9th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2013)

At the 12th International Semantic Web Conference (ISWC2013), Sydney, Australia, October, 2013

SSWS 2013 PC Co-chairs' Message

SSWS 2013 is the ninth edition of the successful Scalable Semantic Web Knowledge Base Systems workshop series. The workshop series is focussed on addressing scalability issues with respect to the development and deployment of knowledge base systems on the Semantic Web. Typically, such systems deal with information described in Semantic Web languages such as OWL and RDF(S), and provide services such as storing, reasoning, querying and debugging. There are two basic requirements for these systems. First, they have to satisfy the applications semantic requirements by providing sufficient reasoning support. Second, they must scale well in order to be of practical use. Given the sheer size and distributed nature of the Semantic Web, these requirements impose additional challenges beyond those addressed by earlier knowledge base systems. This workshop brought together researchers and practitioners to share their ideas regarding building and evaluating scalable knowledge base systems for the Semantic Web.

This year we received 10 submissions. Each paper was carefully evaluated by four workshop Program Committee members. Based on these reviews, we accepted seven papers for presentation. We sincerely thank the authors for all the submissions and are grateful for the excellent work by the Program Committee members.

October 2013

Thorsten Liebig
Achille Fokoue

Program Committee

Mihaela Bornea
IBM Watson Research Center, USA

Oscar Corcho
Univ. Politecnica de Madrid, Spain

Achille Fokoue
IBM Watson Research Center, USA

Raúl García-Castro
Univ. Politecnica de Madrid, Spain

Volker Haarslev
Condordia University, Canada

Pascal Hitzler
Wright State University, Ohio, USA

Anastasios Kementsietsidis
IBM Watson Research Center, USA

Pavel Klinov
Ulm University, Germany

Spyros Kotoulas
IBM Watson Research Center, USA

Adila A. Krisnadhi
Wright State University, Ohio, USA

Thorsten Liebig
derivo GmbH, Germany

Ralf Möller
Hamburg Univ. of Techn., Germany

Jeff Z. Pan
University of Aberdeen, UK

Bijan Parsia
University of Manchester, UK

Padmashree Ravindra
North Carolina State University, USA

Mariano Rodriguez
Free University of Bolzano, Italy

Sebastian Rudolph
Karlsruhe Inst. of Techn., Germany

Takahira Yamaguchi
Keio University, Japan

Additional Reviewers

Norman Heino
Leipzig University, Germany

Amit Joshi
Wright State University, Ohio, USA

Raghava Mutharaju
Wright State University, Ohio, USA

Andrea Reale
University of Bologna, Italy

Yuan Ren
University of Aberdeen, UK

Martin Rezk
Free University of Bolzano, Italy

Kejia Wu
Condordia University, Canada

Table of Contents

Count Aggregation in Semantic Queries

Bogdan Kostov, Petr Křemen

Department of Cybernetics, Czech Technical University,
Prague, Czech Republic
{bogdan.kostov,petr.kremen}@fel.cvut.cz

Abstract. In this paper we study the *distinct count* aggregation function used in queries into expressive ontologies. The main differences in this settings opposed to aggregation in relational database systems are the Open World Assumption and incomplete knowledge. We propose different interpretations useful in different practical use-cases of the *distinct count* function, i.e. *basic count*, *semantic count*, *epistemic count* and *semantic tuple count* some of which use knowledge derived from the ontology in order to obtain results in accordance with the Open World Assumption. We use *interval semantics* to model the uncertainty of the *distinct count* function's result induced by incomplete knowledge in the ontology. We show that *interval semantics* are particularly useful in aggregate queries with filtering clause and when we need the retrieval of the boundaries of the uncertainty of the *distinct count* function's result. We study and present relationships among the different interpretations and decidability of the *semantic tuple count*. We also propose a theoretical approximation of the *semantic tuple count*. Our results are applicable to a wide range of description logic formalisms allowing to express equality/inequality between individuals, concepts and relations, e.g. Web Ontology Language.

Keywords: semantic counting, count function, OWL, semantic query

1 Introduction

Aggregation functions are an important feature of modern query languages. Aggregation is well understood in query languages of database systems with Unique Name Assumption (UNA) principle, e.g. SQL for relational databases. Although this simple well-known interpretation of aggregation can in some cases be used correctly in queries over semantic knowledge bases, in general this interpretation will return incorrect results as it will not use knowledge inferred from the knowledge base with the Open World Assumption (OWA). As opposed to that, semantic aggregation relies on the knowledge inferred from the knowledge base in order to provide correct result of aggregation functions.

Currently the research filed of semantic aggregation is in its infant state. However the need of semantic aggregation is becoming apparent as more and more applications of semantic technologies emerge. In this paper we propose the need of different interpretations of the *distinct count function* in the settings

of OWA and ontology representing incomplete knowledge, i.e. *basic count* ($\mathcal{BC}$), *semantic tuple count* ($\mathcal{STC}$), *epistemic count* ($\mathcal{EC}$) and *semantic count* ($\mathcal{SC}$). We study the relationships among the different interpretations. We also investigate the decidability of the $\mathcal{STC}$ interpretation.

2 Related Work

In recent years, expressive query languages, like SPARQL-DL [1], OWL-SAIQL [2], SQWRL [3] for OWL 2 [4] or SeRQL [5], SPARQL [6] for RDF, have been introduced and implemented in the field of semantic web. There have been deep studies [7–11] evaluating conjunctive queries in RDF and OWL, but few efforts have been spent on an algebra, as well as aggregation functions.

Recently the RDF query language SPARQL [6] has been extended towards the new SPARQL 1.1 [1] [12] including many new constructs, e.g. aggregation functions or negation as failure. There are already publicly available implementations, e.g. ARQ [13], KGRAM [14], RDF::Query [15] or Sesame [16]. Independently on SPARQL 1.1 authors of [17] discuss the topic of aggregation over data structured as RDF graphs rather than on the relational data returned by the query (the result set table). The authors stress the need of different modes of aggregation. A few years ago the SQWRL query language for OWL [3] was proposed. All these efforts implement or interpret aggregation using the basic relational semantics, i.e. the result of aggregation functions is computed over the results of the non-aggregate variant of the query and neglect the impact of the interplay between aggregation functions and the inferred domain elements.

More closely related to our study is the work [18]. It shows that *exact semantics*, of aggregate queries over a *DL-Lite*$_A$ ontology returns results only if the Abox is not empty and if axioms in the Tbox resolve as constraints over cardinalities of the *groups* in the aggregate query. The authors propose *epistemic semantics* of aggregate queries in the settings of data integration use-cases returning the aggregate function's results known from (inferred by) the ontology. The authors propose an evaluation algorithm for a sub set of aggregate queries, i.e. restricted epistemic aggregate queries, defined using functional dependency of query variables w.r.t. the Tbox of the queried ontology.

Although not exactly in the same framework of query answering like in this paper, the authors in [19] propose an approach using ontology design pattern for incorporating a quantification over types without actually using explicit numerical information.

In this paper we examine the *distinct count* aggregation function, its possible interpretations and their use-cases. We focus our research on the *distinct count* aggregation function alone as it is non-trivial but not too overly complicated. We believe that better understanding of the *distinct count* function and its possible interpretations will cover most of the peculiarities of aggregation in the context of expressive knowledge bases assuming the OWA principle and incomplete knowledge, thus contributing to OWL and RDF.

[1] SPARQL 1.1 version recently became a W3C recommendation.

3 Motivation

In this section we present a simple ontology which describes a simple taxonomy of teachers categories and contains assertions about teachers and courses. The ontology is presented using the well known description logic syntax, see [20].

Example 1. A simple example ontology $\mathcal{O}_1$ about teachers and the courses they teach.
Tbox
$\texttt{BusyTeacher} \sqsubseteq \texttt{Teacher}$, $\texttt{Professor} \sqsubseteq \texttt{Teacher}$, $\exists \texttt{teaches} \cdot \texttt{Course} \sqsubseteq \texttt{Teacher}$, $\texttt{BusyTeacher} \sqsubseteq \geq 3\,\texttt{teaches}$, $\texttt{Professor} \sqsubseteq \leq 3\,\texttt{teaches}$
Abox
$\texttt{BusyTeacher}(\texttt{Sara})$, $\texttt{Professor}(\texttt{Steve})$, $\texttt{Professor}(\texttt{John})$, $\texttt{BusyTeacher}(\texttt{John})$, $\texttt{Course}(\texttt{math})$, $\texttt{Course}(\texttt{physics})$, $\texttt{Course}(\texttt{history})$, $\texttt{teaches}(\texttt{Dave}, \texttt{math})$, $\texttt{teaches}(\texttt{Dave}, \texttt{physics})$, $\texttt{teaches}(\texttt{Dave}, \texttt{history})$, $\texttt{teaches}(\texttt{Sara}, \texttt{history})$, $\texttt{math} \neq \texttt{history}$

We will use the following two aggregate queries throughout the paper to demonstrate the differences in the results of the individual interpretations of the *distinct count* function.

Q_1 - Find all teachers and the number of distinct courses they teach.
Q_2 - Find all teachers that teach more than one distinct courses.

Next we will discuss the need of different interpretations of the *distinct count* function. The most natural interpretation of the *distinct count* function is the *semantics count* interpretation. This interpretation enables users to query for the *distinct count* function value or its constraints as entailed by the queried ontology. For example using this interpretation in query Q_1 we obtain that `John` teaches exactly three courses or using it in query Q_2 we will obtain that `Dave`, `Sara` and `John` teach more than one course. We call this the *semantics count* interpretation and we consider it suitable for knowledge retrieval oriented use-cases. This interpretation is a natural extension of the certain answer semantics for the *distinct count* function and it is monotonic.

Although that is the most natural and in fact correct extension of the semantics of the *distinct count* function, there are some use-cases that need different CWA interpretations. As argued in [18] for ontology based data access (OBDA) use-cases, the certain answer semantics for aggregate queries is not practical as they return trivial results, e.g. empty or very restricted result sets. The authors propose that in this context a practical interpretation will be the one that returns the least known number of courses they teach. This is called the *epistemic count* interpretation. As opposed to the *semantic count*, the result set of query Q_1 with the *epistemic count* interpretation will contain for example that `Dave` and `Sara` teach respectively two and three courses.

Note that the *epistemic count* interpretation may count both named and unnamed entities. This makes the use of this interpretation inappropriate in purely data-centric ontology applications. In [21] the authors use OWL to model

integrity constraints (IC) and propose IC CWA semantics to enable instance data validation. We propose an extension to this semantics for the *distinct count* function. We call this the *semantic tuple count* interpretation. As opposed to the previous interpretations the result of Q_2 with the *semantic tuple count* interpretation will contain only `Dave` as he is the one for which the data in the ontology $\mathcal{O}_1$ satisfies the condition in the query. The *semantic tuple count* can be used to depict IC for n-ary relations.

The most common interpretation of the *distinct count* function is the *basic count* interpretation. This interpretation is scalable and it is safe to be used in data oriented use-cases in CWA and UNA systems. The usage of this interpretation in applications assuming OWA may return incorrect results, however, it can still be used as an approximation. For example the result of query Q_1 contains the answer `Dave` teaches three courses, which may or may not be true according to the ontology.

We continue with the formal definition of the different interpretations of the *distinct count* function and the discussion of the results of queries Q_1 and Q_2 in these interpretations.

4 Preliminaries

In this section we will define basic terms and notions used in the rest of the paper. We will start with the definition of ontology followed by the definition conjunctive queries and aggregate queries with *distinct count* function.

4.1 Ontology

Definition 1. *An ontology $\mathcal{O}$ is a pair $\langle S, A\rangle$, where S is a signature and A is a set of axioms. The semantics of ontologies use a first order interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is an interpretation domain and $\cdot^{\mathcal{I}} : S \to \Delta^{\mathcal{I}}$ is an interpretation function mapping elements from the ontology signature S to elements from the interpretation domain $\Delta^{\mathcal{I}}$. An ontology $\mathcal{O}$ is satisfied by an interpretation $\mathcal{I}$, denoted by $\mathcal{I} \models \mathcal{O}$, if all of its axioms are satisfied by the interpretation $\mathcal{I}$, such interpretation $\mathcal{I}$ is called a model of $\mathcal{O}$. We say that a set of axioms A is entailed by the ontology $\mathcal{O}$, denoted by $\mathcal{O} \models A$, if every model $\mathcal{I}$ of the ontology $\mathcal{O}$, is also a model of A, $\mathcal{I} \models A$.*

Next we define the notion of the monotonic extension $\mathcal{O}'$ of the ontology $\mathcal{O}$.

Definition 2. *We say that $\mathcal{O}' = \langle S', A'\rangle$ is an extension of $\mathcal{O} = \langle S, A\rangle$ if $A \subset A'$ and $S \subset S'$. $\mathcal{O}'$ is monotonic if the original ontology $\mathcal{O}$ is entailed by $\mathcal{O}'$, $\mathcal{O}' \models \mathcal{O}$. A model $\mathcal{I}' = (\Delta^{\mathcal{I}'}, \cdot^{\mathcal{I}'})$ of the ontology $\mathcal{O}$ is an extension of the model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}) \models \mathcal{O}$ if $\Delta^{\mathcal{I}'} \subseteq \Delta^{\mathcal{I}'}$ and $\cdot^{\mathcal{I}'} \subseteq \cdot^{\mathcal{I}}$.*

For a full description of the syntax and semantics of different description logic formalisms see [20]. Note that our discussion is also applicable for OWL 2 ontologies [22, 23] since OWL 2 is backed by the $\mathcal{SROIQ}(\mathcal{D})$ description logic.

We consider description logics which allow only monotonic extensions. In section 5.2 where we prove decidability of the *semantic tuple count* we further restrict to a subfamily of description logics that enable expressing whether two entity objects are different, the same or it is not known, which we will refer to as *equality/inequality relation*. We require the OWA assumption over the *equality/inequality relation* because it provides means to model incomplete knowledge, and it is the fundamental source of uncertainty of the *semantic tuple count* function's value.

4.2 Conjunctive Queries

The discussion in this paper about *distinct count* function and its proposed interpretations is set in the context of conjunctive queries.

Definition 3. *We will denote conjunctive queries using the following rule like notation*

$$Q(\bar{x}) \leftarrow \phi(\bar{x}, \bar{z}). \tag{1}$$

The head of the query $Q(\bar{x})$ denotes the name of the query and the result variables $\mathcal{R}_{var}(Q) = \bar{x}$. The body of the query $\phi(\bar{x}, \bar{z})$ is a comma separated list of query atoms interpreted as a conjunctive query, as defined in the SPARQL-DL [2] query language. The query atom list ϕ may contain non result variables $\bar{z}$. Note that the result variables must be distinguished. By $\mathcal{V}_{var}(Q)$ we denote the list of all variables in the query. A binding $\mu : \mathcal{V}_{var}(Q) \rightarrow S$ is a mapping of the variables of the query to elements in the ontology's signature and $Q|_\mu$ is the substitution of the variables in Q by the binding μ. The binding substitution of a tuple of variables $\bar{v} = (v_1, \ldots, v_k)$ is denoted by $\mu(\bar{v}) = (\mu(v_1), \ldots, \mu(v_k))$. We call Q a ground query if there are no variables in the query. A solution to the query Q w.r.t. the ontology $\mathcal{O}$ is a binding μ for which the substitution $Q|_\mu$ is a ground query, the body of which is entailed by the ontology (denoted by $\mathcal{O} \models Q|_\mu$). The set of all possible solutions of Q w.r.t. $\mathcal{O}$ or the model $\mathcal{I}$ of $\mathcal{O}$ is denoted by $Sat_Q^{\mathcal{O}} = \{\mu | \mathcal{O} \models Q|_\mu\}$ or $Sat_Q^{\mathcal{I}} = \{\mu | \mathcal{I} \models Q|_\mu\}$ respectively . The result set of query Q w.r.t. $\mathcal{O}$ denoted by $Q_{\mathcal{O}}$, is a set of bindings of the result variables $\mathcal{R}_{var}(Q)$, formally $Q_{\mathcal{O}} = \{\bar{a} | \bar{a} = \mu(\mathcal{R}_{var}(Q)) \wedge \mu \in Sat_Q^{\mathcal{O}}\}$.

4.3 Aggregate Queries

Here we define the types of aggregate queries along with their simple syntax used for the purpose of representing aggregate queries in this paper. We also define some additional terminology and symbols used later in the paper.

Definition 4. *We will denote distinct count retrieval and distinct count filtering queries respectively as follows:*

$$Q_a(\bar{x}, \mathsf{count}^{\mathcal{S}}_{\mathsf{dist}}(\bar{y})) \leftarrow \phi(\bar{x}, \bar{y}, \bar{z}) \tag{2}$$

[2] See [1] for list of atoms and their interpretations.

$$Q_{ac}(\bar{x}) \leftarrow (\cdot_{\mathsf{op}} n \mathsf{count}^{\mathcal{S}}_{\mathsf{dist}}(\bar{y})), \phi(\bar{x}, \bar{y}, \bar{z}) \tag{3}$$

In the queries of the type Q_a, the head $Q_a(\bar{x}, \mathsf{count}^{\mathcal{S}}_{\mathsf{dist}}(\bar{y}))$ specifies the disjoint sets of grouping $\mathcal{G}_{var}(Q_a) = \bar{x}$ and aggregation $\mathcal{A}_{var}(Q_a) = \bar{y}$ variables. The distinct count aggregation function which returns the number of distinct tuples according to the semantics mode specified by the superscript $\mathcal{S}$ w.r.t. the ontology $\mathcal{O}$ is denoted by $\mathsf{count}^{\mathcal{S}}_{\mathsf{dist}}$. The body of Q_a contains a query atom list.

We also consider distinct count filtering by comparison queries of the form Q_{ac} show in (3). The head of queries of these type contain only the grouping variables. The body of the query contains cardinality restriction atom[3] *where $\cdot_{op} \in \{>, <, \leq, \geq, =, \neq\}$. By Q^* we denote a non aggregate variant of Q, obtained from Q by removing its aggregate function from the head or the comparison predicate in the body. The result variables of Q^* are the union of group and aggregate variables of Q, $\mathcal{R}_{var}(Q^*) = \mathcal{G}_{var}(Q) \cup \mathcal{A}_{var}(Q)$.*

Before we define the general semantics of the result of aggregate queries we clarify and define the auxiliary terms and notations in the following three definitions.

Definition 5. *Let $\mathbb{N}^{0,\infty}$ be the extension of the set of natural numbers with zero and infinity $\mathbb{N}^{0,\infty} = \mathbb{N} \cup \{0, \infty\}$. Let L be a subset of $\mathbb{N}^{0,\infty}$ and let $\mathsf{Int}(L)$ denote the smallest interval containing L,$\mathsf{Int}(L) = \langle \inf(L), \sup(L) \rangle$. We extend the intuitive comparison between elements in $\mathbb{N}^{0,\infty}$ with comparison between sets $L \subseteq \mathbb{N}^{0,\infty}$ and elements $n \in \mathbb{N}^{0,\infty}$. $L \leq n$ ($L < n$) is be true if and only if $\sup(L) \leq n \vee \sup(L) = n = \infty$ ($\sup(L) < n$). $L \geq n$ ($L > n$) is true if and only if $\inf(L) \geq n \vee \inf(L) = n = \infty$ ($\inf(L) > n$). Note that the symbol ∞ is an element of $\mathbb{N}^{0,\infty}$.*

Next we define the interpretation a tuple and a set of tuples.

Definition 6. *Let $\mathcal{O} = \langle S, A \rangle$ be an ontology, $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ a model of $\mathcal{O}$ and T be a set of tuples composed of elements in S. The interpretation of the of the tuple $\bar{t} = (t_1, t_2, \ldots, t_k), \bar{t} \in T$ is $\bar{t}^{\mathcal{I}} = (t_1^{\mathcal{I}}, t_2^{\mathcal{I}}, \ldots, t_k^{\mathcal{I}})$. The interpretation of T is $T^{\mathcal{I}} = \{\bar{t}^{\mathcal{I}} | \forall \bar{t} \in T\}$.*

Next we define aggregate groups or simply groups as the set of tuples with common grouping variable binding.

Definition 7. *Let Q be an aggregate query of type Q_a or Q_{ac}, $\mathcal{O}$ be an ontology, $\mathcal{I}$ a model of $\mathcal{O}$ and let $\bar{k} = \mu(\mathcal{G}_{var}(Q^*))$, where μ is an arbitrary binding. Then the aggregate group, denoted by $\Gamma(\mathcal{O}, Q, \bar{k})$, with key $\bar{k}$ is equal to the set of tuples $\{\mu(\mathcal{A}_{var}(Q)) | \mu(\mathcal{G}_{var}(Q)) = \bar{k} \wedge \mu \in Sat^{\mathcal{O}}_{Q^*}\}$. The aggregate group with key $\bar{k}$ in a tuple set T is $\Gamma(T, \bar{k}) = \{\bar{a} | \forall \bar{t} \in T, (\bar{k}, \bar{a}) = \bar{t}\}$.*

[3] Note that we reuse the well known notation for property cardinality restrictions, see [20].

Note that the definition of the aggregate group $\Gamma(\mathcal{O}, Q, \bar{k})$ w.r.t. to the ontology $\mathcal{O}$ can be used also to obtain an aggregate group w.r.t. to the model $\mathcal{I}$ of $\mathcal{O}$, i.e. $\Gamma(\mathcal{I}, Q, \bar{k})$.

Next we define results of aggregate queries in terms of the counting function $f^{\mathcal{S}}_{[\mathcal{O},Q]}$ and the set $K^{\mathcal{S}}(\mathcal{O}, Q)$. Informally the counting function $f^{\mathcal{S}}_{[\mathcal{O},Q]}$ is an uncertainty aware generalization of the *distinct count* function, it takes as an input the key of the group to be counted and returns a set of possible values w.r.t. $\mathcal{O}$ and Q. The set $K^{\mathcal{S}}(\mathcal{O}, Q)$ is the set of keys of the groups to be counted. The counting function and the key of sets will be defined in each of the concrete distinct count interpretations. The superscript is used to distinguish among the different interpretations.

Definition 8. *Let $\mathbb{D}$ be the set of all tuples of arbitrary length and $\mathcal{P}(\mathbb{N}^{0,\infty})$ be the power set of $\mathbb{N}^{0,\infty}$. Let $\mathcal{S}$ be an aggregate semantic, $f^{\mathcal{S}}_{[\mathcal{O},Q]} : \mathbb{D} \to \mathcal{P}(\mathbb{N}^{0,\infty})$ be the counting function and $K^{\mathcal{S}}(\mathcal{O}, Q) \subset \mathbb{D}$ be the key set in $\mathcal{S}$. The result of an aggregate query Q of type (2) w.r.t. the ontology $\mathcal{O}$ is $Q_{\mathcal{O}} = \{(\bar{k} \in K^{\mathcal{S}}(\mathcal{O}, Q), inf(L)) | L := f^{\mathcal{S}}_{[\mathcal{O},Q]}(\bar{k}) \wedge \inf(L) = \sup(L)\}$ and the result set of an aggregate query Q with comparison filtering, i.e. queries of the form Q_{ac}, is $Q_{\mathcal{O}} = \{\bar{k} \in K^{\mathcal{S}}(\mathcal{O}, Q) | L := f^{\mathcal{S}}_{[\mathcal{O},Q]}(\bar{k}) \wedge L \cdot_{op} n\}$. The interpretation of the last condition $L \cdot_{op} n$ is defined in Definition 5.*

5 *Distinct count* function

In this section we formally define the semantics of each of the various interpretations of the counting function $f^{\mathcal{S}}_{[\mathcal{O},Q]}$ and the set of keys $K^{\mathcal{S}}(\mathcal{O}, Q)$. We present this definitions in order to be able to show formally the relations between the interpretations and in the case of $\mathcal{SC}$ and $\mathcal{STC}$ to be able evaluate comparison filtering queries in OWA. We show and discuss the results of the example queries Q_1 and Q_2 from section 3 in each of the proposed semantics. We also study the relationship between individual interpretations of the *distinct count* function. We prove decidability of the *semantic tuple count* $\mathcal{STC}$ interpretation. Finally we show how to enable distinct count queries in the context of the description logic $\mathcal{SROIQ}$.

In (4) we show the queries Q_1 and Q_2 from example 1 in the notation defined in Definition 4.

$$\begin{aligned} &Q_1(?t, \mathsf{count}_{\mathsf{dist}}?c) \leftarrow \mathtt{PropertyValue}(\mathtt{teaches}, ?t, ?c) \\ &Q_2(?t) \leftarrow (>1\mathsf{count}_{\mathsf{dist}}(?c)), \mathtt{PropertyValue}(\mathtt{teaches}, ?t, ?c) \end{aligned} \tag{4}$$

The tables in this section showing the results of the queries have the following notation. Only cells highlighted with grey background color [4] are part of the result set of the query. The other cells have white background. Note that through out this section $\mathcal{O} = \langle S, A \rangle$ is an arbitrary ontology with a signature S and an axiom set A, Q is an arbitrary aggregate query, $\mathcal{O}_1$ refers to the ontology in example 1 and Q_1 and Q_2 refer to the queries shown in (4).

[4] The use of colors in this paper is intended to be readable in gray scale copies.

5.1 Known interpretations of the *Distinct Count*

In this section we discuss three known interpretations of the *distinct count* function in aggregate queries.

***Basic Count* Interpretation** The *basic count* interpretation ($\mathcal{BC}$) is used originally in the SQL query language, but it is also used in semantic query languages, e.g. SPARQL and SQWRL. This interpretation is not adequate for ontological knowledge as it does not infer the *distinct count* function's value from the ontology. The *basic count* interpretation can be implemented in $n \log n$ time, where n is the size of the tuple set to be counted.

Definition 9. *The set of keys $K^{\mathcal{BC}}(\mathcal{O}, Q)$ is the set of all syntactically distinct result bindings of the grouping variables $\mathcal{G}_{var}(Q)$ of the query Q over the ontology $\mathcal{O}$, i.e. $K^{\mathcal{BC}}(\mathcal{O}, Q) = \{\bar{k} | \bar{k} = \mu(\mathcal{G}_{var}(Q)) \wedge \mu \in Sat^{\mathcal{O}}_{Q^*}\}$. For $\bar{k} \in K^{\mathcal{BC}}(\mathcal{O}, Q)$ the counting function is defined as $f^{\mathcal{BC}}_{[\mathcal{O},Q]} = \{|\Gamma(\mathcal{O}, Q, \bar{k})|\}$.*

The result of the distinct count query Q_1 is shown in table 1(a).

Table 1. The result set of the $\mathcal{BC}$ interpretation of the aggregate queries Q_1 and Q_2.

(a) result of Q_1

$?t$	$\mathsf{count}^{\mathcal{BC}}_{\mathsf{dist}}(?c)$
`Dave`	3
`Sara`	1

(b) result of Q_2

$?t$	$(> 1\mathsf{count}^{\mathcal{BC}}_{\mathsf{dist}}(?c))$
`Dave`	true
`Sara`	false

There are two group keys `Dave` and `Sara`. The number of courses `Dave` teaches is three because there are three courses that `Dave` teaches asserted in the ontology, whose names are syntactically different. For `Sara`, who teaches only one course according to the ontology, the result of the count function is one. We can see that this interpretation ignores the implicit knowledge derived from the fact that `Sara` is a `BusyTeacher` that teaches at least three courses.

***Semantic Count* Interpretation** Informally the *semantic count* ($\mathcal{SC}$) interpretation counts the number of possible tuples entailed by the queried ontology. This interpretation is similar to the *exact semantics* presented in [18]. However our definition of $\mathcal{SC}$ presented here does not return a single value of the distinct count function but a set of possible values. While this definition has no effect on the results of queries of type Q_a it ensures monotonic results in the queries of type Q_{ac}. The decidability of the $\mathcal{SC}$ interpretation is an open problem. Nevertheless we believe that the minimum of the $\mathcal{SC}$ interpretation is more likely to be decidable.

Definition 10. *The semantic count interpretation is denoted by $\mathcal{SC}$. Let $l = |\mathcal{G}_{var}(Q)|$ be the number of grouping variables in Q and $\mathbb{S} = S^l$ is the set of all tuples of length l composed of elements from the signature S. The $\mathcal{SC}$ counting function is defined as follows $f^{\mathcal{SC}}_{[\mathcal{O},Q]}(\bar{k}) = \{|T^{\mathcal{I}}| \, |T := \Gamma(\mathcal{I}, Q, \bar{k}), \forall \mathcal{I} \models \mathcal{O}\}$. The $\mathcal{SC}$ key set is defined as follows $K^{\mathcal{BC}}(\mathcal{O}, Q) = \{\bar{k} \in \mathbb{S} | \sup(f^{\mathcal{SC}}_{[\mathcal{O},Q]}(\bar{k})) > 0\}$.*

Next we discuss the results of the example queries Q_1 and Q_2, with formal representation shown in (4), over the ontology $\mathcal{O}_1$ from example 1. The last columns in the tables 2(a) and 2(b) show the boundaries of the intervals found for each of the group keys from the first column.

Table 2. The result of the aggregate queries Q_1 Q_2 in 4 with $\mathcal{SC}$ interpretation.

(a) result of Q_1

$?t$	$\mathsf{count}^{\mathcal{SC}}_{\mathsf{dist}}(?c)$	$\mathsf{Int}(f^{\mathcal{SC}}_{[\mathcal{O}_1,Q_1]})$
`Dave`	-	$\langle 2, \infty \rangle$
`Sara`	-	$\langle 3, \infty \rangle$
`Steve`	-	$\langle 0, 3 \rangle$
`John`	3	$\langle 3, 3 \rangle$
`math`	-	$\langle 0, \infty \rangle$

(b) result of Q_2

$?t$	$(> 1\mathsf{count}^{\mathcal{SC}}_{\mathsf{dist}}(?c))$	$\mathsf{Int}(f^{\mathcal{SC}}_{[\mathcal{O}_1,Q_2]})$
`Dave`	true	$\langle 2, \infty \rangle$
`Sara`	true	$\langle 3, \infty \rangle$
`Steve`	false	$\langle 0, 3 \rangle$
`John`	true	$\langle 3, 3 \rangle$
`math`	false	$\langle 0, \infty \rangle$

In table 2(a) we show the results of query Q_1. Next we discuss the intuition of the calculation of the values of the $\mathcal{SC}$ counting function for each of groups in table 2(a). In order to obtain the final result of the query we need to apply the comparison semantics from definition 8. The last row in both tables show an unexpected group which the algorithm should process. In fact we have two more groups that we omitted from the table which are with keys `history` and `physics`. This is an effect of the OWA assumption. This anomaly is caused by the fact that the ontology $\mathcal{O}_1$ does not explicitly state that `math`, `history` and `physics` are not teachers and that only teachers can teach and thus allowing the existence of models in which subjects teach something. Moreover the calculated interval is zero to infinity because there are no axioms which constraint it. The $\mathcal{SC}$ can be used to locate such unwanted behavior. The minimum in the first row with group key `Dave` is obtained from the two semantically distinct courses `math` and `history` that `Dave` teaches. The maximum of the first row is infinity because the ontology does not constrain the number of courses `Dave` can teach. The minimum of the second row with group key `Sara` is determined from the constraint that a `BusyTeacher` teaches at least three courses and the fact that `Sara` is a `BusyTeacher` and who teaches `history`. In this case there are two restrictions, 'at least three' and 'at least one' which constraint the minimum of courses `Sara` teaches. In such cases we should select the most specific one. Therefore the minimum of the second row's interval is three since the first restriction is more specific. The evaluation of the maximum in the second row's interval is analogous to the one in the first row. The third row's minimum is

zero because `Steve` is not constrained to teach a minimum number of courses as `Dave` and `Sara` were in the first and second rows. The maximum in the third row is derived from the fact that `Steve` is a professor and the constraint of the `Professor` class which limits its instances to teach at most three courses. Now we apply interval semantics to the first three rows that we discussed. The three rows are not included in the result of query Q_1 with $\mathcal{SC}$ semantics because according to the definition of the results of queries of type Q_a in definition 8 which states that set returned by the counting function must be singleton. The answer contains only the last row because the return value of the counting function is a singleton containing only the number three. This is true because the `John` is both a `Professor` and a `BusyTeacher` the constraints of which were already discussed.

The results of query Q_2 as well as all the relevant group keys of Q_2 are shown in table 2(b). We have discussed the interval boundaries of the intervals for groups of Q_1 and since the Q_2 has identical groups, note that $K^{\mathcal{SC}}(\mathcal{O}_1, Q_1) = K^{\mathcal{SC}}(\mathcal{O}_1, Q_2)$, we skip this explanation for query Q_2. The result of Q_2 contains all rows except for the third one with group key `Steve` that does not satisfies the comparison filter which limits the retrieval of groups with at least two distinct tuples in all models of the ontology $\mathcal{O}_1$.

We point out the importance of interval semantics in the settings of incomplete knowledge. Note that in query Q_2 we obtained results that were not present in the result of Q_1. This proves that the results of query Q_2 obtained by filtering the results of query Q_1 won't contain the full result entailed by the ontology.

***Epistemic Count* Interpretation** The *epistemic count* ($\mathcal{EC}$) interpretation is introduced in the work [18]. Informally this interpretation of the *distinct count* function returns a single value which represents the known number of distinct tuples in an aggregate group. For the number k of known tuples holds that in any model $\mathcal{I}$ of the ontology $\mathcal{O}$ there is at least k distinct tuples for the counted group. Because $\mathsf{count}^{\mathcal{EC}}_{\mathsf{dist}}$ interpretation is equivalent to the infimum of the $\mathcal{SC}$ counting function here we define the $\mathsf{count}^{\mathcal{EC}}_{\mathsf{dist}}$ interpretation in terms of $\mathcal{SC}$ counting function and key set. The decidability of the $\mathcal{EC}$ interpretation is an open problem an it is equivalent to the decidability of the minimum of the $\mathcal{SC}$ interpretation problem.

Definition 11. *The $\mathcal{EC}$ key set is defined as $K^{\mathcal{EC}}(\mathcal{O}, Q) = K^{\mathcal{SC}}(\mathcal{O}, Q)$. The $\mathcal{EC}$ counting function is defined as $f^{\mathcal{EC}}_{[\mathcal{O},Q]}(\bar{k}) = \{\inf(f^{\mathcal{SC}}_{[\mathcal{O},Q]}(\bar{k}))\}$.*

In tables 3(a) and 3(a) we can see the results of the aggregate queries Q_1 and Q_2 respectively with *distinct count* function interpreted with the $\mathcal{EC}$ semantics. The results are identical with those of the minimum of the interval in tables 2(a) and 2(b) and are not discussed further.

5.2 *Semantic Tuple Count* Interpretation

The *semantic tuple count* ($\mathcal{STC}$) interpretation is defined using interval semantics. Informally it counts the same tuples as the $\mathcal{BC}$ interpretation but it uses

Table 3. The result of the aggregate queries Q_1 and Q_2 in (4) with the $\mathcal{EC}$ interpretation.

(a) result of Q_1

$?t$	$\mathsf{count}^{\mathcal{EC}}_{\mathsf{dist}}(?c)$
`Dave`	2
`Sara`	3
`John`	3
`math`	0

(b) result of Q_2

$?t$	$(> 1\mathsf{count}^{\mathcal{EC}}_{\mathsf{dist}}(?c))$
`Dave`	true
`Sara`	true
`John`	true
`math`	false

knowledge in the ontology to derive the *equivalence/inequivalence* relation between the counted tuples which is needed to remove semantically duplicate tuples and also to deal with uncertainty of the count's value.

Definition 12. *The $\mathcal{STC}$ key set $K^{\mathcal{STC}}$ is defined as the key set of the $\mathcal{BC}$ interpretation, i.e. $K^{\mathcal{STC}}(\mathcal{O}, Q) = K^{\mathcal{BC}}(\mathcal{O}, Q)$. We define the $\mathcal{STC}$ counting function as follows $f^{\mathcal{STC}}_{[\mathcal{O},Q]}(\bar{k}) = \{|\Gamma(\mathcal{O}, Q, \bar{k})^{\mathcal{I}}||\forall \mathcal{I} \models \mathcal{O}\}$.*

Table 4. The result of the aggregate queries Q_1 and Q_2 in (4) with the $\mathcal{STC}$ interpretation.

(a) result of query Q_1

$?t$	$\mathsf{count}^{\mathcal{STC}}_{\mathsf{dist}}(?c)$	$\mathsf{Int}(f^{\mathcal{STC}}_{[\mathcal{O}_1,Q_1]})$
`Dave`	-	$\langle 2, 3\rangle$
`Sara`	1	$\langle 1, 1\rangle$

(b) result of query Q_2

$?t$	$(> 1\mathsf{count}^{\mathcal{STC}}_{\mathsf{dist}}(?(c)))$
`Dave`	true
`Sara`	false

Next we discuss the results of queries Q_1 and Q_2 from example 1 with the $\mathcal{STC}$ interpretation. Here we also show all the group keys that should be considered during evaluation of the $\mathcal{STC}$ interpretation and since the $\mathcal{STC}$ interpretation is interval based we also show the evaluated intervals in the last columns of the tables 4(a) and 4(b). In table 4(a) we have the results of Q_1. Now we discuss the values of the interval boundaries in rows one and two. The minimum in the first row is based on the two distinct courses `math` and `history`. The maximum is three because there are three entailed tuples in the group of the first row and because there is no other axioms that constraints the maximum to be smaller. The minimum in the second row is one because we have only one tuple for the `Sara` group. This is also the only possible value for the maximum of row two because there are no other entailed tuples in that group. Apparently from the interval semantics only the second row is returned.

The results for the query Q_2 are shown in table 4(b). We already discussed the interval boundaries in the discussion of query Q_1. Applying the interval semantics we filter the second row because `Sara` teaches only one distinct course and therefore it is not contained in the result. The first is contained in the result

because the minimum boundary, which is 2, is bigger than the less than operand in the filtering atom, which is one.

Decidability of the $\mathcal{STC}$ Interpretation In this section we prove decidability of the $\mathcal{STC}$ interpretation. We present only the statements of the most relevant lemmas and the proof of the theorem at the end of the section. The omitted proofs and lemmas can be found in the technical report [24]. First we define the family of formalisms for which we prove that the $\mathcal{STC}$ interpretation is decidable.

Definition 13. *We assume that the supported formalisms $\mathcal{F}$ (i) are capable of expressing the equality/inequality relation among elements of the ontology (ii) consistency check of ontologies and query answering is decidable in $\mathcal{F}$ and (iii) that $\mathcal{F}$ is monotonic.*

Implementing the evaluation of the $\mathcal{STC}$ counting function based on definition 5.2 is not feasible because the ontology $\mathcal{O}$ might have an infinite number of models. We show that there is a finite number of models sufficient for the calculation of the boundaries of the value of the $\mathcal{STC}$ counting function.

The next proposition states that the interval of the *distinct count* function with $\mathcal{SC}$ and $\mathcal{STC}$ interpretation w.r.t. the ontology $\mathcal{O}$ will be more specific if we extend the queried ontology. The interval calculated w.r.t. the original ontology will include the one calculated from the extended ontology.

Note that in this section we will use identical equality and inequality axioms, denoted by $=_a$ and $\neq_a$ respectively, for all type of entities, i.e. individuals,classes and properties. In the following section we show an equivalent representation of this axioms in the concrete logic $\mathcal{SROIQ}$.

Definition 14. *Let $\mathcal{O} = \langle S, A\rangle$ be an ontology, T be a tuple set composed of elements in the set $D \subseteq S$. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation such that $\cdot^{\mathcal{I}}$ is defined on D then the complete set of equality/inequality axioms satisfied by $\mathcal{I}$ is denoted as $A_{\#}(\mathcal{I}, D) = \{a =_a b | \forall a, b \in D, a^{\mathcal{I}} = b^{\mathcal{I}}\} \cup \{a \neq_a b | \forall a, b \in D, a^{\mathcal{I}} \neq b^{\mathcal{I}}\}$. We denote the set all complete sets of equality/inequality axioms between elements in D w.r.t. the ontology $\mathcal{O}$ as $\mathbb{A}_{\#}(\mathcal{O}, D) = \{A_{\#}(\mathcal{I}, D) | \forall \mathcal{I} \models \mathcal{O}\}$. The cannonic model of $A_{\#} = A_{\#}(\mathcal{I}, D)$, denoted by $\mathcal{I}_{A_{\#}} = (\Delta^{\mathcal{I}_{A_{\#}}}, \cdot^{\mathcal{I}_{A_{\#}}})$ is a model with an interpretation function $\cdot^{\mathcal{I}_{A_{\#}}}$ the domain of which is D.*

Note that adding equality/inequality axioms to the complete set $A_{\#}(\mathcal{I}, D)$ wont change the original set or if it does the resulting set is unsatisfiable. Note also that the cannonic interpretation function $\cdot^{\mathcal{I}_{A_{\#}}}$ can be constructed in polynomial time.

Lemma 1. *Let $\mathcal{O} = \langle S, A\rangle$ be an ontology, $D \subseteq S$, then $\mathbb{A}_{\#}(\mathcal{O}, D)$ is finite.* □

Lemma 2. *Let T be a tuple set composed of elements from set D and two interpretations $\cdot^{\mathcal{I}_1}$ and $\cdot^{\mathcal{I}_2}$ which agree on the equivalence and inequivalence between elements in D. Then the number of elements in the sets $T^{\mathcal{I}_1}$ and $T^{\mathcal{I}_2}$ is the same, $|T^{\mathcal{I}_1}| = |T^{\mathcal{I}_2}|$.* □

The next lemma states that for extensions of the ontology $\mathcal{O}$ with an axiom set from $\mathbb{A}_{\#}(\mathcal{O}, D)$ the $\mathcal{STC}$ counting function returns a singleton set and that the only value in the set can be calculated in polynomial time.

Lemma 3. *Let $\mathcal{O} = \langle S, A\rangle$ be an ontology, Q be an aggregate query, $\bar{k} \in K^{\mathcal{STC}}$, $T = \Gamma(Q^*_{\mathcal{O}}, \bar{k})$ and D be the set of elements composing tuples in T, $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be a model of $\mathcal{O}$, $A_{\#} = A_{\#}(\mathcal{I}, D)$, $\mathcal{O}' = \langle S, A \cup A_{\#}\rangle$ and $\cdot^{\mathcal{I}_{A_{\#}}}$ be the cannonic interpretation function of $A_{\#}$. Then $f^{\mathcal{STC}}_{[\mathcal{O}',Q]}(\bar{k}) = \{c\}$, where $c = |T^{\mathcal{I}}| = |T^{\mathcal{I}_{A_{\#}}}|$.* □

Instead of looking for the boundaries of the $\mathcal{STC}$ interval in the set of all models as the definition 12 suggests, we can search for the boundaries in the finite set of representations of the equality interpretation. We first describe an algorithm which terminates in a final number of steps and then we prove its correctness.

Algorithm 1 : STC - Semantic Tuple Count procedure

```
PROCEDURE STC
  INPUT : O  // the queried ontology,
          T  // set of tuples to be counted
  OUTPUT : <a,b> // the calculated interval
  a := inf; b := 0;
  D := {elements used in tuples in T};
  FORALL A# IN A#(O,D) DO
    A' := union(A, A#);
    IF A' is consistent
      construct cannonic interpretation I#(A#);
      c := |T interpreted by I#(A#)|;
      IF a > c THEN a := c;
      IF b < c THEN b := c;
    END-IF
  END-FORALL
  RETURN <a,b>;
END
```

We will prove decidability by proving correctness and termination of algorithm 1.

Theorem 1. *The algorithm 1 terminates and evaluates correctly the interval $\mathsf{Int}(f^{\mathcal{STC}}_{[\mathcal{O},Q]}(\bar{k}))$ for an ontology $\mathcal{O}$ and aggregate query Q and group key $\bar{k}$.*

Proof. Algorithm 1 terminates since there is a finite number of extensions to check. From lemma 3 we have that the $\mathcal{STC}$ counting function is a subset of the calculated interval. Also from lemma 3 we have that the interval is the smallest because the boundaries correspond to some model of $\mathcal{O}$.□

Corollary 1. *Calculation of the $\mathcal{STC}$ interval is decidable in the family of formalisms defined in Definition 13.* □

$\mathcal{STC}$ Interpretation approximation The proposed algorithm 1 is searching trough all the possible extensions for the set of elements D. As we show in corollary 1 there are $n = 2^{|D|^2}$ possible extensions for each of which we need to make a consistency check. In this section we propose an approximation of the $\mathcal{STC}$ interpretation which can be used for example as optimization of an evaluation algorithm. The approach presented here utilizes the knowledge inferred from the ontology. The prove of the correctness of the approximation is presented in the technical report [24].

Definition 15. *Let $G = (V, E)$ be an undirected graph with nodes V and edges E. A connected component K in G is a subgraph in G maximal with the property, for each pair of nodes in K there is a path in K. A (maximal) clique C is a graph maximal with the property C is a subgraph of G and is complete. The biggest clique C in G is called maximum.*

In order to define the approximation formally we need first to define the auxiliary term *difference graph*, and the notion of *maximum clique*.

Definition 16. *Let $\mathcal{O} = \langle S, A\rangle$ be an ontology, let T be a set of tuples of length l and composed of elements from the set $D \subseteq S$. Let $\bar{s}, \bar{t} \in T$, equality $\bar{s} =_a \bar{t}$ and inequality of tuples w.r.t. the ontology $\mathcal{O}$ is defined respectively as $\mathcal{O} \models \{s_1 =_a t_1, \ldots, s_l =_a t_l\}$ and $\mathcal{O} \models s_i \neq_a t_i$ for some $1 \leq i \leq l$. Let $G_=(\mathcal{O}, T) = (D, E_=(\mathcal{O}, T))$ be the graph with edges $E_=(\mathcal{O}, T) = \{(a, b) | \forall (a, b) \in T^2, \mathcal{O} \models a =_a b\}$. The difference graph is denoted by $\Delta G(\mathcal{O}, T) = (V, E)$ w.r.t. $\mathcal{O}$ and T. The set nodes is the set of connected components of $G_=(\mathcal{O}, T)$, there is an edge between the nodes u and v in V, $\{u, v\} \in E$ if and only if the ontology entails $\neq_a$ axiom between some pair of the set $u \times v$.*

Theorem 2. *Let $\mathcal{O}$ be an ontology, Q an aggregate query, $\bar{k} \in K^{\mathcal{STC}}(\mathcal{O}, Q)$ and let $T = \Gamma(\mathcal{O}, Q, \bar{k})$ be the group to be counted. Let the $\Delta G(\mathcal{O}, T) = (V, E)$ be the difference graph w.r.t. $\mathcal{O}$ and T and $C_{max} = (V_C, E_C)$ be the maximum clique graph of $\Delta G(\mathcal{O}, T)$. $\mathsf{Int}(f^{\mathcal{STC}}_{[\mathcal{O},Q]}(\mathcal{V}_{var}(k))) \subseteq \langle |V_C|, |V| \rangle$.* □

$\mathcal{STC}$ interpretation in $\mathcal{SROIQ}$ In this section we discuss the $\mathsf{count}^{\mathcal{STC}}_{\mathsf{dist}}$ function in a concrete description logic $\mathcal{SROIQ}$. The $\mathcal{STC}$ interpretation is decidable in this description logic since $\mathcal{SROIQ}$ satisfies all the requirements shown in Definition 13. We consider any query language which supports conjunctive queries with mixed Abox, Tbox, Rbox terms, e.g. SPARQL-DL^{NOT} [25]. In order to apply the $\mathcal{STC}$ interpretation in this scenario we need to show representations of equality and inequality between elements of the signature of the ontology which will be used to generate the complete set of axioms in $\mathbb{A}_{\#}$.

5.2 shows the representations for each of the three term types. Individuals $i_k, k \in 1, 2, 3...$, are unique and not contained in the ontology. Note that classes and properties have two possible ways of representing the inequality relation. If one of the representations fails we must also test the other when checking whether two terms of type class or property need to be compared for inequality.

Table 5. Semantics of comparison of different ontology elements

type	$=_a$	$\neq_a$
individuals	$i_1 \doteq i_2$	$i_1 \neq i_2$
classes	$C_1 \equiv C_2$	$\{C_1(i_1), \neg C_2(i_1)\}$ or $\{C_2(i_2), \neg C_1(i_2)\}$
properties	$p_1 \equiv p_2$	$\{p_1(i_1, i_2), \neg\exists p_2.\{i_2\}(i_1)\})$ or $\{p_2(i_3, i_4), \neg\exists p_1.\{i_4\}(i_3)\}$

6 Conclusion

We investigated the *distinct count* function in the context of different semantics, i.e. $\mathcal{BC}$, $\mathcal{SC}$ and $\mathcal{EC}$. We introduced a new interpretation $\mathcal{STC}$ and compared its behavior with the other interpretations. We found the following relationships between $\mathcal{SC}$ and $\mathcal{EC}$, $\inf(f^{\mathcal{SC}}_{[\mathcal{O},Q]}) = \mathcal{EC}$ and also between $\mathcal{SC}$ and $\mathcal{STC}$, $\inf(f^{\mathcal{STC}}_{[\mathcal{O},Q]}(\bar{k})) \leq \inf(f^{\mathcal{SC}}_{[\mathcal{O},Q]}(\bar{k}))$ and $\sup(f^{\mathcal{STC}}_{[\mathcal{O},Q]}(\bar{k})) \leq \sup(f^{\mathcal{SC}}_{[\mathcal{O},Q]}(\bar{k}))$. We proved decidability of the $\mathcal{STC}$ interpretation in the context of the selected family of formalisms, provided an approximation using basic graph problems and showed how to apply the $\mathcal{STC}$ in the $\mathcal{SROIQ}$ description logic.

In future work we would like to focus on the implementation of the evaluation of the $\mathcal{STC}$ interpretation into the SPARQL-DL^{NOT} [25] query language. In order to provide practical implementation research in optimizing the evaluation is needed. In the worst case algorithm 1 will issue 2^{n^2} consistency checks where n is the number of counted tuples. We are also interested in the investigation of the decidability of $\mathcal{SC}$ and $\mathcal{EC}$.

Acknowledgments This work has been supported by the grant by the grant of the Czech Technical University in Prague No. SGS13/204/OHK3/3T/13 Effective solving of engineering problems using semantic technologies. Authors also want to express thanks to the anonymous reviewers for providing useful comments during manuscript preparation.

References

1. Sirin, E., Parsia, B.: SPARQL-DL: SPARQL Query for OWL-DL. In: 3rd OWL Experiences and Directions Workshop (OWLED-2007). (2007)
2. Kubias, E., Schenk, S., Staab, S., Pan, J.Z.: OWL SAIQL - an OWL DL Query Language for Ontology Extraction. In: In Proc. of OWLED-07. (2007)
3. O'Connor, M.J., Das, A.K.: SQWRL: A query language for OWL. In: OWLED. (2009)
4. Group, W.O.W.: OWL 2 Web Ontology Language Document Overview. W3C Recommendation, W3C (October 2009) http://www.w3.org/TR/2009/REC-owl2-overview-20091027, cit. 04.12.2012.
5. Broekstra, J., Kampman, A.: An rdf query and transformation language. In Staab, S., Stuckenschmidt, H., eds.: Semantic Web and Peer-to-Peer. Springer Berlin Heidelberg (2006) 23–39
6. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation, W3C (January 2008) http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115, cit. 3.2013.

7. Horrocks, I., Kutz, O., Sattler, U.: The Even More Irresistible SROIQ. In Doherty, P., Mylopoulos, J., Welty, C.A., eds.: KR, AAAI Press (2006) 57–67
8. Sirin, E., Parsia, B.: Optimizations for Answering Conjunctive ABox Queries. In: Description Logics. Volume 189 of CEUR. (2006)
9. Křemen, P., Kouba, Z.: Conjunctive Query Optimization in OWL2-DL. In: Proceedings of the 22th International Conference on Database and Expert System Applications (DEXA 2011). Volume 6861 of LNCS., Springer Verlag (2011)
10. Glimm, B., Horrocks, I., Lutz, C., Sattler, U.: Conjunctive Query Answering in the Description Logic $\mathcal{SHIQ}$. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007). (2007)
11. Kollia, I., Glimm, B., Horrocks, I.: Query Answering over SROIQ Knowledge Bases with SPARQL. In: Proceedings of the 2011 International Workshop on Description Logic (DL 2011). (2011)
12. Seaborne, A., Harris, S.: SPARQL 1.1 Query. W3C Working Draft, W3C (October 2009) http://www.w3.org/TR/2009/WD-sparql11-query-20091022, cit. 3.2013.
13. Apache: ARQ - A SPARQL Processor for Jena, web site (April 2011) http://jena.apache.org/documentation/query/index.html, cit. 3.2013.
14. Corby, O.: Kgram: a knowledge graph abstract machine, web site http://wimmics.inria.fr/corese, cit. 3.2013.
15. Williams, G.T.: RDF Query 2.909 - RDF::Query - A complete SPARQL 1.1 Query and Update implementation for use with RDF::Trine, web site (November 2012) http://search.cpan.org/dist/RDF-Query/, cit. 3.2013.
16. Arjohn Kampman, Christiaan Fluit, J.B.: Sesame, web site (January 2013) http://sourceforge.net/projects/sesame, cit. 3.2013.
17. Seid, D.Y., Mehrotra, S.: Grouping and aggregate queries over semantic web databases. In: ICSC. (2007) 775–782
18. Calvanese, D., Kharlamov, E., Nutt, W., Thorne, C.: Aggregate queries over ontologies. In: ONISW. (2008) 97–104
19. Martínez, D.C., Janowicz, K., Hitzler, P.: A logical geo-ontology design pattern for quantifying over types. In: SIGSPATIAL/GIS. (2012) 239–248
20. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation, and Applications. In Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: Description Logic Handbook, Cambridge University Press (2003)
21. Tao, J., Sirin, E., Bao, J., McGuinness, D.L.: Integrity constraints in owl. In: AAAI. (2010)
22. Motik, B., Parsia, B., Patel-Schneider, P.F.: OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. W3C recommendation, W3C (October 2009) http://www.w3.org/TR/2009/REC-owl2-syntax-20091027, cit. 12.12.2012.
23. Patel-Schneider, P.F., Motik, B., Grau, B.C.: OWL 2 Web Ontology Language Direct Semantics. W3C Recommendation, W3C (October 2009) http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027, cit. 12.12.2012.
24. Kostov, B., Kremen, P.: Count aggregation in semantic queries - technical report. Technical report, Czech Technical University in Prague, Dept. of Cybernetics (2013)
25. Kremen, P., Kostov, B.: Expressive OWL Queries: Design, Evaluation, Visualization. International Journal On Semantic Web and Information Systems (2012) IGI Publishing. To appear in 2013.

DistEL: A Distributed $\mathcal{EL}^+$ Ontology Classifier

Raghava Mutharaju, Pascal Hitzler, and Prabhaker Mateti

Kno.e.sis Center, Wright State University, Dayton, OH, USA

Abstract. OWL 2 EL ontologies are used to model and reason over data from diverse domains such as biomedicine, geography and road traffic. Data in these domains is increasing at a rate quicker than the increase in main memory and computation power of a single machine. Recent efforts in OWL reasoning algorithms lead to the decrease in classification time from several hours to a few seconds even for large ontologies like SNOMED CT. This is especially true for ontologies in the description logic $\mathcal{EL}^+$ (a fragment of the OWL 2 EL profile). Reasoners such as Pellet, Hermit, ELK etc. make an assumption that the ontology would fit in the main memory, which is unreasonable given projected increase in data volumes. Increase in the data volume also necessitates an increase in the computation power. This lead us to the use of a distributed system, so that memory and computation requirements can be spread across machines. We present a distributed system for the classification of $\mathcal{EL}^+$ ontologies along with some results on its scalability and performance.

1 Introduction

The OWL 2 EL profile [4] is used for modeling in several domains like biomedicine,[1] sensors and road traffic [7], and herein we work on a subset called $\mathcal{EL}^+$ [1]. Even though there are not yet any existing very large ontologies in the $\mathcal{EL}^+$ profile, we can very well imagine ontologies with large ABoxes in those domains.[2] Consequently, reasoners should be able to handle very large amounts of data. And although there are some very efficient reasoners available [3, 6], there is only so much a single machine can provide for.

In this paper, we describe a distributed approach to $\mathcal{EL}^+$ ontology classification. Similar to other distributed systems, the design decisions and the performance of our distributed system, DistEL[3] involve answering the following questions effectively.

Synchronization Is synchronization among the distributed processes required? If so, how is it achieved?

Termination What is the termination condition for the distributed processes and how is it detected?

[1] `http://bioportal.bioontology.org`

[2] $\mathcal{EL}^+$ extended with ABoxes can be handled with essentially the same algorithm.

[3] The source code is available at `https://github.com/raghavam/DistEL`.

Communication How do the distributed processes communicate and how can this be minimized?

Data Duplication Is data duplication required? How many copies are maintained?

Result Collection After all the processes terminate, will the results be spread across the cluster?

Note that several other characteristics of a distributed system such as fault tolerance, transparency, etc. have been excluded since they are not yet supported by our system. In the following sections, we describe how our system solves the issues mentioned above, and show that our system can handle large ontologies.

The plan of the paper is as follows. In Section 2 we recall preliminaries concerning $\mathcal{EL}^+$. In Section 3 we describe our distributed approach. In Section 4 we present and discuss our experimental evaluation. In Section 5 we discuss limitations of our approach and future work. In Section 6 we discuss related work, and in Section 7 we conclude.

2 Preliminaries

$\mathcal{EL}^+$ Profile We present a brief introduction to the $\mathcal{EL}^+$ profile. For further details, please refer to [1]. Concepts in the description logic $\mathcal{EL}^+$ are formed according to the grammar

$$C ::= A \mid \top \mid C \sqcap D \mid \exists r.C,$$

where A ranges over concept names, r over role names, and C, D over (possibly complex) concepts. An ontology in $\mathcal{EL}^+$ is a finite set of *general concept inclusions* $C \sqsubseteq D$ and *role inclusions* $r_1 \circ \cdots \circ r_n \sqsubseteq r$, where $r, r_1, \ldots, r_n$ are role names, $n \in \mathbb{Z}^+$. For a general introduction to description logics, and for the formal semantics of the constructors available in $\mathcal{EL}^+$, please refer to [5].

Classification Classification is one of the standard reasoning tasks. The classification of an ontology refers to the computation of the complete subsumption hierarchy involving all concept names occurring in the ontology.

A classification algorithm for $\mathcal{EL}^+$ using forward-chaining rules is given in [1], and Table 1 presents a slightly modified set of completion rules which is easily checked to be sound and complete as well [10]. We use these rules to compute the classification of input $\mathcal{EL}^+$ ontologies. For our modification, we divided the rule **R3** from [1] into **R3-1** and **R3-2**, as follows.

R3-1: If $A \in S(Y)$ and $\exists r.A \sqsubseteq B \in \mathcal{O}$, then $\exists r.Y \sqsubseteq B$

R3-2: If $\exists r.Y \sqsubseteq B$ and $(X, Y) \in R(r)$, then $S(X) := S(X) \cup \{B\}$

This helps in the division and distribution of work that needs to be done by our reasoner. The axioms in the ontology are in one of the normal forms given on the left column of Table 1. $S(X)$ contains all the subsumers of X i.e., $A \in S(X)$ means $X \sqsubseteq A$. Likewise, $R(r)$ stands for $\{(X, Y) | X \sqsubseteq \exists r.Y\}$. Our

Normal Form		Completion Rule
$A \sqsubseteq B$	**R1-1**	If $A \in S(X)$, $A \sqsubseteq B \in \mathcal{O}$, and $B \notin S(X)$ then $S(X) := S(X) \cup \{B\}$
$A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$	**R1-2**	If $A_1, \ldots, A_n \in S(X)$, $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B \in \mathcal{O}$, $B \notin S(X)$ then $S(X) := S(X) \cup \{B\}$
$A \sqsubseteq \exists r.B$	**R2**	If $A \in S(X)$, $A \sqsubseteq \exists r.B \in \mathcal{O}$, and $(X, B) \notin R(r)$ then $R(r) := R(r) \cup \{(X, B)\}$
$\exists r.A \sqsubseteq B$	**R3-1**	If $A \in S(Y)$, $\exists r.A \sqsubseteq B \in \mathcal{O}$ then $P = P \cup \{\exists r.Y \sqsubseteq B\}$
$\exists r.A \sqsubseteq B$	**R3-2**	If $(X, Y) \in R(r)$, $\exists r.Y \sqsubseteq B \in P$ and $B \notin S(X)$ then $S(X) := S(X) \cup \{B\}$
$r \sqsubseteq s$	**R4**	If $(X, Y) \in R(r)$, $r \sqsubseteq s \in \mathcal{O}$, and $(X, Y) \notin R(s)$ then $R(s) := R(s) \cup \{(X, Y)\}$
$r \circ s \sqsubseteq t$	**R5**	If $(X, Y) \in R(r)$, $(Y, Z) \in R(s)$, $r \circ s \sqsubseteq t \in \mathcal{O}$, $(X, Z) \notin R(t)$ then $R(t) := R(t) \cup \{(X, Z)\}$

Table 1. Axioms (in normal forms) and modified completion rules of CEL

Algorithm 1 Pseudocode for CEL classification fixpoint iteration

$S(X) \leftarrow \{X, \top\}$, for each concept X in the ontology.
$R(r) \leftarrow \{\}$, for each role r in the ontology.
$P \leftarrow \{\}$
repeat
  Used below, Old.$S(X)$ stands for $S(X)$ now, and Old.$R(r)$ stands for $R(r)$ and Old.P stands for P;
  $S(X) \leftarrow$ apply R1-1 using $S(X)$;
  $S(X) \leftarrow$ apply R1-2 using $S(X)$;
  $R(r) \leftarrow$ apply R2 using $S(X)$ and $R(r)$;
  $P \leftarrow$ apply R3-1 using $S(X)$;
  $S(X) \leftarrow$ apply R3-2 using $R(r)$ and P;
  $R(r) \leftarrow$ apply R4 using $R(r)$;
  $R(r) \leftarrow$ apply R5 using $R(r)$;
until ((Old.$S(X) = S(X)$) and (Old.$R(r) = R(r)$) and (Old.$P = P$))

goal is to compute S(X) for each concept X in the ontology $\mathcal{O}$. The $R(r)$ is used in deriving all such subclass relationships. P is a set which holds the axioms generated by rule R3-1. Instead of representing these axioms by P, the other option is to add these axioms back in the ontology $\mathcal{O}$, but we would like to keep the ontology read only.

For classifying $\mathcal{EL}^+$ ontologies, all the rules from Table 1 are processed iteratively until no new output is generated, as shown in Algorithm 1.

Notations We use $U(X)$ to refer to the "inverse" of $S(X)$ i.e., $U(X) = \{A \mid X \sqsupseteq A\}$. The advantage and performance benefit that is obtained by using $U(X)$ instead of $S(X)$ is explained later in Section 3. It is typical in computer science to think of $U(X)$ as applying the definition of a function U to the argument X and thus $U(X)$ does not yield different results on repeated use. In constrast with

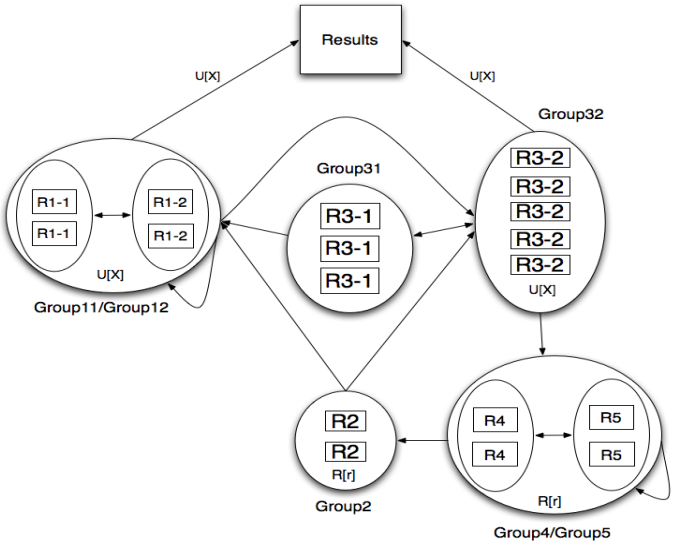


Fig. 1. Node assignment to rules and dependency among the completion rules. Each oval is a collection of nodes (rectangles).

this, we use $U[X]$ to stand for the value stored in an associative array U indexed by a possibly non-integer value X. The same applies to $R(r)$. Hence $U[X]$ and $R[r]$ are conceptually treated as associative arrays, but implementation details might vary. In the rest of the paper we use $U[X]$ and $R[r]$ instead of $S(X)$ and $R(r)$. Sometimes, we refer to all $R[r]$ collectively as *R-data*.

3 Distributed Approach

Architecture Each group of nodes in the cluster (see Figure 1) is dedicated to work on only one particular completion rule R_i, i.e., on axioms belonging to R_i's normal form. Axioms of the ontology are split into disjoint collections based on their normal form. Each disjoint set of axioms are assigned to a particular group, G_i, responsible for processing rule R_i. Within each group, axioms are again split among the nodes of the group. Nodes of Group11, Group12, Group32 produce results which are collected by a single node. We use a set of key-value pairs to represent the axioms and the sets $U[X]$, $R[r]$ that are distributed over the cluster. Figure 1 also shows the dependency among the completion rules i.e., the axioms that are to be processed in the next iteration are determined by the updates done by other nodes. For example, the axioms that will be considered in rule R2 in the next iteration will depend on the output from rules R1-1, R1-2 and R3-2. This is explained further in the optimization section.

Algorithm 2 Pseudocode for rule R1-1

$K \leftarrow 0$
for all axioms of the form $A \sqsubseteq B$ **do**
  $K \leftarrow K + (U[B] \ \cup= \ U[A])$ //add U[A] to U[B]
end for
return K

Algorithm 3 Pseudocode for rule R1-2

$K \leftarrow 0$
for all axioms of the form $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$ **do**
  $K \leftarrow K + (U[B] \ \cup= \ U[A_1] \cap \cdots \cap U[A_n])$
end for
return K

Algorithm 4 RolePairHelper({r, X}, B)

$K \leftarrow$ Send triplet ({r, B}, X) to Group32 //send to D_1
if there exists s such that $r \sqsubseteq s$ **then**
  $K \leftarrow$ Send ({r, X}, B) to Group4 //send to D_1
end if
if there exist s and t such that $r \circ s \sqsubseteq t$ **then**
  $K \leftarrow$ Send ({r, B}, X) to Group5 //send to D_0
end if
if there exist s and t such that $s \circ r \sqsubseteq t$ **then**
  $K \leftarrow$ Send ({s, X}, B) to Group5 //send to D_1
end if
return K

Key-Value Store Redis[4] is an open source, high performance key-value store implementation. It provides several data structures like sets, sorted sets, hash, lists. It also supports atomic operations and server-side Lua[5] scripting along with client-side sharding. All these features are used in our implementation. Redis runs on each node of the cluster holding the axioms, $R[r]$ and $U[X]$ values.

We use a Java client named Jedis[6] to interact with Redis.

Pseudocode for completion rules Pseudocode for some rules use a notation such as D_0, D_1. They denote databases of Redis. Each Redis instance can have several databases associated with it. If not mentioned specifically then all the data goes into database-0 (or D_0). Pseudocode for all the rules captures the total number of updates made and returns this number.

The pseudocode of R1-1 is given in Algorithm 2. The operator $\cup=$ performs the set union and returns the number of elements that were added to the destination set – the latter is needed for termination checking, discussed

[4] `http://redis.io`
[5] `http://www.lua.org`
[6] `https://github.com/xetorthio/jedis`

Algorithm 5 Pseudocode for rule R2

$K \leftarrow 0$
for all axioms of the form $A \sqsubseteq \exists r.B$ **do**
  for all $X \in U[A]$ **do**
    $K \leftarrow K$ + RolePairHelper({r, X}, B)
  end for
end for
return K

Algorithm 6 Pseudocode for rule R3-1

$K \leftarrow 0$
for all r, A, B in axioms of the form $\exists r.A \sqsubseteq B$ **do**
  for all $Y \in U[A]$ **do**
    $K \leftarrow K$ + (Send axiom $\exists r.Y \sqsubseteq B$ to Group32)
  end for
end for
return K

Algorithm 7 Pseudocode for rule R3-2

$K \leftarrow 0$
for all axioms of the form $\exists r.Y \sqsubseteq B$ received from R3-1 **do**
  $T_r := \{X \mid (\{r, Y\}, X) \in D_1\}$; //received from RolePairHelper
  $K \leftarrow K + (U[B] \ \cup= \ T_r)$
end for
return K

Algorithm 8 Pseudocode for rule R4

$K \leftarrow 0$
T_3 := set of triplets received from RolePairHelper();
$T_r := \{r \mid (\{r, X\}, Y) \in T_3\}$;
for all $r \in T_r$ **do**
  for all roles s such that $r \sqsubseteq s$ is an axiom **do**
    $K \leftarrow K$ + RolePairHelper({s, X}, Y)
  end for
end for
return K

below. All $U[X]$ are stored on the result node. So $\cup=$ also implicitly involves contacting the result node for read ($U[A]$) and write ($U[X]$) operations.

The pseudocode of R1-2 is given in Algorithm 3. As mentioned below, it suffices to find the intersection of all $U[A]$ involved in the conjuncts, i.e., $A_1 \ldots A_n$.

Expanding on these two rules, let us briefly come back to an issue mentioned earlier, namely why we chose to use $U[X]$ instead of $S[X]$ for our implementation.

Let $\mathcal{O}$ be an ontology and let $K \sqcap L \sqcap M \sqsubseteq N \in \mathcal{O}$. Furthermore, assume that there are five concepts in the ontology, K, L, M, N and P. During some iteration of the classification assume $S(K) = \{K, L, N, \top\}$, $S(L) = \{L, P, M, \top\}$,

Algorithm 9 Pseudocode for rule R5

```
K ← 0
T_0 := set of triplets received from RolePairHelper();    //in D_0
for all ({r, Y}, X) ∈ T_0 do
    T_r := {r | ({r, Y}, Z) ∈ D_1};
    for all t ∈ axioms r ∘ s ⊑ t do
        K ← K + RolePairHelper({t, X}, Z)
    end for
end for
return K
```

$S(M) = \{M, N, K, \top\}$, $S(N) = \{N, \top\}$, and $S(P) = \{P, K, L, M, \top\}$. Now, according to rule R1-2, we have to check for the presence of K, L and M in each of the five $S(X)$, where $X = K, L, M, N, P$. Since only $S(P)$ has K, L, M, we have to add N to $S(P)$.

On the other hand, use instead $U[K] = \{K, M, P\}$, $U[L] = \{L, K, P\}$, $U[M] = \{M, L, P\}$, $U[N] = \{N, K, M, P\}$, $U[P] = \{P, L\}$. In this case, instead of checking all $U[X]$, we can compute the intersection of $U[K]$, $U[L]$, $U[M]$, which is P. So, $P \sqsubseteq N$ which means $U[N] \cup= \{P\}$. In large ontologies, the number of concepts would be in the millions or more, but the number of conjuncts in axioms like $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$ would be very less in number. So the performance is better by using $U[X]$ since set intersection needs to be performed only on a very small number of sets in this case.

Rules R2, R4 and R5 deal with $R[r]$ values. RolePairHelper, with pseudocode given in Algorithm 4, provides functionality that is common to these three rules. The R-data, R[.], is an associative array indexed by roles. RolePairHelper({role r, concept X}, concept B) is invoked in rules R2, R4 and R5. RolePairHelper() informs all nodes (namely Group32, Group4 and Group5) of these updates. Group4 does not care to know the updates to $R[r]$ unless role r has a super role s, that is for some s, $r \sqsubseteq s$. Similarly, Group5 does not care to know of these updates unless there exist roles s and t such that $r \circ s \sqsubseteq t$. Note that the R[.] across all the nodes will, in general, not be the same because of these selective updates. Note also that replicating R[.] across all nodes causes no semantic harm. This is done to facilitate local reads of R-data on nodes dealing with role axioms, i.e., Group32, Group4 and Group5. The Send primitive in Algorithms 4 and 6, returns 0 if the message sent is duplicate, or 1 otherwise.

Nodes handling rule R2, i.e., Group2, do not make use of $R[r]$ values. So they do not need to be stored locally on Group2 nodes as shown in Algorithm 5. Rules R2, R4 and R5 potentially add new entries to the R-data, and every such update implies a triggering of rules R3-2, R4, and R5. RolePairHelper({r, X}, B) broadcasts such updates.

The pseudocode for rule R3-1 is given in Algorithm 6. Here, newly formed axioms do not need to be sent to all the nodes in Group32 but can be sent to only a specific node. For the sake of clarity, this is not shown in the pseudocode and is explained in the section on optimizations.

Algorithm 10 Modeling of One Iteration, $OIP_i(D)$

$K \leftarrow$ apply rule R_i using (D)
if $K == 0$ **then**
    return true
else
    return false
end if

Algorithm 11 Modeling of a Process, P_i

repeat
    $isNew \leftarrow OIP_i(D)$
    broadcast $isNew$ to all P_j
    receive t_j from all P_j
    $t \leftarrow t_1 \vee t_2 \vee ... \vee t_j$
until $\neg t$

The pseudocode for rule R3-2 is given in Algorithm 7. Here, database-1 (D_1) is queried with key $\{r, Y\}$ and T_r holds all such X.

Regarding R4, in Algorithm 8, T_3 receives only such triples whose r participates in an axiom of the form $r \sqsubseteq s$. T_r is formed for each r found in T_3.

Concerning R5, in Algorithm 9, using the key $\{r, Y\}$, the database D_1 is queried and the results are referenced by T_r. All axioms of the form $r \circ s \sqsubseteq t$ in which r participates in, are retrieved. Note, that the value of s does not matter, since it is already taken care of in RolePairHelper. The following example illustrates how Algorithms 4 and 9 are connected. Let k, m and n be roles, where $k \circ m \sqsubseteq n$, $(X, Y) \in R(k), (Y, Z) \in R(m)$. RolePairHelper({k, X}, Y) sends ({k, Y}, X) to D_0 of Group5. RolePairHelper({m, Y}, Z) sends ({k, Y}, Z) to D_1 of Group5. In Algorithm 9, T_k contains ({k, Y}, X). D_1 is queried with {k, Y} as the key and ({n, X}, Z) is produced. n is obtained from the Rolechain axiom.

Termination One iteration of the process, OIP, is modeled by the pseudocode shown in Algorithm 10. Each OIP reads and writes to a Redis instance D (database).

The appropriate pseudocode for rule R_i is processed and the return value is collected in K, which holds the number of updates made. Depending on the value of K, either true or false is returned.

Each process P_i performs the computations in Algorithm 11. The receive() is a blocking operation; i.e., until a message is received, the calling process (P_i) does not proceed to the next state. However, even though the pseudocode does not show it, we assume that the messages from P_j can be received in any order.

On each node, N_i, the process P_i processes all the axioms local to it and keeps track of whether this resulted in any changes (*isNew*) to either its local Redis database or to the one on other nodes. This boolean value is broadcast to all the processes. When all the *isNew* messages are received, each process on all the nodes knows whether any of the other process made some updates or

not. If at least one process makes an update then all the processes continue with the next iteration. If none of the processes makes an update then all processes terminate.

All the nodes in the cluster keep processing the axioms over several iterations until no new output is generated by any of the nodes. In sequential computation, this is fairly easy to check, but in a distributed system, all the nodes should be coordinated in order to check whether any of them have produced a new output.

The coordination among the processes on all nodes is achieved by message passing. Process P_i on each node is associated with a channel C_i. At the end of each iteration, P_i broadcasts its status message to channels on all the nodes. It then does a blocking wait until it receives messages from all the processes on its channel. This is generally known as barrier synchronization.[7] If all the messages that P_i receives are false, i.e., none of the processes made an update to any of the key-value pairs, then P_i terminates.

Optimizations The following optimizations were put in place to speed up the processing of rules.

1. All the concepts and roles in the ontology are assigned numerical identifiers. This saves space and is easier to process.
2. If $X \sqsubseteq A$, normally this would be stored in a set whose key would be X and value would be A. But we reverse it, and make A the key and X its value. This makes the check $A \in S(X)$, a single read call. This check is required in rules R1-1, R1-2, R2 and R3-1.
3. As shown in Figure 1, the output of a rule can affect the processing of another rule. For example, rule R2 works on axioms of the form $A \sqsubseteq \exists r.B$. R2 then depends on the rules which affect A, which are R1-1, R1-2 and R3-2. If R1-1, R1-2 and R3-2 do not make any changes to $U[A]$, then the axiom $A \sqsubseteq \exists r.B$ need not be considered in the next iteration. We keep track of these dependencies and thereby reduce the number of axioms to work on in subsequent iterations.
4. Extending on the optimization just mentioned, if there is a change in $U[A]$, then not all elements of $U[A]$ need to be considered again. In fact, we need to consider only the newly added elements. This can be achieved by assigning scores to each element in the set $U[A]$. A node working on rule R2 and axiom $A \sqsubseteq \exists r.B$ keeps track of the scores of elements in $U[A]$, i.e., up to what it has read in the previous iteration, and only considers elements whose scores are greater than that.
5. In the pseudocode of Algorithm 6 and Algorithm 4, it is shown that the newly formed axiom and the triple is sent to all the nodes of a particular group. Instead, they can only be sent to a particular node in the group, and this node is selected based on the key. For the same key within the same group, however, we can ensure that always the same node gets selected. This reduces duplication of data.

[7] `http://en.wikipedia.org/wiki/Barrier_(computer_science)`

Ontology	#Logical Axioms	#Concepts	#Roles
Not-Galen	8,015	4,242	413
GO	28,897	20,465	1
NCI	46,870	27,653	70
SNOMED	1,038,481	433,106	62
SNOMED-DUP-2	2,076,962	866,212	124
SNOMED-DUP-3	3,115,443	1,299,318	186
SNOMED-GALEN-GO	1,075,393	456,319	476

Table 2. Sizes of (normalized) ontologies we used

4 Evaluation

To evaluate our implementation, we made use of the seven ontologies that are listed in Table 2. The numbers in Table 2 are obtained after normalizing the ontologies. The first three ontologies have been obtained from `http://lat.inf.tu-dresden.de/~meng/toyont.html`. SNOMED is from `http://www.ihtsdo.org/snomed-ct`. SNOMED-DUP-2 and SNOMED-DUP-3 are ontologies with axioms from SNOMED, but each axiom replicated twice and thrice, respectively, while concept and role names are systematically renamed for each copy. SNOMED-GALEN-GO is a merge of the three ontologies, SNOMED, Not-Galen and GO, which was obtained synthetically as follows: Upon normalization of each of these ontologies, new class names and role names were created which were assigned to a local namespace. However, class names and role names introduced in the normalization are *shared* between the ontologies. We thus obtain a merged ontology which, albeit the merge is synthetic, retains some of the real-life character of each of these ontologies.

DistEL is implemented in Java and makes use of Redis for storage. Our cluster consists of 13 Linux nodes, but our implementation scales to larger clusters. Each node has two quad-core AMD Opteron 2300MHz processors with 16GB RAM. DistEL treats a Redis instance as a node, so in order to show the scalability aspect of our implementation, we ran 2-3 Redis instances on a single node. This allowed us to effectively run tests for more than 13 nodes on the cluster. Since each node has 8 cores, and data on an instance of Redis generally doesn't go beyond 5GB (for our experiments), running 2-3 Redis instances does not adversely affect the evaluation.

Table 3 has the classification time of ontologies when run on Pellet (version 2.3.0), jCEL (0.18.2) and ELK (version 0.3.2). Heap space given to run all the ontologies is 12GB. Timeout limit given was 2 hours. All the reasoners are invoked through the OWL API. Time taken by the OWL API to load the ontology is not taken into consideration. Note that SNOMED-GALEN-GO could not be processed by any of the state-of-the-art systems. This is remarkable because the

Ontology	Pellet	jCEL	ELK
Not-Galen	12.0	3.0	1.0
GO	5.0	5.0	2.0
NCI	6.0	7.0	3.0
SNOMED	1,845.0	327.0	24.0
SNOMED-DUP-2	OutOfMemory	687.0	64.0
SNOMED-DUP-3	OutOfMemory	1149.0	93.0
SNOMED-GALEN-GO	OutOfMemory	TIME OUT	TIME OUT

Table 3. Classification time of ontologies using Pellet, jCEL and ELK

Ontology	7 nodes	9 nodes	12 nodes	15 nodes	18 nodes
Not-Galen	6.76	6.44	6.67	6.67	7.09
GO	11.58	11.65	11.74	12.59	12.51
NCI	21.13	21.57	21.53	22.15	22.80
SNOMED	382.77	385.09	392.09	398.07	393.57
SNOMED-DUP-2	774.34	767.85	787.10	798.58	826.50
SNOMED-DUP-3	2,160.00	2,160.00	2,113.57	2,194.80	2,233.12
SNOMED-GALEN-GO	—	—	—	—	411.72

Ontology	21 nodes	25 nodes	28 nodes	32 nodes
Not-Galen	6.78	6.83	6.74	6.77
GO	12.30	12.46	12.87	12.93
NCI	22.53	22.63	22.66	22.15
SNOMED	396.66	405.94	410.07	412.39
SNOMED-DUP-2	803.43	805.81	828.55	828.78
SNOMED-DUP-3	2,177.13	2315.19	2163.17	2257.94
SNOMED-GALEN-GO	416.99	418.99	419.58	428.43

Table 4. Load times (in seconds) of DistEL

ontology is hardly larger than SNOMED itself.[8] In fact, it shows that realistic handcrafted ontologies of a size which cannot be handled by state-of-the-art reasoners are not far out of reach.

Pre-processing times (in seconds) for the ontologies on DistEL are given in Table 4. Pre-processing includes the time taken by the OWL API[9] to load the ontology in-memory and the time taken to insert the axioms into the nodes of the cluster, and it is approximately constant with respect to the number of nodes. Time taken by SNOMED-GALEN-GO is not mentioned for 7, 9, 12 and 15 nodes because the classification time on these nodes times out.

Table 5 shows the classification times of DistEL with varying numbers of nodes, and a corresponding visualization is given in Figure 2. For the larger ontologies, i.e., SNOMED and larger, we see a steady decrease of classification time with increasing number of nodes used. Note that, for the larger ontologies,

[8] We are not entirely certain yet what causes this explosion. Our guess is that it is caused by the large number of role chains in Galen, together with the sheer size of SNOMED.

[9] `http://owlapi.sourceforge.net`

Ontology	7 nodes	9 nodes	12 nodes	15 nodes	18 nodes
Not-Galen	43	42.27	41.06	39.12	36.70
GO	46.20	49.39	51.83	52.44	53.62
NCI	275	168.96	157.36	156.45	156.82
SNOMED	1,610.00	1,355.81	865.89	886.44	613.53
SNOMED-DUP-2	3,238.19	2,687.75	1,699.73	1,765.31	1,255.87
SNOMED-DUP-3	4,880.78	4,052.00	2,570.29	2,644.40	1,825.51
SNOMED-GALEN-GO	TIME OUT	TIME OUT	TIME OUT	TIME OUT	1,336.28

Ontology	21 nodes	25 nodes	28 nodes	32 nodes
Not-Galen	37.51	36.69	36.11	35.09
GO	51.89	56.76	39.70	50.80
NCI	155.19	154.75	161.41	160.12
SNOMED	529.30	441.74	442.81	383.01
SNOMED-DUP-2	1,064.44	887.19	893.96	755.38
SNOMED-DUP-3	1,571.43	1278.62	1286.50	1146.71
SNOMED-GALEN-GO	1,241.96	702.02	693.51	618.18

Table 5. Classification time (in seconds) of DistEL

the effect of parallelization is very good indeed. E.g., using twice the number of nodes almost halves the runtime in most cases – so the effect of the parallelization is indeed near optimal.

As expected, for the smaller ontologies, the parallelization does not have much effect as soon as a certain threshold is reached. We see, however, that even when using many more nodes than necessary to reach optimal runtime, we do not get a significant amount of additional time lost due to communication overhead.

After having retrieved the runtime figures just discussed, we noticed that some of the measured times for 15 nodes were in fact higher than those of 12 nodes, and a similar effect showed for 28 versus 25 nodes. When additional nodes are added to the cluster, they should ideally be assigned to the slowest processing nodes. But if this is not the case, then we do not see any noticeable improvement in performance. On the contrary, there is a possibility of reduction in performance because of additional communication overhead. Since we are currently assigning nodes by intuition and rule-of-thumb, we have to expect to get such performance drops sometimes.

To further check on this effect, we repeated the run of SNOMED on 15 nodes using a different assignment of nodes to rules, and it resulted in a classification time of 606.05 seconds, which is a significant improvement compared to the timing obtained by the node assignment in Table 5. In fact, it is better than using 18 nodes in the previous assignment. This shows that our manual rule-of-thumb assignments are likely not optimal, and that, in fact, a signifcantly better performance should be achievable by automating the node assignment, or by using methods for dynamic load balancing. However, in this paper we only want to show that significant parallelization can be achieved, and do not yet focus on a most efficient implementation. This is left for future work.

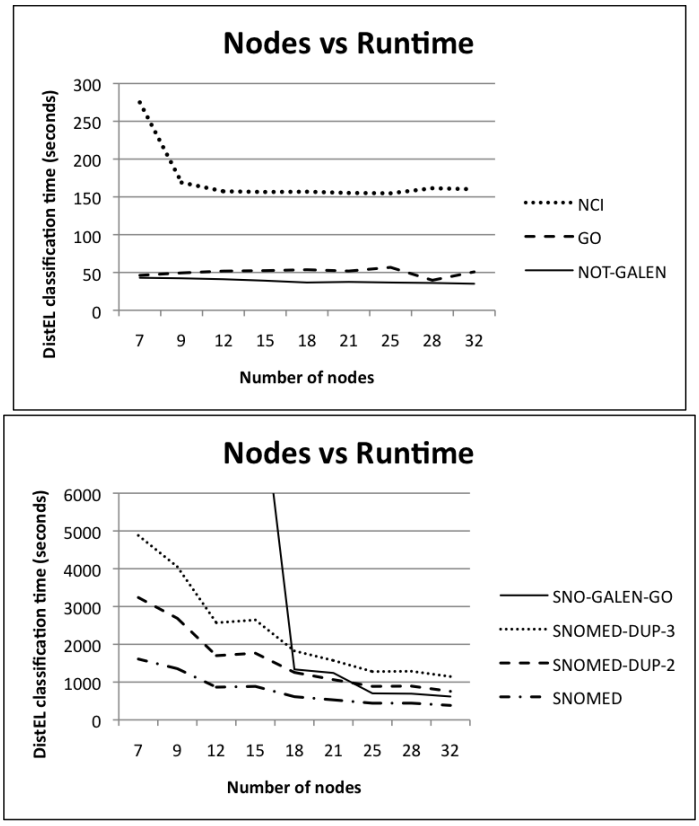


Fig. 2. Visualization of number of nodes vs runtimes

Correctness of the results produced by DistEL is verified by comparing the output with that of ELK, in the cases where ELK does not time out.

There is a significant difference in performance between our distributed implementation and ELK. One of the primary reasons for this difference is that our implementation requires cross-node communication (key-value pairs are sent across the nodes) and among the nodes. On the other hand, our architecture can be extended by adding more nodes, and thus can scale up to datasets which ELK cannot handle, such as SNOMED-GALEN-GO.

We list some further insights which we gained from our implementation and system.

1. For our manual assignment of nodes to rules, it is very important to figure out the slowest processing nodes in the cluster, so that, if additional nodes are available, it would be easy to determine, to which group these new nodes should be assigned to.

2. The majority of the time is in fact spent on reading and writing to local as well as remote databases. Design choices and architecture should be formulated in such a way so as to reduce cross-node communication.
3. We cannot estimate the runtime or node assignment to rules just by counting the number of axioms. Some axioms are harder to process than others although their number might be less. A case in point is SNOMED-GALEN-GO compared with SNOMED-DUP-3.

5 Limitations and Future Work

Some of the limitations and planned next steps are presented below.

1. Compared to other popular distributed frameworks like Hadoop,[10] our architecture does not currently provide support for fault tolerance.
2. Axioms are distributed across the cluster by type rather than load. This leads to improper load balancing. To improve load balancing we plan to implement work stealing so that the idle nodes can work on axioms from other nodes.
3. Completion rules are assigned to nodes manually, for now. This could be done automatically by considering ontology statistics such as the number of role axioms and subclass axioms, or other measures.
4. The OWL API is used to read the axioms from an ontology, and by design it loads the entire ontology into memory. With the size of ontologies that we hope to deal with using our work, this becomes a bottleneck. Going forward, we plan to use a streaming API for XML[11] to read the axioms.
5. We plan to extend our work to include ABox reasoning [11] where there is a greater scope of getting large ontologies and our distributed system could be put to test.
6. We also intend to use multicore threading to take advantage of the number of cores in modern machines.
7. Another possible line of future work is to apply our distributed approach to other $\mathcal{EL}^+$classification algorithms such as the materialization procedure used by ELK.

6 Related Work

Most of the distributed reasoning approaches in the literature are focussed on RDFS inference but there are a few that deal with a fragment of OWL, namely OWL Horst. In [17], Urbani et al., use MapReduce for reasoning over OWL Horst. They look to carry over their work from distributed reasoning over RDFS to OWL Horst, which was possible only to a certain extent. Soma et al., [15] investigate partitioning approaches for parallel inferencing in OWL Horst. Although not distributed, a backward chaining approach [16] is used to scale up

[10] `http://hadoop.apache.org`
[11] `http://en.wikipedia.org/wiki/StAX`

to a billion triples in the OWL Horst fragment. A distributed approach to fuzzy OWL Horst reasoning has also been investigated in [8].

Distributed resolution techniques were used by Stuckenschmidt et al., to achieve scalability of various OWL fragments such as $\mathcal{ALC}$ [12] and $\mathcal{ALCHIQ}$ [13]. There have been attempts at achieving distributed reasoning on the $\mathcal{EL}^+$profile in [10] and [14], but they do not provide any evaluation results. Distribution of OWL EL ontologies over a peer-to-peer network and algorithms based on distributed hash table have been attempted in [2], but again, no evaluation results are provided. Reasoning over Fuzzy-EL+ ontologies using MapReduce [18] has also been attempted but implementation and experiment details are not provided.

We have tried several distributed approaches for $\mathcal{EL}^+$ontology classification, some of which have been unsuccessful. Here, we presented an approach which gave encouraging results due to increase in the number of axioms that are processed locally and decrease in the cross-node communication, when compared to our previous approaches [9]. Our earlier approaches include use of MapReduce (involved many redundant computations) and distributed queue (distribution of CEL's queue approach). The latter approach involves a lot of cross-node communication.

7 Conclusions

With ever increasing data generation rates, large ontologies challenge reasoners from the perspective of memory and computation power. In such scenarios, distributed reasoners offer a viable solution. We presented our distributed approach to $\mathcal{EL}^+$ ontology classification, called DistEL, where we show that our classifier can handle large ontologies and the classification time decreases with the increase in nodes. The results are encouraging, and we plan to go ahead with adding ABox reasoning support to our work as well as explore other possible distributed classification approaches.

Acknowledgements. We would like to thank members of the Redis mailing list `redis-db@googlegroups.com` for their helpful suggestions. This work was supported by the National Science Foundation under award 1017225 "III: Small: TROn – Tractable Reasoning with Ontologies." Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Baader, F., Lutz, C., Suntisrivaraporn, B.: Is Tractable Reasoning in Extensions of the Description Logic EL Useful in Practice? In: Proceedings of the 2005 International Workshop on Methods for Modalities (M4M-05) (2005)
2. Battista, A.D.L., Dumontier, M.: A Platform for Reasoning with OWL-EL Knowledge Bases in a Peer-to-Peer Environment. In: Proceedings of the 5th International Workshop on OWL: Experiences and Directions, Chantilly, VA, United States, October 23-24 2009. CEUR Workshop Proceedings, vol. 529. CEUR-WS.org (2009)

3. Dentler, K., Cornet, R., ten Teije, A., de Keizer, N.: Comparison of Reasoners for large Ontologies in the OWL 2 EL Profile. Semantic Web 2(2), 71–87 (2011)
4. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S. (eds.): OWL 2 Web Ontology Language: Primer. W3C Recommendation (27 October 2009), available from http://www.w3.org/TR/owl2-primer/
5. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)
6. Kazakov, Y., Krötzsch, M., Simancik, F.: Concurrent Classification of EL Ontologies. In: 10th International Semantic Web Conference, Bonn, Germany, October 23-27. Lecture Notes in Computer Science, vol. 7031, pp. 305–320. Springer (2011)
7. Lécué, F., Schumann, A., Sbodio, M.L.: Applying Semantic Web Technologies for Diagnosing Road Traffic Congestions. In: International Semantic Web Conference (2). Lecture Notes in Computer Science, vol. 7650, pp. 114–130. Springer (2012)
8. Liu, C., Qi, G., Wang, H., Yu, Y.: Large Scale Fuzzy pD* Reasoning Using MapReduce. In: 10th International Semantic Web Conference, Bonn, Germany, October 23-27. Lecture Notes in Computer Science, vol. 7031, pp. 405–420. Springer (2011)
9. Mutharaju, R.: Very Large Scale OWL Reasoning through Distributed Computation. In: International Semantic Web Conference (2). Lecture Notes in Computer Science, vol. 7650, pp. 407–414. Springer (2012)
10. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce Algorithm for EL+. In: Proceedings of the 23rd International Workshop on Description Logics (DL 2010), Waterloo, Ontario, Canada, May 4-7, 2010. CEUR Workshop Proceedings, vol. 573. CEUR-WS.org (2010)
11. Ren, Y., Pan, J.Z., Lee, K.: Parallel ABox reasoning of EL ontologies. In: Proceedings of the 2011 Joint International Conference on the Semantic Web. pp. 17–32. JIST'11, Springer, Heidelberg (2012)
12. Schlicht, A., Stuckenschmidt, H.: Distributed Resolution for ALC. In: Proceedings of the 21st International Workshop on Description Logics (DL2008), Dresden, Germany, May 13-16. CEUR Workshop Proceedings, vol. 353. CEUR-WS.org (2008)
13. Schlicht, A., Stuckenschmidt, H.: Distributed Resolution for Expressive Ontology Networks. In: Proceedings of the Third International Conference on Web Reasoning and Rule Systems, RR 2009, Chantilly, VA, USA, October 25-26, 2009. Lecture Notes in Computer Science, vol. 5837, pp. 87–101. Springer (2009)
14. Schlicht, A., Stuckenschmidt, H.: MapResolve. In: Web Reasoning and Rule Systems – 5th International Conference, RR 2011, Galway, Ireland, August 29-30, 2011. Lecture Notes in Computer Science, vol. 6902, pp. 294–299. Springer (2011)
15. Soma, R., Prasanna, V.K.: Parallel Inferencing for OWL Knowledge Bases. In: 2008 International Conference on Parallel Processing, ICPP 2008, September 8-12, 2008, Portland, Oregon, USA. pp. 75–82. IEEE Computer Society (2008)
16. Urbani, J., van Harmelen, F., Schlobach, S., Bal, H.E.: QueryPIE: Backward Reasoning for OWL Horst over Very Large Knowledge Bases. In: 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011. Lecture Notes in Computer Science, vol. 7031, pp. 730–745. Springer (2011)
17. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E.: OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples. In: Proceedings of the 8th Extended Semantic Web Conference (ESWC2010), Heraklion, Greece, May 30–June 3, 2010. Springer (2010)
18. Zhou, Z., Qi, G., Liu, C., Hitzler, P., Mutharaju, R.: Reasoning with Fuzzy-EL+ Ontologies Using MapReduce. In: Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012). Frontiers in Artificial Intelligence and Applications, vol. 242, pp. 933–934. IOS Press (2012)

Rule-based Reasoning on Massively Parallel Hardware

Martin Peters[1], Christopher Brink[1], Sabine Sachweh[1], and Albert Zündorf[2]

[1] University of Applied Sciences Dortmund, Germany,
Department of Computer Science
{martin.peters ‖ christopher.brink ‖ sabine.sachweh}@fh-dortmund.de
[2] University of Kassel, Germany, Software Engineering Research Group,
Department of Computer Science and Electrical Engineering
zuendorf@cs.uni-kassel.de

Abstract. In order to enable the semantic web as well as other time critical semantic applications, scaleable reasoning mechanisms are indispensable. To address this issue, in this paper we propose a rule-based reasoning algorithm which explores the highly parallel hardware of modern processors. In contrast to other approaches of parallel reasoning, our algorithm works with rules that can be defined depending on the application scenario and thus is able to apply different semantics. Furthermore we show how vector-based operations can be used to implement a performant match algorithm. We evaluate our approach by applying the ρdf, RDFS and pD* rule sets to different data sets and compare our results with other recent work. The evaluation shows that our approach is up to 9 times faster depending on the rule set and the used ontology and is able for example to apply the ρdf rules to an ontology with 2.2 million triples (and 1.3 million inferred triples) in less than 6 seconds.

Keywords: rule-based reasoning, GPU, parallel reasoning, Rete algorithm

1 Introduction

The use of ontologies is widely spread whether they are used in scientific applications or to enable the semantic web. One key characteristic of ontologies is the possibility to reason about the given data and to create new knowledge by inferring facts that are implicitly given by the ontology. Depending on the size and the structure of the data, the reasoning process may be very resource consuming, especially with regard to the continuously growing amount of data of the semantic web. On the other side, applications using ontologies for example in the field of ambient assisted living [1] [2] or smart spaces [3] [4] [5] may be time critical and need to process incoming data very fast.

Different approaches exist to speedup the reasoning process and to provide scalable solutions for different levels of expressivity, varying from improvements on the reasoning process itself to distributed reasoning over clusters of computational units. Especially the number of approaches using parallel structures has

increased in the past few years. The ELK reasoner [6] for example takes advantage of multi-core and multi-processor systems of modern computers to perform OWL EL[3] reasoning. Other approaches rely on a distributed cluster and use the MapReduce framework to perform RDFS and pD* (also know as OWL Horst) [7] reasoning on RDF graphs with millions of triples [8] [9].

While these approaches perform very well for a predefined set of rules that define the strength of the semantics and thus the expressivity of the resulting ontology, the use of a cluster of machines for computation may not always be desirable due to the high costs. In addition the predefined set of rules that is implemented may not always be exactly what is needed for a specific application. In this paper we present an approach which uses the massively parallel hardware of a modern graphic processor unit (GPU) to apply a set of freely defined rules to an RDF-based ontology. While a single core of a GPU has not as much computation power like on a CPU, a GPU provides much more processors that are able to perform simple computation tasks in parallel. Thus it makes sense to exploit this highly parallel hardware and to break down the complex workload into fine grained tasks for parallelisation. Our approach uses an adapted version of the Rete algorithm [10] and thus implements a forward chaining rule engine, which is able for example to materialise the complete finite RDFS closure as well as to apply the pD* rules in a scaleable manner on a single machine. Also this approach is more flexible than most other reasoners, because it is not dedicated to a predefined semantic, we achieve a very high performance due to the parallelisation of the time consuming steps. In addition our approach is not required to be executed on a GPU, but also can be executed on a multicore CPU.

The main contribution of this paper will be to show how the Rete algorithm can be used to reason on ontologies in a highly parallel manner. The use of a rule-engine-based approach allows us to provide a reasoner that is not dedicated to one predefined semantic nor to a specified rule order, and thus can be used for different rule sets of various complexity and semantics. We are also going to introduce a concept for an efficient vector-based match algorithm which is one key factor for the performance of our reasoner, called AMR (act-mobile reasoner). In the next section we are starting with taking a deeper look on the related work on high performance and parallel reasoning. In section 3 we describe the Rete algorithm and show, how it can be used for parallel inferencing on parallel processors. We will also provide more details on the OpenCL[4] programming model and show how this has an impact to our approach. To evaluate our concept we use different rule sets applied to some well known ontologies of different sizes. Finally we are going to discuss our results and give a conclusion.

[3] http://www.w3.org/2007/OWL/wiki/EL

[4] OpenCL: open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/

2 Related Work

Particularly with regard to large ontologies of hundreds of millions of triples recently the use of clusters for applying finite rules like for RDFS or OWL Horst semantics were a focus of interest in the research community. In [9] and [8] the authors presented WebPie, an inference engine based on the open-source MapReduce implementation Hadoop, which is able to compute RDFS as well as the pD* semantics on data sets containing billions of triples. The parallelisation is achieved by encoding the necessary rules as a set of Map and Reduce operations which are executed in a given order while the distribution of the workload is handled by Hadoop. Other papers propose similar approaches also based on MapReduce differing in the implemented semantics like OWL 2 EL [11] or Fuzzy pD [12]. Another approach relying on multiple computing machines is presented in [13], where a divide-conquer-swap strategy is applied. The input data is stored on a shared location and divided into smaller chunks which are processed by a grid of compute nodes. The results are exchanged between those nodes for further processing. Just like the previous mentioned approaches, the strategy relies on a predefined semantic. Another strategy for parallel reasoning is presented in [14] where the input data is also partitioned and processes running on a cluster computing the finite RDF schema closure for each partition.

Besides the use of a cluster to increase the scaleability, other approaches try to take advantage of the parallel structures of a single machine like available through modern multicore CPUs and GPUs. [15] and [16] for example propose an approach for concurrent classification (TBox) and parallel ABox reasoning for OWL 2 EL. Another interesting reasoning mechanism is presented in [17], where the ρdf vocabulary, which represents a subset of the RDFS rules, is encoded to be applied highly parallel on the graphic processor. While this approach does not only consider ABox or TBox information of an ontology and shows great performance, it still relies on a pre-defined semantic. Nevertheless, this work is most related to our work and will be used for evaluation in section 4.

As outlined by the discussed approaches, many scaleable solutions exists for reasoning on a cluster as well as on a single machine which support different semantics. Nevertheless, as far as we know, there is no scaleable solution using parallel inferencing for a general purpose reasoning process, where the semantic can be defined by the application in terms of simple rules, irrespective of whether these semantics are based on a RDFS or OWL profile or include application specific rules.

3 Parallelising Rule Execution

3.1 OpenCL programming model

OpenCL is a heterogeneous programming framework that allows to develop applications that execute across a range of device types and supports a wide range of parallelism. While the *device* refers to the typically parallel processors like a multicore CPU or a GPU, the *host* can be seen as the outer control logic that

prepares the execution of logic on the device. The logic executed on the device is called kernel and thus is that part of an OpenCL program, that typically is executed in parallel. To parallelise an application, the workload needs to be partitioned into small chunks where each chunk can be computed by the same code in parallel. Each chunk is handled by a *work-item* which runs in its own thread and has a unique global identifier which can be used to identify that part of the input data, that shall be processed by a work-item. In addition, work-items are grouped into *work-groups* which have a limited size but can share local memory which can be accessed much faster than global memory. Nevertheless, local memory is very limited such that it needs to be used wisely. To achieve a high performance it is important to have a kernel with as less control structures that might be evaluated by two work-items in a different way as possible, because this would lead to idling threads during the parallel execution.

3.2 The Rete algorithm

What distinguishes the AMR reasoner from the aforementioned parallel reasoners is that we do not know which rules shall be applied to an ontology and thus can not provide dedicated methods that each implement one rule. Thus, we have to implement a generic rule engine which is able to handle ontological data. One widely used algorithm for this complex task was provided by Charles L. Forgy [10], the *Rete Match* algorithm, which is able to find all the objects matching a given pattern. The Rete algorithm builds a network of nodes where each node corresponds to a pattern occurring on the left hand side (LHS) of a rule (that part, that defines the match conditions). Thus, each pattern on the LHS corresponds to a match condition such as ($?p$ rdfs:domain $?c$). For each single pattern an alpha node is created, while more than one pattern on the LHS in addition leads to at least one beta node. Thus, a beta node always has more than one pattern. For each node a list of matching objects is stored which is created by propagating the working memory and thus the input data through the network.

Considering the following rules from the RDFS semantic:

$$(?x\ \ ?p\ \ ?y) \rightarrow (?p\ \text{rdf:type}\ \ \text{rdf:Property}) \tag{R1}$$

$$(?x\ \ ?p\ \ ?y)\ \ (?p\ \text{rdfs:domain}\ \ ?c) \rightarrow (?x\ \ \text{rdf:type}\ \ ?c) \tag{R2}$$

The Rete algorithm would create one alpha node from the left hand side of the first rule R1, and one more alpha node for the second pattern of the second rule R2. Because the first pattern of R2 is equal to the pattern of R1, both rules would share one node. Because R2 consists of two patterns, in addition one beta node would be created connecting the two other alpha nodes. Further considering the following working memory from a simple university example, which contains three triples consisting of a subject, predicate and object (s, p, o):

$$\text{Bob uni:publishes Paper1} \tag{WM1}$$

Alice uni:publishes Paper2 (WM2)

uni:publishes rdfs:domain Researcher (WM3)

The Rete network resulting by parsing R1 and R2 to alpha- and beta nodes and propagating the working memory through the network can be seen in figure 1.

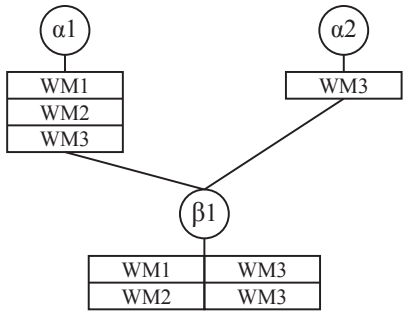


Fig. 1: Rete network for rules R1 and R2 after the working memory has been propagated

The Rete network in figure 1 has three nodes in total, while the final node of R1 is $\alpha 1$ and the final node of R2 is $\beta 1$. A final node always represents a complete rule such that the results stored by that node can be used to fire the corresponding rule. Thus, R1 would fire three times and produce two new triples (and one duplicate) while R2 would fire two times and produce also two new triples. The final working memory would be extended by the following triples:

uni:publishes rdf:type rdf:Property (WM4)

rdfs:domain rdf:type rdf:Property (WM5)

Bob rdf:type Researcher (WM6)

Alice rdf:type Researcher (WM7)

The new triples would also be propagated through the network until no new triples are derived and the rule engine has finished his job (fixpoint iteration).

3.3 Definitions

As can be seen from the previous introduction, the Rete algorithm basically performs three steps to apply rules to a working memory: an alpha-match, a beta-match and the rule firing. Before continuing, we have to define some terms that are used throughout this paper:

Definition 1. *An **alpha node** specifies a node of the Rete network that directly corresponds to one pattern of a rule. An alpha node always has exactly one pattern that consists of three (pattern-)terms.*

Definition 2. *A **beta node** specifies a node of the Rete network that has exactly two parents (p1 and p2). The parents in turn may be alpha- or beta nodes. Depending on the position of a beta node in the network, it has a depth >= 1 that is calculated by the longest distance to an alpha node.*

Definition 3. *The **pattern-width** defines the number of terms of the pattern of one node. Thus, an alpha node always has a pattern-width of 3. A beta node whose parents are both alpha nodes has a pattern-width of 6 and so on.*

Definition 4. *The **matches** of a node are referred to as a set A. Accordingly the number of matches is defined by $|A|$.*

Definition 5. ***Match-conditions** are created for each node and define a function $C(m, ...)$ with $m \in A$, that evaluates to true if a triple or a set of triples (in case of a beta node) conform to the pattern of that node.*

3.4 Rete on Parallel Hardware

To parallelise the overall reasoning process, different strategies were proposed by other approaches which essentially consists of data partitioning and rule partitioning [18]. While data partitioning means the dividing of the data into smaller units and the independent processing of each unit, the rule partitioning approach applies only a subset of the overall rules to the complete data set. Both types of parallelisation require a synchronisation of the results and especially the data partitioning approach may produce duplicates. Nevertheless, these drawbacks might be unavoidable in a cluster-based environment where each of the parallel processes runs in an independent environment.

Using the parallel structures on a single computer, a parallelisation can take place on a different level than rule or data partitioning. This is because in the case of a single computer a synchronisation of interim results of the parallel threads can be performed much more efficient by a single host application. Furthermore, to achieve a high performance for example on a GPU it is important to have lots of tasks that can be computed independently and where each task consists only of a small workload. In order to take these considerations into account, our approach parallelises the reasoning process on a deeper level. For alpha-matching, one kernel is executed where for each triple that needs to be considered a single thread (work-item) is created. This thread checks whether the corresponding triple matches one or more of the alpha node patterns and creates a list containing the matching nodes. Thus, each thread needs to iterate over all of the alpha nodes. Finally the resulting list can simply be transformed to create a match-list for each alpha node containing the matching triples. That means that the number of threads that are executed in parallel is equal to the number of triples available for processing.

Because of its complexity, the beta-match is handled in a different way. Just like the rule partitioning approach it would not be desirable to simply execute one thread for each beta node in parallel because of the low number of nodes and the high computation load. On the other side the number of match-steps, that need

to be computed heavily rises with the number of matches of the parents of one beta node. Considering the example from figure 1, for $\beta 1$ only 3x1 possibilities needed to be computed. If $\alpha 1$ and $\alpha 2$ both had ten matches, the number of possible combinations would arise to 100. By taking this complexity as well as the fact, that the GPU is able to handle millions of work-items in a very efficient way, into account, it is a natural way to apply the parallelism for the beta-step during this match-algorithm. That means, that for each match of one parent of a beta node a work-item is created which iterates over all matches of the second parent of the beta node. Thus, the number i of work-items is defined by:

$$Def: \quad i = |A_{p1}|, \;\; with\; |A_{p1}| >= |A_{p2}| \tag{1}$$

and the number of iterations j each work-item has to perform by:

$$Def: \quad j = |A_{p2}|, \;\; with\; |A_{p2}| < |A_{p1}| \tag{2}$$

Finally the match-algorithm for a beta node is defined as:

$$C(m_i, m_j), \;\; m_i \in A_{p1}, \; m_j \in A_{p2} \tag{3}$$

where m_i denotes the parallelism and i corresponds to the rank of the thread. By defining $|A_{p1}| >= |A_{p2}|$ and $|A_{p2}| < |A_{p1}|$ we ensure, that the number of parallel threads is at least as high as the number of iterations each thread has to perform, which can be computed more efficient on the GPU. Furthermore the match-algorithm needs to be performed for $A_{p1} \times A_{p2}$.

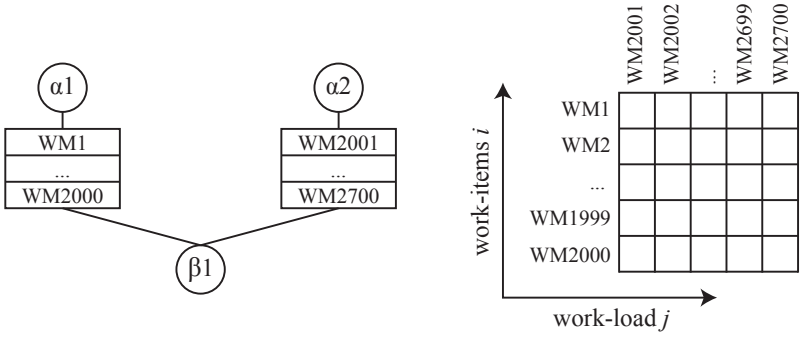


Fig. 2: Parallelise the match process

Figure 2 shows an example where $A_{\beta 1}$ shall be computed. Because $|A_{\alpha 1}| = 2000$ and $|A_{\alpha 2}| = 700$ it follows that $C(m_1, m_2)$ needs to be executed 1.4 million times. To do this, 2000 work-items are created (from each match of $\alpha 1$) where each work-item iterates over 700 matches (from $\alpha 2$) and validates $C(m_1, m_2)$. This task needs to be performed for each beta node, while the beta-match of nodes of the same depth can also run in parallel.

3.5 Applying the OpenCL Programming Model

Programming parallel hardware with OpenCL imposes some restrictions. First of all, strings can not be computed in an efficient way and thus are not suitable for a fast match-algorithm. Thus, all triple terms s, p, *and* o are transformed into integer values, where each term is mapped to one value. Non literal terms are mapped only once such that there is only one index for each term, even if it is used a multiple times within the ontology. Furthermore, the memory used by a kernel needs to be allocated before kernel execution with the consequence, that each alpha- and beta-step has to be performed twice. The first time, only the number of matches is calculated (called match-count) while for the second execution the number of matches is used to allocate the required memory for all matches and thus the final results can be created. Finally, the overall reasoning process consists of an alpha-count where the resulting triples of an alpha match are calculated, an alpha-match, where the final results of the alpha-step are created and an beta-count and beta-match for each depth (see definition 2 in section 3.3). The last step is to fire the rules using the working memories of the final nodes and to create new triples. Eventually the algorithm iterates over the complete process until no new triples are derived (fixpoint). In addition the beta-count processes as well as the beta-match processes can be executed at the same time for all beta nodes of one depth (non blocking operations). This allows an out-of-order execution where the GPU may define the order of given commands to optimise the throughput and is the reason, why we first start all beta-count and beta-match operations and read the result back in a following loop.

3.6 Vector-based Matching

The most computation intensive task during the reasoning process is the beta-matching. Thus it is essential to speed this task up and make the computation as simple as possible. To achieve this, our approach uses a vector-based operation to check, if a combination of triples matches the pattern of a beta node. Vector-based operations can be computed by a GPU in a very efficient way which is why it is desirable to use them. To do this, first of all we use so called unrolled kernels for rules with up to four patterns (which is for example the max pattern-width occurring in the OWL Horst rule set). This means, that instead of using loops iterating over a defined number of items (where in this case the number of items depends on the number of patterns that a beta node has) we just write the command to execute as often as it is needed. The kernels on the other hand are executed by using a kernel implementation depending on the characteristics of the beta node. This also allows us to exactly know the pattern-width of each parent of a beta node during kernel implementation which is the basis for the vector-based match algorithm.

$$(?x\ \ ?p\ \ ?y)\ \ (?p\ \text{rdfs:domain}\ \ ?c) \rightarrow (?x\ \ \text{rdf:type}\ \ ?c) \qquad \text{(R2)}$$

Recapturing rule R2, where the final beta node has two alpha nodes as parents, an unrolled kernel can be executed which assumes a pattern-width of three for both parents. Furthermore it can be seen, that to verify a match only the second term of the first pattern and the first term of the second pattern needs to be checked, such that the value for $?p$ in the first pattern is equal to the value of $?p$ of the second pattern. To do this, in each work-item a vector v_1 is created such that those elements from m_1, that need to be compared to elements of m_2, are placed at the location in v_1 where their corresponding element of m_2 is located. In addition the elements in v_1 need to be negated such that a later performed addition can result in a null vector if the elements of both vectors are equal except of their sign. Besides v_1, another vector u is created which has the same number of elements than v_1 and is filled with elements equal to 1 at those positions, where the second pattern holds an element that needs to be considered to verify a match. All other elements of u are defined as 0. Finally the loop which runs over all matches of A_{p2} only has to create a vector v_2 which holds all elements of m_2. This vector is used to verify for a match as follows:

$$(v_2 * u) + v_1 \tag{4}$$

This operation is performed in a component based manner (meaning that a concurrent, component based multiplication as well as a component based addition is performed) and results in a null-vector, if a match was found. Otherwise, at least one element of the resulting vector is unequal to 0. Furthermore only a simple operation and a minimum of data transfer is necessary within the inner loop of a kernel, which allows a very efficient execution even for a large $|A_{p2}|$.

To continue the example illustrated in figure 1 the working memory of $\alpha 1$ and $\alpha 2$ looks like depicted in figure 3. To improve the readability, not only the

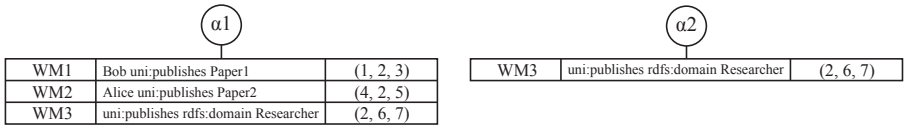


Fig. 3: Working memory of $\alpha 1$ and $\alpha 2$

working memory reference (row 1) is given, but also the data (row 2) as well as the internal representation of that data (row 3). The internal representation, like described before, is a mapping of each subject, predicate and object to an integer value, which is used for computation. Based on the aforementioned description, three parallel threads would be executed to calculated $A_{\beta 1}$. Because every item in the working memory of $\alpha 1$ matches the pattern ($?x$ $?p$ $?y$) and every item in the working memory of $\alpha 2$ matches the pattern ($?p$ rdfs:domain $?c$) (see rule R2) the only calculation necessary to see, if a combination of an $\alpha 1$ and an $\alpha 2$ match also is a match of the beta node $\beta 1$, is to compare the corresponding $?p$ values. To do this, in each thread a constant vector v_1 is created using the corresponding match of $\alpha 1$. Looking at the first thread, v_1 holds the negated

?p-value of the WM1 element, which is positioned at the first vector component, because the corresponding ?p-element of the $\alpha 2$ match is also positioned at the first element. This results in $v_1 = (-2, 0, 0)$. Because only the first component of the vectors need to be considered for R2 (only the ?p elements have to be compared), the vector u is created as $u = (1, 0, 0)$. Now, for each element of $A_{\alpha 2}$ a vector v_2 is created which simply holds the numeric representation of the corresponding triple such that v_2 results in $v_2 = (2, 6, 7)$. The final (component based) calculation is as follows:

$$\begin{pmatrix} 2 \\ 6 \\ 7 \end{pmatrix} * \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -2 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{5}$$

Using the same calculation to match WM3 against WM3 (WM3 is a match of $\alpha 1$ as well as of $\alpha 2$) would not result in a null vector and thus would not be a match:

$$\begin{pmatrix} 2 \\ 6 \\ 7 \end{pmatrix} * \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -6 \\ 0 \\ 0 \end{pmatrix} \neq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{6}$$

In addition to the unrolled and vector based kernels, we also implemented kernels that can be used with a pattern-width larger than 4 which allows to write even more complex rules with an arbitrary size.

4 Evaluation

4.1 ρdf, RDFS and OWL Horst

Because our approach is able to handle a given set of rules independent of the semantic that those rules belong to, we used three different rule sets with a varying complexity, which often were implemented by other reasoners in a manual way. The ρdf vocabulary [19] is a simplified version of the RDFS semantic which consists of all rules of RDFS with at least two rule body-terms. This semantic imposes a more efficient reasoning, while the results of the missing rules are supposed to be created on the fly by a reasoner, if resources are queried. These rules were also implemented by the MapReduce-based approach presented in [9] as well as the by the GPU based approach proposed in [17]. Due to space restrictions, we also refer to [17] for the ρdf rules.

The second rule set consists of the complete RDFS rules like defined by the W3C[5]. This rule set consists of 13 rules with one or two antecedents and is used in several other publications for evaluation purpose [14] [8]. The last rule set formally known as pD* was proposed by Herman J. ter Horst which incorporates RDFS and D entailment and extends these semantics with some basic support for OWL [7]. It provides a complete set of entailment rules and has become a promising ontology language for the semantic web because of its

[5] http://www.w3.org/TR/rdf-mt/#RDFSRules

expressiveness on the one side and its relatively low computation complexity on the other side. For a complete overview of the 22 rules (some of them can be combined as they share the same antecedents resulting in 16 rules) we refer due to space restrictions to [8].

4.2 Test Environment

To achieve comparable results we choose ontologies with various sizes that were already used to evaluate other approaches. Thus, we are using the Vicodi[6] ontology which is an ontology of European history used for semantical indexing of historical documents [20]. The TBox is of a moderate size while the ABox contains a large number of instances. In total, the ontology consists of 146,280 triples and thus is compared to our second ontology, known from the Lehigh University benchmark (LUBM)[7], a small sized ontology. LUBM is a benchmark ontology and defines an TBox for a university scenario. A generator allows to generate university data sets while the number of universities that are created can be defined as an input. Thus, we created 3 LUBM data sets with 268,794 triples which contains two universities (LUBM2), a LUBM5 ontology with 727,265 triples and a LUBM10 ontology with 1,480,366 triples. To use another large ontology we used the DBPedia[8] 3.7 which is a lightweight ontology containing structured data extracted from Wikipedia. For this data set we used a similar setup like described by [17] containing the DBpedia Ontology, Infobox Types and Infobox Properties. We also limited the size of the data set in a similar way by scaling the instance triples by $1/8^{th}$, $1/16^{th}$, and $1/32^{nd}$ of the original size.

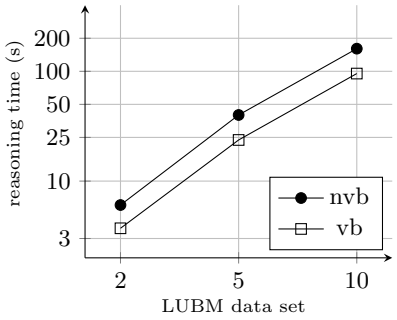
Our implementation is Java based and integrates with Jena[9] applications. Thus, we use the Jena framework to parse the ontologies and create our own data structures for the reasoning process by reading an ontology graph from Jena. To be able to use OpenCL from our Java application, we use jocl[10] as Java bindings. The tests are performed on a work station with a 2.0 GHz Intel Xeon processor with 6 cores and an AMD 7970 gaming graphic card with 3GB of memory running an Ubuntu 12.04. In order to compare our results to other approaches, we performed the same tests with the ρdf rule set with the GPU based reasoner proposed in [17]. In the following we refer to that reasoner as grdfs reasoner. Other parallel reasoners also implementing the complete RDFS or OWL Horst rules on a single computer considering the ABox as well as the TBox were not available. We also run our experiment using the non-vector-based kernel and compare the results with the vector-based version. For each experiment the total time except data transformation (i.e. loading the Jena-graph to AMR and file parsing for grdfs) were measured, which also includes the non parallel rule firing. A dedicated kernel execution time on the GPU is not given due to the

[6] http://www.vicodi.org/about.htm
[7] http://swat.cse.lehigh.edu/projects/lubm/
[8] http://dbpedia.org/About
[9] http://jena.apache.org/
[10] http://www.jocl.org/

execution of multiple kernels which are launched asynchronously such that no precise information are available. Furthermore each test was performed ten times while the average is presented.

4.3 Results and Comparison

The first experiment shows the impact of using the vector-based operations during the beta-match. Therefore we used all three LUBM data sets and executed it using the pD* rule set with the naive implementation of the kernel and with the vector-based kernel. Notice that both kernels are unrolled and the only difference is, that the second kernel uses vector operations instead of calculating each element in a single step. We choose the pD* vocabulary because it is the most computation intensive one of the three used rule sets.



LUBM	triples	entailed	nvb	vb	speedup
2	268,794	38,584	6.05 s	3.72 s	1.63
5	727,265	102,618	40.01 s	23.69 s	1.69
10	1,480,366	207,677	160.80 s	95.39 s	1.69

Fig. 4: Comparison of non-vector-based operations (nvb) and vector-based (vb) operations

The results in figure 4 show that the vector-based kernel provides a constantly better performance. The calculation using the vector-based method is round about 1.7 times faster than using the naive implementation which simply applies the comparison for two matches of the corresponding parent-nodes using single operations. On a different hardware (MacBook Pro) even a speedup of more than 4 could be measured using the vector-based operation.

The next test shall show the scaleability of our approach. Therefore we use the Vicodi ontology as well as the LUBM2 ontology and run our algorithm on the CPU of our test environment. The CPU has less cores than the GPU and is not that fast, but OpenCL allows us to use only a defined number of cores of the CPU. This way it is possible to show the speedup which is achieved by increasing the number of used cores. Because the used CPU has 6 cores where each core can run 2 threads through hyper-threading (resulting in 12 virtual cores), we applied the pD* vocabulary to the input data using 1 to 12 cores.

As can be seen from figure 5 the speedup nearly doubles with a doubling of the number of used processors for both datasets until 6 cores are used. The

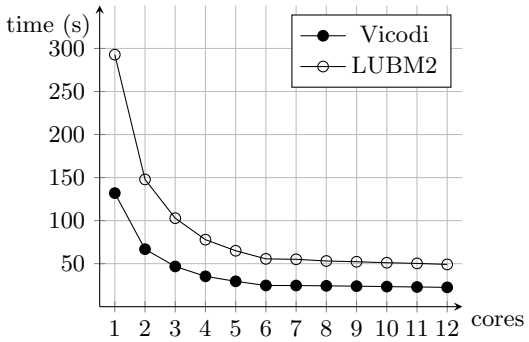


Fig. 5: Scaleability test using the Vicodi and the LUBM2 ontology on the CPU

other 6 cores still contribute to a better performance, but the impact is not that reasonable anymore. We assume that this is because two (virtual) cores always share some resources on one processor and thus do not provide such a speedup as is would be achieved if each core had physically exclusive resources.

Finally we want to compare our results to other approaches. For this we performed a set of tests with the GPU based grdfs reasoner proposed in [17] as well as with AMR. Both reasoners used the same hardware like described before. Because the grdfs reasoner implements the ρdf semantic, the tests were only performed using the corresponding rules. While we can not prove that our implementation works correct, we still get exactly the same results using the general purpose rule engine provided by the Jena framework for ontologies with a limited size which that rule engine is able to handle. The largest ontology we were able to test with Jena using the ρdf vocabulary was the LUBM2 ontology with 268,794 triples (and 146 entailed triples), which took about 27 minutes, while our approach entails exactly the same triples in about 270 ms. We also tested for example the Vicodi ontology using the Jena framework and the RDFS vocabulary which took about 12 minutes and inferred 127,886 triples to a total of 274,233. Our approach again provides exactly the same results using the GPU in less than a second. The final results of our tests including parallel and serial work (the complete reasoning process excluding parsing) are listed in table 1.

	triples	AMR	grdfs	entailed AMR	entailed grdfs	speedup
LUBM2	268,794	276 ms	1,383 ms	146	22	5.02
LUBM5	727,265	447 ms	3,153 ms	146	22	7.05
LUBM10	1,480,366	676 ms	6,207 ms	146	22	9.22
DBPedia 1/32	1,087,364	3,061 ms	7,554 ms	1,087,364	1,085,309	2.47
DBPedia 1/16	2,276,510	5,931 ms	14,681 ms	1,936,950	1,934,887	2.48
DBPedia 1/8	4,523,729	10,954 ms	27,739 ms	3,083,513	3,081,433	2.53

Table 1: Reasoning time for different data sets using AMR and grdfs reasoner

The experiment shows that our approach provides a speedup of a factor of up to 9.2 compared to the grdfs reasoner. While the speedup for the LUBM data sets is more significant than for the DBPedia data sets, it is still more than two times faster. The different speedup factor results from the fact that using the DBPedia data sets many new triples are inferred, such that up to 60% of the reasoning time of the AMR reasoner is needed for the serial implemented rule firing, which also includes operations like dictionary lookup to not infer duplicate triples.

Further results from other work for comparison can be used for example from [14], where the authors evaluated an approach to parallelise the RDFS closure using an Opteron blade cluster, each server in the cluster having two dual-core 2.6 GHz AMD Opteron processors. The LUBM10k/1024 data set from that paper has a slightly smaller size and a similar complexity like the LUBM10 data set used for this paper. While the approach from [14] took about 2 seconds in total to calculate the closure using a cluster of 128 cores, our approach calculates the RDFS closure on a single machine in about 3 seconds. On a MacBook Pro with a Core i7 processor, which has a less powerful GPU but due to the CPU a faster architecture for serial calculations, the same test could even be finished in less than 2.6 seconds using the AMR reasoner. Nevertheless, the approach from [14] is able to handle much larger data sets.

5 Discussion and Future Work

The results in section 4.3 show that our approach offers a good and scaleable performance using a single computer, also for data sets with millions of triples. Nevertheless, there are still restrictions regarding the size of an ontology. On the one hand for a performant execution the main memory of the host computer needs to be large enough to hold the complete ontology as well as inferred matches and data structures that are used for the rule execution. On the other hand the use of integers for the created index structures limits the number of processable triples. This limitation is even stronger regarding the matches-arrays of the single nodes which easily needs to hold a multiple of elements as triples are available. To overcome this issues, the use of 64bit datatypes as well as appropriate collection types to hold the triples and matches should be considered. In addition the use of collection oriented matching like describe in [21] could be considered, where matches are calculated and stored in a collection-oriented way instead of using single tuples. Furthermore a partitioning strategy could be implemented that allows to distribute the workload of large ontologies over multiple GPUs as well as over multiple machines. Thus, a combination of the cluster-based approach used in [14] and the low level parallelisation like described in this paper might be an interesting approach. Another optimisation might be possible by parallelising the rule firing, too, which will require thread safe data structures and a concept to detect duplicates.

Besides optimisations regarding the performance and the ability to handle larger data sets, in the future we are also going to investigate how we can ex-

tend the functionality of our rule-based system to also support operants like $greaterThan(?x, ?y)$ within a rule body. This way our system would offer much more flexibility for scenarios with application specific rules like used in different kinds of smart environments like [2] [5].

6 Conclusion

In the past most of the approaches to parallelise the reasoning process have focused on distributing the workload over multiple machines to use a large number of processors. Only a few approaches already considered the use of the parallel structures available on a single machine. All approaches have in common, that they implement a defined set of rules and can not be configured in an application specific way. In this paper we proposed a rule-based approach that is independent from a specific semantic and uses the parallel structures of modern CPUs as well as of GPUs. The high performance is achieved by parallelising the Rete algorithm and breaking the match-steps into fine grained tasks which can be computed highly parallel. We also introduced a vector-based operation to compute the beta matches, which easily doubles the performance of the algorithm running on a GPU. Finally our results show, that the approach scales well with the number of used cores and can apply a set of rules to an ontology in a very performant way. Thus the parallelisation of a generic rule-based approach to apply rules on ontological data can be very efficient, if the workload is partitioned into an adequate number of units which can be computed highly parallel.

References

1. Ausín, D., Castanedo, F., López-de Ipiña, D.: Benchmarking results of semantic reasoners applied to an ambient assisted living environment. In: Proceedings of the 10th international smart homes and health telematics conference on Impact Ananlysis of Solutions for Chronic Disease Prevention and Management. ICOST'12, Berlin, Heidelberg, Springer-Verlag (2012) 282–285
2. Agostini, A., Bettini, C., Riboni, D.: A performance evaluation of ontology-based context reasoning. In: Pervasive Computing and Communications Workshops, 2007. PerCom Workshops '07. Fifth Annual IEEE International Conference on. (2007) 3–8
3. Pantsar-Syvaniemi, S., Simula, K., Ovaska, E.: Context-awareness in smart spaces. In: Computers and Communications (ISCC), 2010 IEEE Symposium on. (2010) 1023–1028
4. Reinisch, C., Kofler, M., Kastner, W.: Thinkhome: A smart home as digital ecosystem. In: Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on. (2010) 256–261
5. Peters, M., Brink, C., Sachweh, S., Zündorf, A.: Performance considerations in ontology based ambient intelligence architectures. In: Proceedings of the 4th International Symposium on Ambient Intelligence. (2013)
6. Kazakov, Y., Krötzsch, M., Simančík, F.: ELK: a reasoner for OWL EL ontologies. System description, University of Oxford. In: Technical Report (2012)

7. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Web Semantics: Science, Services and Agents on the World Wide Web **3**(2-3) (October 2005) 79–115
8. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: Owl reasoning with webpie: calculating the closure of 100 billion triples. In: Proceedings of the 7th international conference on The Semantic Web: research and Applications - Volume Part I. ESWC'10, Berlin, Heidelberg, Springer-Verlag (2010) 213–227
9. Urbani, J., Kotoulas, S., Oren, E., Harmelen, F.: Scalable distributed reasoning using mapreduce. In: Proceedings of the 8th International Semantic Web Conference. ISWC '09, Berlin, Heidelberg, Springer-Verlag (2009) 634–649
10. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. In Raeth, P.G., ed.: Expert systems. IEEE Computer Society Press, Los Alamitos, CA, USA (1990) 324–341
11. Maier, F., Mutharaju, R., Hitzler, P.: Distributed reasoning with EL++ using mapreduce. Technical report, Kno.e.sis Center, Wright State University, Dayton, Ohio (2010)
12. Liu, C., Qi, G., Wang, H., Yu, Y.: Reasoning with large scale ontologies in fuzzy pd* using mapreduce. Computational Intelligence Magazine, IEEE **7**(2) (2012) 54–66
13. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: A platform for large-scale analysis of semantic web data. In: Proceedings of the WebScience '09, Society On-Line (2009)
14. Weaver, J., Hendler, J.: Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In Bernstein, A., Karger, D., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K., eds.: The Semantic Web - ISWC 2009. Volume 5823 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 682–697
15. Kazakov, Y., Krötzsch, M., Simancík, F.: Concurrent classification of EL ontologies. In: Proceedings of the 10th international conference on The semantic web - Volume Part I. ISWC'11, Berlin, Heidelberg, Springer-Verlag (2011) 305–320
16. Ren, Y., Pan, J.Z., Lee, K.: Parallel abox reasoning of EL ontologies. In: Proceedings of the 2011 joint international conference on The Semantic Web. JIST'11, Berlin, Heidelberg, Springer-Verlag (2012) 17–32
17. Heino, N., Pan, J.: RDFS reasoning on massively parallel hardware. In: The Semantic Web ISWC 2012. Volume 7649 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 133–148
18. Soma, R., Prasanna, V.: Parallel inferencing for OWL knowledge bases. In: Parallel Processing, 2008. ICPP '08. 37th International Conference on. (2008) 75–82
19. Muoz, S., Prez, J., Gutierrez, C.: Minimal deductive systems for RDF. In Franconi, E., Kifer, M., May, W., eds.: The Semantic Web: Research and Applications. Volume 4519 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 53–67
20. Nagypl, G., Motik, B.: A fuzzy model for representing uncertain, subjective, and vague temporal knowledge in ontologies. In Meersman, R., Tari, Z., Schmidt, D., eds.: On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE. Lecture Notes in Computer Science. Springer Berlin Heidelberg (2003) 906–923
21. Acharya, A., Tambe, M.: Collection oriented match. In: Proceedings of the second international conference on Information and knowledge management. CIKM '93 (1993) 516–526

TripleRush: A Fast and Scalable Triple Store

Philip Stutz, Mihaela Verman, Lorenz Fischer, and Abraham Bernstein

DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
{stutz, verman, lfischer, bernstein}@ifi.uzh.ch

Abstract. TripleRush is a parallel in-memory triple store designed to address the need for efficient graph stores that quickly answer queries over large-scale graph data. To that end it leverages a novel, graph-based architecture.
Specifically, TripleRush is built on our parallel and distributed graph processing framework Signal/Collect. The index structure is represented as a graph where each index vertex corresponds to a triple pattern. Partially matched copies of a query are routed in parallel along different paths of this index structure.
We show experimentally that TripleRush takes less than a third of the time to answer queries compared to the fastest of three state-of-the-art triple stores, when measuring time as the geometric mean of all queries for two benchmarks. On individual queries, TripleRush is up to three orders of magnitude faster than other triple stores.

1 Introduction

Many applications such as social network analysis, monitoring of financial transactions or analysis of web pages and their links require large-scale graph computation. To address this need, many have researched the development of efficient triple stores [1, 14, 11]. These systems borrow from the database literature to investigate efficient means for storing large graphs and retrieving subgraphs, which are usually defined via a pattern matching language such as SPARQL. This avenue has had great success both in research [1, 14, 11] and practice.[1] However, many of these systems are built like a centralized database, raising the question of scalability and parallelism within query execution. One way to increase the efficiency of parallel pipelined joins in such centralized databases is the use of sideways information passing [10].

Other approaches focus on adapting triple stores to a distributed setting: MapReduce [3] has been used to aggregate results from multiple single-node RDF stores in order to support distributed query processing [5].

Others have mapped SPARQL query execution pipelines to chains of MapReduce jobs (e.g., [6]). Whilst this provides scalability, the authors point out that the rigid structure of MapReduce and the high latencies when starting new jobs constrain the possibilities for optimizations [6]. Distributed graph processing frameworks such as Pregel [8], GraphLab/Powergraph [7, 4], Trinity [12], and

[1] http://virtuoso.openlinksw.com, http://www.ontotext.com/owlim

our own SIGNAL/COLLECT [13] can offer more flexibility for scalable querying of graphs.

Among the existing triple stores we only know of Trinity.RDF [15] to be implemented on top of such an abstraction. Trinity.RDF is a graph engine for SPARQL queries that was built on the Trinity distributed graph processing system. To answer queries, Trinity.RDF represents the graph with adjacency lists and combines traditional query processing with graph exploration.

In this paper we introduce TripleRush, a triple store which is based on an *index graph*, where a basic graph pattern SPARQL query is answered by routing partially matched query copies through this index graph. Whilst traditional stores pipe data through query processing operators, TripleRush routes query descriptions to data. For this reason, TripleRush does not use any joins in the traditional sense but searches the index graph in parallel. We implemented TripleRush on top of SIGNAL/COLLECT, a scalable, distributed, parallel and vertex-centric graph processing framework [13].

The contributions of this paper are the following: First, we present the TripleRush architecture, with an index graph consisting of many active processing elements. Each of these elements contains a part of the processing logic as well as a part of the index. The result is a highly parallel triple store based on graph-exploration. Second, as a proof of concept, we implemented the TripleRush architecture within our graph processing framework SIGNAL/COLLECT, benefiting from transparent parallel scheduling, efficient messaging between the active elements, and the capability to modify a graph during processing. Third, we evaluated our implementation and compared it with three other state-of-the-art in-memory triple stores using two benchmarks based on the LUBM and DBPSB datasets. We showed experimentally that TripleRush outperforms the other triple stores by a factor ranging from 3.7 to 103 times in the geometric mean of all queries. Fourth, we evaluated and showed data scalability for the LUBM benchmark. Fifth, we measured memory usage for TripleRush, which is comparable to that of traditional approaches. Last, we open sourced our implementation.[2]

In the next section we discuss the infrastructural underpinnings of TripleRush. This is followed by a description of the TripleRush architecture, as well as the functionality and interactions of its building blocks. We continue with a description of the optimizations that reduce memory usage and increase performance. We evaluate our implementation, discuss some of this paper's findings as well as limitations, and finish with some closing remarks.

2 Signal/Collect

In this section we describe the scalable graph processing system SIGNAL/COLLECT and some of the features that make it a suitable foundation for TripleRush.

[2] Apache 2.0 licensed, `https://github.com/uzh/triplerush`

SIGNAL/COLLECT [13][3] is a parallel and distributed large-scale graph processing system written in Scala. Akin to Pregel [8], it allows to specify graph computations in terms of vertex centric methods that describe aggregation of received messages (collecting) and propagation of new messages along edges (signalling). The model is suitable for expressing data-flow algorithms, with vertices as processing stages and edges that determine message propagation. In contrast to Pregel and other systems, SIGNAL/COLLECT supports different vertex types for different processing tasks. Another key feature, also present in Pregel, is that the graph structure can be changed during the computation. The framework transparently parallelizes and distributes the processing of data-flow algorithms. SIGNAL/COLLECT also supports features such as bulk-messaging and Pregel-like message combiners to increase the message-passing efficiency.

Most graph processing systems work according to the bulk-synchronous parallel model. In such a system, all components act in lock-step and the slowest part determines the overall progress rate. In a query processing use-case, this means that one expensive partial computation would slow down all the other ones that are executed in the same step, which leads to increased latency. SIGNAL/COLLECT supports asynchronous scheduling, where different partial computations progress at their own pace, without a central bottleneck. The system is based on message-passing, which means that no expensive resource locking is required. These two features are essential for low-latency query processing.

With regard to the representation of edges, the framework is flexible. A vertex can send messages to any other vertex: Edges can either be represented explicitly or messages may contain vertex identifiers from which virtual edges are created. TripleRush uses this feature to route query messages.

3 TripleRush

The core idea of TripleRush is to build a triple store with three types of SIGNAL/COLLECT vertices: Each *index vertex* corresponds to a triple pattern, each *triple vertex* corresponds to an RDF triple, and *query vertices* coordinate query execution. Partially matched copies of queries are routed in parallel along different paths of this structure. The index graph is, therefore, optimized for efficient routing of query descriptions to data and its vertices are addressable by an ID, which is a unique [subject predicate object] tuple.

We first describe how the graph is built and then explain the details of how this structure enables efficient parallel graph exploration.

3.1 Building the Graph

The TripleRush architecture is based on three different types of vertices. *Index and triple vertices* form the *index graph*. In addition, the TripleRush *graph* contains a *query vertex* for every query that is currently being executed. Fig. 1 shows the index graph that is created for the triple [Elvis inspired Dylan]:

[3] `http://uzh.github.io/signal-collect/`

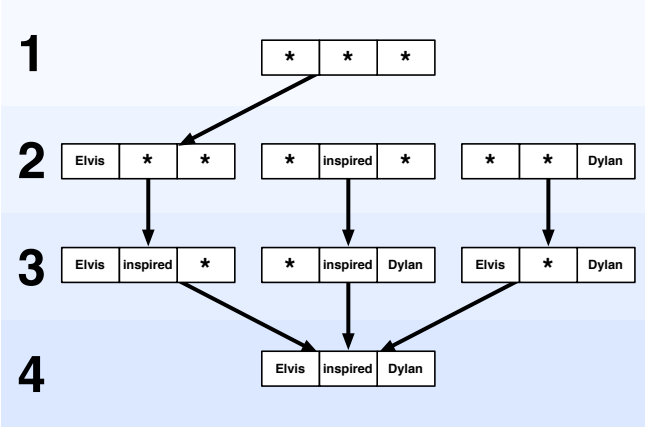


Fig. 1. TripleRush index graph that is created for the triple vertex [Elvis inspired Dylan].

Triple vertices are illustrated on level 4 of Fig. 1 and represent triples in the database. Each contains subject, predicate, and object information.

Index vertices, illustrated in levels 1 to 3 in Fig. 1, represent triple patterns and are responsible for routing partially matched copies of queries (referred to as *query particles*) towards triple vertices that match the pattern of the index vertex. Index vertices also contain subject, predicate, and object information, but one or several of them are wildcards. For example, in Fig. 1 the index vertex [* inspired *] (in the middle of the figure on level 2) routes to the index vertex [* inspired Dylan], which in turn routes to the triple vertex [Elvis inspired Dylan].

Query vertices, depicted in the example in Fig. 2, are query dependent. For each query that is being executed, a query vertex is added to the graph. The query vertex emits the first query particle that traverses the index graph. All query particles—successfully matched or not—eventually get routed back to their respective query vertex. Once all query particles have succeeded or failed the query vertex reports the results and removes itself from the graph.

The *index graph* is built by adding a *triple vertex* for each RDF triple that is added to TripleRush. These vertices are added to Signal/Collect, which turns them into parallel processing units. Upon initialization, a triple vertex will add its three parent *index vertices* (on level 3) to the graph and add an edge from these index vertices to itself. Should any parent index vertex already exist, then only the edge is added from this existing vertex.

When an *index vertex* is initialized, it adds its parent index vertex, as well as an edge from this parent index vertex to itself. Note that the parent index vertex always has one more wildcard than its child. The construction process continues recursively until the parent vertex has already been added or the index vertex has no parent. In order to ensure that there is exactly one path from an index

vertex to all triple vertices below it, an index vertex adds an edge from at most one parent index vertex, always according to the structure illustrated in Fig. 1.

Next we describe how the index graph allows parallel graph exploration in order to match SPARQL queries.

3.2 Query Processing

The index graph we just described is different from traditional index structures, because it is designed for the efficient parallel routing of messages to triples that correspond to a given triple pattern. All vertices that form the index structure are active parallel processing elements that only interact via message passing.

A query is defined by a list of SPARQL triple patterns. Each query execution starts by adding a query vertex to the TripleRush graph. Upon initialization, a *query vertex* emits a single query particle. A query particle consists of the list of unmatched triple patterns, the ID of its query vertex, a list of variable bindings, a number of tickets, and a flag that indicates if the query execution was able to explore all matching patterns in the index graph. Next, we describe how the parts of the query particle are modified and used during query execution.

The emitted particle is routed (by SIGNAL/COLLECT) to the index vertex that matches its first unmatched triple pattern. If that pattern is, for example, [Elvis inspired ?person], where ?person is a variable, then it will be sent to the index vertex with ID [Elvis inspired *]. This index vertex then sends copies of the query particle to all its child vertices.

Once a query particle reaches a triple vertex, the vertex attempts to match the next unmatched query pattern to its triple. If this succeeds, then a variable binding is created and the remaining triple patterns are updated with the new binding. If all triple patterns are matched or a match failed,[4] then the query particle gets routed back to its query vertex. Otherwise, the query particle gets sent to the index or triple vertex that matches its next unmatched triple pattern.

If no index vertex with a matching ID is found, then a handler for undeliverable messages routes the failed query particle back to its query vertex. So no matter if a query particle found a successful binding for all variables or if it failed, it ends up being sent back to its query vertex.

In order to keep track of query execution and determine when a query has finished processing, each query particle is endowed with a number of tickets. The first query particle starts out with a very large number of tickets.[5]

When a query particle arrives at an index vertex, a copy of the particle is sent along each edge. The original particle evenly splits up its tickets among its copies. If there is a remainder, then some particles get an extra ticket. If a particle does not have at least one ticket per copy, then copies only get sent

[4] A match fails if it creates conflicting bindings: Pattern [?X inspired ?X] fails to bind to the triple [Elvis inspired Dylan], because the variable ?X cannot be bound to both Elvis and Dylan.

[5] We use `Long.MaxValue`, which has been enough for a complete exploration of all queries on all datasets that we have experimented with so far.

along edges for which at least one ticket was assigned, and those particles get flagged to inform the query vertex that not all matching paths in the graph were explored. Query execution finishes when the sum of tickets of all failed and successful query particles received by the query vertex equals the initial ticket endowment of the first particle that was sent out.

Once query execution has finished, the query vertex reports the result that consists of the variable bindings of the successful query particles, and then removes itself from the graph.

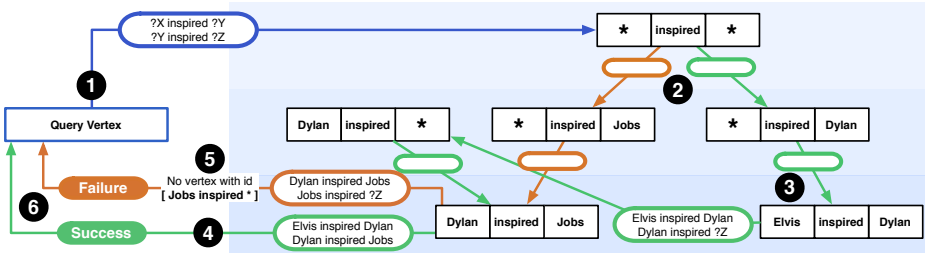


Fig. 2. Query execution on the relevant part of the index that was created for the triples [Elvis inspired Dylan] and [Dylan inspired Jobs].

As an example illustrating a full query execution, consider the relevant subgraph created for the triples [Elvis inspired Dylan] and [Dylan inspired Jobs], shown in Fig. 2 along with the query processing for the query: (unmatched = [?X inspired ?Y], [?Y inspired ?Z]; bindings = {}). Execution begins in the query vertex.

1. Once the query vertex has been added to the graph, it emits a query particle, which is illustrated in blue. Given its first triple pattern, the query particle is routed to the index vertex with ID [* inspired *].
2. Inside the index vertex, the query particle is split into two particles, one colored green and the other one orange (for illustration). The tickets of the blue particle are evenly split among the green and the orange particle. Both particles are sent down to their respective index vertex, the green one to [* inspired Dylan] and the orange one to [* inspired Jobs]. These index vertices, in turn, send the particles further down to their corresponding triple vertices.
3. The first pattern of the green particle gets matched in the triple vertex [Elvis inspired Dylan]. The triple vertex sends the updated particle (unmatched = [Dylan inspired ?Z]; bindings = { ?X=Elvis, ?Y=Dylan }) to the index vertex with ID [Dylan inspired *], which in turn routes the particle towards all triples that match the next unmatched pattern, [Dylan inspired ?Z].
4. From the index vertex, the green particle is routed down to the triple vertex [Dylan inspired Jobs], which binds ?Z to Jobs. As there are no more unmatched triple patterns, the triple vertex routes the particle containing successful bindings for all variables back to its query vertex.

5. The first pattern of the orange particle gets matched in the triple vertex [Dylan inspired Jobs]. The triple vertex sends the updated particle (unmatched = [Jobs inspired ?Z]; bindings = { ?X=Dylan, ?Y=Jobs }) to the index vertex with ID [Jobs inspired *]. The message cannot be delivered, because no index vertex with that ID is found. The handler for undeliverable messages reroutes the failed query particle to its query vertex.
6. The query vertex receives both the successfully bound green and the failed orange particle. Query execution has finished, because all tickets that were sent out with the initial blue particle have been received again. The query vertex reports the successful bindings { ?X=Elvis, ?Y=Dylan, ?Z=Jobs } and then removes itself from the graph.

The architecture and query execution scheme we described captures the essence of how TripleRush works. Next, we explain how we improved them with regard to performance and memory usage.

3.3 Optimizations

The system, as previously shown, already supports parallel graph exploration. Here we describe how we implemented and optimized different components.

Dictionary Encoding While the examples we gave so far represented triple patterns in terms of strings, the actual system implementation operates on a dictionary encoded representation, where RDF resource identifiers and literals are encoded by numeric IDs. Both wildcards and variables are also represented as numeric IDs, but variable IDs are only unique in the context of a specific query.

Index Graph Structure We remove triple vertices, and instead store the triple information inside each of the three third level index vertices that have a compatible triple pattern. We hence refer to these index vertices as *binding index vertices*, because they can bind query particles to triples, which was previously done by the triple vertices. This change saves memory and reduces the number of messages sent during query execution.

One question that arises from this change is: If the subject, predicate and object of the next unmatched pattern of a particle are already bound, where should that particle be routed to? With no single triple vertex responsible anymore, TripleRush load balances such routings over the three binding index vertices into which the triple information was folded.

Index Vertex Representation In Fig. 1, one notices that the ID of an index vertex varies only in one position—the subject, the predicate, or the object—from the IDs of its children. To reduce the size of the edge representations, we do not store the entire ID of child vertices, but only the specification of this position

consisting of one dictionary encoded number per child. We refer to these numbers as *child-ID-refinements*. The same reasoning applies to binding index vertices, where the triples they store only vary in one position from the ID of the binding index vertex. Analogously, we refer to these differences as *triple-ID-refinements*.

Binding index vertices need to be able to check quickly if a triple exists. This requires fast search over these triple-ID-refinements. We support this by storing them in a sorted array on which we use the binary search algorithm.

Index vertices of levels 1 and 2 do not need to check for the existence of a specific child-ID, as these levels always route to all children. Routing only requires a traversal of all child-ID-refinements. To support traversal in a memory-efficient way, we sort the child-ID-refinements, perform delta encoding on them, and store the encoded deltas in a byte array, using variable length encoding.

Note that these array representations are inefficient for modifications, which is why we initially store the child/triple-ID-refinements in tree sets and switch the representation to arrays once the loading is done. This design supports fast inserts at the cost of increased memory usage during loading.

Query Optimization The number of particle splits performed depends on the order in which the triple patterns of a query are explored. One important piece of information to determine the best exploration order is the number of triples that match a given triple pattern, which we refer to as the cardinality of the pattern. Because only relative cardinality differences matter for ordering, we can assume a very large number for the root vertex. The binding index vertices already store all the triples that match their pattern and thus have access to their cardinalities. So we only need to determine the cardinality of index vertices on level 2, below the root vertex and above the binding index vertices. A level 2 index vertex requests cardinality counts from its binding index vertex children and sums up the replies. We do this once after graph loading and before executing queries, but it can be done at any time and could also be done incrementally.

Query optimization happens only once inside the query vertex before the query particle is emitted. To support it, the query vertex first sends out cardinality requests to the vertices in the index graph that are responsible for the respective triple patterns in the query. These requests get answered in parallel and, once all cardinalities have been received, we greedily select the pattern with the lowest cardinality to be matched first. If this match will bind a variable, we assume that the cardinality of all other patterns that contain this variable is reduced, because only a subset of the original triples that matched the pattern would be explored at that point. To this end, we divide the cardinality estimate for each triple pattern containing bound variables by a constant per bound variable. In our experiments we set the constant to 100, based on exploration.[6] If all variables in a pattern are bound (meaning that all its variables appear in

[6] We tried different factors and this one performed well on the LUBM benchmark. It also performed well on the DBPSB benchmark, which suggests that it generalizes at least to some degree.

patterns that will get matched before it), then we assume a cardinality of 1, designating that at most one triple could match this pattern.

Repeating the procedure we choose the next pattern with the lowest cardinality estimate, until all patterns have been ordered. Next, the initial query particle and all its copies explore the patterns in the order specified by the optimization.

Optimizations to Reduce Messaging Each TripleRush vertex is transparently assigned to a SIGNAL/COLLECT worker. Workers are comparable to a thread that is responsible for messaging and for scheduling the execution of its assigned vertices.

Sending many individual messages between different SIGNAL/COLLECT workers is inefficient, because it creates contention on the worker message queues. In order to reduce the number of messages sent, we use a bulk message bus that bundles multiple messages sent from one worker to another. In order to reduce message sizes and processing time in the query vertex, we do not send the actual failed particle back to the query vertex, but only its tickets.[7] We also use a Pregel-like combiner that sums up the tickets in the bulk message bus, to again reduce the number of messages sent.

Because the query vertex is a bottleneck, we further reduce the number of messsages it receives and the amount of processing it does by combining multiple successful particles into one result buffer before sending them. The query vertex can concatenate these result buffers in constant time.

4 Evaluation

In the last section we described the TripleRush architecture and parallel query execution. In this section we evaluate its performance compared to other state-of-the-art triple stores.

4.1 Performance

TripleRush was built and optimized for query execution performance. In order to evaluate TripleRush, we wanted to compare it with the fastest related approaches. Trinity.RDF [15] is also based on a parallel in-memory graph store, and it is, to our knowledge, the best performing related approach. Thus, our evaluation is most interesting in a scenario where it is comparable to that of Trinity.RDF. As Trinity.RDF is not available for evaluation, we decided to make our results comparable by closely following the setup of their published parallel evaluations. The Trinity.RDF paper also includes results for other in-memory and on-disk systems that were evaluated with the same setup, which allows us to compare TripleRush with these other systems in terms of performance.

[7] The flag is also necessary and in practice we encode it in the sign.

Datasets and Queries Consistent with the parallel Trinity.RDF [15] evaluation, we benchmarked the performance of TripleRush by executing the same seven queries on the LUBM-160 dataset (∼21 million triples) and the same eight queries on the DBPSB-10 dataset (∼14 million triples). The LUBM (Lehigh University Benchmark) dataset is a synthetic one, generated with UBA1.7,[8] while the DBPSB (DBpedia SPARQL Benchmark) dataset is generated based on real-world DBpedia data [9].[9] The queries cover a range of different pattern cardinalities, result set sizes and number of joins. The queries L1-L7 and D1-D8 are listed in the Appendix. These queries only match basic graph patterns and do not use features unsupported by TripleRush, such as aggregations or filters. More information about the datasets and the queries is found in [2] and [15].

Evaluation Setup In the Trinity.RDF paper, all triple stores are evaluated in an in-memory setting, while RDF-3X and BitMat are additionally evaluated in a cold cache setting [15].

For evaluating TripleRush, we executed all queries on the same JVM running on a machine with two twelve-core AMD Opteron™ 6174 processors and 66 GB RAM, which is comparable to the setup used for the evaluation of Trinity.RDF.[10] The whole set of queries was run 100 times before the measurements in order to warm up the JIT compiler, and garbage collections were triggered before the actual query executions. All query executions were complete, no query particle ever ran out of tickets. We repeated this evaluation 10 times.[11]

The execution time covers everything from the point where a query is dispatched to TripleRush until the results are returned. Consistent with the Trinity.RDF setup[12], the execution times *do* include the time used by the query optimizer, but *do not* include the mappings from literals/resources to IDs in the query, nor the reverse mappings for the results.

Result Discussion The top entries in Tables 1 and 2 show the minimum execution times over 10 runs. According to our inquiry with the authors of the

[8] http://swat.cse.lehigh.edu/projects/lubm

[9] `http://aksw.org/Projects/DBPSB.html`, dataset downloaded from `http://dbpedia.aksw.org/benchmark.dbpedia.org/benchmark_10.nt.bz2`

[10] The evaluation in [15] was done on two Intel Xeon E5650 processors with 96 GB RAM. The review at `http://www.bit-tech.net/hardware/cpus/2010/03/31/amd-opteron-6174-vs-intel-xeon-x5650-review/11` directly compares the processors and gives our hardware a lower performance score.

[11] The operating system used was Debian 3.2.46-1 x86_64 GNU/Linux, running the Oracle JRE version 1.7.0_25-b15. More details are available in the benchmarking code on GithHub at `https://github.com/uzh/triplerush/tree/evaluation-ssws`, classes com/signalcollect/triplerush/evaluation/LubmBenchmark.scala and com/signalcollect/triplerush/evaluation/DbpsbBenchmark.scala, using dependency `https://github.com/uzh/signal-collect/tree/evaluation-ssws`. The full raw results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldEVWVU05blNLUHhTM1hhVTYySHp2MkE`

[12] We inquired about what is included in the execution time for the systems in [15].

Trinity.RDF paper [15], this is consistent with their measures. Additionally, we also report the average execution times for TripleRush. TripleRush performs fastest on six of the seven LUBM queries, and on all DBPSB queries. For the query where TripleRush is not the fastest system, it is the second fastest system.

On all queries, TripleRush is consistently faster than Trinity.RDF. In the geometric mean of both benchmarks, TripleRush is more than three times faster than Trinity.RDF, between seven and eleven times faster than RDF-3X (in memory) and between 31 and 103 times faster than BitMat (in memory). For individual queries the results are even more pronounced: On query L7 TripleRush is about ten times faster than Trinity.RDF, on L1 it is more than two orders of magnitude faster than RDF-3X (in memory) and on L4 TripleRush is more than three orders of magnitude faster than BitMat (in memory).

These results indicate that the performance of TripleRush is competitive with, or even superior to other state-of-the-art triple stores.

Fastest of 10 runs	L1	L2	L3	L4	L5	L6	L7	Geo. mean
TripleRush	**80.9**	**53.7**	78.5	**1.5**	**0.8**	**1.5**	**63.2**	**12.1**
Trinity.RDF	281	132	110	5	4	9	630	46
RDF-3X (in memory)	34179	88	485	7	5	18	1310	143
BitMat (in memory)	1224	4176	**49**	6381	6	51	2168	376
RDF-3X (cold cache)	35739	653	1196	735	367	340	2089	1271
BitMat (cold cache)	1584	4526	286	6924	57	194	2334	866
Average over 10 runs								
TripleRush	89.3	60.1	84.1	1.7	1.3	2.3	69.4	14.8

Table 1. LUBM-160 benchmark, time in milliseconds for query execution on ∼21 million triples. Comparison data for Trinity.RDF, RDF-3X and BitMat from [15].

Fastest of 10 runs	D1	D2	D3	D4	D5	D6	D7	D8	Geo. mean
TripleRush	**1.8**	**73.3**	**1.1**	**1.3**	**1.2**	**6.1**	**6.4**	**8.2**	**4.1**
Trinity.RDF	7	220	5	7	8	21	13	28	15
RDF-3X (in memory)	15	79	14	18	22	34	68	35	29
BitMat (in memory)	335	1375	209	113	431	619	617	593	425
RDF-3X (cold cache)	522	493	394	498	366	524	458	658	482
BitMat (cold cache)	392	1605	326	279	770	890	813	872	639
Average over 10 runs									
TripleRush	2.0	82.8	1.3	1.8	1.5	8.4	9.1	12.4	5.3

Table 2. DBPSB-10 benchmark, time in milliseconds for query execution on ∼14 million triples. Comparison data for Trinity.RDF, RDF-3X and BitMat from [15].

4.2 Data Scalability

Performance is a very important characteristic for a triple store, but it is also important that the query execution time scales reasonably when queries are executed over more triples.

We evaluate the data scalability of TripleRush by executing the LUBM queries L1-L7 with the same setup as in subsection 4.1, but ranging the dataset sizes from 10 to 320 universities and measuring the average time over 10 runs. The execution time for queries L1-L3 and L7 should increase with the size of the dataset, which is proportional to the number of universities. Queries L4-L6 are tied to a specific university and, given a good query plan, should take approximately the same time to execute, regardless of the dataset size. The left chart in Fig. 3 shows the execution times on a linear scale, while for queries L1-L3 and L7 both number of universities and the execution times are shown on a logarithmic scale. We see that queries L2 and L7 scale slightly sublinearly. Queries L1 and L3 scale almost linearly until LUBM-160, and then with a factor of more than three on the step to LUBM-320. We are uncertain about what causes this difference. As expected, the results in the left chart in Fig. 3 show that for queries L4-L6 the query execution time does not increase with the dataset size.

Overall, this evaluation suggests that TripleRush query execution times scale as expected with increased dataset sizes, but leaves an open question related to the scaling of queries L1 and L3 on LUBM-320.

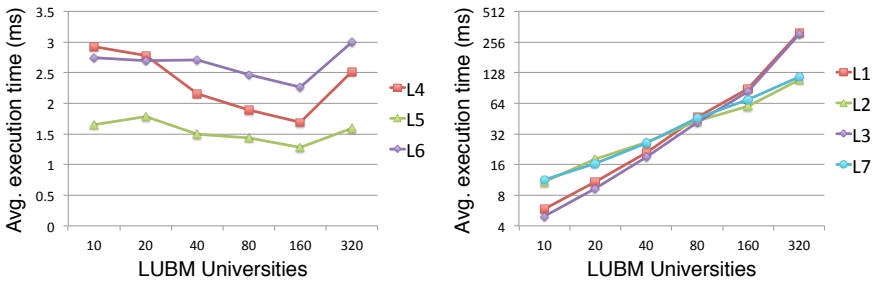


Fig. 3. Average execution times (10 runs) for queries L1-L7 on different LUBM sizes.

4.3 Memory Usage

Another important aspect of a triple store is the memory usage and how it increases with dataset size. In order to evaluate this aspect, we measured the memory usage of TripleRush after loading LUBM dataset sizes ranging from 10 to 320 universities and optimizing their index representations (smallest memory footprint of entire program from 10 runs). Figure 4 shows that the memory usage increases slighly sublinearly for this dataset. The memory footprint of TripleRush

is 5.8 GB when the 21 347 998 triples in the LUBM-160 datset are loaded. This is equivalent to ∼272 bytes per triple for this dataset size. TripleRush requires 3.8 GB for the DBPSB-10 dataset with 14 009 771 triples, which is equivalent to ∼271 bytes per triple. This is between a factor of 2 up to 3.6 larger than the index sizes measured for these datasets in Trinity.RDF [15], but far from the index size of 19 GB measured for DBPSB-10 in BitMat [15].

Currently, graph loading and index optimization for LUBM-160 takes as little as 106 seconds (without dictionary encoding, average over 10 runs). This is because the tree set data structure we use during graph loading supports fast insertions. The flip side is the high memory usage, which causes the graph-loading of the LUBM-320 dataset to take excessively long. Most of that time is spent on garbage collection, most likely because the entire assigned 31 GB heap is used up during loading. After loading is finished, the index representation optimizations reduce the memory usage to a bit more than 11 GB.

Overall, the index size of TripleRush is rather large, but that is in many cases a reasonable tradeoff, given the performance.

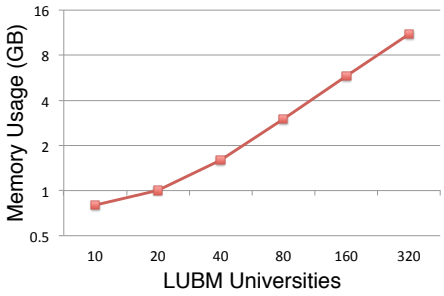


Fig. 4. Memory usage after loading LUBM datasets.

5 Limitations and Future Work

Our current investigation and design has a number of limitations that should be addressed in future work.

First, we need to evaluate TripleRush with additional benchmarks.

Second, and more importantly, we need to investigate the performance of TripleRush on larger graphs, in a distributed setting. Whilst we are optimistic that some of its optimizations will help even in the distributed setting, it is unclear what the overall performance impact of increased parallelism and increased messaging cost will be.

Third, TripleRush was not built with SPARQL feature-completeness in mind. Many SPARQL constructs such as filters and aggregates were not implemented.

Fourth, the current query optimization is very simple and could be improved.

Fifth, the root vertex is a communication bottleneck. Potential avenues for addressing this are to disallow a completely unbound query, which would retrieve the whole database, or to partition this vertex.

Sixth, the memory usage during graph loading should be reduced without overly slowing down the loading times.

Seventh, although the hardware we ran the benchmarks on had a lower performance score, it is desirable to do a comparison with Trinity.RDF on exactly the same hardware.

Even in the light of these limitations, TripleRush is a highly competitive system in terms of query execution performance. To our knowledge, it is the first triple store that decomposes both the storage and query execution into interconnected processing elements, thereby achieving a high degree of parallelism that contains the promise of allowing for transparent distributed scalability.

6 Conclusions

The need for efficient querying of large graphs lies at the heart of most Semantic Web applications. The last decade of research in this area has shown tremendous progress based on a database-inspired paradigm. Parallelizing these centralized architectures is a complex task. The advent of multi-core computers, however, calls for approaches that exploit parallelization.

In this paper we presented TripleRush, an in-memory triple store that inherently divides the query execution among a large number of active processing elements that work towards a solution in parallel. We showed that this approach is both fast and scalable.

Whilst TripleRush has its limitations, it is a step towards providing high-performance triple stores that inherently embrace parallelism.

Acknowledgments We would like to thank the Hasler Foundation for the generous support of the Signal/Collect Project under grant number 11072 and Alex Averbuch as well as Cosmin Basca for their feedback on our ideas.

References

1. D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 411–422. VLDB Endowment, 2007.
2. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 41–50, New York, NY, USA, 2010. ACM.
3. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

4. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.
5. J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.
6. S. Kotoulas, J. Urbani, P. A. Boncz, and P. Mika. Robust runtime optimization and skew-resistant execution of analytical sparql queries on pig. In P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7649 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2012.
7. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.
8. G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.
9. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. Dbpedia sparql benchmark: performance assessment with real queries on real data. In *Proceedings of the 10th international conference on The semantic web - Volume Part I*, ISWC'11, pages 454–469, Berlin, Heidelberg, 2011. Springer-Verlag.
10. T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 627–640, New York, NY, USA, 2009. ACM.
11. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal.The International Journal on Very Large Data Bases*, 19(1):91–113, 2010.
12. B. Shao, H. Wang, and Y. Li. The trinity graph engine. Technical report, Technical Report 161291, Microsoft Research, 2012.
13. P. Stutz, A. Bernstein, and W. W. Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In P. P.-S. et al., editor, *International Semantic Web Conference (ISWC) 2010*, volume LNCS 6496, pages pp. 764–780. Springer, Heidelberg, 2010.
14. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.
15. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. *Proceedings of the VLDB Endowment*, 6(4), 2013.

Appendix A: Evaluation Queries

LUBM evaluation queries, originally used in the BitMat evaluation [2] and selected by them from OpenRDF LUBM Queries.

```
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>

L1: SELECT ?X ?Y ?Z WHERE {
    ?Z ub:subOrganizationOf ?Y.
    ?Y rdf:type ub:University.
    ?Z rdf:type ub:Department.
    ?X ub:memberOf ?Z.
    ?X rdf:type ub:GraduateStudent.
    ?X ub:undergraduateDegreeFrom ?Y.
}
L2: SELECT ?X ?Y WHERE {
    ?X rdf:type ub:Course.
    ?X ub:name ?Y.
}
L3: SELECT ?X ?Y ?Z WHERE {
    ?X rdf:type ub:UndergraduateStudent.
    ?Y rdf:type ub:University.
    ?Z rdf:type ub:Department.
    ?X ub:memberOf ?Z.
    ?Z ub:subOrganizationOf ?Y.
    ?X ub:undergraduateDegreeFrom ?Y.
}
L4: SELECT ?X ?Y1 ?Y2 ?Y3 WHERE {
    ?X ub:worksFor
    <http://www.Department0.University0.edu>.
    ?X rdf:type ub:FullProfessor.
    ?X ub:name ?Y1.
    ?X ub:emailAddress ?Y2.
    ?X ub:telephone ?Y3.
}
L5: SELECT ?X WHERE {
    ?X ub:subOrganizationOf
     <http://www.Department0.University0.edu>.
    ?X rdf:type ub:ResearchGroup.
}
L6: SELECT ?X ?Y WHERE {
    ?Y ub:subOrganizationOf
       <http://www.University0.edu>.
    ?Y rdf:type ub:Department.
    ?X ub:worksFor ?Y.
    ?X rdf:type ub:FullProfessor.
}
L7: SELECT ?X ?Y ?Z WHERE {
    ?Y ub:teacherOf ?Z.
    ?Y rdf:type ub:FullProfessor.
    ?Z rdf:type ub:Course.
    ?X ub:advisor ?Y.
    ?X rdf:type ub:UndergraduateStudent.
    ?X ub:takesCourse ?Z.
}
```

DBPSB evaluation queries, received courtesy of Kai Zeng.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX dbpres: <http://dbpedia.org/resource/>
PREFIX rdfcore: <http://www.w3.org/2004/02/skos/core#>

D1: SELECT ?X WHERE {
    ?Y rdfcore:subject dbpres:Category:First-person_shooters.
    ?Y foaf:name ?X.
}
D2: SELECT ?X WHERE {
    ?Z foaf:homepage ?Y.
    ?Z rdf:type ?X.
}
D3: SELECT ?X ?Y ?Z WHERE {
    ?Z rdfcore:subject dbpres:Category:German_musicians.
    ?Z foaf:name ?X.
    ?Z rdfs:comment ?Y.
}
D4: SELECT ?W ?X ?Y ?Z WHERE {
    ?Z dbo:birthPlace dbpres:Berlin.
    ?Z dbo:birthDate ?X.
    ?Z foaf:name ?W.
    ?Z dbo:deathDate ?Y.
}
D5: SELECT ?X ?Y ?Z WHERE {
    ?Z rdfcore:subject dbpres:Category:Luxury_vehicles.
    ?Z foaf:name ?Y.
    ?Z dbo:manufacturer ?W.
    ?W foaf:name ?X.
}
D6: SELECT ?Z1 ?Z2 ?Z3 ?Z4 WHERE {
    ?X rdf:type ?Y.
    ?X dbpprop:name ?Z1.
    ?X dbpprop:pages ?Z2.
    ?X dbpprop:isbn ?Z3.
    ?X dbpprop:author ?Z4.
}
D7: SELECT ?Y WHERE {
    ?X rdf:type ?Y.
    ?X dbpprop:name ?Z1.
    ?X dbpprop:pages ?Z2.
    ?X dbpprop:isbn ?Z3.
    ?X dbpprop:author ?Z4.
}
D8: SELECT ?Y WHERE {
    ?X foaf:page ?Y.
    ?X rdf:type dbo:SoccerPlayer.
    ?X dbpprop:position ?Z1.
    ?X dbpprop:clubs ?Z2.
    ?Z2 dbo:capacity ?Z3.
    ?X dbo:birthPlace ?Z4.
    ?Z4 dbpprop:population ?Z5.
    ?X dbo:number ?Z6.
}
```

Eviction Strategies for Semantic Flow Processing*

Minh Khoa Nguyen,[1] Thomas Scharrenbach,[1] Abraham Bernstein[1]

University of Zurich, Switzerland
{nguyen,scharrenbach,bernstein}@ifi.uzh.ch

Abstract. In order to cope with the ever-increasing data volume continuous processing of incoming data via Semantic Flow Processing systems have been proposed. These systems allow to answer queries on streams of RDF triples. To achieve this goal they match (triple) patterns against the incoming stream and generate/update variable bindings. Yet, given the continuous nature of the stream the number of bindings can explode and exceed memory; in particular when computing aggregates. To make the information processing practical Semantic Flow Processing systems, therefore, typically limit the considered data to a (moving) window. Whilst this technique is simple it may not be able to find patterns spread further than the window or may still cause memory overruns when data is highly bursty.
In this paper we propose to maintain bindings (and thus memory) not on recency (i.e., a window) but on the likelihood of contributing to a complete match. We propose to base the decision on the matching likelihood and not creation time (fifo) or at random. Furthermore we propose to drop variable bindings instead of data as do load shedding approaches. Specifically, we systematically investigate deterministic and the matching-likelihood based probabilistic eviction strategy for dropping variable bindings in terms of recall. We find that a matching likelihood based eviction can outperform fifo and random eviction strategies on synthetic as well as real world data.

Keywords: Semantic Flow Processing, Linked Data stream processing, eviction strategies, load shedding, matching likelihood estimation on streams

1 Introduction

Information processing increasingly incorporates query matching on dynamically changing information (i.e. information flows) [5]. Based on these Information Flow Processing (IFP) systems, such as C-SPARQL [4], EP-SPARQL [1], or CQELS [10] emerged that are capable of processing flows of semantically annotated data. All of these Semantic Flow Processing (SFP) systems allow defining

* The research leading to these results has received funding from the European Union Seventh Framework Programme FP7/2007-2011 under grant agreement no 296126.

queries in a language that extends SPARQL with algebra adjustments for matching queries on flows of time-annotated RDF triples.

Each SFP system works on limited memory resources. They constrain the number of valid partial variable bindings (i.e., bindings that are not yet complete solutions to a query) by applying a window on the stream. Such a window limits the scope of query matching to, for example, a period of time or to a certain number of triples. On the one hand this limits the scope of answers the system can find. If a match requires two data items that do not fall in the same scope the system will never be able to perform such a match. On the other hand, even the limited scope can not necessarily guarantee that the system will never exceed its memory resources. Consider the situation when a popular TV broadcast/show, such as a football match, starts and many users switch to it at the same time. In this case we may find that a sudden burst in the input data could cause the system to exceed its resources. The situation gets worse when we are not only interested in the number of viewers for a show but want to track the users – the literature refer to the latter case as non-shrinking semantics [3].

To our knowledge *no current SFP approach has a solution for freeing memory in case these exceed the available resources*. While proposals have been made for load-shedding—dropping or ignoring incoming data—in IFP systems, only one of them investigated load-shedding according to a statistical model of the matching history [6]. We believe that instead of ignoring incoming data one should one should drop variable bindings based on their likelihood that they lead to a result. We refer to this as *eviction strategies* in contrast to load shedding.

Eviction strategies are more flexible than load shedding. If we drop a data item we may drop information too early in the matching process. Eviction allows us to define constraints at which point of the matching process we want to delete items. While there exist deterministic approaches that allow for dropping variable bindings, such as *first-in-first-out (fifo)* we propose to use the matching likelihood as the dropping criterion. In this paper we, hence, investigate the performance of different eviction strategies. Note that an eviction strategy is not the same as a garbage collection. While the latter removes those variables bindings that will not contribute to a solution, the former removes variable bindings that have the potential but cannot be kept due to memory limitation. As a consequence applying an eviction strategy could lead to incomplete results for the sake of feasibility.

In this paper *we propose to use a probabilistic based eviction strategy that estimates the likelihood of a binding to contribute to a result in contrast to using a fixed time window for load shedding*. Note that this paper intends to show that a probability based model can outperform fifo and random load shedding but does not provide a model of how to estimate those probabilities on the go. We investigate the effect our approach has on the completeness of results compared with deterministic approaches. We systematically investigated the challenges arising from the limited memory resources of regular joins for SFP systems. If the partial bindings for the potential join partners exceed the memory we start dropping partial results. Our findings enable an SFP to trade off completeness

versus resources. To ensure methodological correctness we followed the PCKS approach [11] for defining stress tests for SFP systems.

As the relevant properties of an SFP system we identify completeness as a Quality of Service constraint. The corresponding challenge we want to tackle is the ability to handle bursts in the data. We therefore define recall as the key performance indicator for the evaluations of the tests we performed in the scope of this paper. In order to test the system we check how the system behaves under restricted resources and under unrestricted resources.

In our experiments we compared our probabilistic methods based on the matching likelihood of partial bindings with the first-in-first-out (fifo) deterministic method as well as random load shedding. We show that our statistical approach can outperform fifo and random load shedding both on simulated data as well as on a real-world dataset of viewership behavior of IPTV users. The improvement ranges from 14% to 50% when using synthetic and real world data.

This remainder of this paper is structured as follows: next we formally specify the context of SFP and then introduce our load-shedding approach. We then evaluate our approach on synthetic and real world data in Section 3. After a discussion of the results and limitations we present related work in Section 5 and conclude in Section 6.

2 Method: Likelihood-based Eviction of Partial Matches

This section first lays the formal groundwork for defining what eviction strategies are. Then, it introduces our likelihood-based eviction strategy as well as the baseline first-in-first-out (fifo) and random strategies.

2.1 Semantic Flow Processing

SFP rely on an algebra that modifies the SPARQL algebra for query matching on flows of data. The result of an algebra operation is a set of variable bindings consisting of a finite number of pairs $?var/val$, where $?var$ is a variable name and val is an RDF term. A query is a tree of algebra expressions where results (i.e., bindings) are propagated from the leaves to the root. An SFP system, hence, consumes time-stamped RDF triples $< s, p, o > [time]$ that are then consumed (or evaluated) by the algebra operations defined in the query tree. Hereby the system updates the bindings for all affected levels of the query tree and bindings at the root level will be emitted as results.

In our discussions we rely upon the definitions of SPARQL and RDF as given in the SPARQL-query specification [8]. Whilst we limit our discussions to joins and aggregates our approach is by no means limited to these operations. A *semantic flow* $\mathcal{F}$ is a finite set of time-stamped RDF triples. We denote algebra expressions by α. The evaluation of an algebra operation α and a finite set of variable bindings Ω on a semantic flow again produces a (possibly new/updated)

set of variable bindings $eval(\alpha, \Omega, \mathcal{F}) = \Omega'$. In a query tree all algebra expressions consume variable (a semantic flow) bindings, perform the corresponding operations, and then emit variable bindings.[1]

SPARQL query processing systems usually implement an iterator-based approach: since they know all relevant data beforehand, they can successively perform the algebra operations recursively and, hence, know a binding was completely processed by an algebra expression. A SFP system, in contrast, has to keep its variable bindings to guarantee completeness. Consuming data in a streaming fashion turns the execution model upside-down. Instead of recursively processing all available data for each algebra expression we have to continuously evaluate newly arriving data for all algebra expressions. Consequently, the algebra expressions in an SFP have never finished evaluating all available data and have to continuously maintain a list of corresponding variable bindings. Consider the simple join of two graph patterns `?a ab ?b` and `?b bc ?c`, for example. Assuming that the SFP system matched the first pattern with the binding variable binding $\mu = \{?a/a_1, ?b/b_1\}$ it is not really possible to say if and when a match for the second graph pattern might happen.

Keeping all variable bindings, however, will eventually exceed the system's memory resources and make processing unfeasible. Therefore, SFP systems typically limit the number of data they process by windowing, which limits the data considered for computing partial matches to a given period of time or number of triples. This allows to mark bindings with an an expiration time-stamp. A garbage collection process can then remove those variable bindings that become out-of-scope.

Unfortunately, windowing has its distinct problems. For example, queries that require to match patterns that unfold over periods longer than a practical window for example cannot be matched. In those cases there are two options: First one can apply load shedding – an approach that drops data from the input stream, for example by only considering every x-th data item for processing. Load shedding is a well-established approach and we discuss its difference to eviction in Section 5 along with related work.

Second, we may delete variable bindings instead of input data. In this paper paper we investigate this alternative approach we refer to as *eviction*. It leaves the input stream untouched but intermediate variable bindings get deleted to make place for new ones. In the following we discuss this approach in detail.

2.2 Eviction: Dropping Variable Bindings

In order to limit memory usage of a SFP system we propose to use eviction, which deletes partial binding from the processing tree. When deleting partial matches (or shedding load in the input stream) we potentially sacrifice completeness. As a consequence, it is imperative to understand what consequence eviction has on completeness as a Quality of Service (QoS) criteria. In this study we, therefore, compare the performance of the following *eviction functions*:

[1] Note that the leaves of the tree typically consume $\mathcal{F}$ and that the root may emit RDF data to a new flow $\mathcal{F}'$.

– random deletion,
– time-based deletion (e.g., delete the oldest bindings first), and
– data-driven deletion of bindings.

While the first is self-explanatory, the latter two require further explanation.

Time-Based Deletion Time-based deletion bases on the assumption that the probability that a binding contributes to a result decreases with the amount of time it stays in the bindings cache. Whilst we know of no empirical evidence that this assumption holds for typical SFP benchmarks it is used in systems such as SASE+ [7], where it is referred to as least-recently-used (LRU). [2]

LRU can be implemented as a priority queue, which can be implemented via a heap. Heap organization has a complexity of $\mathcal{O}(log\ n)$, where n is the number of elements in the bindings cache.

Note that the actual performance of LRU can depend on the time model for the bindings. A time model is either based on system time or on data time or on both. Furthermore, an LRU eviction function must specify which time-stamp of a binding it uses. This time-stamp can be the creation time of the binding or the time it was last updated. In the scope of this work we use data time and the time-stamp of the last update of the binding.

Data-Driven Eviction Data-driven eviction—the core contribution of this paper—bases the eviction decision of a binding μ on the probability that μ contributes to a result. This requires an estimation of the matching probability for μ. To that end we introduce the notion of the transactional closure for a binding μ in Section 2.3, which permits counting the number of results it contributed to for a certain part of the stream data.

Probability estimation can be performed either online or offline. In the online case the contribution probability has to be estimated along with the actual query processing, which is a prerequisite in cases where the data cannot be stored for offline analysis. In this paper we assume that we can compute the probabilities offline and describe the method for estimating the probabilities in Section 2.4.

When applying data-driven eviction on each operator's binding locally it requires the same computational effort as LRU.[3] Akin to the LRU case, data-driven eviction can be implemented using a priority queue. The only difference is the comparison attribute which is now the matching probability rather than the time-stamp. Global data-driven eviction strategies may require additional effort (cf Section 2.5).

[2] Note that in our case LRU gives the same result as fifo since we only forward fresh bindings to next operator instead of performing updates on the current binding.

[3] Assuming that the probabilities, as in our case, are computed offline and do not influence the online processing effort.

2.3 The Transactional Closure

In order to evaluate the performance of any eviction or load shedding operator we have to know whether deleting a data item or a variable binding impacts the results. A query result emerges from applying a set of algebra operators. In order to determine whether deletion of data items or variable bindings has an impact on the results of a query we hence need to know *all* sequences in which the application of a set of data items to the algebra operators of a query leads to a result.

We now define the basics for computing all possible paths to all query results for a flow $\mathcal{F}$. Any result of a query is a variable binding μ_{root} the root of the tree of the query's algebra expressions $\mathcal{T}_{\alpha_{\nabla\wr\wr\sqcup}}$. As such any of these root variable bindings emerges from a sequence of application of algebra expressions from $\mathcal{T}_{\alpha_{\nabla\wr\wr\sqcup}}$. As a result, for each μ_{root} we have a finite set of sets of variable bindings. We refer to this set as the *transactional closure* $(\mu)^+_{\alpha,\mathcal{F}}$ of the binding μ. Indeed, we can define such a transactional closure recursively for all bindings for all algebra expressions in $\mathcal{T}_{\alpha_{\nabla\wr\wr\sqcup}}$.

Having the definition of the transactional closure at hand we now investigate what happens, if we delete some of the variable bindings from the transactional closure. If we delete all entries from $(\mu)^+_{\alpha,\mathcal{F}}$, then the variable binding μ will never be created. In this study we want to know what impact any variable binding $\mu' \in (\mu)^+_{\alpha,\mathcal{F}}$ has on the transactional closure of μ.

We assume that each algebra expression α maintains a *bindings cache*: the finite collection of bindings, which we denote with Ω_α. If we now delete a variable binding μ' from Ω_α we would like to know which potential impact this deletion has on the query results. We hence define the transactional closure with respect to α, Ω, and $\mathcal{F}$ by $(\mu)^+_{\alpha,\Omega,\mathcal{F}}$. It inductively defines those variable bindings that emerge from μ by applying α to $\langle \mathcal{F}, \Omega \rangle$. Let $R_{\alpha,\Omega,\mathcal{F}}$ be the result set of α for Ω and $\mathcal{F}$, then we call $R^\mu_{\alpha,\Omega,\mathcal{F}} = R_{\alpha,\Omega,\mathcal{F}} \cap (\mu)^+_{\alpha,\Omega,\mathcal{F}}$ the result set of μ.

The bindings cache of an algebra expression contains different sets of variable bindings. A join with n join partners, for example, has one set $\Omega_{\alpha,n}$ for each of the n join partners. In this study we assume that each of these sets can hold a limited number of variable bindings, i.e., $|\Omega_{\alpha,n}| \leq \omega_{\alpha,n}$, where $\omega_{\alpha,n}$ is the memory available to store bindings. When the application of α on the flow $\mathcal{F}$ causes some of these sets to exceed its limit ($\omega_{\alpha,n}$), then we apply an eviction strategy $\mathbf{E}$ that removes variable bindings from $\Omega_{\alpha,n}$ such that the above condition will be fulfilled again.

2.4 Estimating Matching Probabilities

We estimate the selectivity of a variable binding μ by the number of results that depend on μ. We define the probability that the result set of a variable binding μ is non-empty as the above count normalized by the total number of results with respect to the bindings cache:

$$Pr(R_{\alpha,\Omega,\mathcal{F}} \neq \emptyset) = \frac{\left|R^{\mu}_{\alpha,\Omega,\mathcal{F}}\right|}{|R_{\alpha,\Omega,\mathcal{F}}|}$$

Note that the choice of α designates for which biding cache $\omega_{\alpha,n}$ this probability is computed.

The above formula, hence, computes the probability that there exists a binding μ' that bases on μ. If α is a query, then the above formula defines the probability that μ contributes to a query's root result.

In some cases it is possible that a binding μ' originates from two different upstream bindings. Consider, for example, the following operation joining two identical basic graph patterns $BGP(\texttt{?x P Y})$. Here, any binding $\mu = \{\texttt{?x/X}\}$ could originate from the first or the second triple pattern matching:

$$JOIN(BGP(\texttt{?x P Y}), BGP(\texttt{?x P Y}))$$

For the computation of the probabilities we count both original variable bindings.

2.5 Eviction Strategies: Local vs Global Approaches

A local eviction strategy operates locally on the binding cache of each operator. A global eviction strategy, in contrast, operates on all binding caches together, attempting to optimize the overall global performance. In other words, for a local eviction strategy all $\omega_{\alpha,n}$ are fixed whereas a global eviction strategy has the additional constraints that $\sum_{\alpha}\sum_{n}\omega_{\alpha,n} \leq \omega$ for all $\omega_{\alpha,n}$. A global eviction strategy hence requires to also solve (or approximate) a constraint optimization problem. Since it is our goal to show that data-driven eviction can lead to superior results over the other strategies we only have to show that one of the two approaches offers advantages over the other, non data-driven ones. We, hence, restrict ourselves to the simpler, local eviction strategy shown in Algorithm 1 and leave the global eviction strategy for future work.

Algorithm 1 Local eviction strategy for an algebra expression $\alpha = \alpha_1, \ldots, \alpha_I$, a flow $\mathcal{F}$, a set of sets of variable bindings Ω_{α_i,n_i}, limits on these ω_{α_i,n_i}

```
Apply F to ⟨α, Ω⟩
for all Ω_{α_i,n} do
  if |Ω_{α_i,n}| > ω_{α_i,n} then
    eviction(Ω_{α_i,n})
  end if
end for
```

3 Evaluation

The main working hypothesis of our paper is that eviction based on the matching likelihood can outperform the fifo and random dropping approach of variable bindings. We therefore have to estimate the matching likelihood. This in turns required implementing the transactional closure in order to be able to compute the counts on which the matching likelihood is defined. In order to calculate the matching likelihood, we implemented an "oracle". This oracle allows to retrieve the cardinality of complete matches in the future. The oracle assumes that required bindings are not been evicted by other join operators. With this oracle we can compute the probabilities as proposed in Section 2.4. We will test probabilistic eviction with probabilities learned from past data in future studies. Please find the discussion of this limitation in Section 4.

In this section we evaluate our approach governed by the above hypothesis. We therefore compared the performance of our likelihood eviction with the *fifo* and *random* strategies with respect to recall as the key performance indicator (KPI).

3.1 Experimental Setup

We performed our experiments on a synthetic and a real world data set. We used the same query that performs the following join:

```
SELECT ?a ?b ?c ?d
WHERE {
    ?a  ab  ?b .
    ?b  bc  ?c .
    ?c  cd  ?d .
}
```

We considered joins without sequential or temporal constraints, because of memory constraints. Our goal was to compute the recall for all potential bindings. Computing the transactional closure for joins requires an exponential number of traces which matching originated from which variable bindings. We discuss this limitation in Section 4.

We performed the experiments considering local eviction. We tested the three different eviction strategies with bindings caches which we increased exponentially. We chose sizes as multiples of 10. The reason for this is that calculating matching statistics is currently a time consuming process and therefore we choose an exponentially increasing size of cache in the evaluation.

For the comparison we assumed the same time for determining whether an element was subject to eviction or not. This holds true for *fifo* and statistical eviction, as they both keep a cache where the elements to be evicted have to be sorted out. We can achieve this by applying Heap-Sort. We assume that the number of elements to be dropped by eviction to be constant in relation to the

size of the bindings cache. This way each eviction step is bounded by $\mathcal{O}(\log(n))$ where n is the size of the bindings cache.

All experiments were conducted using the UZH Katts Semantic Flow Processing engine.[4] While we simulated an uniform distribution of results for the synthetic dataset we used anonymous IPTV viewership logs and joined them with Electronic Program Guide data. The data was gathered in the scope of the ViSTA-TV project.[5] For our experiments we used a data sample from Aug 1st, 2012. The sample comprised about 300'000 events in total for the user log and the EPG data. We thereby only considered such properties that also occur in the query. For the above mentioned exponential explosion for tracing the matching we limited ourselves to a subset comprising 10'000 events.

To retrieve the exact matching probabilities for the bindings we stored those statistics in a sqlite3 in memory database. However, updating the matching probability for each binding every time unit (every time unit can change the likelihood of a match for a binding) is a very time expensive process. Therefore we updated the probability for each binding only once, namely when we added it to the bindings cache.

The following configuration was used to run all our experiments: 2 Quad-core Intel(R) Xeon(R) CPU X5570@ 2.93GHz, 72GB of RAM, 4 TB of disk space, running Fedora Core 12 kernel 2.6.32.14 64bit.

3.2 Results

As Figures 1-2 show, our likelihood based eviction approach always outperforms *fifo* and *random* eviction. The difference for the synthetic data set decreases with increasing size of the bindings cache from 50% to 10% difference. In the real world data set the differences in recall increases from 14% to 29% for the 10'000 events data set and for the 1'000 events data set we can observe a decrease from 32% to 14%.

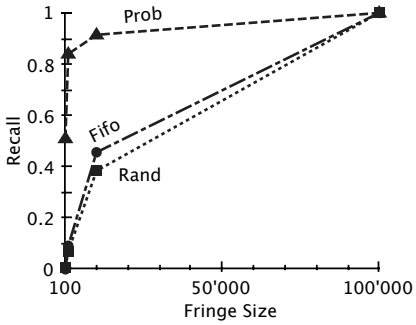
We see that for the extreme size of the binding cache, we always obtain 100% recall for all approaches. We evaluate the overall performance of our approach by the recall measured for the bindings cache size that can hold up to 10% of the bindings of the cache for which we get 100% recall. While the matching likelihood based eviction performs very good on the synthetic data set (up to 91% recall with bindings cache of 10% relative of 100% performance limit), it performs not as good for the real-world data set (up to 60% recall with bindings cache of 10% relative of 100% performance limit).

4 Discussion

Experimental Results The results of our study indicates that likelihood based eviction can outperform *fifo* and *random*. This was the case for all experiments we

[4] KATTS is a recursive acronym for Katts is A Triple Torrent Sieve and is accessible via GitHub at https://github.com/lorenzfischer/katts

[5] http://vista-tv.eu



(a)

	100	1'000	10'000	100'000
fifo	0 (0%)	616 (9%)	3065 (46%)	6718 (100%)
random	41 (1%)	468 (7%)	2578 (38%)	6718 (100%)
prob	3425 (51%)	5642 (84%)	6144 (91%)	6718 (100%)
ground	6718 (100%)	6718 (100%)	6718 (100%)	6718 (100%)

(b)

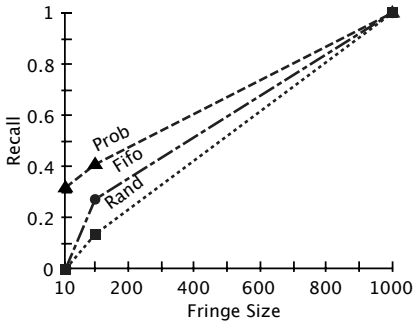
Fig. 1: Recall Generated Dataset with 145'000 events.

conducted. The relative difference between is larger for the synthetic data than for the real world data set. We think that our approach was able to better predict the matching likelihood for synthetic dataset as it was generated following some uniform distributions. Nevertheless an eviction strategy based on the matching likelihood seems to be a promising approach.

We also found that there is a relation between the size of the bindings caches and the recall. The larger the bindings cache the higher the recall. While this result seems not very surprising it yet indicates that future research will have to be performed finding out the exact size of the bindings cache such that the recall becomes 100%. Such research would also investigate at which cost an improvement of the recall using eviction strategies comes.

Shortcomings The most important shortcomings of our study are the limited number of data items we could process and the restriction to a query consisting of a regular join. The limitation to the small number of data items is caused by the exponential number of possible traces from variable bindings to results. Research on more complicated queries and larger data sets has hence to be performed online. This, in turn makes an exact calculation of the matching statistics infeasible. Therefore one has to learn the matching statistic which we tackle in our future work.

In this study we hence concentrated on investigating how well likelihood based eviction works in —*principle*, compared to *fifo* and *random* eviction. We are currently investigating how to best estimate the matching likelihood on the go, i.e., along with the processing of incoming data. Future experiments will



(a)

	10	100	1'000
fifo	0 (0%)	3 (14%)	22 (100%)
random	0 (0%)	6 (27%)	22 (100%)
prob	7 (32%)	9 (41%)	22 (100%)
ground	22 (100%)	22 (100%)	22 (100%)

(b)

Fig. 2: Recall ViSTA-TV Dataset with 1'000 events.

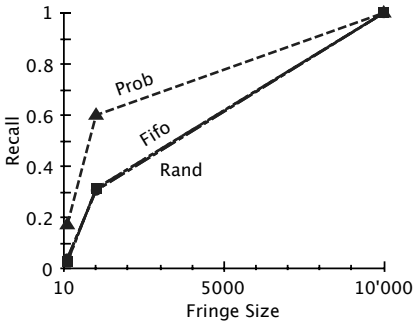
hence systematically investigate different types of algebra expressions. These include aggregates but also joins with sequential and temporal constraints, as these are more typical for SFP systems. We believe that likelihood based eviction can help most for such cases where a lot of variable bindings are produced, e.g., in the case of aggregates, in particular for non-shrinking semantics.

Our simulation did not investigate response time or throughput but recall. While we showed that matching likelihood based eviction outperforms deterministic approaches like *fifo* and *random* eviction in terms of recall, we will investigate at which costs this comes for the former two KPIs. As in the previous case we will have to then estimate the statistics online rather than offline. Following the PCKS paradigm [11] are working on implementing tests for determining how well our approach works for bursts in the data.

In this study we considered a purely local eviction strategy. It would be interesting to see the effect of global constraints. We are currently investigating how to add these to statistical eviction. This requires a well-defined optimization strategy carefully balancing the relation between local and global constraints.

5 Related Work

Research related to this paper roughly covers three areas: Semantic Flow Processing, selectivity estimates on flow-data, and load shedding for IFP/SFP systems. In the sequel we discuss how our work relates to each of these.



(a)

	100	1'000	10'000
fifo	39 (4%)	290 (31%)	943 (100%)
random	27 (3%)	297 (31%)	943 (100%)
prob	164 (17%)	567 (60%)	943 (100%)
ground	943 (100%)	943 (100%)	943 (100%)

(b)

Fig. 3: Recall ViSTA-TV Dataset with 10'000 events.

5.1 Semantic Flow Processing

C-SPARQL [4] performs query matching on subsets of the information flow defined by windows. The decidability of SPARQL query processing on such finite sets of RDF triples causes the number of variable bindings produces to be finite. Their number may still become prohibitively large, for example, when using non-shrinking semantics for aggregates [3]. Since the execution is performed on traditional SPARQL engines, we may apply our approach, for example as a filter for iterator-based implementations such as Jena.[6]

EP-SPARQL [1] is a complex event processing system which extends the ETALIS system by a flow-ready extension of SPARQL for query processing [1]. It provides a garbage collection facility which can "prune outdated events" or "expired events by using periodic events, generated by the system". ETALIS is a prolog engine which seriously hampers the applicability of our approach to ETALIS engine. Note however, that EP-SPARQL may be implemented on other engines, e.g., CQUELS, too.

CQELS [10] "implements the required query operators natively to avoid the overhead and limitations of closed system regimes". It optimizes the execution by dynamically re-ordering operators, because "the earlier we prune the triples that will not make it to the final output, the better, since operators will then process fewer triples". This pruning does, however not make any guarantees

[6] https://jena.apache.org/

about the number of variable bindings created by the processors. Our method should be directly applicable to CQUELS as it provides a native implementation of the operators and these operators maintain a list of active variable bindings.

5.2 Selectivity Estimates

Selectivity estimates for optimizing the execution of SPARQL queries were first investigated by [12]. These are not directly applicable to our approach. Selectivity estimates for SPARQL query optimization base on the selectivity of predicates whereas we are interested in the matching likelihood of variable bindings. More related to our work is the operator scheduling of CQELS [10]. They propose to choose the next operator to execute according to the likelihood that the operator leads to a positive result. They, however, base the decision on heuristics. It would be very interesting to see what effect applying our matching likelihood would have on the performance of CQELS.

To the best of our knowledge, no approach estimates the matching likelihood of bindings to determine their selectivity.

5.3 Load Shedding

Load shedding has been applied to information flow processing. Approaches like [6, 2, 13] perform load shedding by dropping tuples from the stream, i.e., dropping data instead of variable bindings. In contrast to a Complex Event processing (CEP) system they assume a Data Stream Management System, i.e., a set of triples with a relational database execution engine.

An approach we found that follows a similar idea is SASE+ [7]. SASE is has an automata based matching approach which can be considered similar to variable bindings in our case. They do apply some eviction strategy. Yet, they base their eviction on a deterministic approach that is similar to our implementation of *fifo*.

Another approach that resembles our work is proposed in [6]. They perform a simple equi-join on two incoming streams and evict tuples which are less likely to find a join partner. However, the approach in [6] works with a sliding window and with a single equi-join of two streams.

Note that applying a window to the flow of RDF triples can be considered dropping tuples, too. As such all approaches that implement a window on the incoming data effectively offer the window as a load shedding strategy.

6 Conclusion and Outlook

In this paper we proposed to perform load shedding by eviction, i.e., by dropping variable bindings rather than by dropping tuples. While the latter is a well-established technique for Data Stream Management Systems, for CEP our approach is the first that applies (i) load shedding by eviction and (ii) bases

the decision on the matching likelihood of a variable binding rather than on a heuristic.

Our experiments show that eviction is a promising strategy for regular joins for event-driven Semantic Flow Processing systems. We outperform the usually used deterministic approach first-in-first-out by up to 51% recall.

While our approach is currently investigating recall as the only key-performance indicator (KPI) we are confident that we will be able to show that the matching likelihood also performs better than deterministic approaches when different Quality of Service (QoS) constraints require testing a combination of recall with other KPIs such as response time.

Future work will concentrate on extending the algebra expressions to joins with sequential and temporal constraints. We believe that the solution for load shedding will have to incorporate statistical information from the data flow and the query log. We will also investigate how the matching likelihood can be learned from the stream on-the-go.

In the future we will also plan to evaluate our approach with standardized queries and data as proposed in [14] or [9].

We are convinced that only systematic investigations on the relation between using windows and likelihood-based eviction for load shedding will reveal this solution from the data.

References

1. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: Proceedings of the 20th International Conference on World Wide Web. pp. 635–644. ACM (2011)
2. Babcock, B., Datar, M., Motwani, R.: Load Shedding for Aggregation Queries over Data Streams. In: Proceedings of the 20th International Conference on Data Engineering. pp. 350—-. ICDE '04, IEEE Computer Society (2004)
3. Barbieri, D., Braga, D., Ceri, S.: Incremental reasoning on streams and rich background knowledge. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) The Semantic Web: Research and Applications: 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 2, 2010. LNCS, vol. 6088, pp. 1–15 (2010)
4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: A Continuous Query Language for RDF Data Streams. International Journal of Semantic Computing 4(1), 3–25 (2010)
5. Cugola, G., Margara, A.: Processing flows of information. ACM Computing Surveys 44(3), 1–62 (Jun 2012)
6. Das, A., Gehrke, J., Riedewald, M.: Approximate join processing over data streams. In: Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03. p. 40. ACM Press (2003)
7. Diao, Y., Immerman, N., Gyllstrom, D.: Sase+: An agile language for kleene closure over event streams. Tech. rep., University of Massachusetts Amherst, Department of Computer Science (2008)
8. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C (2011)
9. Le-Phuoc, D., Dao-Tran, M., Pham, M.: Linked stream data processing engines: facts and figures. The Semantic Web– ... 1380(24761), 1–12 (2012)

10. Le-phuoc, D., Dao-tran, M., Parreira, J.X., Hauswirth, M.: A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) The Semantic Web – ISWC 2011: 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I. Lecture Notes in Computer Science, vol. 7031, pp. 370–388. Springer Berlin / Heidelberg (2011)
11. Scharrenbach, T., Urbani, J., Margara, A., Valle, E.D., Bernstein, A.: Seven Commandments for Benchmarking Semantic Flow Processing Systems. In: Proc.ESWC 2013 (to appear). pp. 1–15. No. 296126 (2013)
12. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In: Proceedings of the 17th International World Wide Web Conference (WWW). ACM (Apr 2008)
13. Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load Shedding in a Data Stream Manager. In: 29th International Conference VLDB. vol. 54, pp. 309–320. VLDB Endowment (2003)
14. Zhang, Y., Duc, P.M., Corcho, O., Calbimonte, J.p.: SRBench : A Streaming RDF / SPARQL Benchmark. In: The Semantic Web - ISWC 2012: 11th International Semantic Web Conference, ISWC 2012, Boston, MA, USA, Nov 12-14, 2012, Proceedings (2012)

Scalable Linked Data Stream Processing via Network-Aware Workload Scheduling

Lorenz Fischer, Thomas Scharrenbach, Abraham Bernstein

University of Zurich, Switzerland*
{lfischer,scharrenbach,bernstein}@ifi.uzh.ch

Abstract. In order to cope with the ever-increasing data volume, distributed stream processing systems have been proposed. To ensure scalability most distributed systems partition the data and distribute the workload among multiple machines. This approach does, however, raise the question how the data and the workload should be partitioned and distributed. A uniform scheduling strategy—a uniform distribution of computation load among available machines—typically used by stream processing systems, disregards network-load as one of the major bottlenecks for throughput resulting in an immense load in terms of inter-machine communication.
In this paper we propose a graph-partitioning based approach for workload scheduling within stream processing systems. We implemented a distributed triple-stream processing engine on top of the Storm realtime computation framework and evaluate its communication behavior using two real-world datasets. We show that the application of graph partitioning algorithms can decrease inter-machine communication substantially (by 40% to 99%) whilst maintaining an even workload distribution, even using very limited data statistics. We also find that processing RDF data as single triples at a time rather than graph fragments (containing multiple triples), may decrease throughput indicating the usefulness of semantics.

Keywords: semantic flow processing, stream processing, linked data, complex event processing, graph partitioning, workload scheduling

1 Introduction

In today's connected world, data is produced in ever-increasing volume, velocity, variety, and veracity [20]: sensor data is gathered, transactions are made in the financial domain, people post/tweet messages, humans and machine infer new information, etc. This phenomenon can also be found on the Web of Data (WoD), where new sources are made available as linked data. In order to process these

* The research leading to these results has received funding from the Europ. Union 7th Framework Programme FP7/2007-2011 under grant agreement no 296126 and from the Dept. of the Navy under Grant NICOP N62909-11-1-7065 issued by Office of Naval Research Global.

growing data sources many have proposed the use of distributed infrastructures such as Hadoop [20]. The batch-oriented synchronous nature of these solutions, however, may not be suited to ensure the timeliness of data processing. To address this shortcoming stream processing approaches based on information-flow processing have been proposed [8]. These systems continuously ingest new data as it arrives and process it online rather than storing it for batch-like processing. This continuous processing puts a significant load on employed systems and is, obviously, limited by the capacity of the employed hardware infrastructure.

To cope with increasing demands distributed stream processing systems have been proposed, which usually ensure scalability by partitioning the data and distributing the processing thereof to multiple machines. Note that deciding how to partition the data and distribute the associated processing is a non-trivial task and can have dire consequences on performance.

Distributed processes communicate with each other by sending messages containing partial results of the overall processing task. Processes on different machines communicate over the network and the resulting network load limits its scalability in two ways: First, network traffic is several orders of magnitude slower than in-machine communication.[1] Second, the bandwidth of each machine limits the amount it can possibly communicate to processes residing on other machines. *As a consequence, finding a good distribution strategy for distributed stream processing is crucial to ensure scalability.* Note that the variety and potential schemalessness of linked data further aggravates the problem as a Semantic Flow Processing (SFP) systems (1) cannot rely on the schema for data partitioning and distribution and (2) the triple-nature (rather than the reliance on n-tuples or relations) of the underlying data model potentially further subdivides the smallest unit of data increasing the number of possible partitioning (and hence, distributions).

As distributed stream processing becomes more important in many areas of business and science, researchers have proposed various ways to schedule workload in such systems. Interestingly, we found no previous work that employs existing graph partitioning algorithms to the problem of workload scheduling.

In this paper, *we propose the use of graph partitioning algorithms to optimize the assignment of tasks to machines*: we regard the data-flow within an SFP system as a graph, where the edges' weight represents the required bandwidth for information passing and the vertices' weight represents the computational load required to process incoming messages. Specifically, we operationalize the edge weights as the number of messages sent from one process to another, based on the two assumptions: first, that messages are approximately the same size, and second the computational load to be proportional to the number of messages received by a processing vertex, assuming further that all messages need the same time to be processed on all tasks. We then use a graph partitioning algorithm to optimize the distribution of processes to machines whilst addressing two possibly

[1] Numbers from 2009: 500 times for latency, 40 times for bandwidth. See http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf for more details.

opposing goals: we try to minimize costly network messages whilst distributing computation load as evenly as possible.

To evaluate our distribution approach we implemented an SFP system on top of the Storm realtime computation framework.[2] Our implementation allows compiling certain SPARQL queries, modified for stream processing, to a Storm topology. To enable parallelization of the processing functions the data flows in the topology are partitioned using appropriate hash functions. Next we empirically determine the weight of the nodes and edges, partition the graph, and "schedule" the processes accordingly on machines. In our evaluation using two real-world datasets we show that our graph-partitioning strategy can decrease inter-machine communication substantially (by 40% to 99%) whilst maintaining an even workload distribution, even using very limited data statistics.

The remainder of this paper is structured as follows: next we succinctly summarize the most relevant related literature (Section 2) before introducing the details of our approach (Section 3). This is followed by a description of our experiments (Section 4), a discussion of the results in the light of its limitations (Section 5), and an exploration of the implications and future work (Section 6).

2 Related Work

This study relates to (distributed) Information Flow Processing (IFP)[3], Semantic Flow Processing (SFP) [18], and workload scheduling. We provide an overview of the most relevant work in these fields.

Information Flow Processing: The field of IFP is vast and a survey is beyond the scope of this paper [15, 12, 8]. Here we only discuss the Aurora/Borealis [1, 2] systems, as they are most closely related to our research.
Aurora [1] lets the user specify a query using a set of operators (boxes), which are connected by links (arrows). The *Borealis* system [2] extends Aurora to include—among other things—the distribution of the workload across multiple machines. Load distribution is achieved by *query partitioning* – the assignment of operators (boxes) or a collection thereof (superboxes) to worker machines. This approach has two drawbacks: First, it limits the degree of parallelism to the number of boxes. Second, in its naïve setup, all information exchange between operators goes over the network consuming enormous amounts of network bandwidth. The only improvement to this strategy is to group operators onto the same machine. Our approach, in contrast, proposes to improve load distribution through *data partitioning*, where operators themselves are replicated across many machines.

Semantic Flow Processing: The *C-SPARQL* system [6] performs query matching on subsets of the information flow defined by windows. For query matching on the subsets it uses a regular SPARQL engine that is extended by some stream related

[2] http://storm-project.net
[3] The term Information Flow Processing has been suggested by [8] as the term *Stream Processing* is ambiguous due to its usage by both the Complex Event Processing (CEP) and the Data Stream Management Systems (DSMS) community.

features. A distributed version was implemented using Apache S4 platform;[4] yet with no particular scheduling strategy [9].

EP-SPARQL [4] is a complex event processing system, which extends the ETALIS system with a flow-extension of SPARQL for query processing [4]. ETALIS is a Prolog engine for which no distributed version exists yet.

CQELS [13] "implements the required query operators natively to avoid the overhead and limitations of closed system regimes". It optimizes the execution by dynamically re-ordering operators to "prune the triples that will not make it to the final output" thus limiting processing. As the implementation makes no assumptions about scheduling with regards to messages sent between algebra components CQELS could benefit from a scheduler based on graph partitioning.

$SPARQL_{Stream}$[7] is a streaming extension to SPARQL that allow users to query relational data streams over a set of stream-to-ontology mappings. The language supports powerful windowing constructs and *SRBench* [23] uses it as the default engine for evaluation.

INSTANS [17] and *Sparkwave*[11] are based on a RETE network. Both their implementations are non-distributed implementation, yet very efficient. Both stream querying systems support the RDF, and RDFS, and—in the case of Sparkwave—OWL entailment. It would be interesting to investigate, to what extent our approach could be built on top of a RETE-network.

Workload Scheduling: Earlier work on scheduling in stream processing concentrated on operator scheduling in wide area networks [16] and admission control [21, 22], recent work also targets the usecase in which workload of a stream processor in a compute cluster has to be scheduled [5].

SODA [21] is an admission control system and task scheduler for System S [3]. The task scheduler within SODA is based on a mixed-integer optimization program and also uses techniques from the network flow literature.

Pietzbuch et al. [16] present an decentralized algorithm that is geared towards minimizing the overall latency of a stream processor whose operators are spread out across a wide area network, while taking CPU load into account.

Xia et al. [22] map the problem of task scheduling to a multicommodity flow network and present a distributed scheduling algorithm in which the amount of communication between nodes is incrementally analyzed and reduced.

Aniello et al. [5] present two algorithms which are both geared towards reducing the number tuples tranferred over the network of a storm cluster. Their static "offline" scheduler takes characteristics of the topology into account while their "online" scheduler collects network statistics, before optimizing the schedule by moving nodes connected by *hot edges*, i.e. edges that exhibit high data volumes, on the same server. Their evaluations conducted using a synthetic and a real-world dataset show, that online-scheduling results in much lower latency than static or uniform scheduling.

[4] http://incubator.apache.org/s4

3 Problem Statement, Formal Definitions, and System Description

In this section we provide the technical foundations for our study. These include a brief introduction to the data- and processing models employed. Next, we introduce the three concepts of data partitioning, scheduling, and load balancing and how they affect the performance of a distributed system, before presenting a formal problem description. We then show, how the multi-constraint optimization problem of scheduling can be solved using a graph partitioning algorithm, before we, finally, introduce the system we built to test our hypothesis.

3.1 Data- and Processing Model

A linked data stream processing system essentially continuously ingests large volumes of temporally annotated RDF triples and emits the results again as data stream. Such systems usually implement a version of the SPARQL algebra that has been modified for processing dynamic data. In our case, we focus on a subset of those defined in the queries of the *SRBench* benchmark [23]. The processing model considered is a directed graph, where the nodes are algebra operators and data is sent along the edges. Hence, each query can be transformed to a query tree of algebra expressions – the topology of the processing graph.
While the system consumes temporally annotated RDF triples ($< s, p, o > [ts]$, where ts denotes the time-stamp), internal operators in the topolgy consume and emit sets of *variable bindings* when performing the operations associated with the respective operator. These variable bindings comprise a finite number of variable/value pairs $?var/val$, where $?var$ is a variable name and val is an RDF term. Note that source operators (i.e., the input to the topology) consume timestamped RDF triples instead of bindings and output operators may also output RDF triples if so specified by the query.

3.2 Scheduling, Data Partitioning, and Load Balancing

In order to scale the system horizontally (i.e., executing its parts on multiple processing units concurrently) we may replicate parts (or the whole) of the query's topology and execute clones of the operators in parallel. We refer to these clones as *tasks* or *task instances*. Hence, each operator will be instantiated as a finite number of n tasks (where $n \geq 1$). These task instances, and thus the workload of the system, can then be distributed across several machines in a compute cluster. We refer to the assignment of tasks to machines as *scheduling*. Figure 1 shows an example operator topology and one possible schedule distributing tasks to two servers of a compute cluster. As there are now multiple instances of each operator of the topology the data needs to be *partitioned* in accordance to the operator's needs. For stateless operators, such as filters or binders, it does not matter which variable bindings they receive for processing. For stateful operators, in contrast, such as aggregators or joins, the system needs to provide some guarantees about

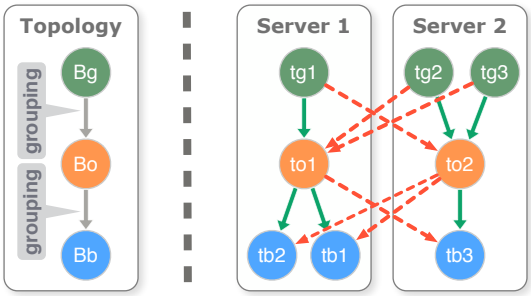


Fig. 1. A topology for an example query with three operators or algebra expressions (left side) and a possible schedule with eight tasks distributed over two servers (right side). Data flows from top to bottom. Green (solid) arrows indicate fast intra-machine communication; red (dashed) arrows indicate costly inter-machine communication.

what data gets delivered to which task instance. To this end, a topology configuration contains *grouping strategies* (or information about the data partitioning function) on the edges between operator nodes (see also Figure 1 on left).

In this study we assume the number of messages sent between the machines as the key performance indicator (KPI). This is seems to be a prudent choice, as the network can become a bottleneck of a distributed system that needs to scale horizontally. We acknowledge that our choice has limitations and therefore provide a discussion in Section 5.

Given these definitions *the goal of our approach is to find a schedule (i.e., assignment of tasks to machines) for a given topology that minimizes the total number of data messages transferred over the network, whilst maintaining an even workload distribution across machines in terms of CPU cycles.*

To achieve this goal we partition the data into logical units. We then regroup these using graph-partitioning which provides us with an optimization procedure to minimize the number of messages sent between machines. As a result, we propose the following hypothesis: *Combining data partitioning between tasks with a scheduler that employs graph partitioning to assign the resulting task instances outperforms a uniform distribution of data and tasks to machines.*

Our hypothesis assumes that different distribution strategies significantly influence the number of messages sent between the machines. Most stream processing platforms attempt to uniformly distribute compute loads possibly incurring high network traffic. Approaches like Borealis schedule the processors according to the structure of the query, where every operator is is assigned to one machine. This approach has an upper limit in parallelization equal to the number of operators and may incur high network traffic between to machines containing active operators. Instead we propose to parallelize the operators and minimize network traffic allowing for more flexibility for distributing the workload. Most importantly, *we propose that the scheduling strategy should optimize the amount of data sent between machines.*

3.3 Formal Problem Description

In principle, a linked data stream processing system can be conceived as a query graph $Q_{SFP} =< Op, MC >$, where Op is a finite set of operators op_i (i.e., $Op = \cup_{i=1}^{I} op_i$) and each op_i executes one or more algebra operations. The flow of information between the operators is established by a set of edges $mc \in MC$ (message channels) that denote a flow of messages (time-stamped variable bindings $\mu[ts]$) between the operators (op_i).

Since we want to enable parallelism and distribution each operator is instantiated in parallel as a finite number of tasks $T_{op_i} = \cup_{j=1}^{J_i} t_{i,j}$, where J_i denotes the degree of parallelism of op_i. We refer to the set of all tasks as

$$T = \cup_{i=1}^{I} T_{op_i} = \cup_{i=1}^{I} \cup_{j=1}^{J_i} t_{i,j}$$

Furthermore, each message channel $mc \in MC$ is instantiated via a finite set of channels $c_{ij} \in C$ that connect the tasks $t_i, t_j \in T$. Specifically, connected tasks send messages, i.e., time-stamped variable bindings $\mu[ts]$ to each other.

Thus for each query graph Q_{SFP} there exists a parallelized Task Graph $TG =< T, C_T >$, where in addition to the mapping of each task to exactly one operator (as specified above) each channel $c \in C$ maps to exactly one mc and each mc has at least one c to ensure connectivity. Hence, to ensure a correct mapping we require that $\forall op_a, op_b, mc_{op_a,op_b} : \exists t_{a,j}, t_{b,j}, c_{t_{a,j},t_{b,j}}$, where (i) mc_{op_a,op_b} is a message channel that connects op_a and op_b, and (ii) $c_{t_{a,j},t_{b,j}}$ connects the corresponding tasks. In addition, we require that $\forall t_{a,j}, t_{b,j}, c_{t_{a,j},t_{b,j}} : \exists_{\text{exactly one}} op_a, op_b, mc_{op_a,op_b}$ to ensure the one-to-n mapping of operators and message channels to tasks and channels.

Graph Partitioning A partitioning divides a set into pairwise disjoint sets. In our case we want to partition the vertices of a graph $G = (V, E)$ with a finite set of vertices V and a finite set of edges $E \subset V \times V$. A partitioning $P = \{P_1, \ldots, P_K\}$ for V separates the set of vertices such that

- it covers the whole set of vertices: $\bigcup_{k=1}^{K} P_k = V$ and
- the partitions P_k are pairwise disjoint: $\bigcap_{k=1}^{K} P_k = \emptyset$

In addition, we denote (i) a *partitioning function* by $part : V \rightarrow P$ that assigns every vertex $v_1, \ldots v_l$ the partition $P_k \in P$ it belongs to, (ii) a *cost function* by $cost(P) \in \mathbb{R}$, which denotes some kind of cost associated with the partitioning that is subject to optimization, and (iii) a *load imbalance factor* $stdv(P)$ that ensures that the workload of the tasks is evenly distributed over the machines.

We can easily map our problem of minimizing the number of messages that are sent between machines to a graph partitioning problem with a specific cost function. First, we define the graph to be partitioned as the Task Graph TG. A partitioning of TG maps each task to exactly one machine.

Second, in our case the cost function to minimize is the number of messages sent between machines. We operationalize the cost function for the network traffic as $cost(P) = \sum_{k=1}^{K} cost(P_k)$. Each $cost(P_k)$ denotes the cost of transmitting

messages, i.e. the bindings, across the network to a task situated in partition P_k. Hence, we increment the cost for $cost(P_k)$ by one, iff a message is being sent from task t_1 to t_2 and the two tasks are *not in the same partition*, i.e. $part(t_1) \neq part(t_2)$ and $t_2 \in P_k$.

Third, when optimizing the costs for the partitions we add the constraint that the partitions shall be balanced with respect to the computational load. We approximate the computation load for a partition by counting the number of messages that are sent over a channel for which task t is the receiving task: $load(P_k) = \sum_{c_{t_a,t_b} \in C} count(m)$ iff $part(t_b) = P_k$, where $m \in M$ is a message sent over channel c_{t_a,t_b}. Note that we count all incoming messages for each task, regardless of wether they had to be transferred over the network or not, as they have to be processed and hence consume computation power in both cases.

In order to make optimal use of the available resources, a balanced load distribution is desirable. The standard deviation $stdv(P)$ of the load for all partitions shall hence not exceed a certain threshold C:

$$C < stdv(P) = \sqrt{\frac{\sum_{p \in P} (\frac{load(P)}{K} - load(p))^2}{K}}$$

All graph partitioning in this paper were computed using the *METIS* algorithms for graph partitioning [10] – a well established graph partitioning package.

3.4 KATTS

In order to test our hypothesis we built a research prototype of a distributed linked data stream processing engine called *KATTS*.[5] In order to keep the programming overhead minimal, we chose to build our system on top of the *Storm* realtime computation framework.[6] While our current prototype is built using *Storm* it is important to note, that our findings are not only valid in the context of *Storm*, but for any system that uses partitioned data streams.

A *Storm* application is a graph, consisting of compute nodes that are connected by edges. Edges are configured using a partitioning function or *grouping strategy*. Using the abstractions of *Storm* we implemented a set of stream operators, a configuration environment, and a monitoring suite. Topologies can be specified using XML and will output the sending behavior of the topolgy: the communication graph. In addition to input operators that read time annotated n-triple files and an output operator, the current set of supported operators contains an aggregation operator (min, max, avg), a filter operator, a bind operator, and a temporal join operator. Every incoming edge of each consuming operator can be configured with a grouping strategy. If no grouping strategy is configured *local or shuffle grouping* will be used.[7] We always used the *field grouping* strategy, which partitions the data based on the value of a tuple field (i.e., the value

[5] *KATTS* is a recursive acronym for *Katts is A Triple Torrent Sieve*. The code will be made accessible at https://github.com/uzh/katts upon publication of the paper.
[6] http://storm-project.net
[7] https://github.com/nathanmarz/storm/wiki/Concepts#stream-groupings

of a variable). For partitioning *Storm* uses the *hashCode()* method of the field value, which in our case is an object of type *java.lang.String*.[8] In addition to the configuration parameters of the particular node implementation, each node of the topolgy can be configured with a value that defines its degree of parallelism, which is the number of task instances that should be created for this operator.

The monitoring suite has two main components: (1) a data collection facility, which records communication behvaior, and (2) a runtime monitoring component that keeps track of the number of input sources the system is receiving data from. When all sources have been fully processed, the data aggregation process will be executed and the topology as well as the *Storm* cluster will be halted.

4 Evaluation

In this section we evaluate if our proposed strategy is indeed better in terms of network load whilst maintaining a comparable host load when compared to a baseline strategy of trying to attain a uniform distribution of load. To that end we first present the experimental setup and then present the results.

4.1 Experimental Setup

Datasets We evaluated our system using two example queries that are built around a real world streaming use case: *SRBench* [23], which works with streams of weather measurements, and an open government dataset, which we collected through public sources.

SRBench is an RDF/SPARQL streaming benchmark consisting of weather observations about hurricanes and blizzards in the USA during the years 2001 and 2009 and contains 17 queries on LinkedSensorData,[9] which originated from the MesoWest project of the University of Utah.[10] Given that our approach only implements joins, simple aggregates, and triple-pattern matching, we restricted ourselves to Q3, which searches for weather stations observing hurricane-like condition. These are defined as "a sustained wind (for more than 3 hours) of at least 33 m/s or 74 mph." Without the prefix declaration a C-SPARQL [6] like version of this query would look as follows:[11]

```
ASK
FROM STREAM <http://www.cwi.nl/SRBench/observations> [RANGE 3h STEP 1h]
WHERE {
  ?observation om-owl:procedure         ?sensor ;
               om-owl:observedProperty  weather:WindSpeed ;
               om-owl:result            [ om-owl:floatValue ?value ] .
}
GROUP BY ?sensor
HAVING ( MIN(?value) >= "74"^^xsd:float )
```

[8] http://docs.oracle.com/javase/6/docs/api/java/lang/String.html#hashCode()

[9] http://wiki.knoesis.org/index.php/LinkedSensorData

[10] http://mesowest.utah.edu/index.html

[11] We employ a minimum aggregate rather than the average value used in http://www.w3.org/wiki/SRBench, as we think the word "sustained" means that the wind speed has to be "at least" 74 miles per hour and not "on average". We also use a step size of 1 hour instead of 10 minutes.

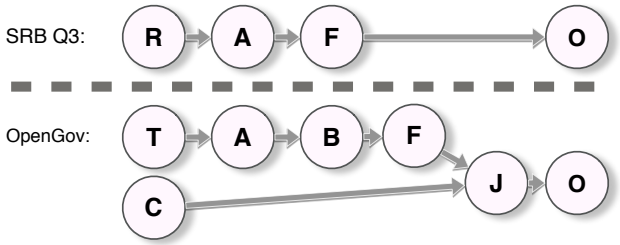


Fig. 2. The topologies for Query 3 of the SRBench and the OpenGov query.

The resulting topology has four processes depicted in Fig.2: First, the reader node (R) reads the incoming stream and scans it for the triple patterns contained in the where-clause. Second, the matched bindings are then sent to the aggregator node (A), which creates the minimum aggregate over the temporal window of three hours and a step size of one hour. Third, the output of the aggregator is then sent to the filter node (F), which filters all occurrences that are smaller than 74 mph and sends all remaining instances to the output node (O). Finally, the output node writes the occurrences into a file on disk.

OpenGov Dataset: To complement the regular setup of *SRBench*, which consists mostly of weather station measurements, we gathered a second data set, which combines data on public spending in the US with stock ticker data.[12] We devised a query that would highlight (publicly traded) companies, that double their stock price within 20 days and are/were awarded a government contract in the same time-frame. This query requires the system to scan two sources, aggregate/filter values, and finally join certain events that may have a causal relation to each using a temporal condition. The C-SPARQL representation of the query, for example, looks as follows:

```
REGISTER QUERY PublicSpendingStock AS
SELECT { ?company_name ?agency_name ?contract_id ?min_price ?max_price ?factor }
FROM STREAM <wrds.crsp/ticker.trdf> [RANGE 20 DAY STEP 1 DAY]
FROM STREAM <usaspending.org/contracts.trdf> [RANGE 20 DAY STEP 1 DAY]
WHERE { GRAPH <wrds.crsp/ticker.trdf> {
          ?ticker_id wc:PRC     ?ticker_price ;
                     wc:COMNAM ?company_name ;
                     wc:TICKER ?ticker_symbol .
        } UNION GRAPH <usaspending.org/contracts.trdf> {
          ?contract_id us:agencyid         ?agency_name ;
                       us:obligatedamount ?contract_amount ;
                       us:vendorname      ?company_name .
        }}
AGGREGATE { (?min_price, MIN, {?ticker_price}) }
AGGREGATE { (?max_price, MAX, {?ticker_price}) }
BIND (?max_price / ?min_price AS ?factor)
FILTER(?factor > 2)
```

The resulting topology (Fig.2) first aggregates (A) the ticker-sourced (T) data to compute the minimum and maximum over a time window of 20 days. It computes the ratio between these numbers (B), and then filters those solutions where that ratio is smaller than or equal to two (F). The remaining company tickers are then joined (J[13]) with the ones that where awarded government contracts (C). The joined tuples are then sent to the output node (O).

[12] http://www.usaspending.gov, https://wrds-web.wharton.upenn.edu/wrds

[13] We use a hash join with eviction rules for the temporal constraints.

Evaluation Criteria In accordance with the Properties-Challenges-KPIs-Stress-tests (PCKS) paradigm for for benchmarking SFP systems [18] we tested the performance of our Distributed Flow Processing System by choosing the number of inter-machines network messages as a key performance indicator (KPI). As a secondary performance indicator (SPI) we chose the uniformity of load distribution as measured by number of messages received per machine.

Procedure We used the following procedure to measure the performance of our approach. First, we took each dataset and partitioned it to 12 files (as we had 12 machines at our disposal). The two queries were compiled into the topologies described above and instantiated to allow 48 tasks for each node that is neither a reader nor an output node.[14] We then recorded the number of messages that were sent between tasks at runtime.

Second, to test our hypothesis we needed to partition the resulting communication graph based on the network load of each channel. Since the channel loads are not known before running the query we chose two experimental scenarios. In the first scenario we assume an *oracle* optimizer that would know the number of messages that would flow along every channel. This scenario allows to establish a hypothetical upper bound of quality that our method could attain, if it were to have an oracle. In a second scenario we assumed a *learning* optimizer that first observes channel statistics for a brief period of time and then partitions the graph accordingly. To that end we sliced the *SRBench* data into daily and the *OpenGov* data into monthly slices. We then measured the the performance of our approach based on learning during the preceding one to three time-slices, essentially providing a adaptively learning system.

Third, to partition the graph we employed *METIS* [10]. We used the *gpmetis* in its standard configuration, which creates partitions of equal size, and only changed the *-objtype* parameter to instruct *METIS* to optimize for total communication volume when partitioning, rather than minimizing on total edgecut.[15]

4.2 Results

The Suitability of Graph Partitioning for Scheduling The results of our evaluation are shown in Figure 3, which plots the number of network messages divided by the number of total messages as a measure for the optimality of the distribution. As the figure shows on the left, the *SRBench* data can be optimally partitioned by the id of the reporting weather station even when using only the data of the immediately preceeding time slice (Prev.1). All further computation can be managed on a local machine, as no further joins are necessary. This clearly indicates that some queries can be trivially distributed when a good data partition is either known or can be learned.

On the right we find the results for the *OpenGov* dataset. This evaluation is not quite as clear-cut, as the join operation requires a significant redistribution of messages. The results here are quite interesting. First, we find that our approach

[14] As we we ran our experiments on machines with more than 12 cores, we were able to achieve better capacity utilization by using more than 12 tasks per node.

[15] We used v5.0.2 with default partitioning (kway) and default load imbalance of 1.03.

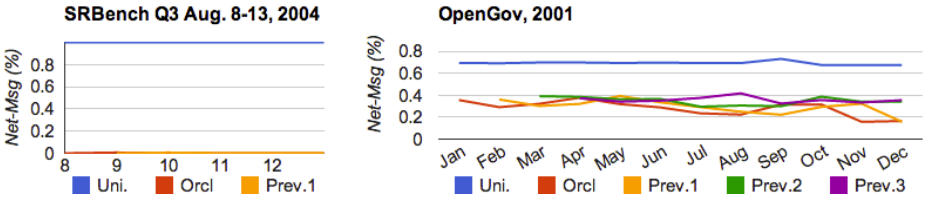


Fig. 3. Percentage of messages sent over the network for the uniform distribution and the graph partitioned setup, using either the test data itself (oracle) or data from the previous one to three time-slices as input for the graph partitioning algorithm.

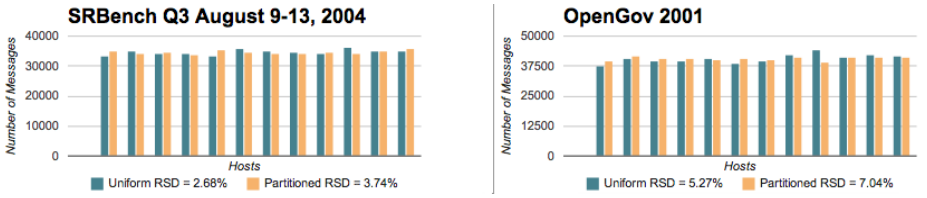


Fig. 4. Average computation load distribution for all time-slices of each dataset. RSD = Relative Standard Deviation

clearly outperforms the uniform distribution strategy by a factor of two to three. Second, it is interesting to observe that even longer learning periods, using two (Prev.2) and even three previous time slices (Prev.3), do not necessarily improve the overall performance - maybe due to over-fitting or concept drift [19].

For the sake of brevity we only show data for three time-slices of each evaluation in Table 1: on the left side again the results for the embarrassingly parallel *SRBench* query, which shows a reduction in network usage by over 99%. The right side of the table is more interesting as it exhibits the gain of our approach in a non-trivial case. Even for the *OpenGov* query, workload distribution using a graph partitioning approach yields savings of network bandwidth of over 40%.

Balancing Computation Load Next to keeping the bandwidth usage to a minimum, a distributed system must also make good use of the available computational power. For this reason we analyzed how many messages were processed by all tasks on each host for the two queries. Figure 4 shows the results of this

SRBench Q3 August, 2004				*OpenGov*, 2001			
Slice	Uniform	Oracle	Prev.1	Slice	Uniform	Oracle	Prev.1
Aug 9	1	0.7%	0.0%	Feb	69,5%	29.0%	36.0%
Aug 10	1	0.0%	0.7%	Mar	69,8%	32.1%	30.1%
Aug 11	1	0.0%	0.0%	Apr	69,8%	37.7%	32.1%

Table 1. Percentage of messages sent over the network for the uniform distribution, the partitioning based on the test data itself (oracle), and the preceeding time slice ("Prev.1" in Table) as input for the graph partitioning algorith; three time slices each.

evaluation: The load distribution resulting from the graph partitioned task assignment only differs slightly from the one found by uniform task distribution (average relative standard deviation [RSD] *OpenGov*: 7.04% for partitioning vs. 5.27% for uniform baseline; SRBench: 3.74% for partitioning vs. 2.68% for uniform baseline).

The Influence of Data Partitioning The results above are very encouraging. One of the major limitations of our measurements, however, is that we assumed that the data came partitioned into meaningful groups. Whilst this assumption is often true in practice (the input from weather stations comes as grouped messages from one station, data about one stock usually arrives from one source, etc.). But in some worst-case scenarios the data might be mixed (even if a total random intermixing is unlikely). To investigate the robustness of our procedure against this assumption we ran our approach under two different partitioning regimes: first, we made sure to partition the data along a different hash function that chosen by our system (which relies on the *Storm* hash partitioning) and second, we ensured employing the same partitioning. The results of this sensitivity analysis are shown in Figure 5 for the *SRBench* query, which graphs a Sankey chart of the inter-task communication under both conditions, where the width of the lines corresponds to the number of messages. As the figure clearly shows the mixed hashing setting requires to reshuffle all data from the readers to the processing nodes, while the equally partitioning setting provides a clean stream setting. As a consequence, we can expect that badly pre-partitioned data would not exhibit as good results as the ones we exhibited above.

5 Discussion and Limitations of the Results

The results shown in the section above show that using a graph partitioning algorithm to schedule tasks instances on machines does indeed reduce the messages sent over the network whilst only having a slightly less even load distribution.

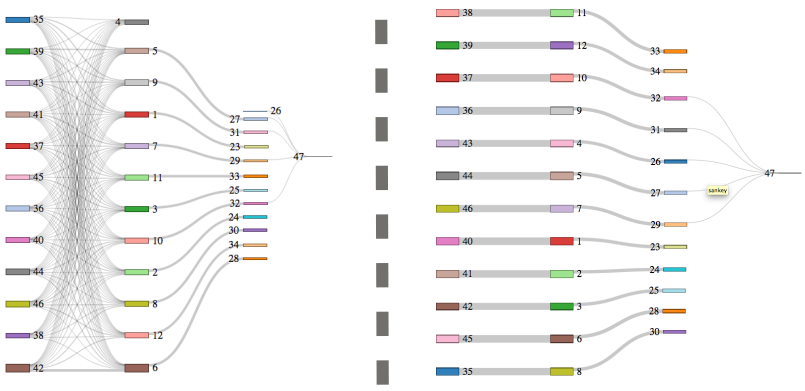


Fig. 5. Two communication graphs (data flows from left to right)
Left: Input partitioned using different hash function than the one used by Storm.
Right: Input partitioned using identical hash function as the one used by Storm.

The first part of the finding could be seen as almost tautological: it could be understood as showing that graph-partitioning using a well-establish algorithm is better than a partitioning that ignores network traffic but "only" focuses on load distribution. We believe, however, that there are subtle considerations that are less than obvious.
First, the *critical element is to realize that the operators can be parallelized with an adequate data partitioning approach* not to "just" use graph partitioning. It is the interplay of the two partitionings that enables the graph partitioning to find a good schedule: inadequate data partition can lead to highly suboptimal schedules as the results about the influence of data partitioning show.
Second, the principle of *finding the smallest possible partition given the desired degree of parallelism* (see also Section 3.2) seems important. How important needs to be investigated. Whilst the idea seems simple its details are intricate and required careful analysis—a task that we will have to continue in the future by further exploring the interactions between data and graph partitioning and devise an automated model for optimizing it.

One somewhat surprising outcome of our analysis is that the overall efficiency of the system heavily depends not only on the *consistent use* of the same partitioning function, but also on the *compatibility of the values* over which the data is being partitioned. Using incompatible data partioning functions can result in very poor performance as seen in section 4.2. If a topology contains operators that partition over incompatible fields such as in the OpenGov query, graph partitioning is still useful, but much less effective as when working with compatible fields. It is this observation which contains an interesting insight: *linked data stream processors should work with graph fragments rather than triples.* "Naturally" occuring graph fragments often contain interdependent graph elements. If one pulls these fragments apart due to some partitioning function one might have to gather them in a later join. Hence, it seems prudent to favor an approach that leaves these fragment together if a later join is foreseeable.

Our current analysis is based on some *underlying assumptions.* First, we assumed that some *network statistics are available* at the onset. Whilst this may not always be given, our findings shows that even a small amount of statistics seem to produce adequate schedules. Hence, it seems straightforward to start with an uniform distribution and then apply an incremental graph partitioning approach [14] improving the schedule during run-time.
Second, we assumed that the *processing load (both CPU and RAM) is proportional to the number of messages received* (i.e., constant operator complexity). Whilst this assumption is definitely true for some operators (e.g., computing the average) others may require more computational effort. We intend to address this issue in future work.
Third, we assumed that the *query topology was given.* Obviously, queries could be translated to various topologies; each of which would require its own schedule. Hence, it would make sense to combine our approach with a query optimizer – a task beyond the scope of this paper.

Also, our current evaluation has some limitations. First, it is limited to two datasets and queries. Whilst the queries seem representative of many settings we have seen we intend to significantly extend our evaluation in the future in terms of number of datasets and queries. Second, all our evaluations were run on a cluster with 12 machines, 1GB ethernet, and 24 cores each. Obviously, we will have to extend our evaluation to investigate the interactions between number of machines and cores available and the degree of parallelism "granted." Third, we will have to run throughput-analyses in real-world setups in addition to our current network analysis adding number of messages ingested per second as KPI.

6 Conclusion and Outlook

In this study we investigated whether and how scheduling the tasks of Distributed Semantic Flow Processing (DSFP) systems benefits from applying graph partitioning. We implemented our approach on the Katts DSFP engine and evaluated it using a query of the *SRBench* benchmark and a usecase inspired by the open government movement with regards to network load. The results show that using a graph partitioning algorithm to schedule task instances on machines does indeed reduce the number of messages sent over the network. We also found that this only leads to a slightly less even load distribution.

The critical element for optimizing the scheduling using graph partitioning is an adequate data partitioning for parallelizing the operators. Future work will investigate whether the principle of finding the smallest possible data partition given the desired degree of parallelism is as important as our experiments indicate.

Our study's most important shortcomings are its limitation to two datasets and queries and the fixed setup of the distributed system. For the first we intend to systematically extend our evaluation in the future in terms of number of datasets and queries. For the latter, is it the interactions between number of machines and cores available and the degree of parallelism that require further research. Especially the impact of such interactions on throughput in terms of messages ingested per second is of interest here.

We are confident that our findings help making DSFP systems more scalable and ultimately enable reactive systems that are capable of processing billions of triples or graph fragments per second with a negligible delay. It is our firm belief that the key to addressing these challenges needs to and will have to be revealed from the data itself.

Acknowledgements We would like to thank Thomas Hunziker, who wrote the first prototype of the *KATTS* system during his master's thesis in our group.

References

1. Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Erwin, C., Galvez, E., Hatoun, M., Maskey, A., Rasin, A., Et Al.: Aurora: a data stream management system. In: Proc. of the 2003 ACM SIGMOD. pp. 666–666 (2003)

2. Abadi, D.J., Ahmad, Y., Balazinska, M., Hwang, J.h., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: Proc. CIDR2005. pp. 277–289 (2005),
3. Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Park, Y., Venkatramani, C.: Spc: A distributed, scalable platform for data mining. In: Proc. Workshop on Data Mining Standards, Services and Platforms, DM-SSP (2006)
4. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW2011. pp. 635–644 (2011)
5. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in storm. In: DEBS2013 (2013),
6. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: A Continuous Query Language for RDF Data Streams. Int. J. of Sem. Comp. 4(1), 3–25 (2010)
7. Calbimonte, J.p., Corcho, O., Gray, A.J.G.: Enabling Ontology-based Access to Streaming Data Sources. In: Proc. ISWC 2010 (2010)
8. Cugola, G., Margara, A.: Processing flows of information. ACM Computing Surveys 44(3), 1–62 (Jun 2012),
9. Hoeksema, J., Kotoulas, S.: High-performance Distributed Stream Reasoning using S4. In: First International Workshop on Ordering and Reasoning (2011)
10. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM J. on Scientific Comp. 20(1), 359–392 (Jan 1998),
11. Komazec, S., Cerri, D.: Sparkwave: Continuous Schema-Enhanced Pattern Matching over RDF Data Streams. In: DEBS 2012 (2012)
12. Lajos, J.F., Toth, G., Racz, R., Panczel, J., Gergely, T., Beszedes, A.: Survey on Complex Event Processing and Predictive Analytics. Tech. rep., Citeseer (2010),
13. Le-phuoc, D., Dao-tran, M., Parreira, J.X., Hauswirth, M.: A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In: Proc. ISWC 2011. vol. 7031, pp. 370–388 (2011)
14. Ou, C.W., Ranka, S.: Parallel incremental graph partitioning. Parallel and Distributed Systems, IEEE Transactions on 8(8), 884–896 (1997)
15. Owens, T.: Survey of event processing. Tech. Rep. December, Air Force Research Laboratory Public Affairs Office (2007),
16. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-Aware Operator Placement for Stream-Processing Systems. In: Proc. ICDE2006 (2006)
17. Rinne, M., Nuutila, E., Seppo, T.: INSTANS : High-Performance Event Processing with Standard RDF and SPARQL. In: ISWC 2012 Post. & Demos. pp. 6–9 (2012)
18. Scharrenbach, T., Urbani, J., Margara, A., della Valle, E., Bernstein, A.: Seven Commandments for Benchmarking Semantic Flow Processing Systems. In: ESWC 2013 (2013)
19. Vorburger, P., Bernstein, A.: Entropy-based Concept Shift Detection. In: Proc. ICDM2006. pp. 1113–1118 (2006)
20. White, T.: Hadoop: The definitive guide. O'Reilly Media, Inc., 3 edn. (2012),
21. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.L., Fleischer, L.: SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In: Proc. Middleware2008 (2008)
22. Xia, C., Towsley, D., Zhang, C.: Distributed resource management and admission control of stream processing systems with max utility. In: Proc. ICDCS2007 (2007)
23. Zhang, Y., Duc, P.M., Corcho, O., Calbimonte, J.p.: SRBench : A Streaming RDF / SPARQL Benchmark. In: Proc. ISWC 2012 (2012)

A Distributed Directory System

Fausto Giunchiglia and Alethia Hume

Department of Information Engineering and Computer Science
University of Trento, Italy
{fausto,hume}@disi.unitn.it
http://www.disi.unitn.it

Abstract. We see the local content from peers organized in directories (i.e., on local ordered lists) of local representations of entities from the real world (e.g., persons, locations, events). Different local representations can give different "versions" of the same real world entity and use different names to refer to it (e.g., George Lombardi, Lombardi G., Prof. Lombardi, Dad). Although the data from these directories are related and could complement each other, there are no links that allow peers to share and search across them. We propose a Distributed Directory System that constructs these connecting links and allows peers to: (i) maintain their data locally and (ii) find the different versions of a real world entity based on any name used in the network. We evaluate the approach in networks of different sizes using PlanetLab and we show that the results are promising in terms of the scalability.

Keywords: Name-Based Entity Search, P2P, Entity Directory

1 Introduction

We see Internet as a network of peers (a P2P network) organizing their content in directories, which digitally represent their own versions of *entities* that exist in the real world. Entities can be of different types (e.g., person, location, event and others), they have a name, and are described by attributes (e.g., latitude-longitude, size, birth date), which are different for different entity types [1]. Different versions of an entity can represent different points of view, they could show different aspects of the entity or the same aspects with different level of details. In a way, the local representations from peers can be seen as pieces of information about a particular entity that are stored in a distributed manner in the network.

In this network, the different directories contain related data and, to some extent, they can complement each other. One problem that prevents us from exploiting the relation between these data is that there are no links connecting the local directories from peers. An effort to connect related data on the web is that of Linked Data[1], which allowed linking important datasets like, dbpedia, Freebase, DBLP, ACM, and others. Nevertheless, this approach leaves out of

[1] http://linkeddata.org/

the semantic web the individual users (i.e., simple normal peers) and the data from their local directories stored in personal devices (e.g., smart-phones, PDAs, notebooks, etc.). We propose building a distributed directory that constructs the connecting links among the local directories at this level, i.e., the level of simple peers with personal devices. It it important to note that the whole directory can be seen as another dataset, which could be included as another node in the Linked Data graph. In this way, the directory would become the bridge that allows simple peers to participate as part of the semantic web as opposed to act only as consumers of it.

As in any directory, a peer normally identifies and distinguishes an entity from others by means of names (e.g., George Lombardi, Trento, Italy, University of Trento), which play a different role from the other attributes because they are identifiers rather than descriptions [2]. The values of other types of attributes have a meaning that can be understood, e.g., by mapping them to concepts from a knowledge base, like WordNet[2]. Names, on the other hand, are strings that behave similarly to keywords. Real world entities can be called by multiple names as a consequence of variations and errors. Moreover, the set of names used in different local representations to identify the same real world entity can be different, at the same time that the sets of names used to identify different real world entities can overlap.

The approach we propose for a *Distributed Directory System (DDS)* incorporates the notion of a real world entity described by different local representations from peers. This notion is used to organize the references to the local representations in order to allow finding all the available information about entities. Our system offers two main features:

- First, it takes into consideration that multiple, possible different, names can be used to identify the same real world entity (e.g., George Lombardi vs. G. Lombardi and Italy vs. Italia).
- Second, it allows peers to have control over the privacy of their data because the *DDS* stores only the names of the entity and a link to the local representation.

As a result, any name that is used in some local representation to identify an entity can be used to find all the different versions of that entity that are stored in the network of peers.

The paper is structured as follows. Section 2 presents a motivating example that shows a world of related directories, while Section 3 formalizes the basic notions that link the different directories. In Section 4, we explain the name matching problem that arises when linking different directories. Then, a distributed entity directory is proposed in Section 5 and the algorithms to perform search in such directory are explained in Section 6. The implementation and the evaluation details are discussed in Section 7. Finally, the related works are discussed in Section 8 and the conclusions are presented in Section 9.

[2] `http://wordnet.princeton.edu/`

2 A World of Directories

Nowadays, most of the organization of our data is done in terms of directories. A well known and old example is the telephone book directory, used to organize address and phone numbers of people and companies. Newer forms of directories can be seen, for example, in contact lists, document directories, event directories (i.e., calendars or agendas) used by peers in current devices (e.g., computers, PDAs, smart-phones) to organize the local representation of entities of their interest. Moreover, the data from different directories (possibly from different peers) can be related. Different peers attending to the same event might store local representations of the event. Each of them might also have the contact information of the other peers attending to the event, e.g., a meeting.

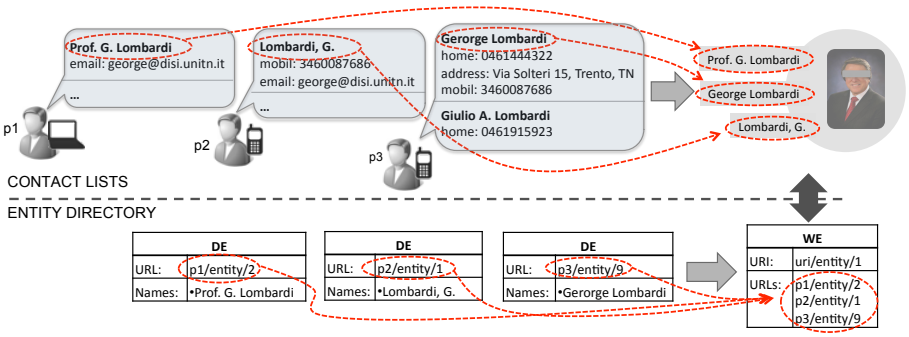


Fig. 1. Contact Lists Example

Let us consider in details the example of contact lists in different devices from the peers of a network that connects students, researchers and professors among them (e.g., SmartCampus[3]), and with their family members. The first part of Figure 1 (upper part) shows that the contact list of each device can be seen as a local directory of people. Different peers in this network can have different information about the people in their contact lists, like phone numbers, email addresses, skype user and others, which show different ways to get in touch with them. For example, suppose that p_1 is a student that is taking a course with prof. George Lombardi and therefore p_1 has, in its contact list, the university email address of the professor. A researcher p_2 that is working with him could have more information, like his email and mobile phone number. On the other hand, a family member p_3 may have his home address and phone number but not the university email (because such information is not relevant for p_3).

Now, suppose that another researcher in the network, let us call it p_4, hears about prof. Lombardi work and wants to contact him. We can see that:

[3] `http://www.smartcampuslab.it`

1. First, the information that p_4 needs is distributed in the network and the problem is knowing where the different pieces are stored
2. Second, the different peers can call the same person using different names, e.g., Prof. Lombardi, George Lombardi, G. Lombardi. In our example, this means that p_4 need to be sure that the other peers (i.e., p_1, p_2 and p_3) are all referring to the same person as he is.
3. Third, the contact information can change in time. The work email of Prof. Lombardi will change if his affiliation changes, his phone numbers can change at any time, and his address will change if he changes residence.
4. Finally, the privacy and the sensitiveness of the information have to be considered. Most likely the phone number and address of the home of prof. Lombardi would be more private than the university email. As a consequence, $p3$ will not share such information with everyone.

3 Linking Directories

We define a Directory of Entities that formalizes the links between data from different directories through the distinction between a Digital Entity (DE) and a Real World Entity (WE). A DE is defined as a local representation of an entity that exist in the real world. A URL (Uniform Resource Locator) is used in order to uniquely identify a DE and it can be used (by dereferencing) to obtain the full local description (i.e., based on attributes). We also consider a set of names $\{N\}$ as the human readable identifiers used in DEs to refer to a WE and distinguish it from others. Formally,

$$DE = \langle URL, \{N\} \rangle \tag{1}$$

On the other hand, a WE represents the real world entity and is modeled as a class of DEs. We use a URI (Uniform Resource Identifier) to uniquely identify each WE. Formally,

$$WE = \langle URI, \{URL\} \rangle \tag{2}$$

where $\{URL\}$ is a non-empty set of identifiers of different DEs that describe WE. As a consequence of the composition of these definitions we can see that multiple sets of names are given to a WE through DE definitions from different peers that describe the same WE.

In the second part of Figure 1 (lower part) we show how the example from Section 2 can be formalized in terms of these notions (i.e., DEs and WEs). We can see a *one-to-one* mapping between the WE from an entity directory and the real person represented in different contact lists. Moreover, we see that an entry from a contact list is translated into a DE in the directory (i.e., also a *one-to-one* mapping). There is a *one-to-many* relation between WEs and DEs which shows that each single entry in a contact list correspond to one person but one person can be described in many different entries (possibly from different peers). Finally, the relation between $Names$ and WEs introduces a name matching problem that is better discussed in the following section.

Note that these notions allow the separation between "what" is being represented and "where" is being represented. This separation is needed in order to model the issues stated in items 1 and 2 from the example of Section 2. The DEs model the different pieces of information that p_4 needs and their $URLs$ tell us where they are. The WE models the link that connects different DEs and its URI identify what they represented. Regarding item 2, we can see that different sets of names are given in DEs, which models the fact that p_1, p_2 and p_3 can define the different names that they use to call an entity.

On the other hand, the distinction between the two notions (DE and WE) also provide the infrastructure to deal with the issues introduced by the other two items (i.e., items 3 and 4 in Section 2). The dynamism of the information about the entities and the privacy of local data are constrained to affect DEs. In this way, when the email of Prof. Lombardi changes (see Figure 1), p_2 (the researcher) updates its local representation (i.e., the DE). The corresponding WE definition is not affected by this update, nevertheless the information (available in the P2P network) about Prof. George Lombardi is updated. Similarly, access control can be implemented over the data associated to each single DE representation, which do not affect WE definitions. Note that such implementation (i.e., access control implementation) is out of the scope of this paper, but the interested readers are invited to see (for example) [3].

4 Name Matching

Names are human readable identifiers that serve the purpose of distinguish an entity from others. They are labels composed by a combination of words, numbers and symbols [2]. In the context of our entity directory, we define the set of names that identify a WE as the union of the names used in DEs that locally represent that WE in different peers. Names are different from other attributes because they play the role of keywords rather than been mapped to concepts from a knowledge base. As such, names can suffer from different types of variations. Following the results from the study performed in [4], we can distinguish among the following types:

- **Format.** The format variations have a strong dependence with entity type and affect mostly to people names. They include the variation of the order in which the words of a name can be written (e.g., *George Lombardi* and *Lombardi, George*) and the multiple abbreviations that can exist for the same full name (e.g., *Giulio Augusto Lombardi* can be abbreviated as *G. A. Lombardi*, *Giulio A. Lombardi* and others). It is also important to notice that the abbreviation of a name can be a valid reference to many different full names (e.g., *G. Lombardi* is valid for *George Lombardi* but also for *Giulio Lombardi*).
- **Full translations.** Names sometimes are written differently in different languages (e.g., *Trento* in Italian, *Trient* in German or *Trent* in English).
- **Part-of translations.** In other cases only one part of the name changes in different languages. This is the case of names composed by common and

proper nouns, where the common noun is called trigger word in [4] and is the only part that is affected by the translation (e.g., *University of* Trento vs. *Università di* Trento).
- **Misspellings.** Names can be misspelled, either in the definition of a DE or during the specification of a search query. The misspellings can be a consequence of variations in the punctuation, capitalization, spacing, omissions, additions, substitutions, phonetic variations (e.g., *Fasuto* vs. *Fausto*, *G Lombardi* vs. *G. Lombardi*).
- **Pseudonyms.** Entities also have pseudonyms that are not (necessarily) variations of a name but rather alternative names for an entity, which can be defined (and used) in different contexts. This is the case for some arbitrary nicknames that are sometimes used by peers to refer to a DE (e.g., *Fede* is commonly used as a nickname for *Federico* or *Federica* and *The King of Rock and Roll* is a common nickname for *Elvis Presley*).

The name variations together with the DE definition presented above, show that the relation between names and DEs is of the type *many-to-many*. In turn, this leads to a name-matching problem when we intend to search an entity based on its names [2]. This problem, in the context of the entity directory, can be decomposed in:

1. The problem of matching names inside the network: A name used in a DE can be a variation of the name used in another DE that represent the same WE. We need to take into consideration all the multiple names (including name variations) used in the network to identify a WE and match them to all the different DEs that describe WE. In the example from Figure 1, if the user is searching an entity with the name *"George Lombardi"*, the directory should be able to return all the DEs (i.e, *p1/entity/2*, *p2/entity/1* and *p3/entity/9*) that represent the different versions of *uri/entity/1* rather than only returning the one that give it such name (i.e., *p3/entity/9*).
2. The problem of matching queries with the names used in the network: This case considers query names that are unknown to the entity directory, but that are however variations of one or more known names. We say that a name is unknown to the directory if there is no DE in the network that uses such name to identify a WE. The easiest example is a query name that is misspelled with regard to the DEs of the directory. In the example from Figure 1, if the user input the query *"Goerge Lombardi"*, the search should be able to find that *"George Lombardi"* is a candidate match.

5 A Distributed Directory System

In this paper we propose a Distributed Directory System (DDS) that organizes information about entities incorporating the notions of WE and DE, which were presented in Section 3. These notions allow the separation of the problem of finding the DEs that represent different versions of a WE from the problem of

finding WEs that are identified with multiple names. We exploit this separation by building two different indexes, one to deal with each problem.

A *DEindex* is created to map WEs (i.e., $URIs$) to DEs (i.e., $URLs$) and can be formally defined as,

$$DEindex = \{WE \rightarrow DE \mid \nexists WE' \rightarrow DE \in DEindex \; s.t., WE' \neq WE\} \quad (3)$$

We can see that this index encodes the *one-to-many* relation between WEs and DEs because the mapping of different WEs to the same DE is not allowed. On the other hand, a *WEindex* is created to map the names that are given (in local representations) to WEs (i.e., $URIs$). Let us call $\{N^{DE}\}$ to the set of names of a digital entity DE. Then, the *WEindex* can be formally defined as,

$$WEindex = \{N \rightarrow WE \mid \exists WE \rightarrow DE \in DEindex \; s.t., N \in \{N^{DE}\}\} \quad (4)$$

We can see that this index encodes the *many-to-many* relation between $Names$ and WEs because the only constraint on the mappings is related to the existence of a local representation that gives "support" to such mapping.

Let us now discuss in more details how the publication, maintenance and search of entities are done in the DDS:

The *publication and deletion of DEs* in the network are the two main events that modify the DDS by affecting the content of the indexes defined above. The publication of a DE affects both indexes in a straightforward manner. First, the DE is associated to the WE that it represents by adding the corresponding mapping (i.e., $WE \rightarrow DE$) to the $DEindex$. Second, the mappings $N_i^{DE} \rightarrow WE$, of each name N_i^{DE} in $\{N^{DE}\}$ to the WE that is associated to the DE, are added to the $WEindex$. In order to do this, we assume that the peer locally caches the identifier (i.e., the URI) of the WE that is represented by its DE[4]. On the other hand, when a DE is deleted from the network, only the $DEindex$ is directly affected. The same mapping $WE \rightarrow DE$ that is added when the DE is published, is then removed from the $DEindex$ when the peer deletes the DE. Regarding the $WEindex$, we say that it is not directly affected because the mappings of names can be removed only after verifying that they are no longer valid to identify the corresponding WE. Such verification is further discussed as part of the DDS maintenance.

The *maintenance* of the DDS is performed through periodic checks over the indexes in order to detect and remove entries that are no longer valid. In the $DEindex$, an entry can be considered invalid if it contains mapping to a DE that has been unreachable for a long time. In order to detect this situation, each entry is attached with a timestamp corresponding to the last time when the DE was reachable. This timestamp is updated in every periodic check. When the DE is not reachable, the interval between the last reachable time and the current time is verified. The corresponding entry is removed from the $DEindex$ if such interval exceeds a given threshold. An entry from the $WEindex$, on the

[4] Note that the initial identification of the WE described by a DE is a problem of identity management and is out of the scope of this work. See for example [5, 6]

other hand, is considered invalid if it contains a mapping that do not complies with the constraint established by the index definition presented in equation 4. This means that a mapping between N and WE has to be removed from the $WEindex$ when there are no DEs in the network using the name N to refer to such WE. In other words, when none of the available entities provide support to such mapping.

Search in the DDS can be performed using two different types of identifiers, $URIs$ and *names*. In this context, having as input a URI means that the target WE has been uniquely and fully identified. Therefore, the goal of the search is to obtain all the different representations (i.e., the DEs) of the WE. On the other hand, in a search based on names, we need to find the candidates WEs (to be the right answer) as a consequence of the *many-to-many* relation between names and WEs. After the candidates WEs has been found, we can use the search by URI to find the different representations of them. In what follows, the search by names is considered in more details while the search by $URIs$ is included as a part of the former.

A query is formally defined as $Q = \{N^Q\}$, where $\{N^Q\}$ is the non-empty set of names used to identify one target WE. Then, the problem of searching entities based on their names can be seen as retrieving WEs that are described in the network by at least one DE, such that, the intersection between $\{N^{DE}\}$ and $\{N^Q\}$ is not empty. This definition considers a partial matching between $\{N^{DE}\}$ and $\{N^Q\}$ in order to allow finding a WE from any of the names given to it on different DEs. In turn, this can be translated in the formal definition of the Query Answer (QA) as follows:

$$\begin{aligned} QA = \{\langle WE, \{DE\}\rangle \mid \exists N' \in \{N^Q\} : N' \rightarrow WE \ \in \ WEindex \\ \wedge \ \forall DE' \in \{DE\} : WE \rightarrow DE' \in DEindex\} \end{aligned} \tag{5}$$

As we mentioned before, this answer is build in two steps. The algorithms that perform the two steps are presented in Section 6.

6 Algorithms

We assume that the indexes offer non-blocking APIs (to allow the parallelization of index lookups), which mean that a call to the GET function on the indexes returns immediately a reference to an object that will be filled with the results from the index lookup. In Algorithm 1, we define the global data structures, which are strictly related to the indexes. They are used across the different functions involved in the search. We use the statement *for all* (line 6 in Algorithm 2 and line 8 in Algorithm 3) to denote the concurrent execution of the statements that are in its body (i.e., line 7 in Algorithm 2 and lines 9 to 24 in Algorithm 3).

The Search Entity function is presented in Algorithm 2 and is the main entry point for the search by names. This function receives the query names and returns a set of candidate WEs according to the constraints given in Equation 5. In order to measure how relevant each candidate WE is, we count the number of

query names that match with the names associated to the WE. This relevance is associated to each candidate WE and included in the resultset. In line 7, the first step of the search by names is initiated with the call to the $GetWEindex$ function of the $WEindex$. The object returned by the function is given to the corresponding handler function, which knows how to process it.

Algorithm 1 Global Data Structures

```
1: WEAnswer : ⟨isComplete, name, weAnsValues⟩
2: DEAnswer : ⟨isComplete, URI, deAnsValues⟩
3: isComplete : boolean                    ▷ TRUE when the index lookup is finished
4: weAnsValues : NULL OR {URI} OR {URL} OR {{URI} ∪ {URL}}
5: deAnsValues : {URL}                                    ▷ not empty set of URLs
```

Algorithm 2 Search Entity

```
1: function SearchEntity(names : {name}) → {⟨WE, relevance⟩}
2:     WEs : {⟨WE, relevance⟩}                              ▷ stores search results
3:     WE : ⟨URI, {URL}⟩                  ▷ {URL}.size == 1 when URI == NULL
4:     relevance : integer
5:     WEs := {}
6:     for all name ∈ names do                                  ▷ Parallel threads
7:         HandleWEAnswer(GetWEindex(name), WEs)
8:     end for
9:     return WEs
10: end function
```

The Algorithm 3 shows the $HandleWEAnswer$ function, which is in charge of processing the values retrieved from the $WEindex$. We can see from lines 4 to 6 the loop that waits until the answer is completed. Then, in line 8, we start one execution thread to process each retrieved value. A value returned from the $WEindex$ represents a WE, it can be a URI or a URL (see line 4 from Algorithm 1). In the former case, we say that the WE identity is known. The corresponding instance is created (line 10 in Algorithm 3) with the global identifier and an (up to now) empty set of DEs. In the later case, the URL identifies a WE with no global identifier and we assume that there is only one DE that describes it (line 18 in Algorithm 3).

In lines 11 and 19, we check whether the WE is already in the result-set. If it is, we call the function $relevanceWE{+}{+}$, which increments the count of the relevance that is associated with the WE. Otherwise, we *add* the WE to the result-set with a relevance count initiated to 1 (lines 14 and 22). At this point, if we are in the case of a WE with global identifier (i.e., with a URI), the second step of the search is initiated with the call to the $GetDEindex$ function

of the *DEindex* (see line 15). The object returned by the function is given to the *HandleDEAnswer* function, which then process it.

Algorithm 3 Handler of the WE Answers

```
 1: function HandleWEAnswer(weAnswer : WEAnswer, WEs : {⟨WE, relevance⟩})
 2:     waitingTime : integer
 3:     waitingTime := 5                                  ▷ parameterizable waiting time
 4:     while weAnswer.isComplete = FALSE do
 5:         WAITms(waitingTime)                           ▷ specified in milliseconds
 6:     end while
 7:     if weAnswer.weAnsValues ≠ NULL then
 8:         for all weAnsValue ∈ weAnswer.weAnsValues do  ▷ Parallel threads
 9:             if isURI(weAnsValue) then
10:                 wEntity := ⟨weAnsValue,{}⟩
11:                 if wEntity ∈ WEs then
12:                     relevanceWE++(WEs, wEntity)
13:                 else
14:                     add(WEs,⟨wEntity,1⟩)
15:                     HandleDEAnswer(GetDEindex(weAnsValue), WEs)
16:                 end if
17:             else
18:                 wEntity := ⟨NULL,{weAnsValue}⟩
19:                 if wEntity ∈ WEs then
20:                     relevanceWE++(WEs, wEntity)
21:                 else
22:                     add(WEs, ⟨wEntity,1⟩)
23:                 end if
24:             end if
25:         end for
26:     end if
27: end function
```

Algorithm 4 Handler of the DE Answers

```
1: function HandleDEAnswer(deAnswer : DEAnswer, WEs : {⟨WE, relevance⟩})
2:     waitingTime : integer
3:     waitingTime := 5
4:     while deAnswer.isComplete = FALSE do
5:         WAITms(waitingTime)
6:     end while
7:     addDE2WE(WEs, deAnswer.key, deAnswer.deAnsValues)
8: end function
```

Finally, the Algorithm 4 shows how the values retrieved from the $DEindex$ are handled. First, we wait until the answer is completed (see the loop from line 4 to line 6) and then the values are used to update the resultset. Note that the function *addDE2WE* takes the key (i.e., the URI) to identify, in the resultset, the WE that has to be updated. The values (i.e., the $URLs$) are then associated to such WE in order to complete the QA. We say that this function (called in line 7 in Algorithm 4) adds DEs to a given WE from a given set.

7 Implementation and Evaluation

We implement the distributed directory on top of a P2P network, where the distribution of the indexes is done using a Distributed Hash Table (DHT). DHTs[5] allow the peers participating in the network to store and retrieve pairs of key and value. In particular, we use TomP2P[6], an advanced DHT library that extends the basic functions of DHTs. The library supports storing multiple values mapped to the same key and distinguishes between different index domains. The execution of the operations over different index domains can be seen as having different DHTs, i.e., one for the $DEindex$ and other for the $WEindex$.

We are interested in the evaluation of the approach under realistic network conditions and we want to measure how much the performance decreases when the size of the network grows (i.e., the scalability). The performance is considered here in terms of the time that takes the system to process a query. We use *PlanetLab*[7] as a testbed because we believe it gives us the realistic network conditions that we need. PlanetLab provides a network of computers (i.e., nodes) that are distributed around the world, connect to each other through the internet and are available for research purposes. We perform the evaluations on networks of 50, 100 and 150 peers and the data extracted from the proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)[8] are used to generate the data-sets. We use the titles of publications, names of authors and names of locations related to the conference.

Each data-set is produced by generating triples of $\langle Name, URI, URL \rangle$. The names and $URIs$ are replicated in order to simulate different WEs having the same name and different peers storing DEs that describe the same WE. Let us call p_n to the popularity of a name n (i.e., number of WEs that are called by n) and p_{we} to the popularity of a WE (i.e., number of DEs that represent WE). First, for each name n, we generate p_n triples with the same name (different URI and URL). Second, for each URI, we generate p_{we} triples with the same name and URI but with different $URLs$. The popularities p_n and p_{we} follow a Zipf[9] distribution, which means that there is a long tail of unpopular names and WEs. The distribution of both popularities are independent, which means

[5] `http://en.wikipedia.org/wiki/Distributed_hash_table`
[6] `http://www.tomp2p.net/`
[7] `https://www.planet-lab.eu/`
[8] `http://ijcai.org/`
[9] `http://en.wikipedia.org/wiki/Zipf's_law`

that a popular WE do not necessarily has a popular name and vice versa. We assume that the local entity base of each peer contains, in average, 2000 DEs. We have overall around 100000, 200000 and 300000 DEs. The query set for each peer is generated by randomly selecting a set of 1400 names from the initial set of entity names.

During the evaluation, we first index the data-set for the corresponding network size and then the peers begin the search evaluation process pseudo-simultaneously. In this process, each peer performs the following steps: (i) takes a query from the query set, (ii) runs the search, (iii) measures and logs the time that the system takes to respond to the query, (iv) waits a random interval of time (between 1 and 3 seconds), and (v) go back to step (i). These steps are repeated until the end of the set of queries. Once all the peers end the search process, we compute the average query time for the network. We show the results for the different network sizes in Table 1. The values for the average query times

Table 1. Average query time

Network Size	50 peers	100 peers	150 peers
Avg. Query Time (in seconds)	2.77	2.75	2.61

are stable with the network growth and we believe this is a promising result regarding the scalability of the directory. On the other hand, when comparing to information retrieval systems (in general), the average times for search are still high.

In order to have better understanding of the query times that contribute to these averages, we analyze the distribution of the query time in the different networks. In Figure 2 we show the results of this analysis, where we can see that also the query time distribution is stable with regard to the network growth. Also in Figure 2 we can notice that more than 55% of the queries are actually answered in less than a second, while in almost 70% of the cases the response arrives in less than 2 seconds (which is less than the average time). Moreover, only 9% of queries take more than 5 seconds to be answered.

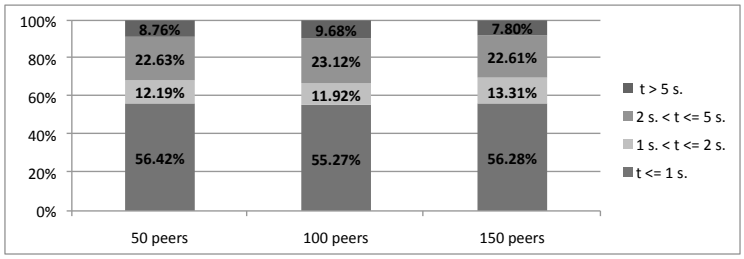


Fig. 2. Query time of different networks

It has to be noted that the results are returned after the query answer is complete, i.e., once all the lookups involved in the query have ended. This means that

a single slow lookup is enough to delay the computation of a query answer and therefore increase the query time. Furthermore, we know that particularly slow peers can produce this problem when a lookup has to be routed through them. We believe that, in the big picture, the scalability of the approach is a promising and important result. On the other hand, there are some techniques to perform result catching or to avoid routing through slow peers (see for example [7]) that can be implemented to reduce the effect of slow peers at query time.

8 Related Work

The work introduced in this paper involve the approaches that are capable of managing information about entities in a P2P network. More specifically, our approach deals with the distributed indexing and searching of entities based on their identifiers. To the best of our knowledge there are no approaches that integrates these areas, i.e., that performs search of entities over a p2p network. Nevertheless, we give an overview of related approaches from both areas.

Some entity aware approaches concentrate the attention on the definition of models and structures for the representation of entities [1]. In [6] an entity name system (ENS) is proposed in order to provide support for the generation and reuse of globally unique identifiers for entities across different and independent RDF repositories. The local repository of a single user is not considered as a source of data and the users need a special access permit in order to contribute with the definition of entities. As a first step towards searching, the work presented in [8] proposes a model that analyzes the query specification and performs the disambiguation of the desired type of entity. In [9], named entities are extracted by analyzing queries based on syntactic matching of patterns. These approaches do not directly address the search, but their results are relevant for the definition of the directory proposed in this paper.

Other approaches that perform search following an entity centric perspective can be found in the literature [10–12]. Entity search engines are proposed in [10, 12], heuristic rules are used in [11] to identify entities appearing in a collection of documents and a service to find documents that contain statements about particular resources is provided in Sindice [13]. Most of these approach collect data from multiple web sources (i.e., by crawling) but do not consider distribution at the level of single users (i.e., a p2p network). In particular, [12] automatically aggregates descriptions from the different sources and allows subsequent navigation to related entities. Distribution is considered in terms of clusters of computers that allow parallel processing and scalable storage but the search is centralized (i.e., they build centralized indexes). In contrast to these approaches, our approach performs a distributed search in a P2P network and allows users to maintain their data locally.

On the other hand, we have P2P approaches, which perform distributed search but are not aware of entities [14, 15]. They are mainly classified as unstructured and structured approaches. The first unstructured networks (e.g.,

Gnutella[10]) have scalability problems due to the number of messages generated and do not guarantee that all answers will be found. Other approaches use clustering techniques [16–20], their goal is to find the best group to answer a query and then send the query to the peers in that group. Our approach can find all available answers and has proven to be promising in terms of scalability.

We can find also more structured approaches that aim to guarantee the location of the content shared on the network (e.g., CAN [21], Chord [22] Pastry [23] and Tapestry [24] They store pairs of $\langle key, value \rangle$ in a Distributed Hash Table (DHT) and then retrieve the value associated with a given key. Other approaches perform multi-keyword search using DHTs but they can be very expensive in terms of required storage and generated traffic (e.g., see [25]). Hierarchical structures combine clustering techniques with the structure of DHTs [26–29]. In general, P2P approaches provide the techniques needed in order to build our solution. The novelty of our approach is in the domain of application of such techniques.

9 Conclusions

We presented and approach for a distributed directory of entities that introduces the notions of DE and WE in order to link local directories of different peers. The directory provides search services based on entity identifiers. In particular, we presented the algorithms for searching entities based on their names. We discussed the name matching problem that appears as a consequence of the *many-to-many* relation between names and WEs. Then, we showed that, by its design, our directory deals with the problem of matching names inside the network (i.e., the first part of the name matching problem).

The data from peers are stored locally, only the identifiers and the links to the local representations are indexed. This infrastructure allows the implementation of access control mechanisms on the local representations in order to deal with privacy issues. At the same time, the changes made by peers in local representations, are available in the directory in a straightforward manner. The indexes are distributed using a Distributed Hash Table (DHT) but the directory definition is independent from a specific underlying DHT implementation.

The evaluation of the search was performed on networks of 50, 100, and 150 peers running on PlanetLab. The average query time (as a measure of the performance) for different network sizes were presented as well as the distribution of the query times. The results can be considered promising in terms of scalability because the performance is stable with the network growth.

As part of the future works, we want to study and integrate (possibly existing) approaches to deal with the problem of matching queries with the names used in the network (i.e., the second part of the naming problem). Additionally, we want to better understand the different elements that influence the search performance in order to find and implement techniques to reduce the query times.

[10] `http://en.wikipedia.org/wiki/Gnutella`

References

1. Bazzanella, B., Chaudhry, J.A., Themis Palpanas, Stoermer, H.: Towards a General Entity Representation Model. 5th Workshop on SWAP (2008)
2. Holloway, G., Dunkerley, M.: The Math, Myth and Magic of Name Search and Matching. 5th edn. Search Software America (2004)
3. Giunchiglia, F., Zhang, R., Crispo, B.: Relbac: Relation based access control. In: Proceedings of the 2008 Fourth International Conference on Semantics, Knowledge and Grid. SKG '08, Washington, DC, USA, IEEE Computer Society (2008) 3–11
4. Bignotti, E.: Semantic name matching. Master's thesis, University of Trento (2012)
5. Hogan, A., Zimmermann, A., Umbrich, J., Polleres, A., Decker, S.: Scalable and distributed methods for entity matching, consolidation and disambiguation over linked data corpora. JWS: Science, Services and Agents on the World Wide Web **10** (2012)
6. Bouquet, P., Stoermer, H., Niederee, C., Maña, A.: Entity name system: The backbone of an open and scalable web of data. In: Proceedings of the 2nd IEEE ICSC, Washington, DC, USA, IEEE Computer Society (2008) 554–561
7. Rhea, S., Chun, B.G., Kubiatowicz, J., Shenker, S.: Fixing the embarrassing slowness of opendht on planetlab. In: Proc. of the 2nd conference on Real, Large Distributed Systems. WORLDS'05, Berkeley, CA, USA (2005) 25–30
8. Bazzanella, B., Stoermer, H., Bouquet, P.: Searching for individual entities: a query analysis. Technical report, University of Trento (2009)
9. Paşca, M.: Weakly-supervised discovery of named entities using web search queries. In: Proceedings of the sixteenth ACM conference on CIKM '07, New York, NY, USA, ACM (2007) 683–690
10. Cheng, T., Chang, K.C.C.: Entity search engine: Towards agile best-effort information integration over the web. In: CIDR 2007. (2007) 108–113
11. Hu, G., Liu, J., Li, H., Cao, Y., Nie, J.Y., Gao, J.: A supervised learning approach to entity search. In: AIRS'06. Volume 4182 of LNCS. (2006) 54–66
12. Hogan, A., Harth, A., Umbrich, J., Kinsella, S., Polleres, A., Decker, S.: Searching and browsing linked data with swse: The semantic web search engine. JWS: Science, Services and Agents on the World Wide Web **9**(4) (2011) 365 – 401
13. Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., Tummarello, G.: Sindice. com: a document-oriented lookup index for open linked data. International Journal of Metadata, Semantics and Ontologies **3**(1) (2008) 37–52
14. Risson, J., Moors, T.: Survey of research towards robust peer-to-peer networks: Search methods. Computer Networks **50** (2006) 3485–3521
15. Lua, E.K., Crowcroft, J., Pias, M., Sharma, R., Lim, S.: A survey and comparison of peer-to-peer overlay network schemes. IEEE Communications Surveys and Tutorials **7** (2005) 72–93
16. Bawa, M., Manku, G., Raghavan, P.: Sets: Search enhanced by topic segmentation. In: Proceedings of ACM SIGIR Conference. (2003) 306–313
17. Cohen, E., Kaplan, H., Fiat, A.: Associative search in peer to peer networks: Harnessing latent semantics. In: Proceedings of IEEE INFOCOM. (2003)
18. Spripanidkulchai, K., Maggs, B., Zhang, H.: Efficient content location using interest-based locality in peer-to-peer systems. In: Proceedings of IEEE INFOCOM. Volume 3. (2003) 2166–2176
19. Crespo, A., Garcia-Molina, H.: Semantic overlay networks for p2p systems. Technical report, Stanford University (2002)

20. Joseph, S.: Neurogrid: Semantically routing queries in peer-to-peer networks. In: Proc. Intl. Workshop on Peer-to-Peer Computing. (2002) 202–214
21. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: Proc. of SIGCOMM'01, NY, USA, ACM (2001) 161–172
22. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proc. of SIGCOMM'01, NY, USA, ACM (2001) 149–160
23. Druschel, P., Rowstron, A.: Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proc.of ACM SIGCOM. (2001)
24. Zhao, B., Huang, L., Stribling, J., Rhea, S., a.D. Joseph, Kubiatowicz, J.: Tapestry: A Resilient Global-Scale Overlay for Service Deployment. IEEE Journal on Selected Areas in Communications **22**(1) (January 2004) 41–53
25. Li, J., Thau, B., Joseph, L., Hellerstein, M., Kaashoek, M.F.: On the feasibility of peer-to-peer web indexing and search. In: IPTPS'03. (2003)
26. Ganesan, P., Gummadi, K., Garcia-Molina, H.: Canon in g major: designing dhts with hierarchical structure. In: ICDCS'04. (2004) 263 – 272
27. Janakiram, D., Giunchiglia, F., Haridas, H., Kharkevich, U.: Two-layered architecture for peer-to-peer concept search. In: 4th Int. Sem Search Workshop. (2011)
28. Papapetrou, O., Siberski, W., Nejdl, W.: Pcir: Combining dhts and peer clusters for efficient full-text p2p indexing. Computer Networks **54**(12) (2010) 2019–2040
29. Garcés-Erice, L., Biersack, E.W., Felber, P., Ross, K.W., Urvoy-Keller, G.: Hierarchical peer-to-peer systems. In: Euro-Par. (2003) 1230–1239