latency cyclic dataflow computational model used by Naiad [28], a system dedicated to the execution of data parallel computation. This computational model has an extended and more modular implementation in Rust and it is now also provided as a Rust framework. Differential Dataflow is built on top of Timely Dataflow, and it extends Timely Dataflow functionalities offering a more idiomatic way to express computation borrowing idioms from SQL and Streaming applications. In Graph Processing scenarios the framework has provided outstanding results: as we can see in Table 3.2 and in Table 3.3, Differential Dataflow outperforms most of the state-of-the art systems even when run on a low-resource machine. The experiments related to the figures are respectively Graph Reachability on the left, and Connect Components on the right, performed in two different data sets (livejournal, orkut) representing graph with up to 600 million edges.

Timely Dataflow and Differential Dataflow offer features that can be used in the context of Stream Reasoning.

[1] Timely Github repository: `https://github.com/TimelyDataflow/timely-dataflow`
[2] Differential Github repository: `https://github.com/TimelyDataflow/differential-dataflow`

Table 3.2: Differential Dataflow results on the reachability problem

Reachability	Cores	Livejournal	Orkut
GraphX	128	36s	48s
SocialLite	128	52s	67s
Myria	128	5s	6s
BigDatalog	128	17s	20s
Differential	1, 2	**7s, 4s**	**15s,9s**

Table 3.3: Differential Dataflow results on the connected component problem

Connected Comp.	Cores	Livejournal	Orkut
GraphX	128	59s	53s
SocialLite	128	54s	78s
Myria	128	37s	57s
BigDatalog	128	27s	33s
Differential	1, 2	**7s, 4s**	**15s,9s**
update	1,2	**98us, 109us**	**200us, 216us**

This framework's name suggests the two main ideas:

- Dataflow: it is based on dataflow computation, where data availability drives the computation instead of a list of instruction imperatively executed.
- Timely: this framework extends dataflow computation with timestamps that represent logical computation points. This allows an efficient and lightweight coordination mechanism.

The main goal of this framework is to provide high expressive power and high performance. It offers the high throughput of batch processors and the low latency of stream processor. Its expressive power allows the user to perform iterative computation (something that the standard dataflow paradigm does not allow) and incremental computation.
Differential Dataflow focuses on providing fast and efficient incremental maintenance of the dataflow computation and it uses Differential Computation to achieve this. In Table 3.3, measurements for the random update of an edge in the graph are provided. The average time shown in the table is in the order of hundreds of microseconds, with a 20 millisecond worst case scenario. Many other systems would require to re-compute the computation from scratch. The features that we have just described fit perfectly in the context of our research. So, ultimely, we aim at exploring solutions to the following problem:

Can we take advantage of the features offered by Timely/Differential Dataflow to achieve results in Stream Reasoning applications? In particular, can we implement the core logic of DynamiTE through a Timely Dataflow paradigm?

These questions are the main focus of this dissertation. To achieve meaningful results we need to keep in mind the requirements formulated at higher level of analysis. Those will translate as the following:

R.1 The Dataflow computation must scale with the Volume of the input data.

R.2 We must model the data according to the RDF data model to encompass Data Variety.

R.3 Timely/Dataflow stream data structure must provide support for timestamped data.

R.4 Functional dataflow computation must happen within well-specified bounds in terms of throughput and latency, depending on the applications.

R.5 Timely/Differential Dataflow allow iterative computation, this allows us to express reasoning services.

R.6 Dataflow computation must support Complex Domain data to perform reasoning tasks.

Chapter 4

Software Design

In this chapter, we outline the design of the system that we developed, that supports the dataflow computation on RDF data using Timely Dataflow and Differential Dataflow. The system provides highly generic and abstract interfaces allowing the user to use it in a modularized way. This system will be used to perform incremental materialization, so we will describe the design of the Dataflow computation and the logic to implement it. We chose to target DynamiTE as a starting reference system because it focuses on the reasoning procedures and on the incremental maintenance of the materialization.

4.1 Design Choices

With the background information we provided so far we are able to fully collocate our application scenario. We can now list some design choices to set some boundaries that we are going to need in the implementation when approaching DynamiTE through a Timely Dataflow paradigm.

First of all, the reasoning that is performed by DynamiTE is related to the ρDF ontology language, the minimal RDFS fragment that we introduced in Section 2.1.3. The Dataflow computation that we provide is restricted to this specific ontology language, as it takes advantage of its structural properties. The Rules are shown in Table 4.1. The input RDF Dataset is represented as a set of Datalog facts $T_e(s,p,o)$. T_e represents *edb* predicates and s, p, o are literals for the subject, the predicate and the object of the RDF triple. The implicit rule is needed to make the edb predicates T_e available to to the idb predicates T_i used in other rules. The constants are represented in Table 4.2, that also shows what they stand for.

4.1.1 Encoding Requirements

Our system provides support for user defined Encoding. This operation has requirements that will constrain the development of our system. Encoding RDF

Table 4.1: ρDF Reasoning rules in Datalog

	Head	**Body**
ρDF Rules	T_i(A, SPO, C)	$\leftarrow$ T_i(A, SPO, B),T_i(B, SPO, C)
	T_i(A, P, B)	$\leftarrow$ T_i(Q, SPO, P),T_i(A, Q, B)
	T_i(A, TYPE, C)	$\leftarrow$ T_i(B, SCO, C),T_i(A, TYPE, B)
	T_i(A, SCO, C)	$\leftarrow$ T_i(A, SCO, B),T_i(B, SCO, C)
	T_i(A, TYPE, D)	$\leftarrow$ T_i(P, DOMAIN, D),T_i(A, P, B)
	T_i(A, TYPE, R)	$\leftarrow$ T_i(P, RANGE, R),T_i(B, P, A)
Implicit Rule	T_i(A, P, B)	$\leftarrow$ T_e(A, SPO, B)

Table 4.2: List of abbreviations used in the ρDF inference rules

Abbreviaton	**RDFS Property**
SCO	*rdfs:subClassOf*
SPO	*rdfs:subPropertyOf*
TYPE	*rdf:type*
RANGE	*rdfs:range*
DOMAIN	*rdfs:domain*

Terms from strings to a numeric type is generally a requirement for Stream Reasoning applications, as the String datatype may be too heavy in a context where Data Velocity matters. The encoding function must be a *bijective function* with the set of RDF Terms as domain and the numeric type as codomain, i.e. a function that maps one RDF Term to one and only one number that belongs to the codomain. Bijective functions are reversible functions, this allows the definition of two basic operations required by any encoding function:

- *locate()*: takes as input an RDF Term and returns the encoded identifier of the numeric type.
- *extract()*: takes as input the encoded identifier and returns the corresponding RDF Term.

To reduce the complexity of the work, we make simplifying assumptions that will target the focus at this stage of the work. At this stage we aim at exploring the different challenges brought by introducing new paradigms in the Stream Reasoning context.

A.1 At this stage of the research we do not plan yet a support to SPARQL query answering, as we want to focus on the problem of the incremental maintenance first. For this reason we are only going to use plain files to store the materialization and the incremental updates.

A.2 We assume that the data can fit in memory. We will load all the knowledge base in memory from files and perfom incremental materialization in-memory.

A.3 We assume all the data is provided in the N-Triples format. At this stage, the system will be able to handle triples according to that serialization technique.

A.4 No support is provided yet to an automatic Datalog program translation into the correspondent dataflow computation. We assume the user provides the datalog program as a timely dataflow computation.

4.1.2 System Design Goals

The main goals of the design of the system are the following:

G.1 Using a low level system programming language to favor performances without giving up the advantages of abstraction-based programming environments.

G.2 Hide all implementation details when integrating Timely Dataflow to perform incremental materialization on RDF data: ideally the user of the system should not know the details of Timely and Differential Dataflow to use it.

G.3 Support to generic input data: this allows the user to define the type of data to run the computation on, e.g. RDF triples, timestamped RDF Triples.

G.4 High modularization: we want the user to define externally the parsing procedure, encoding and materialization logic and feed it into the system that in turn will streamline these operations.

These goals will orient both the design of the software and the implementation. We will try to motivate how we think we meet these goals in the following subjects.

4.2 System Design

In this section, we will be describing the main components and the workflow of the system. Although the encoding of the data is not the central topic of this thesis to maintain high modularization (G.4) we want the system to support encoding and to be extensible to a user defined encoding procedure.

4.2.1 System Components

To meet G.4 and G.1 design goals, we need to decouple the Timely Dataflow implementation details from the materialization logic and the encoding scheme. For this reason, we identify five main external components that the user will need to provide to the system:

- *Materialization Logic.* The dataflow logic to produce the materialization of the RDF Dataset.
- *Encoding Logic.* Represents the function that maps an RDF Term to a numeric type. As per encoding requirements, (Section 4.1.1) this function must be bijective.
- *Parsing Logic.* Defines the parsing function. RDF terms are read as strings. This function maps this strings to an intermediate representation (if needed) before applying the encoding logic.
- *Bijective Map.* This is the type of the dictionary-like data structure that keeps track of the mapping provided by the Encoding Logic. As per encoding requirements, (Section 4.1.1) this data structure needs two provide the *locate()* and *extract()* operations.
- *Encoded Dataset Type.* This represents the type of the Encoded Dataset. Generally we expect this to be a Vector, but to favor extensibility and modularization defer the definition to the user.

As it can be seen in the component diagram in Figure 4.1, the system (called here Reasoning Service Package) is made of three main components:

- *DataStrFactory*: this component requires the definition of the two data structures: the Bijective Map and the Encoded Dataset type. The purpose of this component is to provide the functionalities of creating the map and parse and encode the dataset according to that map and the parsing function and return the encoded dataset. The parsing and encoding function at this component are defined as generic and will be specialized by the *Encoder* component.
- *Encoder*: this component glues the Parsing Logic, Encoding Logic and the DataStrFactory together. In particular, we must avoid to have the same dictionary to represents two different encoding or parsing functions (this could break the encoding requirements we defined in Section 4.1.1 that requires the encoding function to be bijective). For this reason, this encoder requires the implementation of external interfaces of Parsing and Encoding Logic by the user. This component uses the *DataStrFactory* component to provide the functionalities of encoding and parsing by specializing the Parsing and Encoding Logic to what the user specified.

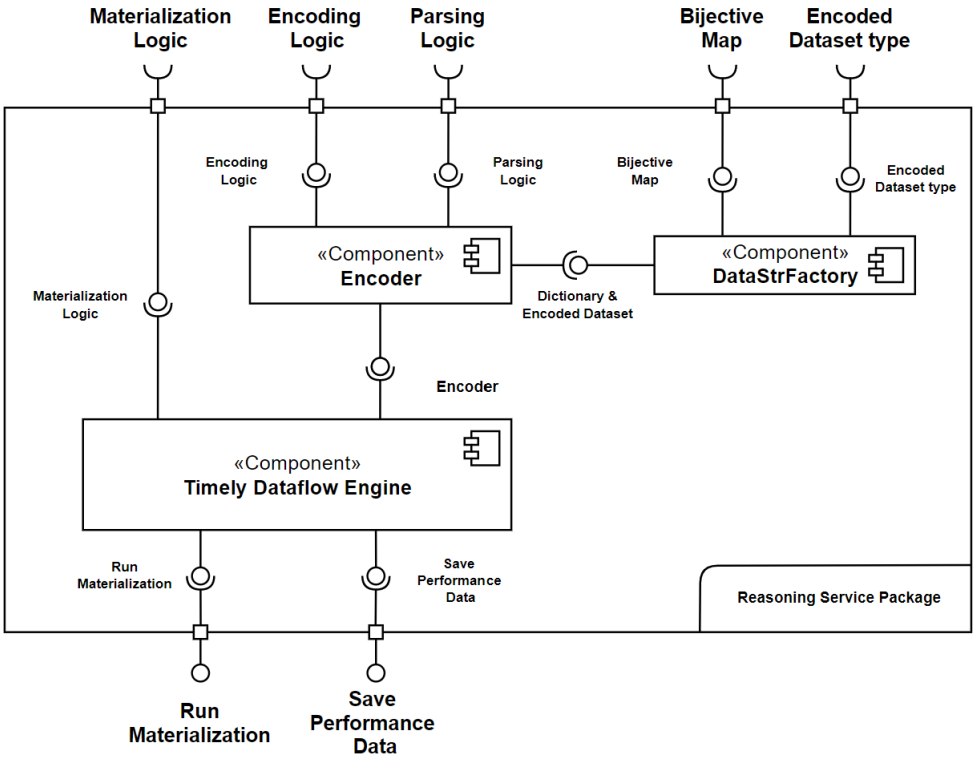


Figure 4.1: Component diagram showing main parts of the system

- *Timely Dataflow Engine*: this component represents the core of the computation. It requires the definition of the Materialization Logic, that is the Dataflow computation corresponding to the type of reasoning the user wants to perform on the RDF dataset. This component uses the *Encoder* output (encoded triples) to load the dataflow and perform the reasoning specified by the Materialization logic. After that, it decodes the materialization using the *Encoder* (that has the ability of decoding thanks to Bijective Map of the *DataStrFactory* component, as per encoding requirement) and provides in output the materialized data set. On top of this, it provides performance statistics relative to all the major steps of the system's workflow.

This design allows us to make some considerations on the goals we specified in Section 4.1.2:

G.2 The design choices we made at component level suggest that we meet G.2 as the component *Timely Dataflow Engine* hides the low level details to pay attention to when integrating the timely dataflow framework. This allows the user to ignore most of the these details (we will see in the next chapter what details at the current stage of the system are exposed).

G.4 All components require external definitions that coincide with the main modules of our application. What is important, is that these definitions happen external to the system. The user defines independently the implementation of the parsing and encoding functions along with the data structure that represents the bijective mapping and the encoded dataset and only then feed these into the system. This allows re-usability of the components without rewriting any code. For this reason we are confident we meet goal G.4.

4.2.2 Incremental Maintenance

In this section, we discuss how we rely on Differential on Differential Dataflow for computing the incremental maintenance of the materialization. Differential Dataflow offers transparent and automatic incremental maintenance based on the same data flow implemented for the full materialization. According to the Differential Dataflow specifications, in order to propagate the input changes to the output there is no need to modify the implementation of the dataflow. We will test this, believing that this can be a worthy starting point for more sophisticated explicit incremental maintenance using the Timely/Differential Dataflow paradigm. This is performed thanks to the internal implementation of the dataflow operators to compute on differences of collections in input to an operator rather than the whole collection. Differential Computation is the technique that is employed in this scenario.

Differential Computation

Given a collection trace A_0, A_1, ..., A_t in order to compute $Op(A_0)$, $Op(A_1)$, ..., $Op(A_t)$ we can apply the operator to every A_t independently of the *version* (i.e., the subscript index of the collection) it has. If successive versions contain common records, an incremental evaluation would be more efficient. We can define a *difference trace* that contains for each version t:

$$\delta A_t = \begin{cases} A_0 & \mathrm{t} = 0 \quad (4.1) \\ A_\mathrm{t} - A_{\mathrm{t\text{-}1}} & \mathrm{t} > 0 \quad (4.2) \end{cases}$$

With this definition, we can compute the 'current' version of the collection as:

$$A_t = \sum_{s \leq t} \delta A_s$$

An operator can compute the output difference, δB_t given the input difference δA_t. Figure 4.2 shows that incremental computation uses differences of collections as sequences. Incremental Systems generally compute:

$$\delta B_t = Op(A_{t-1} + \delta A_t) - Op(A_{t-1})$$

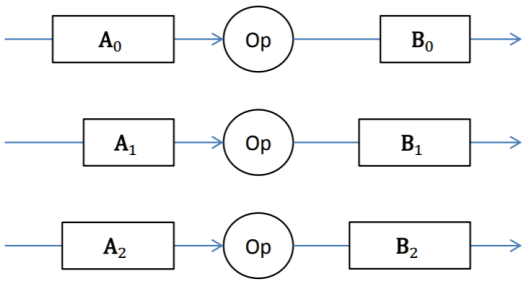


Figure 4.2: Incremental computation computes the differences of the output based on the differences on the input [6]

.

At this point, an incremental system would incorporate the δA$_t$ and δB$_t$ respectively in their collections and discard them as they are not used. Incremental computation can be performed over cyclic dataflow graphs, where the output gets fed back to the input until the fix point is reached. This systems have the limitation that differences can only be used to update the input collections or to perform iteration but not both. Indeed, the incremental computations require the versions to be *totally ordered*, i.e. different versions of collections are considered as sequences. Differential Computation extends this limit by allowing multiple predecessors versions, hence the versions may be *partially* ordered. Since A_t might not have a single well-defined predecessor A_{t-1} we have that:

$$\delta A_t = A_t - \sum_{s<t} \delta A_s$$

where s and t range over elements of a partial order and $<$ is the less-than relation. So we have that:

$$\delta B_t = Op(\sum_{s \leq t} \delta A_s) - \sum_{s<t} \delta B_s$$

The biggest consequence to the partial ordering is that now the operator is able to use both input collection differences and iteration differences to perform incremental maintenance over cyclic dataflow graphs. In fact, if we suppose we have a collection A$_{ij}$ that has different values for each round i and interation j of a loop containing it. We consider the partial order:

$$(i_1, j_1) \leq (i_2, j_2)) \text{ iff } i_1 \leq i_2 \text{ and } j_1 \leq j_2$$

Figure 4.3 shows how differential computation uses the partial order to compute the differences. In this context, δA$_{11}$ is computed is computed as the following:

$$\delta A_{11} - (\delta A_{00} + \delta A_{01} \delta A_{10})$$

. This means that it is considering:

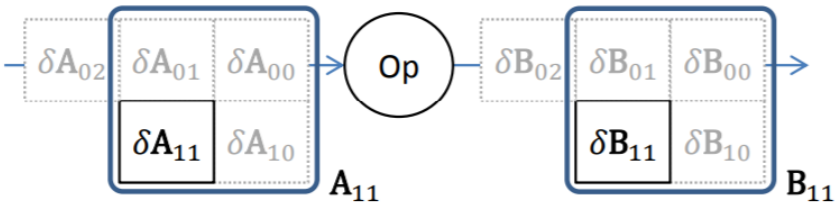


Figure 4.3: Multiple independent collections B_{ij} are computed using differential compuation and the rounded boxes are the differences that are considered when forming the collection A_{11} [6]

.

- δA_{00}: the initial collection A_{00}
- δA_{01}: A_{00} advances to the second iteration.
- δA_{10}: the collection A_{00} advances to the second version A_{10}

This would not be possible in the previous scenario where version of collections were *totally* ordered. Differential computation, unlike standard incremental computation, stores each difference δA_{ij}. This allows the difference δA_{ij} to be contained making up for great performance gain.

4.2.3 System Workflow

Consistent with DynamiTE (Appendix C), our system main functionalities are the following:

- Perform Full Materialization of input Dataset.
- Perform Incremental Materialization of data to be inserted (if any).
- Perform Incremental Materialization of data to be deleted (if any).

Figure 4.4 shows our system workflow. The input RDF Dataset and any available updates are provided into the system with the N-Triples (Section 2.1.1) format as a plain *.nt* file. The system will scan the lines of each file and parse the Strings into a user-defined intermediate representation (e.g., a RDF Term ADT). Then, it will use the encoding components to turn these parsed triples into encoded identifiers according to the logic the user of the system supplies. The system saves the encoding into the file system as a plain *.nt*. After that, it sets up the timely dataflow computation preparing the stream of data and the different computational nodes of the dataflow. The data is loaded into these streams and the full materialization is computed. The result is decoded using the dictionary map generated in the encoding phase and it is subsequently saved on a file as a *.nt*.

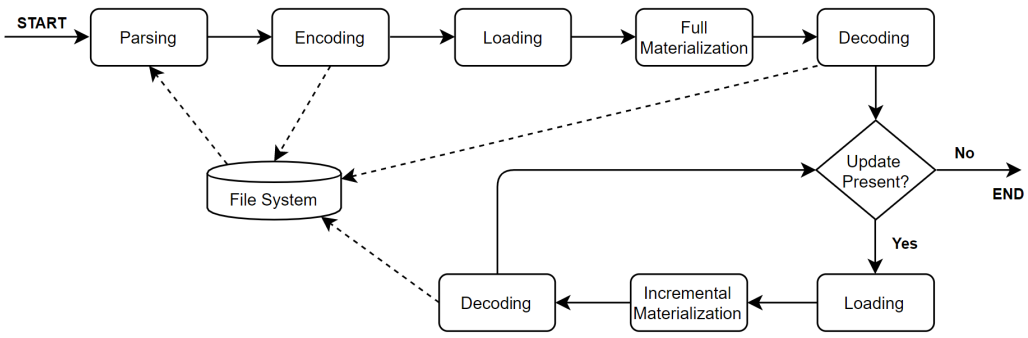


Figure 4.4: System workflow, with interaction with file system.

If there is no update pending, the system exits, otherwise it loads the data of the updates into the dataflow, and performs Incremental Materialization on the fully materialized dataset according to Differential Computation. It decodes the result and saves it persistently on a plain *.nt* file. Then in case there are more updates it repeats the same procedure until all updates are applied. On top of this, at the end each Loading phase, Full/Incremental Materialization phase and Decoding to file phase, the system records latency statistics useful for performance analysis.

4.3 Dataflow Analysis for ρDF

The Reasoning Service Package allows the user to implement externally as a required interface the Materialization Logic. In the next chapter, we will test the system we designed to perform Reasoning on LUBM data according to the *univ-benchmark*. In particular we will compute the materialization using ρDF entailment (Table 4.1). The implementation of the Materialization has to adapt to a Timely/Differential Dataflow context. The Full Materialization is the result of the fix-point computation of the rules in Table 4.1. For the evaluation of the ρDF inference rules we can make one observation. We can break down the fix-point evaluation of all the rules at the same time into separated fix-point evaluation of single rules. To get a correct result using this approach, we must respect an order between the rule evaluation. In fact, if we enumerate the rules as shown in Table 4.3, in order to assure a correct computation of the full materialization, we need Rule 1 to evaluate and reach a fix point before Rule 2. This because Rule 2 has a literal in the body that is instantiated by triples that are generated from the evaluation of the Rule 1.
Following this reasoning, we can define a relation between rules that we call *rule precedence* "$\rightarrow$":

$$\forall m, n \in [1, ..., 6], R_m \rightarrow R_n \Leftrightarrow \exists l \in body(R_n) : head(R_m) \equiv l$$

Where $body(R_n)$ represents the set of literals in the body of rule R_n while

Table 4.3: ρDF Reasoning rules enumerated

Rule #	**Head**	**Body**
Rule 1	T_i(A, SPO, C)	$\leftarrow$ T_i(A, SPO, B),T_i(B, SPO, C)
Rule 2	T_i(A, P, B)	$\leftarrow$ T_i(Q, SPO, P),T_i(A, Q, B)
Rule 3	T_i(A, TYPE, C)	$\leftarrow$ T_i(B, SCO, C),T_i(A, TYPE, B)
Rule 4	T_i(A, SCO, C)	$\leftarrow$ T_i(A, SCO, B),T_i(B, SCO, C)
Rule 5	T_i(A, TYPE, D)	$\leftarrow$ T_i(P, DOMAIN, D),T_i(A, P, B)
Rule 6	T_i(A, TYPE, R)	$\leftarrow$ T_i(P, RANGE, R),T_i(B, P, A)

$head(R_m)$ represents the only literal of the head. We say that R_m *precedes* R_n if $R_m \rightarrow R_n$. Note that the $\equiv$ symbol represents equivalence with variable renaming. For example: T_i(A, SPO, C) of Rule 1 is equivalent to T_i(A, SPO, C) of Rule 2, renaming A to Q and C to P. Therefore Rule 1 *precedes* Rule 2.
We can now build a graph representing the *rule precedence* relation among the inference rules of ρDF obtaing the graph in Figure 4.5, where a node R_m is connected to R_n if R_m precedes R_n This graph suggests us two things:

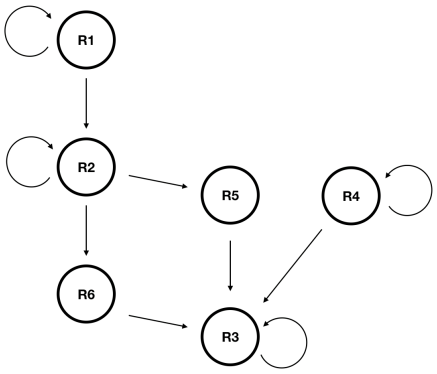


Figure 4.5: *rule precedence* graph on ρDF inference rules

- Nodes with self-loop edges require the iterative evaluation up to fix point. In fact a self loop node means that a rule generates facts that may be use in the body of the fact itself.

- We can see that without considering self-loops, the graph is a Directed Acyclic Graph, so this allows us to find a topological order of the graph, e.g. *R4 - R1 - R2 - R5 - R6 - R3*. This order of rule evaluation gives us a guarantee of the correction of the full materialization.

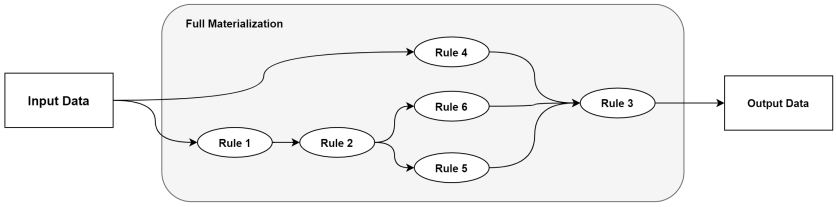


Figure 4.6: Full Materialization high-level dataflow

These two aspects translate in our scenario in the following design specification: Rules for which the corresponding node in the *rule precedence* graph has a self loop require an iterative dataflow that calculates the fix point on database J, moreover, the full materialization has to be computed following a topological order of the *rule precedence* graph, in order for the materialization to be correct: if rule R_m *precedes* R_n then R_n has to be evaluated on the result of the materialization of J with respect to R_m.
It is important to notice the fact that this procedure works *ad hoc* for ρDF reasoning and it would hardly apply to a more generic case where the *rule precedence* graph contains cycles.

4.3.1 Dataflow Design

Based on the *rule precedence graph* in Figure 4.5, we can divide the computation of the materialization as the fix-point computation of the rules taken separately. We can distinguish three groups of rules that share the implementation:

1. Rule 1 and Rule 4: both contain the same schema triple (a triple that has a keyword of Table 4.2 as property, as opposed to a generic triple) both in the Head and the Body of the rule.

2. Rule 5 and 6: the only non recursive rules using respectively *rdfs:domain* and *rdfs:range*.

3. Rule 2 and Rule 3: both generate generic triples, using one generic triple combined with a schema triple

Following a topological order over the rule precedence graph we obtain the high level dataflow of the full materialization, depicted in Figure 4.6. The separation among nodes in this picture is purely logical: Timely Dataflow implements trasversal parallelism, distributing the dataflow among all the workers. Each worker will implement all the nodes in Figure 4.6. Nevertheless, the separation is still valid as the data in input to each different rule represented in the figure respects the partial order of the *rule precedence graph*. Each node inside the dataflow is itself a dataflow and represents the computation related to the evaluation of the

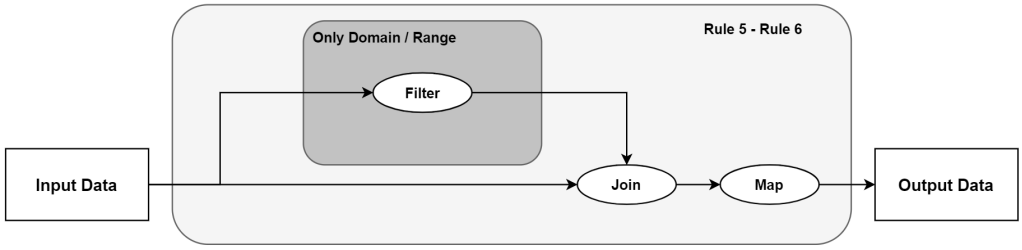


Figure 4.7: Rule 5 and Rule 6 evaluation dataflow

corresponding rule. The dataflow associated to the rule depends on the groups aforementioned.

Rule 5 and Rule 6

We recall:

- Rule 5: *T(A, TYPE, D)* ← *T(P, DOMAIN, D),T(A, P, B)*
- Rule 6: *T(A, TYPE, R)* ← *T(P, RANGE, R),T(B, P, A)*

Except for the predicate used by the first rule of the Body (respectively DOMAIN, RANGE), their dataflow coincides and it is shown in Figure 4.7. We identify the following operators:

- Filter: selects only the triples with respectively *rdfs:domain* / *rdfs:range* as triple predicate. This triples correspond to the first literal of the body of the rules.
- Join: joins the triples in the previous point with the original input collection to obtain the derivations. These derivations do not contribute to produce new facts within the same rule. For this reason the rule does not need to be iterative.
- Map: this operator is needed to select the correct variable bindings to generate a fact according to the Head of the rule.

As an example, suppose we have the Knowledge Base in Listing 4.1

```
1 teaches, rdfs:domain, Person,
2 Alice, teaches, ComputerScience
```

Listing 4.1: Rule 5 and 6 example KB

The *only_domain* subdiagram corresponds to the literal $T(P, DOMAIN, D)$ in the body of the fifth rule. This represents a new stream that only contains the triples that have domain as a predicate. The join operator joins this stream with the original input data on the subject of the first stream and the predicate of the second stream. The *Map* operator is required to select the right binding of the variables with respect to the head of the rule. So the output data will contain the triple shown in Listing 4.2

```
1 Alice, rdf:type, Person
```

Listing 4.2: Rule 5 and 6 derived triple

Rule 1 and Rule 4

We recall:

- Rule 1: *T(A, SPO, C)* ← *T(A, SPO, B),T(B, SPO, C)*
- Rule 4: *T(A, SCO, C)* ← *T(A, SCO, B),T(B, SCO, C)*

Except for the predicate used (respectively, SPO and SCO) their dataflow coincides. Figure 4.8 shows the dataflow. We identify the following operators:

- Filter: filters the input data so that only schema triples with predicate *rdfs:subClassOf* / *rdfs:subPropertyOf* are considered. This narrows the whole data input to only the input triples that are going to be considered by the body of the rule.
- Join: input triples are joined with themselves through the definition of a loop variable. This variable will contain the output of the computation at each iteration so that it will be joined with the input data to obtain all the possible derivations until a fixed point is reached.
- Map: this operator is needed to select the right bindings for the production of the head of the rule.
- Concatenate: concatenates the triples produced at previous iterations to allow to return the full computation of the materialization.
- Distinct: this operator will remove duplicates from the collection. This is required to reach the fixed-point, since otherwise the number of triples in the collection would grow indefinitely.

The In and Ex nodes in the figure are fictitious operators that represent respectively, the entering and the exit of a record from the loop context. To show how the data flows into the dataflow we can consider the dataset in Listing 4.3:

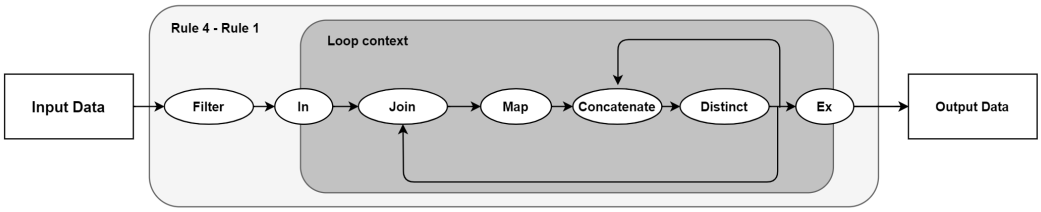


Figure 4.8: Rule 1 and Rule 4 evaluation dataflow

```
FullProfessor, rdfs:subClassOf, Professor,
Professor, rdfs:subClassOf, Faculty,
Faculty, rdfs:subClassOf, Employee
```

Listing 4.3: Rule 1 and 4 example KB

For each iteration the rule produces i-hop derivation for the transitivity property of the *subPropertyOf* and *subClassOf* predicates: the data first gets filtered to consider only triples of predicates respectively *rdfs:subPropertyOf* and *rdfs:subClassOf* resulting in another stream containing all the aforementioned triples. This stream gets joined with itself and then mapped for correct variable binding. At this iteration the join and the map operator will produce the triples shown in Listing 4.4

```
FullProfessor, rdfs:subClassOf, Faculty,
Professor, rdfs:subClassOf, Employee,
```

Listing 4.4: Rule 1 and 4 first iteration derived triples

The Concatenate operator has no effect as there is no rule generated at the iteration before the first. The distinct has no effect on the collection either. This result gets fed back into the loop the second iteration starts. These newly generated triples get joined with the input dataset so the last possible derivation is computed.

```
FullProfessor, rdfs:subClassOf Employee
```

Listing 4.5: Rule 1 and 4 second iteration derived triple

This result gets concatenated with the result at the first iteration, so that we can gather all the results, that are fed back into the loop. This time, the third iteration will clearly not generate anything new. In fact, the *Join* will produce an empty collection. The Concatenate is important so that the previous result (that happens to be the materialization with respect to rule 5) gets included and the system is able that the fix-point has been reached.

Rule 2 and Rule 3

We recall:

- Rule 2: $T(A, P, B) \leftarrow T(Q, SPO, P), T(A, Q, B)$
- Rule 3: $T(A, TYPE, C) \leftarrow T(B, SCO, C), T(A, TYPE, B)$

Even though the rules look different their dataflow design is very similar.

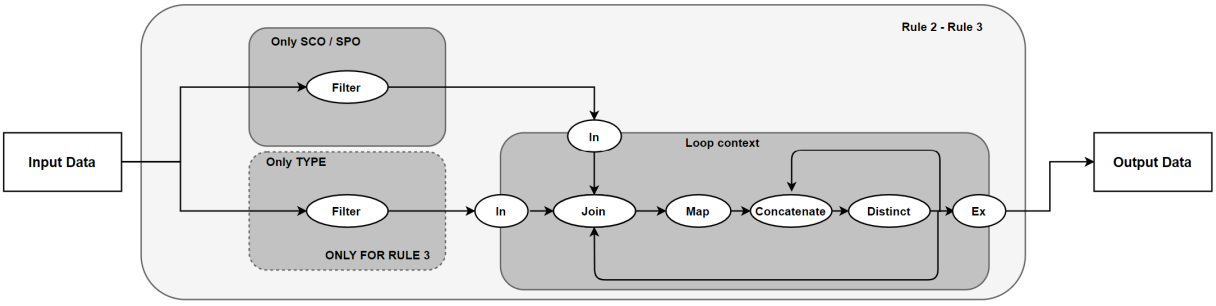


Figure 4.9: Rule 2 and Rule 3 evaluation dataflow

We identify the following operators for the correct execution of the computation:

- Filter: this operators selects only the schema triples with predicate *rdfs:subPropertyOf* / *rdfs:subClassOf* that correspond to the first literal of the Body. For rule 3 the second literal of the Body of the rule contains all triples that contain the *rdf:type* predicate, so for this reason Rule 3 performs another filter operator to select those triples.
- Join: the join operator joins the two literals. Since the rules derive triples that might be joined to derive further facts, the join operator includes the loop variable that contains the triples derived in the previous iteration.
- Map: performs the right variable bindings to produce the right facts.
- Concatenate: it concatenates facts derived in previous iterations, so that newly derived triples can be joined with them to produce all possible derivations.
- Distinct: this operator eliminates duplicates to guarantee termination according to the monotonicity of the *Immediate Consequence Operator* defined in Appendix B.

In line with the previous cases, if we consider Rule 3 and assume we have the dataset shown in Listing 4.6:

```
1 Alice, rdf:type, FullProfessor,
2 FullProfessor, rdfs:subClassOf, Professor
3 Professor, rdfs:subClassOf, Person
```

Listing 4.6: Rule 2 and 3 example KB

We consider Rule 3 as it is more generic. The collection gets first split into two collections *only_sco* and *only_type*. These represent the two literals in the body of the rule. After this the iterative computation starts. In the first iteration the *only_type* collection gets joined with the *only_sco* and than mapped to retrieve the triple specified by the head of the rule. The *Concatenate* and *Distinct* operators have no effect as this is the first iteration. The result of the iteration is show in Listing 4.7.

```
1 Alice, rdf:type, Professor
```

Listing 4.7: Rule 2 and 3 first iteration derived triple

This new triple gets fed back in the loop and it gets joined with the *only_sco* collection so that through the map operator we are able to produce the triple shown in Listing 4.8.

```
1 Alice, rdf:type, Person
```

Listing 4.8: Rule 2 and 3 second iteration derived triple

This gets concatenated with the previous result so we can return all the triples that may produce new derivations in the next iterations. This result is fed back into the loop and the *Join* operator will not produce any new derivations. The *Concatenate* will restore the result at the previous iteration so that the system will be able to understand that the fix point has been reached and the iterative computation can end.

Chapter 5

System Implementation

In this section we describe how we implement the components designed in Section 4 and we show how these interoperate to perform the functionalities. The code can be found at the following link:
`https://github.com/JimmyHypi/Stream-Reasoning-with-Differential-Dataflow`.

5.1 System External Interfaces Implementation

Parsing Logic

To represent the Parsing Logic we use a trait with an Associated Type:

```
pub trait ParsingLogic<T>: Send + Sync {
    type TripleType: Triple<T>;

    fn parse_triple(&mut self, input: &str) -> Self::TripleType;
}
```

Listing 5.1: Parser Trait

The parsing trait requires the definition of the *TripleType* Associated Type. This type is bound to the *Triple*$< T >$ trait. This requires *TripleType* to have a subject, a property and an object of type T. T is the generic parameter that represents the intermediate representation of the RDF Term before being encoded. The trait requires the implementation of the *parse_triple()* function that takes as input a single N-Triple line read as a *&str* and returns the *TripleType* defined be the Associated Type.

Encoding Logic

The Encoding Logic interface is a trait that requires the implementation of the *encode()* function. The contract of this function is that the provided function has to be bijective, as per encoding requirements defined in Section 4.1.1. It is worth noticing that the trait is generic with respect to two generic parameters:

```rust
pub trait EncodingLogic<K, V>: Send + Sync {
    fn encode(&mut self, string: K) -> V;
}
```

Listing 5.2: Encoding Logic Trait

- K is the domain of the encoding function. The type of the input RDF Term to be encoded, input type of the *encode()* function required by the implementors of the trait.
- V is the codomain of the encoding function. It is the encoded identifier associated to the RDF Term, output of the *encode()* function.

Moreover, the *encode()* function takes as input a mutable reference to *self*. In Rust, that is the way to say that the function is a *method* and needs to be called on an instance (Appendix A). This instance is accessed mutably so it changes the state. This is important because it will constitute a key aspect when analyzing performances.

BijectiveMap

The Bijective Map represents the data structure that keeps track of the encoded RDF Terms. It is generic with respect to the same parameters defined for the encoding logic, K and V. It provides the following functionalities:

- *locate()* and *extract()*, as per requirements of the Encoding Logic defined in Section 4.1.1.
- *insert()*: inserts a pair (K, V) inside the map, mutating its inner state.
- *translate()* used for decoding the Encoded Dataset made of triples of type V (codomain), returning triples of type K (domain).

```rust
pub trait BiMapTrait<K, V>
{
    fn locate(&self, left: &K) -> Option<&V>;

    fn extract(&self, right: &V) -> Option<&K>;

    fn insert(&mut self, left: K, right: V);

    fn translate<E>(&self, encoded_dataset: E::EncodedDataSet)
      -> DecodedTriples<&K>
    { /* ... */ }
}
```

Listing 5.3: Bijective Map Trait

Encoded Dataset Type

At this stage, the only requirement we have on the Encoded Dataset Type is that it has to implement the *IntoIterator* trait and it will be defined by the *DataStrFactory* component interface that uses the interface. When a type implements the *IntoIterator* trait in Rust it means that the type can be turned into an Iterator that yields the element of itself. We can consider this Encoded Dataset Type for now as the following type alias:

```
type EncodedDataset<V> = Vector<Triple<V>>;
```

Listing 5.4: Encoded Dataset Type

This type is generic with respect to the type V the represents the codomain of the encoding function.

Materialization Logic

The Materialization Logic is provided into the system as closure. The way to represent that in Rust is by constraining a generic parameter to the *Fn* trait. This is declared by the user of this interface, the *Timely Dataflow Engine*. We show here the trait bind:

```
M: Fn(
            &Collection<
            /*..*/,<E::EncodedDataSet as IntoIterator>::Item>,
            &mut ProbeHandle<usize>
      ) -> TraceAgent</*...*/>
```

Listing 5.5: Materialization Logic trait binds

In Section 4.2.1, we motivate how the design helps hiding almost all the implementation details of Timely and Differential Dataflow to the external user. Here is the point where we find ourselves leaking some of the cumbersome details of the frameworks. M is bound to the *Fn* trait. This trait allows you to define what the parameters and the returning type of the closure are. Input parameters:

- Reference to an object of type *Collection*, which is the fundamental type of a Differential Dataflow computation. A Collection is defined by a few generic parameters that for notation simplicity we omit in Listing 5.5, e.g., the Scope of the Dataflow, the type of the Timestamp. The other type that is required to define when using a Collection is the type of Data inside the collection. We can get this type thorough a chain of **::** notation: Associated Types can be "generically" referenced via this notation. At first, we see *E::EncodedDataset*. This means that the generic type E must have an Associated Type *EncodedDataset*. The EncodedDataSet must implement the *IntoIterator* trait (Section 5.1). In turn, this trait defines an associated type that represents the type of the data yielded by the Iterator. We reference that type using another **::** notation. For us, this type is going to be Triple<V>. So that that input parameter represents a Collection of data of type Triple<V>, that is a collection of encoded triples.

- The *ProbeHandle* is a data structure defined in Timely Dataflow that allows the user to probe the dataflow at different points of the computation. This component

is fundamental to make progress, and can be used by the client in the definition of the dataflow. It is defined in relation to the timestamp generic type parameter.

The output is an object of type *TraceAgent*, a data structure defined in Timely Dataflow that contains the progress of the computation at different times. This data structure is used by the system to decode and output the result of the materialization to file.

5.2 System Components Implementation

In this section, we show the implementation of the different components that make use of those interfaces:

5.2.1 DataStrFactory

The main purpose of this component is to bind the Bijective Map representing the encoding function with the Encoded Dataset. This component will provide a generic interface that will perform parsing and encoding as well as storing the encoding mapping in the Dictionary Structure and form the corresponding encoded dataset. For this reason we define the *DataStrFactory* component as:

```
pub trait DataStrFactory<K, V>
{
    type MapStructure: BiMapTrait<K, V> + Send + Sync;
    type EncodedDataSet: IntoIterator;
    /* .. */
}
```

Listing 5.6: DataStrFactory Associated Types

The MapStructure and the EncodedDataSet are coupled together by being both Associated Types of the same trait. The two Associated Types are bound to *BiMapTrait* and *EncodedDataSet* that are the externally defined interfaces. This trait provides the basics functions shown in Listing 5.7.

```rust
pub trait DataStrFactory<K, V>
{
    fn parse<P: ParsingLogic<K>, W: Path>(
        file_name: W,
        parser: &mut P,
    ) -> Vec<P::TripleType>;

    fn load_from_parser_output<F, P>(
        parsed_triples: Vec<P::TripleType>,
        encoding_logic: &mut F,
    ) -> (Self::MapStructure, Self::EncodedDataSet)
    where
        F: EncodingLogic<K, V>,
        P: ParserTrait<K>;
}
```

Listing 5.7: DataStrFactory provided functions

- The *parse()* function is generic with respect the *ParsingLogic*. Takes as input the path of the file of the RDF Dataset to parse and encode. This function accesses the file, feeding all the read lines to the parser object in input of type *Parsing Logic*. This will parse one N-Triple read as string, into an element of the generic type K. The function will store these results on a vector and return it in output.
- *load_from_parser_output()* takes as input the parsed triples that the *parse()* function produces. The triples in the vector will be encoded using the encoding logic given in input. The function will save the mapping and build the encoded dataset and return both structures to the caller.

It's important to notice that the parsing and encoding logic are given as parameters. Although this favors abstraction, This makes it possible to use different logics with the same structures. However, we must avoid such a behavior, in order to preserve the integrity of the encoding requirements. The Encoder component will make sure to call the functions giving always consistent input parameters.

5.2.2 Encoder

The *Encoder* component makes sure the integrity between the data structures is preserved. It is defined as:

```
pub struct Encoder<K, V, E, P, F>
where
    E: DataStrFactory<K, V>,
    P: ParsingLogic<K>,
    F: EncodingLogic<K, V>,
{
    parser: P,
    encoding_logic: F,
    bijective_map: Option<E::MapStructure>,
}
```

Listing 5.8: Encoder Struct

The Encoder struct is generic with respect to:

- K, V: the data types before and after the encoding.
- P: the parsing logic
- F: the encoding logic
- E: The *DataStrFactory* that binds and deals with the parsing and the encoding of the data.

As we can see, the parser and the encoding logic are members of a struct. The Encoder is generic over a type implementing the DataStrFactory too. This object makes sure to call the functions provided by the DataStrFactory with its member's functions only, therefore guaranteeing integrity:

```
pub fn encode<W: AsRef<Path>>(
        &mut self,
        file_path: W,
    ) -> E::EncodedDataSet { /* .. */ }
```

Listing 5.9: Encoder Struct main function

Every time the Timely Dataflow Engine will need to encode a dataset, it will call the Encoder *encode()* function. This function will insert in the *bijective_map* attribute all the new triples encountered using the same parsing and encoding function.

5.2.3 Timely Dataflow Engine

The main core of the computation is provided as a procedure. Figure 4.1 shows how the Timely Dataflow Engine uses the Encoder component and the Materialization Logic defined by the user. Consistently with this, the procedure shown in Listing 5.10 takes two parameters.

```
pub fn run_materialization<L, R, E, P, F, M>(
    mut encoder: Encoder<L, R, E, P, F>,
    materialization: M,
) -> Result<(), String>
```

Listing 5.10: Timely Dataflow Engine procedure signature

The Materialization Logic is provided as a generic parameter bound to the *Fn* trait (Section 5.1). On the other hand, the encoder parameter is the object of type Encoder explained (Section 5.2.2).

Input Parameters

The Timely Dataflow engine takes as program arguments:

- Path to TBox Dataset
- Path to Abox Dataset
- Path to Output Folder: where all the materialization output is going to be saved
- An optional update path and type of update to perform incremental materialization.
- Any other Timely Dataflow argument (see Appendix D for a list of possible arguments)

In Figure 4.4, we define the system workflow. The Timely Dataflow Engine component takes care of all the different steps defined in the workflow. In particular, if performs the parsing and the encoding of both the TBox and the ABox and of all the update files the user specified in input of the program. For every new dataset it encodes, it updates the *bijective_map* with new entries of the encoder parameter. The *run_materialization()* procedure sets up the dataflow computation, as shown in Listing 5.11. The *dataflow()* function called on *worker* creates a scope in which a dataflow computation is defined. First we create a new *Collection* on this scope and a *data_input* structure that we are going to use to insert the data inside the dataflow. Then, we use the materialization specified by the user as a closure, passing the data_collection and the probe_handle as parameters.

```
let mut probe = timely::dataflow::ProbeHandle::new();
let (mut data_input, mut result_trace) =
  worker.dataflow::<usize, _, _>(|scope| {
            let (data_input, data_collection) =
                scope.new_collection::</*..*/>();
            let res_trace = materialization(
              &data_collection, &mut probe);
            (data_input, res_trace)
        });
```

Listing 5.11: Timely Dataflow Engine dataflow setup

We return from the dataflow function the *data_input* handle to insert the data with and the *res_trace* that represents the Trace Agent that will contain the content of the computation at the different time stamps.

Data Insertion and Materialization

In order to run the computation we need to:

- Load the data in the scope of the dataflow computation. to this extent, we read the encoded dataset from file, previously saved by the *encoder*. To the best of our knowledge, this is the only working way to correctly introduce the data into the dataflow. The data gets loaded into a vector of encoded triples, uniformly distributed among all the worker threads defined by the user.

- Insert the starting data into the dataflow: the *run_materialization()* procedure runs the dataflow computation specified by the user first on the ABox and the Tbox. Only after the full materialization is computed it applies incrementally the updates. The system uses the data_input handle and its *insert()* function.

```
1  for triple in t_box {
2        data_input.insert(triple);
3    }
4  for triple in a_box {
5        data_input.insert(triple);
6    }
7  data_input.advance_to(1);
8  data_input.flush();
```

Listing 5.12: Timely Dataflow Engine initial data insertion

It's important to notice that once the data is inserted into the dataflow, the worker advances its timestamp, notifying the hole system that he will not insert any more data at timestamp 0. This is critical to allow synchronization between workers and ultimately to correctly run the computation.

- Once the data gets inserted, it only queues in Timely Dataflow internal buffers. In order to run the computation, we need to instruct the worker to make progress while a certain condition is met. The *data_input* handle, has a function *time()* that returns the lowest timestamp of an existing data inside the dataflow. When all the workers finish inserting their data. The time of the data input will be 1. Using the *probe* we can get the timestamp the computation is currently at. Before running the materialization this time is 0. Listing 5.13 shows how to run the computation on the data inserted at time 0.

```
1  while probe.less_than(data_input.time()) {
2            worker.step();
3        }
```

Listing 5.13: The system tells the workers thread to make progress

The systems keeps track of the amount of time it takes to perform the materialization and outputs it to file for performance analysis. After these steps are completed, the system decodes the materialization using the *encoder* parameter and saves it to file in the output folder specified by the user. The system keeps track of how much it takes for this step too, for a complete analysis on performances.

Incremental Materialization

Differential Dataflow supports incremental materialization implementing the Differential Computation described in Section 4.2.2. This is transparent to the user. Indeed, to incrementally maintain the materialization, Differential Dataflow uses the same function *insert()* for additions and implements the function *remove()* for data deletion. The important aspect that distinguishes the full materialization from the incremental materialization is the timestamp. We know that the computation of data at timestamp 0 relates to the Full Materialization, while other timestamps correspond to updates. Listing 5.14 shows the implementation of the updates.

```
for (i, (path, mode)) in update_paths.iter().enumerate() {
    match mode {
        Addition =>
            add_data::<E, _, _>(data, &mut data_input, 2 + i),
        Deletion =>
            remove_data::<E, _, _>(data, &mut data_input, 2 + i),
    }
}
```

Listing 5.14: Incremental Materialization on updates

The system loops over all the updates, and in case of Addition it uses the *insert()* function to insert every element of *data* and it uses the *advance_to()* function to increment by a unit the timestamp of the dataflow. Analogous is the case of Deletions, where instead of using the function *insert()* it uses the function *remove()*. After performing the incremental materialization, the system uses the TraceAgent, accessed at different timestamps, to save the result of the incremental maintenance. For each update file and incremental materialization. The system keeps track of the loading time, materialization time and save-to-file time of the computation.

Closing Notes

In light of the Design described in the previous chapter we described the implementation of our proposed approach. In Section 4.1.2 we listed 4 goals that we aim at achieving with the design and the implementation of our system. We motivated that the component structure of the system is able to succeed in G.2 and G.4. Rust provides features that are oriented towards obtaining the best performances without giving up the advantages that models based on abstractions or Object Oriented environments provide (with *Zero Cost Abstraction*). Appendix A goes over a few of the most unique characteristics of the programming language. The main focus of Rust is to push most of the work at compile time, with the goal of providing minimum run-time overhead. This can cause the compilation time to be substantially longer, but this favors better performances at run time. In our context, we accept this trade off as we are looking for the best performances at run time. After describing the implementation, we want to motivate why we think we meet the remaining goals:

G.1 The implementation of the interfaces we described in this chapter is highly based on Rust powerful Trait system that is at the base of the Zero Cost Abstraction and

the use of generics. The *monomorphization* of the code by the compiler, avoids any *dynamic dispatching* that may slow down the run time. Using this techniques was oriented to achieve best performances using Rust effective features.

G.3 The use of generics meets goal N.3, as the user of the system is able to define the computation on the type needed.

Chapter 6

Experiment and Evaluation

In this chapter we will demonstrate how we can use the system we built and described so far, to tackle a use case typical of Stream Reasoning. We use DynamiTE as a reference model to compare the result to. This will allow us to have an indication on how our system performs compared. The code can be found at:
`https://github.com/JimmyHypi/Stream-Reasoning-with-Differential-Dataflow/tree/master/experiments/simple_logic_lubm_data`

6.1 Experiment Description

Consistently with DynamiTE, we will use the LUBM [36] benchmark for performances evaluation. As stated in [7], this benchmark is the standard for measuring performance of reasoning on RDF data. The benchmark provides a data generator [1] that creates extensional data over the *Univ-Bench* ontology that allows the user to control the amount of random data generated, allowing reproducible tests through a seed value. The ontology domain deals with Universities structures and roles. The Ontology Language we are considering is RDFS [19]. In particular, we aim at providing full materialization with incremental maintenance of its subset ρDF [21]. Considering the Design we described in Chapter 4 and the Implementation in Chapter 5 to perform efficient reasoning we need to specify:

- Domain and Codomain of Encoding Logic
- Parsing Logic
- Encoding Logic
- Bijective Map
- Encoded Dataset
- DataStrFactory
- Materialization Computation

[1] `https://github.com/rvesse/lubm-uba`

6.2 Components Definition

Domain and Codomain of Encoding Logic

The domain of the Encoding Logic represents the type NTriples are going to be parsed as. There is a technical problem that does not allow us to simply accept String as the Domain. For sure, we want to avoid using and moving Strings around as they are inefficient. We need to hide the String behind a reference so what we move around is the reference. The problem of references in Rust is that they can only have one owner (Appendix A) and we might find ourselves in situations where that is not enough. We want to avoid cloning the String as it would result in poor performances. What we could use is Rust's *Rc<String>* smart pointer, that allows shared ownership. In our system, Strings are stored in the *bijective_map* parameter of the *encoder* object. The encoder is used inside the worker threads of the Timely Dataflow computation. The worker interface allows only for Thread Safe types to circulate in the worker's closure. For this reason, it is safe to use Rust's *Arc<String>* smart pointer, that is thread safe. So The Domain type of the Encoding Logic will be Arc<String>. Encoding per se is a very broad topic. Many techniques have been developed to cope for a certain type of applications (Section 2.4.3). With our system we provide the ability to define any type of encoding, but web believe that is out of the scope of this project. We will encoded RDF Terms as unsigned integers on 64 bits, *u64*.

Parsing Logic

The Rust ecosystem has a varied support for parser generators. For our application we use LALRPOP [2]. This framework generates a parser from the definition of a grammar and exports the object as a struct available for your code. To use it we need to bind this struct to the *ParserLogic* trait:

```
type ParsedTriple<T> = (T, T, T);
pub struct NTriplesParser {
    lalrpop_parser: ntriples::ArcStringTripleParser,
}
impl ParserTrait<Arc<String>> for NTriplesParser {
    type TripleType = ParsedTriple<Arc<String>>;

    fn parse_triple(&mut self, input: &str) -> Self::TripleType {
        self.lalrpop_parser
            .parse(input)
    }
}
```

Listing 6.1: Parser definition

Encoding Logic

The Encoding Logic we use for the experiment is straightforward: it enumerates the the triples and associates the enumeration to the triple as its encoding. This function maps

[2] https://crates.io/crates/lalrpop

Arc<String> representing RDF Terms to *u64*. This function is stateful: it requires to keep track of the count of the encoded triples. The biggest limit of this approach is that theoretically the cardinality of the codomain is much less than the cardinality of element of types *Arc<String>*. In reality, for our application 2^{64} combinations are enough and there is no risk of conflict as the count is increment for every new triple.

```
pub struct EnumerateLogic {
    current_index: u64,
}

impl EncodingLogic<Arc<String>, u64> for EnumerateLogic {
    fn encode(&mut self, _string: Arc<String>) -> u64 {
        let res = self.current_index;
        self.current_index += 1;
        res
    }
}
```

Listing 6.2: Encoding Logic definition

Bijective Map

The Bijective Map is the data structure that saves the encoding mapping and provides the *locate()* and *enxtract()* methods (Section 4.1.1). To achieve this we use Rust's *BiMap* data structure. A BiMap is internally represented as two HashMaps, giving the ability to retrieve both right and left elements.

```
pub struct BijectiveMap<K, V> {
    bijective_map: BiMap<K, V>,
}

impl<K, V> BiMapTrait<K, V> for BijectiveMap<K, V>
{
    fn locate(&self, right: &V) -> Option<&K> {
        self.bijective_map.get_by_right(&right)
    }
    fn extract(&self, left: &K) -> Option<&V> {
        self.bijective_map.get_by_left(&left)
    }
    fn insert(&mut self, left: K, right: V) {
        self.bijective_map.insert_no_overwrite(left, right)
    }
}
```

Listing 6.3: Bijective Map definition

Encoded Dataset and DataStrFactory

We use a Vector to store the encoded triples. Then, we create the DataStrFactory that will take care of encoding the data and storing it in the specified data structures.

```
type EncodedTriple<T> = (T, T, T);
pub struct BiMapEncoder {}

impl DataStrFactory<Arc<String>, u64> for BiMapEncoder {
    type MapStructure = BijectiveMap<Arc<String>, u64>;
    type EncodedDataSet = Vec<EncodedTriple<u64>>;

    /* functions to instantiate the data structures */
}
```

Listing 6.4: Encoded Dataset and DataStrFactory definition

6.2.1 Materialization Computation

The Full Materialization follows the approach we designed in Section 4.3. We start showing the structure of the computation at high level and then describe the computation of every group of rules. Listing 6.5 contains the definition of the function to compute the full materialization.

```
/// Computes the full materialization of the collection
type EncodedTriple<T> = (T, T, T);
pub fn full_materialization<G, V>(
    data_input: &Collection<G, EncodedTriple<V>>,
    mut probe: &mut ProbeHandle<G::Timestamp>,
) -> TraceAgent</* .. */> {
  /* .. */
}
```

Listing 6.5: *full_materialization() function signature*

The function is generic with respect to some type G and V. G represents the type of the scope required by the Collection type. V is the encoding function codomain type EncodedTriple<V> is a type alias for triples inside the Collection. The function takes the *data_input* that is a reference to a Differential Dataflow Collection of triples, and a mutable *probe* of time *ProbeHandle*. The return type is the TraceAgent that containts the trace of computation over time. Collection traces represent the history of the collection and can be scanned to retrieve the events that happened to the collection. The first part of the full materialization algorithm according to 4.6 is implemented in Listing 6.6.

```
let sco_transitive_closure = rule_4(&data_input);
let spo_transitive_closure = rule_1(&data_input);
let data_input = data_input
  .concat(&sco_transitive_closure)
  .concat(&spo_transitive_closure);
// ...
```

Listing 6.6: full materialization implementation #1

Listing 6.6 shows that Rule 4 and Rule 1 are called on the original input collection according to the design of the algorithm. In fact, Rule 1 and Rule 4 are not in any *rule_precedence* relation. At line 3, we concatenate the input collection with the result of the iterative computation of Rule 1 and Rule 4, so that the derived data is available to downstream operators.

```
let spo_type_rule = rule_2(&data_input);
let data_input = data_input
  .concat(&spo_type_rule);
// ...
```

Listing 6.7: full materialization implementation #2

Listing 6.7 shows how Rule 2 uses the data produced by rule number 1, according to the *rule precedence* relation defined previously. Similarly, Listing 6.8 shows how Rule 3 uses the data from Rule 4 that has been concatenated to the data input at the beginning of the algorithm.

```
let domain_type_rule = rule_5(&data_input);
let range_type_rule = rule_6(&data_input);

let data_input = data_input
  .concat(&range_type_rule)
  .concat(&domain_type_rule);
let sco_type_rule = rule_3(&data_input);
let data_input = data_input
  .concat(&sco_type_rule);
// ...
```

Listing 6.8: full materialization #3

At this point the dataflow is complete. The algorithm ends with the following lines of code:

```
let arrangement =
  data_input.arrange_by_self();

arrangement
  .stream.probe_with(&mut probe);

arrangement.trace
```

Listing 6.9: full materialization #4

We call the the *arrange_by_self()* operator to get the arrangement of the collection. This contains two fields:

- *stream*: this field represents a Timely Dataflow stream that reproduces the events happened in the collection.
- *trace*: this field represent the data structure that keeps track of all the events that happened in the collection. This allows a collection to be moved from a dataflow to another dataflow and to get a *Cursor* that allows us to access it and retrieve the information inside the trace (i.e., the arranged elements of the collection)

In our case, the algorithm uses the stream field to put the probe passed as parameter and then it returns its trace. The probe is used by the workers to get information about the progress made by the whole system, allowing safe coordination between them.
We now show the core implementation of the rules, according to the division performed in Section 4.3.1.

6.2.2 Rule 5 and Rule 6

We start filtering the data input looking for all the schema triples that contain respectively, DOMAIN and RANGE predicate, as shown in Listing 6.10.

```
let only_domain / only_range =
        data_collection
            .filter(|t| t.predicate == "DOMAIN / RANGE");
```

Listing 6.10: Rule 5 and 6 implementation #1

After that we join the original input data with the triples we just selected, and map them according to the bindings of the Head of the rule. We show the implementation of Rule 5 in Listing 6.11:

```
let domain_type_rule =
   data_collection
        .map(|t| (t.predicate, (t.subject, t.object)))
        .join(&only_domain.map(|t| (t.s, (t.p, t.o))))
        .map(|(_key, ((a, _b), (_dom, d)))| (a, TYPE, d));
```

Listing 6.11: Rule 5 and 6 implementation #2

The *join()* operator takes as input collection of *(Key, Value)* and returns a collection of *(Key, JoinedValue1, JoinedValue2)*. So the *map()* operations are needed to format correctly the input and output elements. Rule 5 and Rule 6 do not require any iterative computation, according to the design of the code.

6.2.3 Rule 1 and Rule 4

Rule 1 and 4 introduce iterative computation, performed through the *iterate()* operator.

```
data_collection
   .filter(|t| t.predicate == "SCO/SPO")
   .iterate(|inner| {
      inner
         .map(|t| (t.o, (t.s, t.p)))
         .join(&inner.map(|t| (t.s, (t.p, t.o))))
         .map(|(_o, ((s1, p1), (_p2, o2)))| (s1, p1, o2))
         .concat(&inner)
         .distinct()
  });
```

Listing 6.12: Rule 1 and 4 implementation

As shown in Listing 6.12, the *iterate()* operator takes as a parameter a closure that represents the loop context. The *inner* variable represents the loop variable that contains the newly produced triples that are fed back to the loop until no new triple is derived.

6.2.4 Rule 2 and Rule 3

Rule 2 and Rule 3 require both iterative computation and the join between two collections. These two collections are shown in Listing 6.13

```
let sco_only = data_collection
                  .filter(|t| t.predicate == "SPO/SCO");
let candidates = data_collection
                    // This only for Rule 3
                    .filter(|t| t.predicate == "TYPE");
```

Listing 6.13: Rule 2 and 3 implementation #1

The variable *sco_only* retains all the triples that have respectively SPO and SCO as predicates and the variable *candidates* that for Rule 3 contains all the triples that have TYPE as the predicate. Regarding Rule 2, this coincides with the input data contained in the variable *candidates*. With these two collections defined, the materialization can be computed through an iterative computation. This iterative computation requires both collection to enter the loop context. The *iterate()* operator automatically introduces the collection that the function is called on into the loop context. For other collections to enter it, the *enter()* function is required. This function takes the looping scope as a parameter, and extends the collection timestamp with the iterative count related to that scope. The implementation in Listing 6.14 relates to Rule 3.

```
candidates
   .iterate(|inner| {
      let sco_only_in =
            sco_only
               .enter(&inner.scope());

      inner
         .map(|t| (t.o, (t.s, t.p)))
         .join(&sco_only_in.map(|t| (t.s, (t.p, t.o))))
         .map(|(_key, ((x, typ), (_, b)))| (x, typ, b))
         .concat(&inner)
      .distinct()
  });
```

Listing 6.14: Rule 2 and 3 implementation # 2

6.3 Main Function

Now that we defined all the computation parts, we are ready the assemble all the components together.

```
fn main(){
  let parser = NTriplesParser::new();
  // The EnumerateLogic starting number is 0.
  let encoding_logic = EnumerateLogic::new(0);
  let bi_map_encoder: Encoder<_, _, BiMapEncoder, _, _> =
      Encoder::new(parser, encoding_logic);

  run_materialization(bi_map_encoder,
      move |data_input, mut probe| {
          full_materialization(&data_input, &mut probe)
      });
}
```

Listing 6.15: Main Function

We show this function in order to display how the high modularization of the system allows the code to be clear to read and easy to write. Line 2 and 4 instantiate respectively the parser and the encoding logic. In line 5 we instantiate the Encoder that is passed to the core function of the system, *run_materialization()*. The second parameter of the function is the Closure of generic type M defined in Section 5.1. This is where the logic of the computation is described. In the closure we only call the *full_materialization()* function defined in Section 6.2.1.

6.4 Evaluation

In this section we will report the evaluation of the performances of the approach we described in this thesis work applied to the ρDF materialization computation with incremental maintenance of LUBM data. We compare the results with DynamiTE.

This analysis will help us unveil meaningful modifications and optimizations to further explore this topic of research and make some important considerations that lay down the foundations for future works.

Hardware Used

The experiment was carried by a Laptop PC with a Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz, 6 Physical Cores, 12 Logical Processors, 30 GB of available physical main memory and 250 GB available space in a SSD, running *5.8.5.-arch1-1 GNU/Linux*. This current scenario limits the possibility of testing our approach on very large scale dataset, but still gives us insights about efficiency of the system.

6.4.1 Materialization Correctness

To evaluate the correctness of the computation we used the LUBM test queries[3]: These are SPARQL [18] queries that test specific reasoning procedures. Along with the queries, LUBM provides the answers [4]. In order to check the correctness of the materialization resulting from our approach, we built our own local benchmark with the following features:

- Takes the test queries and the answers as defined in the LUBM benchmark.
- Takes the materialization output of the Timely Dataflow computation.
- Evaluates each query against the materialization.
- Checks if the corresponding query answers provided by the LUBM benchmark is equal to the result.
- If any query provides wrong results reports which query failed and which triple caused the Query to fail.

This benchmark is written in Java and uses Apache Jena's [37] SPARQL Query library for Query Evaluation. We did not include the code implementation in this document as it is only used to validate the functionalities of our system. The Test Queries are 14 and they offer different trade-offs between query selectivity and input size. They cover different reasoning procedures coming from both ρDF [21] and OWL [20]. Moreover, the queries only address the full materialization problem, with no support for incremental materialization validation. Our implementation deals only with ρDF reasoning rules so for this reason the benchmark tests only a subset of Queries provided by LUBM. We can make an observation: some of the queries that test OWL reasoning rules can be tested by adding a triple in the dataset, that would be derived through the higher expressive OWL Ontology Language. This allows us to use some of the queries meant for OWL reasoning to check the correctness our Incremental Computation. With this we can use up to 11 out of 14 queries to check the correctness of both our full and incremental materialization.

[3] `http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt`
[4] `http://swat.cse.lehigh.edu/projects/lubm/answers.htm`

6.4.2 Performance Evaluation

We will follow DynamiTE's steps for performance evaluation. In particular we aim at:

- Analyzing the performances of the full materialization.
- Analyzing the performances of applying updates to existing computation.

We were able to perform the same computation using DynamiTE on the same hardware, but only the full materialization was able to terminate, while, when computing the updates, DynamiTE remains stuck on a infinite loop for un unknown reason. Therefore, we are going to show a direct comparison only for the full materialization. We also want to make considerations as to where the system lacks in performance, outlining possible solutions to be implemented in future works. In order to do this, we provide performance analysis on:

- Parsing and encoding of RDF datasets.
- Loading RDF datasets into the dataflow.
- Decoding and saving materialization to file.

Full Materialization

The maximum number of Universities we are able to use to perform computation on is 50. This corresponds of a dataset of approximately 6.9 million triples. The computation is run over 8 workers. The machine has 6 physical cores, so we expect no improvement for values greater than that. We include the computation over 8 workers to show this fact. For LUBM(50) results are shown in Figure 6.1.

As we can see, the time decreases as the number of worker approaches to 6 and increases after that value. In particular we get the best result at 5 workers with a materialization time of 7175 ms. The throughput for LUBM(50) using Timely Dataflow is ∼960K Triples/s. It is interesting to see how the full materialization time scales with the size of the input dataset. We compare with what we were able to obtain using DynamiTE on the same hardware. The results are shown in Figure 6.2. As we can see, on the left, the figure shows how the Full Materialization time increases as the dataset input size increases (in terms of number of Universities produced by the LUBM data generator). We can see that Timely Dataflow scales more steadily, with an average throughput of ∼1 Million Triples/s against ∼250K Triples/s of DynamiTE.

Incremental Updates

Our approach, at this stage of the work, performs Graph Level Entailment (Section 2.4), therefore the window of occurrence of the RDF Triples gets merged, so that updates are provided as batches of triples (RDF Graphs). Likewise, DynamiTE performs Graph Level Entailment and tests the incremental maintenance over 6 different types of updates for triple addition and 6 different types of updates for triple removal. The authors provide access to these updates[5], so that we were able to use them to test our approach. We report here the updates that are defined in [7]:

[5] `https://github.com/jrbn/dynamite/tree/master/updates`

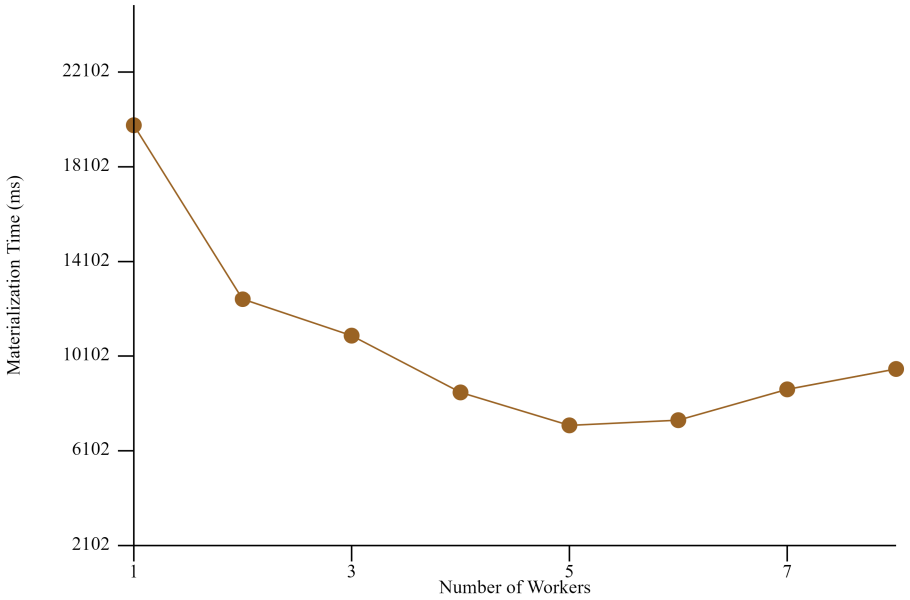


Figure 6.1: Materialization time per worker

- Update 1 adds and removes one triple that does not trigger reasoning.
- Update 2 adds and removes almost 16k triples that do not trigger reasoning.
- Update 3 adds and removes almost 8k triples that produce a fixed number of reasoning
- Update 4 adds and removes TBox triples that trigger reasoning proportional to the input size.
- Update 5 adds and removes one University.
- Update 6 adds and removes two Universities

. 6.1 shows the results of the updates performed for LUBM(50)

Table 6.1: Incremental maintenance time for LUBM(50) in ms

Update	Addition	Deletion
1	<1	<1
2	29	46
3	28	34
4	34	24
5	202	210
6	430	456

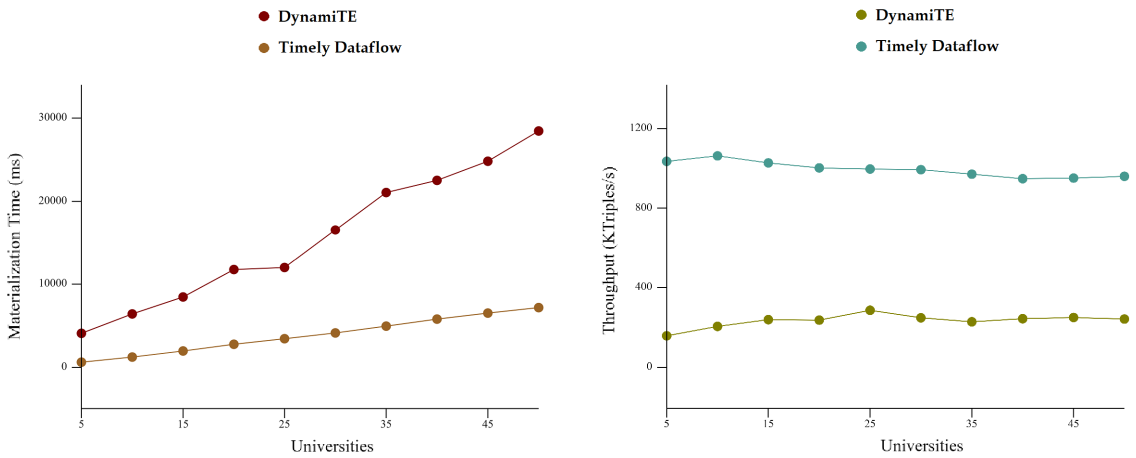


Figure 6.2: (Left) Full Materialization time per number of Universities. (Right) Throughput per number of Universities

Since update 6 is the update that adds/deletes and triggers the most reasoning, we show how the update time increases with the number of universities in 6.3. The graphs show

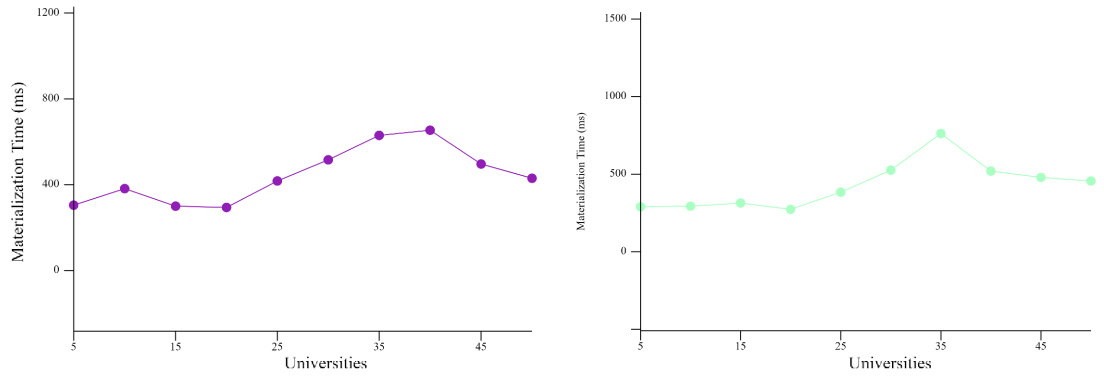


Figure 6.3: (Left) Update 6 Addition time per University. (Right) Update 6 Deletion time per Univeristy

one impressive feature. The update time does not show a major increasing tendency. In the graph, there is an increase that starts at 20 Universities and end at around 40 Universities. After that, the update time goes back down. Although it looks like the increase amount is high, the scale of the axis might deceive. The update times vary in an interval of 200 ms: $\sim$500ms $\pm$ 200ms which for datasets of millions of triples is a very small amount. Within [5-50] Universities the update time does not manifest any significant increase in the trend. Arguably, it makes sense to believe that the update time does not depend on the size of the input data. It would be interesting to see if this happens for any number of Universities and if not how this time increases. We leave as a future work the evaluation of the update trend for very large datasets.

Loading and Save-to-File time

For Loading and Save-to-File time we cannot make any meaningful comparisons as DynamiTE and Timely Dataflow are based on completely different models and perform these steps differently. Both operations have similar trends:

- Figure 6.4 confirms that the best result is obtained for a number of workers equals to the number of physical cores. In particular we obtain that:
  - Best Load Time results with 5 workers: 606ms, for 11,3 Million Triples/s of throughput value
  - Best Save-to-File Time results with 7 workers: 18s, for 382,7K Triples/s of throughput value
- Figure 6.5 and Figure 6.6 on the left show that both the load time and the save-to-file time have good scalability. They increase linearly with respect to the size of the input dataset.
- Figure 6.5 and Figure 6.6 on the right show that the throughput fluctuates over the following intervals:
  - Load time: ∼11,6M ± 0.4M Triples/s
  - Save-to-File time: ∼380K ± 20K Triples/s

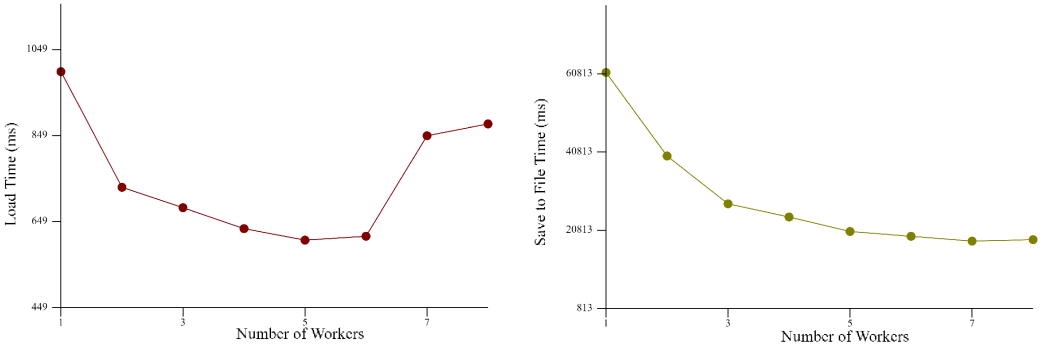


Figure 6.4: (Left) LUBM(50) Load time per workers. (Right) LUBM(50) Save-to-File time per workers

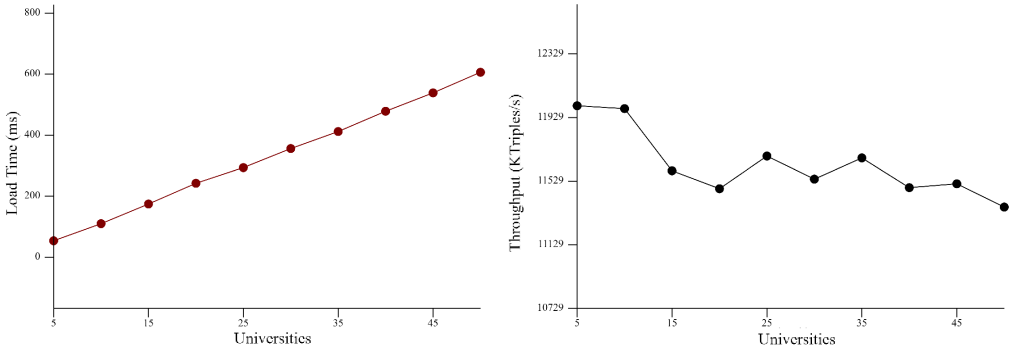


Figure 6.5: (Left) Load time per number of Universities. (Right) Load throughput per number of Universities

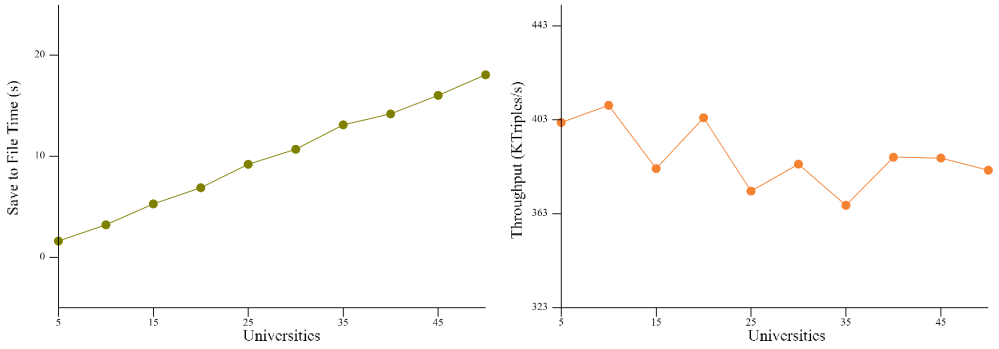


Figure 6.6: (Left) Save-to-file time per number of Universities. (Right) Save-to-File throughput per number of Universities

Encoding

We developed a system able to abstractly define the encoding function. For this experiment, we decided not to focus on the performances of the encoding routine as many techniques could be considered and new approaches could be developed. We leave such topic for a future work. The encoding function that we use for this experiment (Section 6.2) runs single-threaded. For this reason, we expect this operation to be the main bottleneck for the overall performances. We show the results related of the Encoding for completion in Figure 6.7. We obtain that the Encoding phase scales linearly as the input data increases, with a throughput that fluctuates between ∼73K and ∼78K Triples/s. This confirms our expectations: this phase is considerably slower than all the other phases.

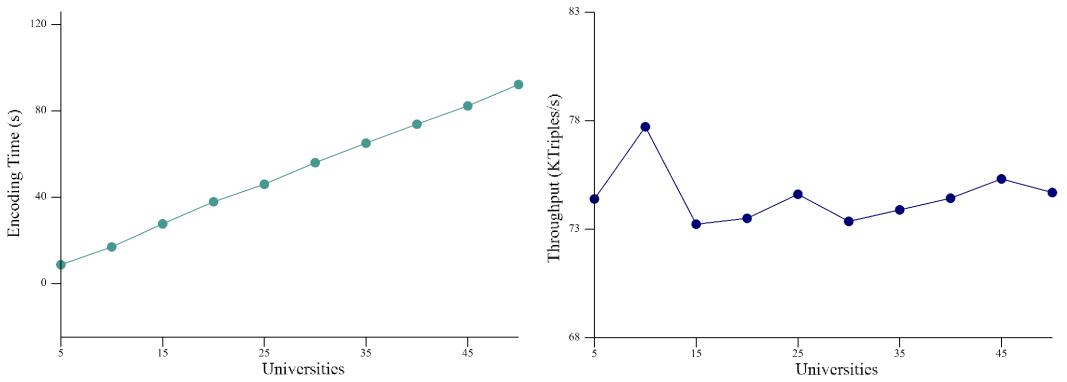


Figure 6.7: (Left) Encoding time per number of Universities. (Right) Encoding throughput per number of Universities

6.4.3 Performance Analysis

From the performance data we showed we are able to have a comprehensive view of the efficiency of the system. Our main focus for this thesis work is to engineer a Timely Dataflow system in a way to allow efficient reasoning on RDF dataset with incremental materialization maintenance. We use the system to perform ρDF reasoning on LUBM generated data. The Timely Dataflow system outperforms DynamiTE in all tasks. The system scales well with the size of the input data as all steps show a linear increase with respect to an increase on the input data. The system is able to materialize $\sim$1 million triples per second, or in other words, it takes $\sim$1 microsecond to process a triple on average. This is a positive result if we compare to our reference model. We find our most exciting results in the incremental maintenance. We are able to add/remove $\sim$300K triples (sixth update) from a dataset of $\sim$6.9M triples and maintain the output of the result in less than a second (400 ms). This result makes this feature very promising in view of comparing it to state-of-the-art systems.

The analysis on the performances allows as to identify the main bottlenecks of the system.

Bottleneck Analysis

The main bottleneck of the system is the encoding logic. This is performed single-thread and provides a throughput of $\sim$75K triples encoded per second. This can be easily improved by providing a more sophisticated technique run on multiple threads. The system modularization makes the integration of a different encoding logic very simple.

We can identify two more bottlenecks that could potentially limit the performances:

- The materialization logic (Section 6.2.1) makes heavy use of the costly *join* operator. Reducing the number of *join* operations would improve performances. One possible idea is to introduce semantic encoding. For example, *LiteMat* (Section 2.4.3) provides a technique to introduce ρDF semantic inside the encoding, such

as the SuperClass or SuperProperty of a resource. Having this information, the join operation is not needed to find the SuperClass or SuperProperty of resources as it can be deduced from the encoding. The drawback of this technique is that it complicates the encoding logic. So, ultimately, a trade-off is required to obtain the best performances.

- The system makes one distinction between TBox and ABox triples. ABox triples are divided uniformly among all workers while TBox triples are copied for each worker and inserted in their portion of the dataflow. This design has two important limits:
  - For larger TBox-es, this would constitute a major bottleneck on performances as each triples needs to be replicated for each worker.
  - The system cannot correctly compute the materialization of *datalog* rules that require the join of triples from the ABox. This poses a major limit in the expressiveness this system can achieve.

  In our application, these limits are negligible:
  - The TBox triples represent only a fraction of the overall data and copying the data among workers does not result in a loss in performances.
  - The datalog program corresponding to ρDF reasoning (Table4.1) does not present any rule that has two ABox triples on its body. This means that no join between ABox triples is required.

Conclusions

In this dissertation work, we laid down the foundations for a new approach to Stream Reasoning using the timely dataflow computation model for computing the Full Materialization of a Knowledge Base and its Incremental Maintenance.

After defining the context of the thesis work listing all the required information in the application domain in Chapter 2, in Chapter 3 we used the Macro-Meso-Micro [13] model to formalize our Problem Statement. In the Macro level of analysis we elicited requirements that come from the question: *Can we make sense, in real-time or near real-time, of vast, heterogeneous data streams that originate from different complex domains, in order to be able to perform useful analysis?* To address this high level domain, we narrowed down the scope of the problem noticing that Graph Stream Processing offers techniques that can be related to Stream Reasoning. Hence, we asked ourselves: *can Graph Stream Processing be used to make Stream Reasoning?* In the Micro Level of Analysis we identified Timely Dataflow as a promising tool that offers features that provide outstanding results in Graph Stream Processing application. We also identified DynamiTE, IMaRS and RDFox as systems that offer different points of view to tackle the different challenges of this scenario. In the thesis we chose DynamiTE as the starting point. So the question at this level is: *can we implement DynamiTE core functionalities in Timely Dataflow?*

In Chapter 4 we started formulating 4 goals to lead us during the design and implementation of the system. We designed a library that is highly modularized and offers low level details encapsulation. Such library offers the interface necessary for the development of a Stream Reasoner. We demonstrate this system by building a Reasoner capable of performing ρDF entailment and incrementally maintain the computation. This requires to design the corresponding dataflow computation. We saw how we can break up the datalog evaluation fix-point computation into the fix-point computation of single datalog rules. We proved that this is safe to do and how this leads us to an elegant design of the dataflow of the computation. The design we propose is able to meet two of the four goals listed.

Based on the previous design, in Chapter 5 we show the most important features of the implementation of the different components of the library and how they cooperate to perform efficiently the functionalities required by the client. In this chapter, we motivate also how we think we achieve the remaining two goals formulated in Chapter 4.

In Chapter 6, we demonstrate how to use the library to build a Reasoner that performs full materialization with incremental maintenance of updates. In particular, we provide the implementation of the abstract component of the system and use the generic interface provided by library to perform the computation of the materialization. We

test this Reasoner with the LUBM benchmark showing performance results and comparing it to DynamiTE. The Reasoner built outperforms DynamiTE showing promising results, especially about the incremental maintenance.

6.4.4 Requirement Analysis

In Chapter 3, we started formulating requirements at high level. Progressively we narrowed down the scope of the problem, formulating more concrete requirements. In this Section we list our main contributions in the perspective of these requirements:

R.1 The performance evaluation provided in Section 6.4.2 shows that the computation time of all the different steps increases linearly with respect to the size of the input data with a very steady throughput. On top of this, we obtain very promising results with the incremental maintenance: the increase rate of the incremental materialization time is negligible.

R.2 The reasoner we built in Chapter 6 provides a Parser that feeds RDF data into the system, that will compute the logic on them.

R.3 The system interface is highly generic. This allows the user to define the type of data he/she wants to perform the computation on. This data can be timestamped data. The system requires the user to specify the computation so that it does not constrain the type of data used.

R.4 In our application, we compare the Reasoner we built with DynamiTE. Based on the performance evaluation in Section 6.4.2 we obtain results that outperform DynamiTE. This is still not enough to claim that the reasoner can compete with the state of the art. In fact, we tried to run the same computation on RDFox [16]. The full and incremental materialization take less than a second in both cases. This means that we have work to do to compete with the state of the art, but nonetheless the results are promising especially the ones related to the incremental maintenance.

R.5 Timely Dataflow includes iterations in its operators. This enhance the expressivity of the framwork. In this thesis work we perform ρDF reasoning. This uses the *iterate()* method provided by the framework.

R.6 We rely on the RDF data model to deal with data coming from Complex Domain models.

6.4.5 Future Work

Problem Statement

One of the main goals of the project is to provide a system that is extensible and modular enough to support the development of future works. In this thesis work, taking DynamiTE as a reference, we focused on the aspect of *Graph Level Entailment*, the Full Materialization and Incremental Maintenance against batches of input changes. This is

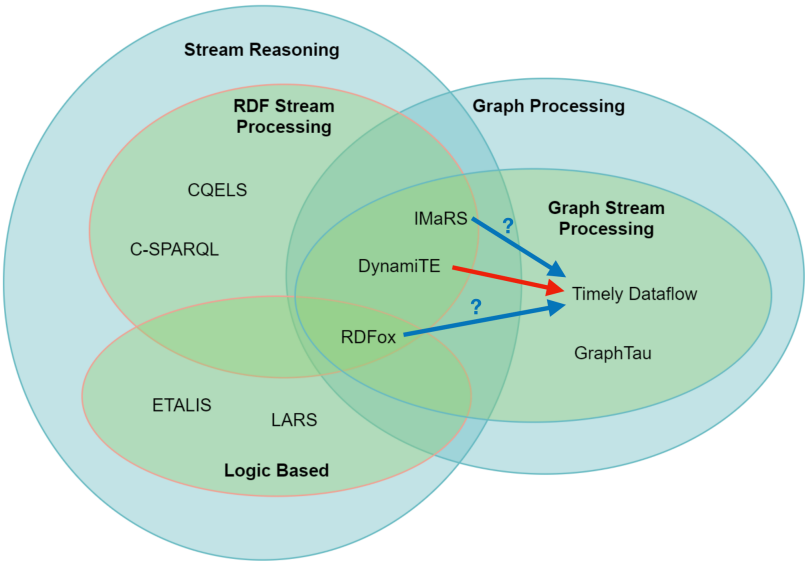


Figure 6.8: Future work visual representation

the first step towards the definition of a complete reasoner able to perform *Window* and *Stream Level Entailment* according to the model in Figure 2.6. Efficient Incremental Maintenance is required to deal with entailment within specific time frames. We used Differential Dataflow's differential computation to perform Incremental Maintenance. DynamiTE introduces the counting algorithm as an alternative approach to perform Incremental Maintenance. A possible future work is to extend our implementation to cover the counting algorithm in a Timely Dataflow computation, comparing it to the differential computation.

Additionaly, in Chapter 3 we mentioned IMaRS and briefly introduced its approach. IMaRS faces *Window Level Entailement*, offering a new logic to perform Incremental Maintenance based on expiration times. A possible future work is extending the Timely Dataflow to take into account the timing aspect associated to triples to see if we can implement IMaRS' logic to perform Incremental Maintenance on timestamped data.

At this level, we are considering only the ρDF Ontology Language. A more ambitious work consists in extending reasoning through Timely Dataflow to the OWL ontology language and ideally make it so that the Timely Dataflow system evaluates generic Datalog rules. In this way we are able to explore how and if logic based approaches, like RDFox [16] can be performed with this system. With the development of these features we will be closer to a system able to collocates itself in the Stream Reasoning context.

System Design

The system we designed in Chapter 4 is oriented towards extensibility. There is one major limitation on the expressiveness able to be achieved by the system that prevents the system to be extended to a broader scope. Currently, the system replicates the TBox

triples in each worker. This design choice optimizes the ρDF entailment logic, but it poses a limit: each worker only has a portion of the ABox, therefore the system is not able to transparently support exchanges of ABox triples among workers required in cases where joins between ABox triples are required (e.g., transitivity property). A possible future work is to extend the design of the system to transparently handle the exchange of ABox triples in order to be able to represent a higher level of expressiveness computation. The system design would then be able to capture different, more complex reasoning procedures and deal with more sophisticated Ontology Languages (e.g., RDFS+, OWL). In future work, we expect the system we developed to be the core used by other systems that offer progressively more functionalities. It would be interesting to explore if and how it is possible to perform the following:

- Automatically translate Datalog rules into Differential Dataflow computation and feed it into our system. This would make the program very easy to use as the user would only need to give a Datalog program instead of the definition of the materialization logic. Clearly, this is a complex task and most likely it will require to constrain the Datalog program to a tractable subset to meet performance goals.
- Our system has capabilities to be used as a SPARQL query engine with support to Incremental Maintenance. In this case, the dataflow computation fed into the system represents the Query logic. Ideally, an automatic procedure would translate the Query into a Differential Dataflow computation and use out system to run it on encoded data, incrementally maintaining the result.

Encoding and Indexing Schemes

In Chapter 6.4, we saw how the Encoding step is the major bottleneck of the system. Currently, the encoding is performed outside the dataflow computation. This is to allow thread safety when dealing with stateful encoding. In fact, accessing mutably the state of the Encoder inside a dataflow computation requires the worker thread to lock the Encoder. This creates a double layer of synchronization (locks of the encoder and system timestamps) that makes coordination between workers susceptible to deadlocks. For this reason, the dataset gets encoded outside the dataflow computation. Two are the possible solutions we leave as a future work to explore:

- Keep the computation outside the dataflow but providing parallel encoding.
- Rearrange computation in a way it is deadlock-safe to perform the encoding in parallel in Timely Dataflow worker threads.

With that done, we can implement more complicated encoding functions, such as the ones that include semantic information in the encoded identifiers (2.4.3).

Another interesting extension would be to study how compression and indexing schemes can improve the computation. DynamiTE, for example, indexes the data into B-Trees to support querying. A possible future work consists in studying how to interface the indexes structures with the Differential Dataflow basic data structures to benefit the computation. The system supports such an extension through the *EncodedDataSet* defined in Section 5.2.1, but it lacks the integration with the dataflow computation.

Ideally, we need to enrich the trait bound on the *EncodedDataSet* and define a higher level trait that represent Indexing Schemes.

Performance Evaluation and Benchmarking

In Section 6, we demonstrated how to use the system we built to implement a basic reasoner able to perform the full materialization and incremental maintenance using the LUBM benchmark. For this thesis work, we were able to test the computation on limited datasets (LUBM(5) to LUBM(50)). In order to get a deeper performance analysis, a future work will test the same computation on larger datasets. This will allow us to validate or refute our evaluation providing a better insight on our system capabilities.

Another interesting future work is to use different benchmarks to test the functionalities of the system. In particular, in this thesis work, we perform bulk deletion using the LUBM benchmark. We saw in Section 1.3 that bulk deletion is critical when dealing with real world situation, such as providing the *right to be forgotten* of the GDPR. The LUBM benchmark lacks the entities and properties typical of social networks that would make the *right to be forgotten* relevant. For this reason, in a future work, we will test our system with the LDBC Social Network Benchmark [38], that provides an ecosystem closer to a social-network-like one, so we can show how bulk updates performed by our system are able to provide a solution to a real world problem, the *right to be forgotten* of the GDPR.

Bibliography

[1] Philip Russom. Big data analytics. Technical report, The Data Warehousing Institute (TDWI), 2011.

[2] W3C. Semantic web stack. https://www.w3.org/2003/Talks/0624-BrusselsSW-IH/26.svgz.

[3] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. On the semantics of sparql. In *Semantic Web Information Management*, pages 281–307. Springer, 2010.

[4] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44, 06 2012.

[5] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, 1:59–83, 2017. 1-2.

[6] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[7] Jacopo Urbani, Alessandro Margara, Ceriel Jacobs, Frank Van Harmelen, and Henri Bal. Dynamite: Parallel materialization of dynamic rdf data. In *International Semantic Web Conference*, pages 657–672. Springer, 2013.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[9] Apache software foundation: Apache hadoop. https://hadoop.apache.org/.

[10] Apache software foundation: Apache spark. https://spark.apache.org/.

[11] World wide web consortium: Semantic web. https://www.w3.org/standards/semanticweb/.

[12] Rdf 1.1 concepts and abstract syntax. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[13] Sandro Serpa and Carlos Ferreira. Micro, meso and macro levels of social analysis. *International Journal of Social Science Studies*, 7:120, 04 2019.

[14] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, November 2009.

[15] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[16] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. Rdfox: A highly-scalable rdf store. In Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015*, pages 3–20, Cham, 2015. Springer International Publishing.

[17] Daniele Dell'Aglio, Marco Balduini, and Emanuele Della Valle. *Applying Semantic Interoperability Principles to Data Stream Management*, pages 135–166. Springer International Publishing, Cham, 2015.

[18] Steve Harris and Andy Seaborne. Sparql 1.1 query language, w3c reccomendation, w3c, 2013. https://www.w3.org/TR/sparql11-query/.

[19] Rdfs ontology language, w3c reccomendation, w3c, 2014. https://www.w3.org/TR/rdf-schema/.

[20] Owl 2 ontology language, w3c reccomendation, w3c, 2012. https://www.w3.org/TR/owl2-overview/.

[21] Sergio Munoz, Jorge Pérez, and Claudio Gutierrez. Minimal deductive systems for rdf. In *European Semantic Web Conference*, pages 53–67. Springer, 2007.

[22] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. *International Semantic Web Conference*, page 370–388, 2011.

[23] Jean-Paul Calbimonte, Hoyoung Jeung, Oscar Corcho, and Karl Aberer. Enabling query technologies for the semantic sensor web. *International Journal On Semantic Web and Information Systems (IJSWIS)*, 8:43–63, 01 2012.

[24] Raphael Volz, Steffen Staab, and Boris Motik. Incrementally maintaining materializations of ontologies stored in logic databases. *J. Data Semantics*, 2:1–34, 01 2005.

[25] Olivier Curé, Hubert Naacke, Tendry Randriamalala, and Bernd Amann. Litemat: a scalable, cost-efficient inference encoding scheme for large RDF graphs. *CoRR*, abs/1510.03409, 2015.

[26] Lei Zou and M. Tamer Özsu. Graph-based rdf data management. *Data Science and Engineering*, 2(1):56–70, Mar 2017.

[27] John Liagouris, Nikos Mamoulis, Panagiotis Bouros, and Manolis Terrovitis. An effective encoding scheme for spatial rdf data. *Proc. VLDB Endow.*, 7(12):1271–1282, August 2014.

[28] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[29] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Stream reasoning and complex event processing in etalis. *Semant. Web*, 3(4):397–407, October 2012.

[30] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Christian Folie. Stream reasoning with lars. *KI - Künstliche Intelligenz*, 32(2):193–195, Aug 2018.

[31] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[32] Apache software foundation: Apache giraph. https://giraph.apache.org/.

[33] Maciej Besta, M. Fischer, Vasiliki Kalavri, M. Kapralov, and Torsten Hoefler. Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. *ArXiv*, abs/1912.12740, 2019.

[34] Arturo Diaz-Perez, Alberto Garcia-Robledo, and Jose-Luis Gonzalez-Compean. *Graph Processing Frameworks*, pages 875–883. Springer International Publishing, Cham, 2019.

[35] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of CIDR 2013*, January 2013.

[36] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: a benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3:158–182, 10 2005.

[37] Apache jena. https://jena.apache.org/index.html.

[38] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasić, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. The ldbc social network benchmark, 2020.

[39] S. Abiteboul, R. Hull, and V. Vianu. *Foundation of Databases*. Addison-Wesley, 1995.

Appendix A

Rust

A.1 Rust

Rust is a system programming language designed for highly performant and highly safe systems. For this reason, among the many features it provides, we can highlight the following.

A.1.1 Memory Safety

Rust is designed to be memory safe without using an automated Garbage Collector:

- It does not allow null pointers, as it requires the variable to be initialized before use. Null pointers are replaced with the *Option* types that is an enumeration with two variants: *Some* and *None*. The advantage of such design choice is that when using variable of type *Option*, the compiler requires the user to explicitly handle both cases, *Some* and, more importantly, *None*. With this, the null pointer represented by the *None* variant is sure to be handled and this check is done at compile-time, which will not add any run-time overhead.

- It prevents dangling pointer at compile time through the explicit use of reference *lifetimes*. The user in this case has to specify the reference lifetimes. The borrow-checker is the component of the compiler that checks the consistency of lifetimes and assures that there is not going to be any reference that points to deallocated memory.

- Data races are prevented at compile time. A data race occurs when there is no synchronization control when multiple pointers access the same data simultaneously and at least one of them is used for writing the data. The compiler wants the user to specify whether a reference variable is mutable or immutable. It then uses this annotation to apply the following rule: 'At any moment, you can have *either* one mutable reference *or* any number of immutable references'. Through this rule the compiler avoids data races that are renown to be difficult bugs to detect.

A.1.2 Ownership

Ownership allows the rust compiler to make memory safety guarantees, without the need for a Garbage Collector. Ownership deals with how the memory is managed through the run of the program. The guarantees are made at compile-time through the application of rules, so that the ownership system does not slow down the run-time. The rules are the following:

- All the values in Rust have a variable called their *owner*
- Only one owner is allowed at a time
- It is sufficient that the owner goes out of scope for the value to be dropped from memory

Because of these, the user does not have to write code for the deallocation of an object and it does not require a garbage collector whose periodic routine would slow down the run time of the program.

A.1.3 Types and Polymorphism

Rust is a statically typed programming language with support to *type inference*, for which the compiler can determine the type or raise a compile-time error if either a variable is never assigned or assigned with two constant of two different type domains (a variable cannot be bound to two types)
Rust supports generic parameters, and implements *monomorphization* for resolving them. This means that the compiler will generate separate code for each type that instantiate a generic type. Compilation time and the size of the resulting binaries size increases significantly. Rust supports *ad hoc polymorphism* through the definition of *Traits* and *Trait bounds*. *Traits* and *Structs* allow for the definition of Classes and the former enables Inheritance and dynamic dispatching, common idioms in Object Oriented Programming (OOP). This support to OOP is based on the principle of *Zero Cost Abstraction* that constitutes one of Rust's main selling points. This principle allows rust to achieve OOP-like behavior, keeping the cost coming from this abstractions negligible.
Thanks to the features described so far, Rust provides support for very efficient and reliable programming. On top of this Timely Dataflow defines its computation providing an environment for data driven processing.

Appendix B

Datalog

B.1 Datalog

Reasoning procedures on knowledge Bases are defined as a Datalog Program. A Datalog Program [39] is a set of rules in the following form:

$$R_1(w_1) \leftarrow R_2(w_2), R_3(w_3), ..., R_{\mathrm{n}}(w_{\mathrm{n}})$$

Where:

- $\mathrm{R_i(w_i)}$ is called a literal.
- $\mathrm{R_i}$ is the predicate.
- $\mathrm{w_i}$:= $\mathrm{t_1}$, ...,$\mathrm{t_m}$ is a tuple of terms.

A term $\mathrm{t_i}$ is either a variable or a constant term. In our case, the constant term is a RDF resource. And the literals will represent triple patterns: $\mathrm{T(t_s,\ t_p,\ t_o)}$. The left hand side of a Datalog rule is referred to as the Head of the rule and the right hand side is called the Body of the rule. Datalog imposes one important constraint:

$$\forall v \in var(R_1(w_1)) \text{ there must be an } i \in (2..n) \text{ such that } v \in var(R_{\mathrm{i}}(w_{\mathrm{i}}))$$

where the term $var(R_{\mathrm{i}}(w_{\mathrm{i}}))$ refers to the set of variables of a literal. In other words, for each variable in the Head of the rule, there must be a literal in the body that contains that variable. These guarantees the correct characterization of a the solution, without the presence of free variables.
Literals that contain only constant terms are called facts. Datalog distinguishes between extensional predicate symbols (*edb*), that never appear in the Head of a rule so they are defined by facts, and intensional predicate symbols (*idb*) that can appear in the head of some rules, and so they are defined by rules.
In our scenario, we deal with only one predicate (T) that represents triple patterns. The *immediate consequence operator* T_P of a Datalog Program P maps a database J, to another one $\mathrm{T_P(J)}$ containing all facts produced as a direct consequence for J and P.
A fact f is said to be a *direct consequence* for J and P if it either belongs to the database

J or there is a rule in P such that f is instantiated by it and each literal in the body is instantiated by a fact that belongs to J. These definitions allow us to formally define the concept of *materialization*.
We first define a useful properties of the operator T_P:
T_P is *monotonic*: for each database I and J, $I \subseteq J$ implies $T_P(I) \subseteq T_P(J)$
This last property leads to the fact that executing the T_P operator repeatedly over augmented versions of a database J will eventually stop producing new derived facts. Being:

$$T\,{}_P^n(J) = \begin{cases} J & \text{n} = 0 \quad \text{(B.1)} \\ T_P(T\,{}_P^{n-1}(J)) & \text{n} > 0 \quad \text{(B.2)} \end{cases}$$

Then, there exist a N such that $T\,{}_P^{N+1}(J) = T\,{}_P^N(J)$. In these terms, $T\,{}_P^N(J)$ is called the fix-point of T_P and it is denoted with $T\,{}_P^\omega(J)$. The *materialization* of J is equivalent to $T\,{}_P^\omega(J)$
To compute $T\,{}_P^\omega(J)$, two are the evaluation algorithms that are very well known in this context:

- *Naive evaluation*: starting from the original database J (n = 0), execute T_P, increase n and repeat the execution on the derived database, until no new derivations are derived (fix-point reached). This approach is clearly inefficient as executing T_P at each iteration would recompute all the derivations that were derived in the previous iterations too.

- *Semi-naive evaluation*: a rule r is instantiated only if it uses at least one fact that was derived in the previous iteration. This considerably decreases the amount of facts re-derived, thus, sensibly enhancing the performance of the evaluation.

Appendix C

DynamiTE

C.0.1 System Workflow

DynamiTE implements three main tasks:

- Full materialization of the input dataset J to obtain T $^{\omega}_{P}$(J).
- Materialization maintenance against a set of triples to be added δ^{+}: T $^{\omega}_{P}$(J $\cup\delta^{+}$)
- Materialization against a set of triples to be deleted δ^{-}: T $^{\omega}_{P}$(J $\setminus\delta^{-}$)

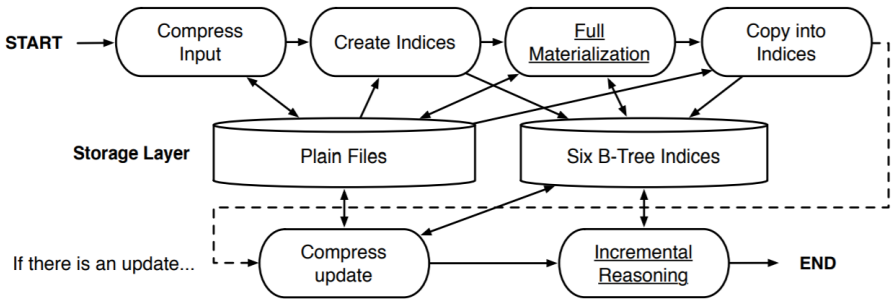


Figure C.1: Dynamite workflow as described by the authors in [7]

Figure C.1 shows how the system performs these steps. As we can see, the system utilizes two storage structures: plain files and B-tree indices. The reason behind this design choice is that DynamiTE aims at dealing with data that can become too large to fit in main memory. The system is able to perform generic querying and it is supported by six on disk B-Trees aimed at storing all the permutations of input data.
In the workflow we see the following components:

- Compress Input: the input data comes into the system encoded with the N-Triples serialization technique. In this part, the input strings are converted to numbers using the Dictionary Encoding technique, implemented through a distributed version of the MapReduce [8] framework.

- Create Indices: the compressed data is stored in the B-Trees together with the dictionaries used for compression.
- Full Materialization: in this step the materialization is computed. The generic triples are stored in plain files, because B-Trees are not efficient enough in supporting the operations required in this step. Schema triples on the other hand are always replicated on the B-Trees to improve their retrieval.
- Copy into Indices: copies the full materialization computed in the previous step into the B-Trees.

At this point, in the presence of updates it:

- Compress Update: compresses the update triples the same way as in the first step.
- Incremental Reasoning: incremental reasoning algorithms are performed.

The materialization procedures are based on three types of rules. These types differ from the type and number of triples they are made of:

- *Schema Triples* are all the triples that have SPO, SCO, DOM or RANGE as a predicate.
- *Generic Triples* are all the other triples.

The implementation of the materialization algorithms are based on the assumption that schema triples are significantly smaller than he rest of the triples. With this distinction, DynamiTE identifies rules of:

- Type 1 rules are the rules that contain only schema triples. These rules have effect only on the Ontology (T-box) of the knowledge base.
- Type 2 rules contain only one literal in their body, and they can be instantiated by generic triples
- Type 3 rules have exactly two literals in their bodies and one of then can be instantiated exclusively by schema triples and the other exclusively by generic triples.

Each rule type has a different instantiation strategy. Refer to [7] for a description of how the instantiation is performed. DynamiTE uses this distinction in the computation of the full materialization of the database. The iteration starts with rules of type 1. Each rule is instantiated in a separate thread, each of which concurrently access the B-Trees to retrieve the data to instantiate the literal in the head. The output literals are stored in both B-Trees and files. Next, DynamiTE evaluates rules of Type 2 and 3 that deal with generic triples and so they require to scan the whole input. This last operation is performed through dividing the input data into smaller files each of which accessed by a different thread. Each thread, then, instantiates Type 2 Rules. Rules of type 3 are then generated, taking into considerations both the original input and the data output of the

previous step. This is performed using the MapReduce framework. This iterations are performed following the Datalog semi-naive evaluation, so only rules with at least one literal instantiated by a triple derived in the previous iteration are considered. To allow this, the system has to keep track of the iteration when a fact was derived. It does that marking the triples with a *step* attribute. When the materialization is completed, the resulting data is copied into the B-Trees for efficient querying.

C.0.2 Incremental Evaluation: additions

Once the full materialization is complete, in case of additions in the input dataset:

- Load the update set δ in memory
- Perform semi-naive evaluation, where a rule is instantiated only if it uses at least one fact in δ
- Add the obtained triples into the indices so that they are available for querying.

The semi-naive evaluation updates the set of updates δ every iteration adding the set of newly derived facts. Moreover, now the updates are stored in main memory while the materialization is stored in B-Trees indexes, so rules of type 1 read schema triple both from main memory (δ) and B-Tree indexes. Rules of type 2 read only generic triples from the update set in memory. Rule of type 3 are split in three different blocks, to allow a more efficient memory read, accessing only triples that can produce new derivations from the B-Trees. This is done by fetching one triple in δ and then retrieving only matching

- schema triples fetched from B-Trees, generic triples from δ in memory.
- both schema and generic triples from δ in memory.
- schema triples from δ in memory and generic triples from B-Trees

These cases cover all the possible updates that can produce new updates. Once the threads run concurrently, the system waits for them to join and removes duplicate. Then, it continues to iterate until the fix point is reached.

C.0.3 Incremental Evaluation: deletions

In Section 2.4, we introduced the topic of Incremental Reasoning. If the triples in δ represent deletions, then we need to delete from the dataset all the triples in δ and all the triples that cannot be derived in J $\setminus\delta$. DynamiTE implements two algorithms for deletions, the DRed (see Section 2.4) and Counting algorithm.

Counting Algorithm

The Delete phase computes an overestimation of the data to be deleted, for this reason we need to re-derive the overestimated deletions. This represents the main performance issue of the algorithm. To cope with this fact, DynamiTE introduce the Counting

Algorithm. This aims at keeping track of the possible derivations of a triple to early identify which ones have alternative derivations and, by contrast what triples to safely delete from the dataset. To allow this, all the triples are annotated with an additional attribute called *count*. The count attribute corresponds to the number of rules that instantiated the triple as a *direct consequence* (incremented by one, if present in the input). In this way, in the Delete phase we compute the derived triples and decrease their count by 1. If the count goes to zero, then it means that that triple could only be instantiated through that rule, hence it can be safely deleted. If a triple had count 0, it would mean that there exists a rule that uses (maybe another) so it would not be safe to delete. This algorithm allows us to skip the Rederive phase.
To implement this algorithm, we need to add a spacial and temporal overhead for managing the counts values. The additional cost of this overhead is less significant that the Rederive phase and quickly amortized after few updates.

Appendix D

Timely and Differential Dataflow

D.1 Timely Dataflow Basics

Unlike standard Dataflow implementation, Timely Dataflow is designed to assign to each worker the whole view of the dataflow. The crate *timely* provides (re-exports) functions to set up the environment in which a user-defined number of workers are defined. The most common one is shown in Listing D.1:

```
timely::execute_from_args(std::env::args(), |worker| {
  // per-worker logic
})
```

Listing D.1: *execute_from_args() function*

This function starts a Timely Dataflow computation from configuration information and per-worker logic. The configuration information are provided by the first parameter that represents the Rust Iterator over command line arguments. These arguments are followed by the number of workers and can assume the following values:

- *-w*: number of per-process threads
- *-n*, *-p* : number of processes involved in the computation used in combination with *-p* to specify the index of the current process with respect to the *-n* defined.
- *-h*: a text file that represents all the host names in the form "hostname:port" in order of process identity for distributed computation

The per-worker logic is given through a closure. The closure contains the dataflow and the data management that each worker is going to share and takes as a parameter the worker itself. This parameter provides useful information as shown in Listing D.2. The *peers()* function returns the total number of workers; The *index()* function returns the worker identifier among all the workers defined and so it ranges in the interval *0..peers*; The *timer()* function returns the timer associated to the worker and it can be polled to obtain the execution time up to that point.

```
timely::execute_from_args(std::env::args(), |worker| {
  let peers = worker.peers();
  let index = worker.index();
  let timer = worker.timer();
})
```

Listing D.2: functions to retrieve important information about the worker thread

The per-worker logic is logically divided in two parts. In the first part, the dataflow diagram is generated. Listing D.3 shows the implementation.

```
// --snip--
let input = worker.dataflow::<u64,_,_>(|scope|{
  let (input, stream) = scope.new_input();
  stream
      .exchange(|&x| x)
      .inspect(|x| println!("w: {:?}\t{:?}", index, x));

  input
});
```

Listing D.3: definition of a dataflow computation

The dataflow is created through the $dataflow ::< u64, \_, \_ > ()$ function on the worker. This notation is called "Turbofish" notation and it is used to specify the concrete types on generic functions and it basically says that the worker is going to create a dataflow that uses *64bit unsigned* integers as logical timestamps to support data-driven computation. This function takes a scope as a parameter and returns a generic type R. In our case the returned value is the input handle that the worker is going to use to supply data into the stream. The scope can be seen as the canvas, or context in which the dataflow diagram is drawn. Inside the dataflow closure we define the input stream and the input handle through the *new_input()* function called on the scope that returns a tuple. After that the dataflow diagram is specified. In this example the dataflow consists in the *exchange()* operator followed by the *inspect()* operator. The *exchange()* operator shuffles data in between workers based on the value given in the closure, modulo the number of workers. This is very useful in the cases where the computation requires particular data to be located in the same worker for the correct execution of the functionalities. The *inspect()* operator takes a closure and applies the code in the closure to all the elements in the stream. In this example we use it to print to stdout all the elements seen by the workers. The function ends returning the input handle. Timely Dataflow provides the feature of progress tracking. Each worker can learn about the possibility of timestamped data at a certain location in the dataflow graph. This is done with the *probe()* operator show in Listing D.4.

```
// --snip--
let (input, probe) = worker.dataflow::<u64,_,_>(|scope|{
  let (input, stream) = scope.new_input();
  let probe =
  stream
      .exchange(|&x| x)
      .inspect(|x| println!("w: {:?}\t{:?}", index, x))
      .probe();
  (input,probe)
});
```

Listing D.4: Dataflow computation example with probe

The *probe()* operator returns a *ProbeHandle* that can be used to learn about the progress the computation has made in the dataflow. In the example so far, both the input and the probe handles are not used. The dataflow does not contain any data yet so it is not performing any computation. Data can be supplied into the stream in the second part of the worker's closure.

```
// dataflow definition as in the previous listing
for round in 0..10 {
    input.send(round);
    input.advance_to(round+1);
    while probe.less_than(input.time()) { worker.step(); }
}
```

Listing D.5: How data is fed into the dataflow

Listing D.5 shows how one can provide data into the stream. Each worker, in this case, sends every number between 0 and 9 into the stream by calling the *send()* function on the input handle. The input handle has a timestamp associated with it and each data get associated to that timestamps when sent into the stream. Listing D.6 shows how progress is tracked in the framwork. The *advance_to()* method restricts the timestamps we can use to those greater or equal to the time provided as parameter. This function represents the moment in the computation where progress is made.

```
input.advance_to(round+1);
while probe.less_than(input.time()) {
  worker.step();
}
```

Listing D.6: Progress tracking

In fact, our program reveals to the system that soon we can consider complete a certain time (records bearing that timestamp might be flowing in the dataflow graph). Next, we consult our probe that was put at the end of the stream using the *less_than()* method. This method returns true if it is possible to see a time less than the argument in the location the probe was inserted. So the while statement, says that, as long as there can be data with timestamp less than the input time we can call the *step()* function on the worker. This function schedules the operators that are then executed moving the data through the dataflow.

D.2 Differential Dataflow Basics

Differential Dataflow [35] is built on top of Timely Dataflow, so it borrows most of the infrastructure that allowed us to define the examples in the previous section. This leads to the fact that differential dataflows are defined in the same way timely dataflows are defined and very similar is the progress tracking and worker scheduling. However, they bring major changes too. In Timely Dataflow the data model that the framework uses is a stream of timestamped data. In Differential Dataflow data is modeled as a Collection containing *(data, time, diff)* elements. The *data* and *time* items are the same as in Timely Dataflow, the *diff* item represents their frequency in the dataset. Common values of *diff* are +1 for data additions and -1 for data deletions. So any data in a differential data is considered as an addition or deletion into the dataflow.
To define a collection, Differential Dataflow implements function like *new_collection()* to define a pair of input handle and collection or *as_collection()* that converts a Timely Dataflow stream of an appropriate record type *(data, time, diff)* into a Differential Dataflow collection. This can be used the same way Timely Dataflow *new_input()* is used.
Differential Dataflow extends timely dataflow operators with idioms that come from the MapReduce and SQL framework. Some of these operators are:

- *Map*: maps a collection to another collection applying a supplied function to each element of the first collection.
- *Reduce*: takes an input collection in the form of *(Key, Value)* and it applies a reduce closure defined by the user.
- *Filter*: applies the supplied condition to all elements of a collection, retaining only those elements that evaluate the condition to true.
- *Join*: takes in input two input collections that have data in the form of *(Key, Value)* and produces in output a collection with *(key, (Value1, Value2))* records
- *Concat*: concatenates two collections that have the same data type. In case of duplicates it sums their *diff*
- *Consolidate*: makes sure that every record of a collection has only one corresponding physical tuple, adding the *diffs*. It does not change the input collection.
- *Iterate*: this operator takes an input collection and a closure. It iteratively runs the closure on the input collection until the computation converges once the fixpoint has been reached.
- *Arrange*: returns an Arrangement of the collection. Arrangements control how data are stored, allowing the user to specify how to index data. This allows for an efficient computation of operators like *Join*.

Another difference is represented by the API used to insert data into the dataflow: instead of using the method *send()*, Differential Dataflow InputSession supports the following functions:

- *insert(item)*: increment the *diff* of the item at the current time of the input.
- *remove(item)*: decrement the *diff* of the item at the current time of the input.
- *update(item, diff)*: support an arbitrary change to an item (either positive or negative) of an arbitrary magnitude for
- *update_at(item, time, diff)*: change to an item at a time in the in present or in the future with respect to the input time of an arbitrary magnitude.

This is due to the nature of the input data as *Collections*.