FLUID: A Common Model for Semantic Structural Graph Summaries Based on Equivalence Relations

Till Blume[a,b], David Richerby[c], Ansgar Scherp[c,d]

[a]Kiel University, Germany
[b]Ernst & Young GmbH WPG – GSA R&D, Germany
[c]University of Essex, UK
[d]Ulm University, Germany

arXiv:1908.01528v3 [cs.DB] 4 Jan 2021

Abstract

Summarization is a widespread method for handling very large graphs. The task of structural graph summarization is to compute a concise but meaningful synopsis of the key structural information of a graph. As summaries may be used for many different purposes, there is no single concept or model of graph summaries. We have studied existing structural graph summaries for large-scale (semantic) graphs. Despite their different concepts and purposes, we found commonalities in the graph structures they capture. We use these commonalities to provide for the first time a formally defined common model, FLUID (FLexible graph sUmmarIes for Data graphs), that allows us to flexibly define structural graph summaries. FLUID allows graph summaries to be quickly defined, adapted, and compared for different purposes and datasets. To this end, FLUID provides features of structural summarization based on equivalence relations such as distinction of types and properties, direction of edges, bisimulation, and inference. We conduct a detailed complexity analysis of the features provided by FLUID. We show that graph summaries defined with FLUID can be computed in the worst case in time $\mathcal{O}(n^2)$ w.r.t. n, the number of edges in the data graph. An empirical analysis of large-scale web graphs with billions of edges indicates a typical running time of $\Theta(n)$. Based on the formal FLUID model, one can quickly define and modify various structural graph summaries from the literature and beyond.

Keywords: Structural graph summary, Semantic graphs, Parameterized formal model

1. Introduction

Representing data as a graph is increasingly popular, since graphs allow a more efficient and a more flexible implementation of certain applications compared to traditional relational databases [23]. In general, such data graphs contain labeled vertices and edges, which describe the data and their relationships. However, in huge, heterogeneous data graphs like the Linked Open Data cloud,[1] several tasks are computationally expensive, such as cardinality computations for queries [33], data exploration [2, 31, 35, 41], data visualization [18], vocabulary term recommendations [39], and related entity retrieval [11]. Graph summarization facilitates the identification of meaning and structure in data graphs [29]. Thus, graph summaries can be used to tackle the above tasks more efficiently, e. g., by serving as indices.

Different attempts have been made to classify the different graph summarization approaches [6, 10, 26, 29]. In this work, we focus on structural graph summaries, i. e., graph summaries that precisely capture specific structural features of the data graph [10]. Structural graph summaries are condensed representations of graphs such that a set of chosen (structural) features of the graph summary are equivalent to the original graph. To achieve this, structural graph summaries partition vertices based on equivalent subgraphs. To determine subgraph equivalences, only structural features are used, such as specific combinations of labels.

[1]https://lod-cloud.net/, last accessed: October 16, 2020.

Accepted author manuscript to appear in Theoretical Computer Science

Structural graph summaries are usually an order of magnitude smaller than the input graph but are equivalent to the original graph regarding the chosen structural features [4]. Structural graph summaries do not include statistical approaches such as sampling or pattern-mining approaches [10].

Semantic structural graph summaries extend structural graph summaries by supporting concepts from ontologies, such as semantic labels for vertices and edges [10, 27, 41]. Ontologies model hierarchical relationships between concepts, i. e., relationships between types and properties with which the vertices and edges are labeled, e. g., the type `Proceedings` is a subtype of the type `Book`. Following this subtype relation allows us to infer the type `Book` for all vertices that are labeled with the type `Proceedings`.

Many different structural graph summaries have been developed for different purposes, capturing different structural features of graphs [2, 8, 11, 12, 18, 25, 27, 31–33, 39–41, 43]. The problem with this plethora of existing summary models is that each model defines its own data structure that is designed for solving only a specific task [2, 11, 19, 31, 33, 35, 39, 41]. Our observation is that the different tasks cannot be sufficiently supported by any of the existing structural graph summaries [4]. Furthermore, it is difficult to compare structural graph summaries, as they were designed, implemented, and evaluated in isolation, for their individual task only and using different queries, datasets, graph models, and metrics.

However, when analyzing the fundamental concepts underlying the various existing (semantic) structural graph summaries, one can identify a common theoretical grounding. Either explicitly, or often only implicitly, the existing graph summary models are defined using equivalence relations [2, 8, 11, 12, 18, 25, 27, 31–33, 39–41, 43]. This paper summarizes our extensive, in-depth analysis of the existing (semantic) structural graph summaries. From this analysis, we have identified common features in existing structural graph summaries. We model these features as equivalence relations, which can be flexibly combined to form new equivalence relations, and by this be tailored for each individual purpose. The result is a small set of operators that make up our common model FLUID (short for FLexible graph sUmmarIes for Data graphs), which defines a language for flexibly defining semantic structural graph summaries. Since only a few operators are needed to define these equivalence relations, we are able to produce a single, parameterized algorithm that computes all structural graph summaries defined with FLUID. Any graph summary defined with FLUID can be computed in worst case in time $\mathcal{O}(n^2)$ w.r.t. n, the number of edges in the data graph. Analyzing large-scale graphs on the web with billions of edges shows that, empirically, their computation time is in $\Theta(n)$.

In summary, the contributions of this work are:

(I) A formally defined language FLUID for flexibly defining and adapting structural graph summaries and semantic structural graph summaries.

(II) A parameterized, sequential algorithm that computes all graph summaries defined with FLUID.

(III) The worst case time complexity of computing summaries with FLUID is in $\mathcal{O}(n^2)$ w.r.t. n, the number of edges in the input graph. An empirical analysis of large-scale web graphs with billions of edges indicates a typical computation time of $\Theta(n)$.

By introducing FLUID, we lay the foundation for a new generation of flexibly defined and easy to use semantic structural graph summaries. Our approach scales for complex tasks and large-scale semantic graphs with billions of edges. Furthermore, the implementation of our graph summarization framework is freely available.[2]

This paper is an extension of our previous workshop paper [3]. The previous work defines FLUID's schema elements and five parameterizations and conducts a first complexity analysis. In this work, we extend the functionality of FLUID by introducing the new set parameterization and the related property instance parameter for the instance parameterization. Additionally, we formally introduce the notion of payload to FLUID, which is needed to adapt graph summaries to different application tasks. Furthermore, we provide an extended literature review including a detailed description of the features provided by related works. We also provide explanations and examples for all schema elements and their parameterizations.

[2] https://github.com/t-blume/fluid-framework

We also present our single, parameterized algorithm and provide an extensive and more precise complexity analysis of our algorithm. Finally, we also included experiments to estimate the impact of the inference parameterization in typical scenarios.

The remainder of this paper is structured as follows. In the next section, we describe our problem statement in detail. We analyze existing (semantic) structural graph summaries in Section 3. In Section 4, we formally define the FLUID language and show how to define structural graph summaries. We present our parameterized algorithm to compute structural graph summaries defined with FLUID and analyze its complexity in Section 5. Finally, in Section 6, we analyze four large-scale, real-world datasets with billions of edges to substantiate that the typical computational complexity is $\Theta(n)$, before we conclude in Section 7.

2. Problem Statement

Structural graph summarization is the task of finding a condensed representation SG (short for "summary graph") of an input graph G such that a set of chosen (structural) features are equivalent in SG and the original graph [10]. Intuitively, structural graph summarization means that we can conduct specific tasks – e. g., counting the vertices with a specific type label – directly on G or, alternatively, obtain the same information from the structural graph summary SG. The fundamental idea of structural graph summaries is that the task can be completed much faster on the graph summary than on the original graph. In this paper, without loss of generality, we consider Resource Description Framework (RDF) graphs as input graphs. We chose RDF graphs due to their popularity and because they are standardized by the World Wide Web Consortium (W3C) [13].[3] Below, we formally introduce the RDF graph model. Subsequently, we introduce the notion of semantic structural graph summaries.

2.1. RDF Graphs

Conceptually, an RDF graph is an edge-labelled directed graph. An RDF graph [13] is a set of triples (s, p, o), with a subject s, predicate p, and object o. Each triple denotes a directed edge from the subject vertex s to the object vertex o, the edge being labeled with the predicate p. RDF graphs distinguish three kinds of vertices, namely International Resource Identifiers (IRIs) [15], blank nodes, and literals, each of which has a different role.

Formally, an RDF graph is defined as $G \subseteq (V_I \cup V_B) \times P \times (V_I \cup V_B \cup L)$, where V_I denotes the set of IRIs, V_B the set of blank nodes, $P \subseteq V_I$ the set of predicates (also identified by IRIs), and L the set of literals (represented by strings) [13]. IRIs conceptually correspond to real-world entities and are globally unique: an IRI may be included in more than one RDF graph, but this corresponds to stating different facts about the same real-world entity. In contrast, blank nodes are only locally defined, within the scope of a specific RDF graph, to serve special data modeling tasks. Through skolemization, blank nodes can be turned into Skolem IRIs, which are globally unique [13, Section 3.5]. Literal vertices are finite strings of characters from a finite alphabet such as Unicode [13]. Thus, two literal vertices which are term-equal (i. e., are the same string) [13, Section 3.3] are the same vertex. Although IRIs, blank nodes, and literals have different roles in RDF, this distinction is not relevant in this work and we treat them equally.

Predicates $p \in P$ act as edge labels. The RDF standard includes the predicate `rdf:type`, which is used to simulate vertex labels: the triple $(s, \texttt{rdf:type}, o)$ denotes the vertex s having label o. Such vertex labels are called RDF types. This indirect representation of vertex labels is a design decision of RDF and contrasts with the direct use of vertex labels in labeled property graphs [23]. Edges in an RDF graph which are not labeled with `rdf:type` are called RDF properties.

We define the set $\ell_T(s) := \{o \in V_I \mid (s, \texttt{rdf:type}, o) \in G\}$ as the **type set** of a vertex s in an RDF graph [20]. We define the set $\ell_P(s) := \{p \in P \mid (s, p, o) \in G \text{ for some } o \text{ and } p \neq \texttt{rdf:type}\}$ as the **property set** of a vertex s in an RDF graph [20]. We write V_C for the set of all RDF types, i. e., the set

[3] The Linked Open Data cloud, available at https://lod-cloud.net/, is a popular depiction of the use of RDF graphs on the web since May 2007. The current depiction of the cloud from July 2020 formally registers 1,260 datasets stemming from different organizations and domains.

$\{o \in V_I \mid (s, \texttt{rdf:type}, o) \in G$ for some $s\}$. Furthermore, we write $\Gamma^+(s) = \{o \mid (s,p,o) \in G$ for some $p\}$ for the set of outgoing neighbors of s in the RDF graph G and $\Gamma^-(o) = \{s \mid (s,p,o) \in G$ for some $p\}$ for the set of o's incoming neighbors. Finally, we define $\ell_P^-(o) := \{p \mid (s,p,o) \in G$ for some s and $p \neq \texttt{rdf:type}\}$ as the **incoming property set**.

An example RDF graph is shown in Figure 1. On the left-hand side, the RDF graph is shown as set of triples. On the right-hand side, it is depicted as a graph. The vertex v_1 has the type set $\ell_T(v_1) = \{\texttt{Proceedings}\}$ and the property set $\ell_P(v_1) = \{\texttt{author, title}\}$. The vertex v_2 has the type set $\ell_T(v_2) = \{\texttt{Person}\}$ and the property set $\ell_P(v_2) = \{\texttt{name}\}$. But, v_1 has predicates $\{\texttt{author, title, rdf:type}\}$ and outgoing neighbors $\Gamma^+(v_1) = \{\texttt{Proceedings}, v_2$, "Graph Database"$\}$.

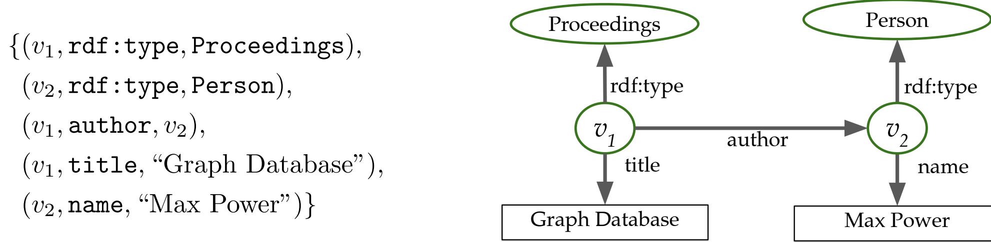


Figure 1: Simple RDF graph to demonstrate the relation between a set of triples (left) and its visualization as a graph (right).

A special characteristic of RDF graphs is their support for semantic labels, which allows the inference of implicit information. Such semantic labels are from ontologies, where semantic relationships between types and properties are denoted, e. g., with predicates from the RDF Schema vocabulary. RDF Schema (RDFS) and its entailment rules are standardized by the W3C [7]. A comprehensive overview of these rules is presented in [1]. For example, the semantics of a triple $(p, \texttt{rdfs:subPropertyOf}, p') \in G$ is that for any subject vertex s with $(s,p,o) \in G$, we can infer existence of the additional triple (s, p', o). This means, when using RDF Schema inference, each vertex may have more types and properties in its type set and property set, respectively.

Finally, RDF supports the definition of named graphs [13]. Following the W3C standard [13], each named graph is a pair consisting of the graph name, which is an IRI, and an RDF graph as defined above.Following Harth et al. [21], we formalize named graphs by extending each triple (s,p,o) to a tuple $((s,p,o),d)$. The context of a tuple denotes the name of the data source from which the triple originated on the web [45]. In this work, the context d of a tuple is only used as payload, and not to determine the equivalence of vertices.

2.2. Semantic Structural Graph Summarization

To compute a (semantic) structural **graph summary** SG for a given data graph G, we partition the data graph into disjoint sets of vertices. In the case of semantic structural graph summaries, we partition the vertices based on equivalent subgraphs. In contrast to classic subgraph matching, we match only selected parts of the subgraphs, e. g., match the type set and the property set of each vertex. We call the respective subgraphs containing the information necessary to determine the equivalence of two vertices the **schema structure** of the vertices. Which **features** of the input graph are considered to determine equivalent schema structures is defined by the **graph summary model**. For different tasks, different features of the summarized vertices are of interest, e. g., the number of summarized vertices for cardinality computation or the data source (context d) for data search. This information about the summarized vertices is called the **payload**. Equivalence relations describe any graph partitioning in a formal way. Thus, for FLUID, we define graph summarization via equivalence relations over vertices.

Definition 1. An **equivalence relation** on a set X (e. g., a set of graph vertices) is a subset $\text{EQR} \subseteq X \times X$ that is reflexive, symmetric, and transitive. When $(x, y) \in \text{EQR}$, we say that x is equivalent to y and write $x \sim y$. For any $y \in X$, the set $\{x \in X \mid x \sim y\}$ is called the **equivalence class of** y, typically denoted by $[y]_{\text{EQR}}$. We denote by "$\top$" the **tautology equivalence relation** $X \times X$, in which all elements of X are equivalent. We denote by "id" the **identity equivalence relation** $\{(x, x) \mid x \in X\}$, in which no two distinct elements of X are equivalent.

Remark 1. For a given equivalence relation, any two equivalence classes either are disjoint or equal, so equivalence relations on a set X correspond precisely to partitions (decompositions) of X [16]. Furthermore, the intersection of two equivalence relations over X is also an equivalence relation.

While the intersection of equivalence relations results in an equivalence relation, the same does not hold true for the union. For example, consider the equivalence relations $\sim_A = \{(1, 1), (2, 2), (1, 2), (2, 1), (3, 3)\}$ and $\sim_B = \{(1, 1), (2, 2), (3, 3), (2, 3), (3, 2)\}$. The union of $\sim_A$ and $\sim_B$ is not transitive, as it contains $(1, 2)$ and $(2, 3)$ but not $(1, 3)$. Thus, to combine equivalence relations in an "or-like" fashion to give a new equivalence relation, we define an extended union operator $\cup_{\text{ex}}$.

Definition 2. Let $\sim_A$ and $\sim_B$ be equivalence relations over the same set. We define the **extended union** $\sim_A \cup_{\text{ex}} \sim_B$ to be the transitive closure of $\sim_A \cup \sim_B$. That is, $\sim_A \cup_{\text{ex}} \sim_B$ is the unique equivalence relation $\sim_{\text{ex}}$ such that $\sim_A \cup \sim_B \subseteq \sim_{\text{ex}}$ and for every equivalence relation $\sim'$ such that $\sim_A \cup \sim_B \subseteq \sim'$, we have $\sim_{ex} \subseteq \sim'$.

For FLUID, we define equivalence relations over the set of vertices appearing in the graph. RDF graphs are defined as sets of triples (s, p, o). However, since each triple has exactly one subject vertex s, an equivalence relation EQR_v over vertices induces an equivalence relation EQR_t over the triples, given by $((s, p, o), (s', p', o')) \in \text{EQR}_t$ iff $(s, s') \in \text{EQR}_v$. Any partitioning of G's vertices into disjoint subsets can be expressed as an equivalence relation over vertices in the data graph G. Thus, we can describe graph summarization using an equivalence relation EQR. When two vertices are placed into the same partition, we say they are **summarized**. The **graph summary** for G with respect to EQR is a labeled graph SG. Each equivalence class $[y]_{\text{EQR}}$ is represented in SG in a so-called **vertex summary**. The vertex summary vs_y is a subgraph in SG that is equivalent to the subgraphs of all summarized vertices $v \in [y]_{\text{EQR}}$ in G under EQR. For vertex summaries, we distinguish **primary vertices**, which are equivalence classes of EQR, and **secondary vertices**, which are equivalence classes of the relations from which EQR is defined [5]. Furthermore, we can attach **payload** [19] to each primary vertex in a vertex summary $vs \subseteq SG$. The payload can be tailored for different purposes, e. g., to contain the number of summarized vertices. For FLUID, we define a set of **payload elements** PAY to implement a specific task. Payload elements map vertex summaries to payload. Formally, we can now denote a (semantic) structural graph summary model as a 3-tuple of a data graph G, an equivalence relation EQR, and a set of payload elements PAY.

Definition 3. A **structural graph summary model** is a tuple $(G, \text{EQR}, \text{PAY})$, where G is the data graph, EQR is an equivalence relation over vertices in G, and PAY is a set of payload elements.

3. Analysis of Existing Semantic Structural Graph Summary Models

Throughout the literature, different attempts have been made to classify and group the different graph summarization approaches [6, 10, 26, 29]. In this work, we focus on structural graph summaries, i. e., graph summaries that precisely capture the structure of the input graph [10]. This excludes in particular statistical approaches that generate approximate data descriptions [10]. As pointed out by Fan et al. [17], many real-life applications require exact matches. Still, there exists a large variety of exact (semantic) structural graph summaries [10].

In the following, we analyze in detail existing (semantic) structural graph summaries with respect to the captured schema structure, i. e., what features of the input graph are used to summarize vertices. Existing surveys about graph summaries cover a wider range of approaches and, thus, lack this level of

Table 1: Structural graph summary models found in the literature and what features they use (X) and do not use (-) to capture the schema structure of vertices. The features are grouped by features that use triple information only (triple features), features that define how features of multiple vertices are combined (subgraph features), and features that define explicit semantic rules (semantic rule features).

		Triple features				*Subgraph features*				*Semantic rule features*			
	Graph summary \ **Feature**	**Property sets**	**Type sets**	**Label sets**	**Neighbor vertex ID**	**Neighbor triple**	**Predicate path**	**(k-)bisimulation**	**Incoming property sets**	**OR combination**	**Related properties**	**RDF Schema**	**OWL SameAs**
Simple	Attribute-based Collection [8]	X	-	-	-	-	-	-	-	-	-	-	-
	Class-based Collection [8]	-	X	-	-	-	-	-	-	-	-	-	-
	Characteristic Sets [33]	X	-	-	-	-	-	-	X	-	-	-	-
	SemSets [11]	-	-	-	X	-	X	-	-	-	-	-	-
Complex	SchemEX [27]	X	X	-	-	X	X	-	-	-	-	-	-
	SchemEX+U+I [4]	X	X	-	-	X	X	-	-	-	-	X	X
	ABSTAT [41]	X	X	-	-	X	X	-	-	-	-	(X)	-
	LODex [2]	X	X	-	-	X	X	-	-	-	-	-	-
	Loupe [31]	X	X	-	-	X	X	-	-	-	-	-	-
	TermPicker [39]	X	X	-	-	X	-	-	-	-	-	-	-
	Weak Summary [18]	X	-	-	-	X	X	X	X	X	X	X	-
	Strong Summary [18]	X	-	-	-	X	X	X	X	-	X	X	-
	Typed Weak Summary [18]	X	X	-	-	X	X	X	X	X	X	X	-
	Typed Strong Summary [18]	X	X	-	-	X	X	X	X	-	X	X	-
	Tran et al. [43]	X	-	X	-	X	X	X	-	-	-	-	-
	A(k)-index [25]	X	-	-	-	X	X	X	-	-	-	-	-
	T-index [32]	-	-	-	-	X	X	X	X	-	-	-	-
	Consens et al. [12]	X	X	-	-	X	X	X	-	-	-	-	-
	Schätzle et al. [40]	X	-	-	X	-	X	X	-	-	-	-	-

granularity [6, 10, 26, 29]. Table 1 shows a cross-table of each analyzed structural graph summary model and its features. In total, we analyzed 19 graph summary models and identified 12 different features. We organize the structural graph summary models and the features into different groups. We distinguish features that only use triple information (triple features), features that define how features of multiple vertices are combined (subgraph features), and features that define explicit semantic rules such joining and inference (semantic rule features). Each group of features adds another level of complexity, i. e., intuitively, the computational complexity grows when features of different groups are used. As one can see, there is no single graph summary model that supports all features. However, we see common combinations of features. In the following, we discuss the graph summary models along the identified features shown in Table 1 from left to right.

3.1. Triple Features

Triple features are solely based on outgoing triples of vertices. To compute the equivalence of two vertices s and s', we only compare triples where the subject is s or s'.

Property sets. The most commonly used feature in structural graph summaries is using properties to compute the schema of vertices. More specifically, for each vertex s in the data graph the property set

$\ell_P(s)$ is compared. Campinas et al. [8] proposed so-called “Attribute-based Collections”, a graph summary that relies solely on property sets to compute the schema structure of vertices. If two vertices s, s' share the same property set, i. e., $\ell_P(s) = \ell_P(s')$, they are considered equivalent, thus, are summarized together.

Type sets. Another commonly used feature is using the vertices' types to compute the schema. Here, for each vertex s in the data graph, the type set $\ell_T(s)$ is compared. If two vertices s, s' share the same type set, i. e., $\ell_T(s) = \ell_T(s')$, they are considered equivalent. Campinas et al. [8] proposed a second graph summary, called “Class-based Collections”, which uses only the vertices' type sets to compute the schema. Both graph summaries, Attribute-based Collections and Class-based Collections, were developed to enhance SPARQL query formulations by providing recommendations [8].

Label sets. Tran et al. [43] proposed the feature of label parameterization for graph summaries. With the label parameterization, only a subset of all edge labels are used to compute the schema. More precisely, one defines a set of predicates P_l, the so-called label set, which are ignored when determining the equivalence of vertices. Tran et al.'s graph summary combines property sets $\ell_P(s)$ with label sets. Furthermore, they combine this with k-bisimulation (see below).

Neighbor vertex ID. The final triple feature is using the identity of outgoing neighbors $\Gamma^+(s)$ to determine the equivalence of vertices. It appears that no existing graph summary summarizes vertices solely by comparing the neighbor identities. However, SemSets [11] summarize vertices that share the same outgoing predicates, which are linked to the same vertices. To check if two vertices s, s' are equivalent under SemSets, all triples where s or s' are the subject vertices are compared. For each triple $(s, p, o) \in G$ there has to be a triple $(s', p, o) \in G$, and vice versa. Thus, they combine neighbor vertex identifiers $\Gamma^+(s)$ with predicate paths (see below). SemSets were developed to discover semantically similar sets of vertices in graphs and to use these vertex sets to improve keyword-based ad-hoc retrieval.

3.2. Subgraph Features

The neighbor vertex identifier is the most direct approach to incorporate neighbor information that leads to a wider range of summary models that consider neighbor information, e. g., vertices in $\Gamma^+(s)$. We classify features as subgraph features when they combine triple features of multiple vertices.

Neighbor triple. SchemEX [27], SchemEX+U+I [4], ABSTAT [41], LODeX [2], and Loupe [31] summarize vertices s and s' based on a common type set and common properties linking to vertices with the same type sets. This means, in order to compute the schema of one vertex s, also the type sets of outgoing neighbors $\Gamma^+(s)$ are required to be equivalent, i. e., we compare neighbor triples. In contrast to SemSets [11], these approaches do not use the neighbor vertex identifiers $\Gamma^+(s)$ but, instead, use the type set $\ell_T(o)$ for each $o \in \Gamma^+(s)$. SchemEX [27], SchemEX+U+I [4], ABSTAT [41], LODeX [2], and Loupe [31] combine type sets $\ell_T(s)$, property sets $\ell_P(s)$, and neighbor type sets $\ell_T(o)$ using predicate paths (see below). The summary models were developed for data source search and exploration.

Predicate path. As indicated in the previous two features, almost all analyzed graph summaries that use neighbor information combine the schema structures using predicate paths, i. e., they compare which predicates link to which neighbors. A predicate path compares via which path of predicates a vertex is connected to neighboring vertices' schema structures. For example, SchemEX [27], SchemEX+U+I [4], ABSTAT [41], LODeX [2], and Loupe [31] consider which property links to which type set. TermPicker [39] follows a different strategy to integrate the schema of neighboring vertices. TermPicker summarizes vertices s based on having the same type set $\ell_T(s)$, the same property set $\ell_P(s)$, and the same type set $\ell_T(o)$ of all $o \in \Gamma^+(s)$. Consequently, TermPicker's graph summaries compress all type sets of all neighbors into a single type set. Thus, TermPicker's graph summaries do not contain information about which *specific* property links to which neighbor.

(k-)bisimulation. Many graph summaries compute the schema of vertices by taking into account the schema of neighbors over multiple hops [25, 27, 36, 43]. This is commonly defined as a bisimulation. Bisimulation operates on state transition systems and defines an equivalence relation over states [38]. Two states are equivalent (or bisimilar) if they change into equivalent states with the same type of transition. Interpreting a labeled graph as a representation of a state transition system allows us to apply bisimulation on graph data to discover structurally equivalent parts.

In practice, graph summaries usually define a stratified k-bisimulation [25, 27, 36, 43]. A stratified bisimulation restricts the maximum path length to k edges in the connected subgraph. This increases the chance that two vertices are considered equivalent. An efficient k-bisimulation algorithm was proposed by Kaushik et al. [25] but there are also efficient implementations specialized to their corresponding graph summary, e. g., by Konrath et al. [27] and Goasdoué et al. [18]. Tran et al. [43] propose, in addition to the label parameterization feature described above, also the height parameterization feature, formulating k-bisimulation as feature for graph summaries.

Some graph summaries combine the feature of using only incoming or only outgoing properties with the k-bisimulation feature [12, 32, 40]. This is referred to as backward k-bisimulation and forward k-bisimulation, respectively [18]. Milo and Suciu [32] developed the so-called T-index to support path queries in semi-structured databases. This summarizes vertices s based on the common set of incoming property-paths, i. e., they use k-bisimulation only on incoming property sets $\ell_P^-(s)$. Consens et al. [12] propose a structural graph summary model to support navigational SPARQL queries, so-called Extended Property Paths (EPPs). They summarize vertices s based on the common set of outgoing property-paths, i. e., they use k-bisimulation only on outgoing property sets $\ell_P(s)$. In addition, for each hop, the type sets $\ell_T(s)$ have to be equivalent. Schätzle et al. [40] developed a similar approach like Consens et al. [12]. The difference is that Schätzle et al. do not consider type sets, but object equivalences. Furthermore, they do not distinguish between types and properties. Thus, they use all predicates. This means, for each hop the same vertex is connected over the same predicate (neighbor vertex ID feature). For $k = 1$, Schätzle et al.'s summary model is equivalent to the SemSets [11] model.

Incoming property sets. Incoming property sets are frequently used in combination with k-bisimulation. To the best of our knowledge, there is no graph summary model that summarizes solely based on incoming property sets $\ell_P^-(s)$. But, Characteristic Sets [33] summarize two vertices s, s' that have the same outgoing property sets $\ell_P(s) = \ell_P(s')$ and the same incoming property sets $\ell_P^-(s) = \ell_P^-(s')$. Characteristic Sets were designed for cardinality estimations of queries in RDF databases. Analogously, Goasdoué et al. [18] define the Strong Summary. The Strong Summary summarizes vertices s and s' if they have the same property sets $\ell_P(s) = \ell_P(s')$ and the same incoming property set $\ell_P^-(s) = \ell_P^-(s')$. Furthermore, they propose the Typed Strong Summary, which summarizes vertices s and s' based on two conditions: (1) if they have empty type sets $\ell_T(s) = \ell_T(s') = \emptyset$ and they have the same property sets $\ell_P(s) = \ell_P(s')$ and the same incoming property set $\ell_P^-(s) = \ell_P^-(s')$ or (2) if they have the same non-empty type sets $\ell_T(s) = \ell_T(s') \neq \emptyset$ [18]. Both the Strong Summary and the Typed Strong Summary also allow k-bisimulation and semantic rule features such as taking so-called "related properties" into account and exploit RDF Schema inferencing (see below).

3.3. Semantic Rule Features in Graph Summaries

The last group includes features that define explicit (semantic) rules.

OR combination. Goasdoué et al. [18] define the Weak Summary using OR combination (see extended union $\cup_{\text{ex}}$, Definition 2). In the Weak Summary, two vertices s and s' are equivalent if they have the same property set $\ell_P(s) = \ell_P(s')$ or the same incoming property set $\ell_P^-(s) = \ell_P^-(s')$ (or both). Analogously to the Typed Strong Summary, they define the Typed Weak Summary, which relaxes the first condition of the equivalence of the property set and the incoming property set using the OR combination of the Weak Summary [18].

Related properties. Goasdoué et al. [18] also propose to include property relations. Two properties p and p' are source-related if they co-occur in any property set $\ell_P(s)$ of any vertex s and they are target-related if they co-occur in any incoming property set $\ell_P^-(s)$ of any vertex s. They developed this feature to generate comprehensible graph visualizations.

RDF Schema. Several semantic structural graph summaries use RDF Schema inferencing to enhance their summaries. ABSTAT [41] exploits RDF Schema type hierarchies to compute so-called minimal patterns. They select the minimal number of types, i. e., they only keep the most specific types from the RDF Schema type hierarchy. Goasdoué et al. [18] exploit RDF Schema type hierarchies, property hierarchies, and RDF Schema domain and RDF Schema range. With domain and range, types for the subject vertex and the object vertex can be inferred. They also propose a so-called shortcut for inferencing, which improved the time needed to perform the inferencing in graph summaries by up to 94% [18].

OWL SameAs. SchemEX+U+I [4] also uses the full RDF Schema inferencing but also exploits the semantics of the `owl:sameAs` property. This property is part of W3C's Web Ontology Language (OWL) [30], which is heavily used in the context of RDF graphs. The `owl:sameAs` property defines an equivalence relation [30, Section 4.2], intended to identify vertices that represent the same real-world entity. To compute the schema structure of one vertex v, the schema structures of all vertices v' in the weakly connected components in an `owl:sameAs`-labeled subgraph of G are merged (see Ding et al. [14] for details on `owl:sameAs` networks). SchemEX+U+I was developed for a data search task.

3.4. Summary

There exists a large variety of structural graph summary models that use different combinations of features. However, none of the existing approaches covers all features. In the past, the idea of a single, generalizable framework to compute structural graph summaries has been discussed a few times. To the best of our knowledge, the idea of using equivalence relations to define structural graph summaries, i. e., to partition vertices of a graph, has been proposed before [9, 36, 43]. In particular, the concept of bisimulation has been used to define a single, adaptive framework for graph summaries based on equivalence relations [36, 43]. However, none of the existing approaches define a language to define these equivalence relations, which also allows a single, parameterized algorithm to compute them. The proposed features and the corresponding algorithms only implement a subset of features, as shown in Table 1. Thus, a language for semantic structured graph summarizes is needed that incorporates all of the different features and that allows us to flexibly define (semantic) structural graph summaries. In the next section, we define our flexible language called FLUID and demonstrate how it covers all features found in the literature.

4. Definition of FLUID

We formally define the elements and parameterizations of FLUID. An overview is given in Table 2. First, we define three different simple schema elements in Section 4.1. Based on simple schema elements, we define complex schema elements. Second, we define in Section 4.2 six parameterizations to further specialize the schema elements. Third, we define three payload elements in Section 4.3. Finally, in Section 4.4, we show how all graph summary models analyzed in Section 3 are defined with FLUID.

4.1. Schema Elements

We distinguish simple schema elements and complex schema elements. Simple schema elements enable triple-based vertex summarization (compare triple features in Table 1). Intuitively, this means they summarize vertices without considering any kind of neighbor information, i. e., they capture the "local" schema of vertices. Recall from Section 2.1 that we assume the graph to be defined using triples of subject vertex, edge label (predicate), and object vertex, i. e., (s, p, o). Local in this sense means that equivalence of two vertices s, s' depends only on the edges where s or s' is the source vertex, i. e., the subject of the triples. In contrast, complex schema elements allow us to capture the schema beyond the scope of a single vertex (compare subgraph features in Table 1). In the following, we formally define three simple schema elements as well as the complex schema element.

Table 2: Overview of FLUID's schema elements and parameterizations.

Schema Elements (SE)	Description	Details
Simple Schema Element (SSE)	Triple-based summarization of vertices	Definitions 4 to 6, Figures 2b to 2d
Complex Schema Element (CSE)	Summarization of vertices using combinations of Schema Elements (SEs)	Definition 7, Figure 3
Parameterizations	**Description**	**Details**
Label parameterization $lp(SSE, P_l)$ (short: SSE_{P_l})	SSE ignores existence of predicates not in P_l	Definition 8
Set parameterization $sp(SSE, S)$ (short: $SSE\|_S$)	Application of SSE is restricted to vertices containing only specified predicates and/or objects	Definition 9
Chaining parameterization $cp(CSE, k)$ (short: CSE^k)	Recursively apply CSE for each connected vertex up to k hops	Definition 10
Direction parameter $dp(SE, \delta)$ (short: δ-SE)	Summarize vertices following the direction parameter $\delta = \{\mathrm{o}, \mathrm{i}, \mathrm{b}\}$ based on outgoing predicates (o), incoming predicates (i), or outgoing and incoming predicates (b)	Definition 11
Inference parameterization $op(G, VG)$ (short: G_{VG})	Include ontology reasoning by inferring additional triples from a vocabulary graph VG, e. g., using VG_{RDFS}	Definition 13
Instance parameter $ip(SE, \Delta)$ (short: $SE[\Delta]$)	Unions of explicitly equivalent vertices	Definition 14

4.1.1. Simple Schema Elements

Simple schema elements summarize vertices s, s' by comparing all triples $(s, p, o) \in G$ and all triples $(s', p', o') \in G$. We define three simple schema elements: predicate-object cluster (POC), predicate cluster (PC), and object cluster (OC). Each simple schema element compares vertices following a different strategy, i. e., compare only the predicates, only the objects, or both (see analysis of triple features in Section 3.1). Figure 2 gives an example RDF graph and the corresponding vertex summaries using simple schema elements.

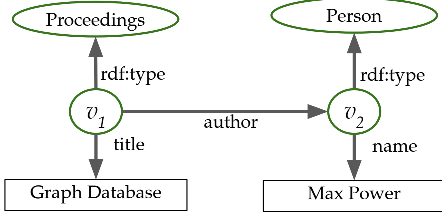
Definition 4. The **Predicate–Object Cluster** (POC) partitions the data graph by comparing the triples based on common predicates linking to common objects. For two vertices s, s', the equivalence relation POC holds true, iff for each triple $(s, p, o) \in G$ there is a triple $(s', p, o) \in G$, and vice versa.

Figure 2b shows two vertex summaries that summarize v_1 and v_2, respectively, following the predicate-object cluster POC equivalence.

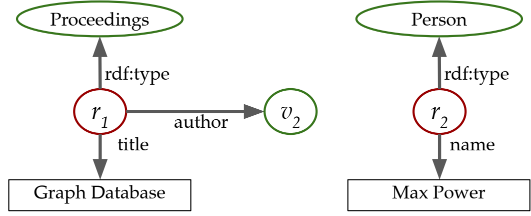
Definition 5. The **Predicate Cluster** (PC) partitions the data graph by common predicates of vertices: $(s, s') \in \mathrm{PC}$ iff, for each triple $(s, p, o) \in G$, there is a triple $(s', p, o') \in G$ for some o', and vice versa.

Note that the predicate cluster is not equivalent to saying $\ell_P(s) = \ell_P(s')$ due to the special treatment of the RDF predicate `rdf:type` (see Section 2.1). As shown in Figure 2c, the predicate cluster includes the `rdf:type` predicate. However, the `rdf:type` predicate is explicitly excluded from property sets. Rather, `rdf:type` defines type sets (see again Section 2.1).

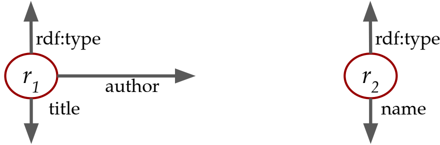
Definition 6. The **Object Cluster** (OC) partitions the data graph by common objects of vertices: $(s, s') \in \mathrm{OC}$ iff, for each triple $(s, p, o) \in G$, there is a triple $(s', p', o) \in G$ for some p', and vice versa.



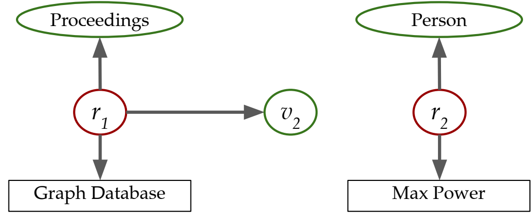
(a) Example RDF data graph as introduced in Figure 1.



(b) The two vertex summaries that summarize v_1 and v_2 according to the POC equivalence relation.



(c) The two vertex summaries that summarize v_1 and v_2 according to the PC equivalence relation.



(d) The two vertex summaries that summarize v_1 and v_2 according to the OC equivalence relation.

Figure 2: An example data graph (a) and the vertex summaries in the graph summary SG that summarize vertices v_1 and v_2 using POC (b), PC (c), and OC (d). The vertex summary identified by the primary vertex r_1 summarizes v_1 and the vertex summary identified by the primary vertex r_2 summarizes v_2. Both vertex summaries are the subgraphs starting from their primary vertex (see Section 2.2).

Note that POC $\neq$ PC $\cap$ OC. Two instances are equivalent under PC $\cap$ OC if they contain the same predicates and the same objects, whereas POC requires the same predicate–object pairs. For example, consider the novel *David Copperfield* by Charles Dickens and a hypothetical biography *Charles Dickens* by the illusionist David Copperfield. These two books are equivalent under PC $\cap$ OC because each has properties “author” and “title” and each has objects “David Copperfield” and “Charles Dickens”. However, they are not equivalent under POC because, e. g., one has “Charles Dickens” as author and the other does not. This acknowledges the predicate path feature in Table 1.

4.1.2. Complex Schema Elements

FLUID’s three simple schema elements summarize vertices by comparing outgoing triples. However, we also need to support schema structures that go beyond the scope of a single vertex. Thus, we define the complex schema element as a generalization of a simple schema element. The simple schema elements are combinations of equivalence relations by using the identity equivalence id on properties and/or objects. Complex schema elements extend on this and allow arbitrary equivalence relations over subjects, properties, and objects. Thus, they can be considered as templates to combine any number of (simple) schema elements. This allows, e. g., applying equivalence relations defined by simple schema elements on objects, i. e., use the local schema of neighboring vertices.

Definition 7. A **complex schema element** is a 3-tuple $CSE := (\sim^s, \sim^p, \sim^o)$, where $\sim^s$, $\sim^p$, and $\sim^o$ are equivalence relations. Two vertices s, s' are equivalent under this CSE iff, for every triple $(s, p, o) \in G$, there is a triple $(s', p', o') \in G$ such that $s \sim^s s'$, $p \sim^p p'$, and $o \sim^o o'$, and vice versa.

The following examples show how to combine equivalence relations using complex schema elements. This allows us to incorporate the schema of neighboring vertices $\Gamma^+(s)$ into the schema of a summarized vertex s.

Example 1. Let CSE-1 $= (\top, \mathrm{id}, \mathrm{id})$, where $\top$ denotes the tautological equivalence relation, in which all vertices are equivalent and id denotes the identity equivalence relation. CSE-1 considers vertices s and s' equivalent iff, for every triple $(s, p, o) \in G$, there is a triple $(s', p', o') \in G$ such that $p' = p$ and $o' = o$. In

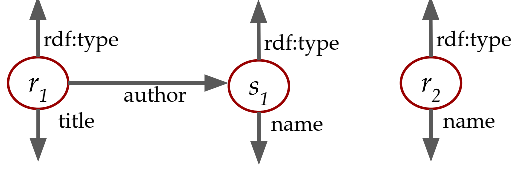


Figure 3: Following the equivalence relation CSE-2 defined in Example 2, the two vertex summaries identified by the primary vertices r_1 and r_2 summarize the vertices v_1 and v_2 from Figure 2a, respectively. To represent the complex schema structure of v_1 in this vertex summary, we use a secondary vertex s_1 in addition to the primary vertex r_1 (see Section 2.2).

other words, it is identical to the simple schema element POC. Similarly to the previous example, PC is identical to the CSE $(\top, \text{id}, \top)$ and OC is the CSE $(\top, \top, \text{id})$. Formally, we may regard the simple schema elements as abbreviations for the corresponding CSEs; however, we implement them separately for efficiency.

Example 2. Let $CSE\text{-}2 = (\top, \text{id}, \text{PC})$. We can see that this is a relaxation of CSE-1 from Example 1, in that CSE-1 requires objects to be identical, whereas CSE-2 only requires objects to have the same set of predicates. In full, CSE-2 declares vertices s and s' to be equivalent iff for every triple $(s, p, o) \in G$, there is a triple $(s', p', o') \in G$ such that $p' = p$, and o' and o, in turn, have the same predicates, and vice versa. That is, for every predicate p that links s to a neighbor o, p also links s' to a neighbor with the same set of predicates as o, and vice versa. This means that CSE-2 summarizes vertices using, in part, the schema of neighboring vertices $\Gamma^+(s)$. In Figure 3, we illustrate the vertex summaries for the example data graph visualized in Figure 2a according to the CSE-2 equivalence relation.

4.2. Parameterizations of Schema Elements

Simple and complex schema elements are the basic building blocks of FLUID. FLUID provides six parameterizations to specialize the schema elements' behavior. For example, a parameterization can be applied that ignores a certain set of predicates or constructs predicate paths with a specific length. In the following sections, we define six parameterizations, which are again motivated by the analysis of existing graph summaries in Section 3. The first two parameterizations address the triple features identified in Section 3.1, the next two address the subgraph features from Section 3.2, and the final two address the semantic rule features from Section 3.3.

4.2.1. Label Parameterization

The label parameterization can be used to restrict schema elements to consider only specific predicates. This allows us to define, e. g., the type cluster OC_{type}, an object cluster that only compares objects connected over the predicate `rdf:type`. Hence, OC_{type} compares vertices based on type sets.

Definition 8. The **label parameterization** is a function $lp(SSE, P_r)$, which takes as input a simple schema element SSE and a set of predicates $P_r \subseteq P$ and returns a schema element SE_{P_r}. The returned schema element SE_{P_r} is defined analogously to SE (Definitions 4, 5, and 6) but each existential and universal quantifier is restricted to tuples (s, p, o) with $p \in P_r$. In detail, $lp(SE, P_r)$ defines vertices s and s' to be equivalent iff, for every triple $(s, p, o) \in G$ with $p \in P_r$, there is a triple $(s', p', o') \in G$ with $p' \in P_r$ such that:

- $SE = \text{POC}$ and $p = p'$ and $o = o'$; or
- $SE = \text{PC}$ and $p = p'$; or
- $SE = \text{OC}$ and $o = o'$.

Using the predicate `rdf:type`, the label-parameterized object cluster $lp(\text{OC}, \{\texttt{rdf:type}\})$ summarizes vertices that have the same set of vertices connected with the predicate `rdf:type`, i. e., vertices with the same type sets. To ease notation, we refer to this label-parameterized object cluster as the **type cluster** OC_{type}. Two vertices s and s' are equivalent under OC_{type}, iff $\ell_T(s) = \ell_T(s')$. Analogously, we define the label-parameterized predicate cluster $lp(\text{PC}, P \setminus \{\texttt{rdf:type}\})$, which summarizes vertices that have the same predicates explicitly excluding the RDF specific `rdf:type` predicate. We denote this label-parameterized predicate cluster as property cluster PC_{rel}. Two vertices s and s' are equivalent according to the PC_{rel}, iff $\ell_P(s) = \ell_P(s')$.

4.2.2. Set Parameterization

The set parameterization sp is applied to simple schema elements. In addition to the requirements of the SSE itself, the set parameterization also requires that all predicate and/or objects must be element of a given set S. In contrast to the label parameterization, which just ignores predicates not in S, the set parameterization says that two vertices are automatically equivalent if they have any predicate not in S.

Definition 9. The **set parameterization** is a function $sp(SSE, S)$, which takes as input a simple schema element SSE and a set of IRIs $S \subseteq V_I$ and returns a simple schema element $SSE|_S$. $SSE|_S$ defines an equivalence relation EQR such that $(s, s') \in EQR$ iff

- if $SSE \in \{\text{PC}, \text{POC}\}$,
  - $S \neq \emptyset$, and s and s' both have at least one outgoing edge; or
  - for all p, p', where there are triples $(s, p, o) \in G$ and $(s', p', o') \in G$, it must follow that $p, p' \in S$ and s, s' are equivalent under SSE; or
  - there are triples $(s, p, o) \in G$ and $(s', p', o') \in G$, where $p, p' \notin S$
- if $SSE \in \{\text{OC}, \text{POC}\}$,
  - $S \neq \emptyset$, and s and s' both have at least one outgoing edge; or
  - for all o, o', where there are triples $(s, p, o) \in G$ and $(s', p', o') \in G$, it must follow that $o, o' \in S$ and s, s' are equivalent under SSE; or
  - there are triples $(s, p, o) \in G$ and $(s', p', o') \in G$, where $o, o' \notin S$

The set parameterization can be combined, e. g., with the label parameterization. We demonstrate this below.

Example 3. Let us apply the set parameterization to the type cluster OC_{type} using the empty set $\emptyset$. The set-parameterized type cluster $\text{OC}_{\text{type}}|_\emptyset$ splits the data graph into at most two equivalence classes. The first equivalence class (vertex summary) contains all vertices that have empty type sets (no type information provided in the data graph) and the second equivalence class (vertex summary) contains all vertices that have at least one type, if any such vertices exist.

Example 4. Let us apply the set parameterization to the property cluster PC_{rel} using the set $S = \{p_1, p_2\}$. The set-parameterized property cluster $\text{PC}_{\text{rel}}|_S$ compares vertices v, v' based on the same property set only if the property set is a subset of S. Any vertex v where there is a $p \in \ell_P(v)$ that is not in S will be summarized by the same vertex summary. Also, the corner case $\ell_P(v) = \emptyset$ will be summarized by that same vertex summary. The remaining vertices are summarized based on the equivalence of their property sets. Thus, there can be no more than four vertex summaries for any input graph, i. e., the ones corresponding to property sets $\{p_1\}$, $\{p_2\}$, and $\{p_1, p_2\}$, and the one that summarizes all other vertices.

The set parameterization can be used like a filter. When such a set S of predicates and/or objects is provided, the graph summary may contain a vertex summary that summarizes all vertices that have predicates and/or objects not in S. More importantly, it contains vertex summaries with exactly these predicates and/or objects. Subsequently, one could either filter out these vertex summaries or all other vertex summaries. In Example 3, we may only want to visualize all vertices with a non-empty type set. In Example 4, we may only want to visualize all vertices that exclusively use properties p_1 and/or p_2.

4.2.3. Chaining Parameterization

Complex schema elements take the schema of two directly connected vertices into account. The chaining of k complex schema elements extends this to vertices within distance k. As discussed in Section 3, this corresponds to a stratified k-bisimulation. The chained complex schema element is denoted by CSE^k.

Definition 10. The **chaining parameterization** is a function $cp(CSE, k)$, which takes a complex schema element $CSE := (\sim^s, \sim^p, \sim^o)$ and a chaining parameter $k \in \mathbb{N}_{>0}$ as input and returns an equivalence relation CSE^k that corresponds to recursively applying CSE to a distance of k hops. CSE^k is defined inductively as follows:

$$CSE^1 = (\sim^s, \sim^p, \sim^o)$$
$$CSE^{k+1} = (\sim^s, \sim^p, CSE^k) \qquad \text{for } k \geq 1\,.$$

Example 5. Recall CSE-2 $:= (\top, \mathrm{id}, \mathrm{PC})$ from Example 2 and denote by $\sim_2$ the equivalence relation it defines. Consider the chained complex schema element CSE-3 $:= cp(CSE\text{-}2, 2)$. By Definition 10, this corresponds to the CSE $(\top, \mathrm{id}, \sim_2)$. Let CSE-3 define the equivalence relation $\sim_3$ and consider vertices s and s'. We have $s \sim_3 s'$ iff the following conditions are met.

- s and s' must have the same predicates, because predicate equivalence is defined by equality using the identity equivalence id.
- For every neighbor $o \in \Gamma^+(s)$ via any predicate, there is a neighbor $o' \in \Gamma^+(s')$ via the same predicate, such that the vertices o and o' are equivalent under CSE-2. That is, for each predicate p that links o to a neighbor with some predicates, the predicate p also links o' to a neighbor with the same predicates.
- Vice versa, for every neighbor o' of s', a similarly corresponding neighbor o of s exists.

Intuitively, CSE-3 determines the equivalence of vertices based not only on their neighbors but also on their neighbors' neighbors. As the chaining parameter k increases, we consider wider neighborhoods.

4.2.4. Direction Parameterization

The direction parameterization dp is applied on schema elements to use only outgoing predicates (parameter $\delta = \mathrm{o}$), only incoming predicates ($\delta = \mathrm{i}$), or both ($\delta = \mathrm{b}$). Our schema elements OC, PC, POC, and CSE only take outgoing predicates into account. Schema structures like Characteristic Sets [33] use also incoming predicates. To address incoming predicates, a parameterized version of the three simple schema elements can be defined by using the incoming triples $(x, p, v) \in G$ with the summarized vertex v in object position.

Definition 11. The **direction parameterization** is a function $dp(SE, \delta)$, which takes as input a schema element SE and one direction parameter $\delta \in \{\mathrm{i}, \mathrm{o}, \mathrm{b}\}$ and returns a schema element δ-SE, respectively. Direction i uses only incoming predicates, o uses only outgoing predicates, and b uses both, incoming and outgoing predicates. The returned schema element δ-SE is a modification of SE in which all assertions

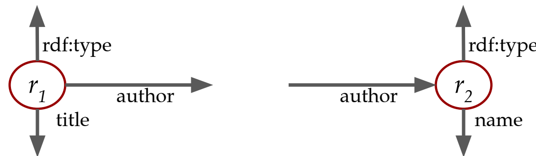


Figure 4: Following the b-PC equivalence relation, the two vertex summaries identified by the primary vertices r_1 and r_2 summarize the vertices v_1 and v_2 from Figure 2a, respectively.

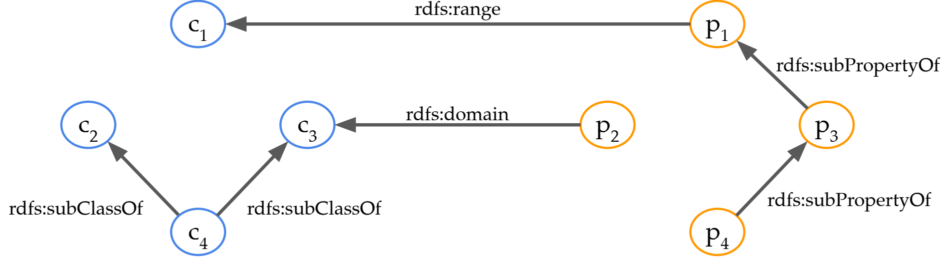


Figure 5: Simple example of a RDFS vocabulary graph. Each vertex represents either a type or a property. To infer implicit information, starting from a given type or property vertex, inferrable information is collected by following edges. The rules on how to collect inferrable information are defined in the RDFS standard [7].

made by SE about the triples are applied on the incoming edges only, the outgoing edges only, or on both. This means that o-$SE = SE$, i-SE is SE modified to use incoming predicates instead of outgoing and b-SE = i-$SE \cap$ o-SE.

Example 6. The bidirectional predicate cluster b-PC partitions the data graph by comparing the triples based on common incoming and outgoing predicates. Thus, the equivalence relation b-PC holds true, iff for each triple $(s_1, p_1, o_1) \in G$ there exists a triple $(s_2, p_1, o_2) \in G$, and vice versa, and for each triple $(s_3, p_3, s_1) \in G$ there exists a triple $(s_4, p_3, s_2) \in G$, and vice versa. The bidirectional predicate cluster b-PC is visualized in Figure 4.

4.2.5. Inference Parameterization

Some graphs contain semantic information, e. g., ontologies described with the RDF Schema (RDFS) vocabulary which, like RDF, is standardized by the W3C [7]. The inference parameterization is the first parameterization to include the semantics of the data graph in the structural graph summary and also falls under our category of semantic rule features (see Section 3.3). The inference parameterization *op* is applied on the graph G and enables ontology reasoning using a vocabulary graph VG. Applying RDFS inferencing means that a vocabulary graph VG_{RDFS} is constructed from the data graph G, which stores all hierarchical dependencies between types and properties denoted by RDFS properties found in the data graph G. To construct the RDFS vocabulary graph, we first extract all triples containing RDFS vocabulary terms, namely all properties $P_{\text{RDFS}} = \{\texttt{rdfs:subClassOf}, \texttt{rdfs:subPropertyOf}, \texttt{rdfs:range}, \texttt{rdfs:domain}\}$. Subsequently, we add the corresponding hierarchical dependencies of `rdfs:subClassOf` and `rdfs:subPropertyOf` in a polytree[4] structure with further cross connections regarding `rdfs:range` and `rdfs:domain` [3]. An example of such a vocabulary graph is illustrated in Figure 5.

Definition 12. An **RDFS vocabulary graph** is an edge-labeled directed multigraph $VG_{\text{RDFS}} \subseteq (V_C \cup P) \times P_{\text{RDFS}} \times (V_C \cup P)$. The set of vertices is the union of types V_C and predicates P in G. The edges are labeled with predicates $p \in P_{\text{RDFS}}$.

With hierarchical dependencies of types and predicates represented using our vocabulary graph, additional triples can be inferred using our inference parameterization. For a given triple in the data graph G, we can look up the vertex representing the used predicates or type in VG. Starting from this vertex in VG, the information needed for inference is contained in a polytree with the vertex itself as root. The entailment rules for VG_{RDFS} are defined by the W3C [7]. A comprehensive overview of these rules is presented in [1].

Definition 13. The **inference parameterization** is a function $op(G, VG)$ which takes any data graph G and a vocabulary graph VG as input and, based on the entailment rules defined in VG, returns a data graph G_{VG}, which also includes all inferred triples.

[4] A polytree is an orientation of an undirected tree.

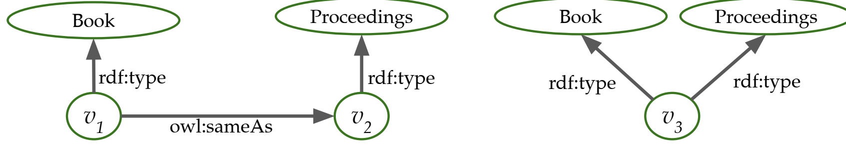


Figure 6: Sample data graph containing three vertices v_1, v_2, and v_3. The object cluster assigns v_1, v_2, and v_3 to different vertex summaries. The instance parameterized object cluster using SameAs instances $[v]_\sigma$ summarizes v_1, v_2, and v_3 into a single vertex summary.

With our notion of inference, we can infer implicitly stated triples, e. g., by adding the implicitly stated types and properties of vertices to the type sets and property sets of the respective vertices. For example, if a vertex has the type `Proceedings` and the vocabulary graph contains the triple (`Proceedings`, `rdfs:subClassOf`, `Book`), then the type `Book` can be inferred and added to the vertex' type set.

FLUID's formalization does not pose any restrictions on when the actual inference happens. In general, we can distinguish inference *inside* the structural graph summary and *outside* the structural graph summary. Inference inside means that all inferrable information is added to the original graph and the graph summary is computed for the new data graph. However, the order of these two is interchangeably. Inferring on the data graph and then summarizing can be equivalent to summarizing the data graph and then inferring on the graph summary [18, 28]. The latter one then may require another update on the graph summary.

Inference outside means that we compute the graph summary for the original graph and keep the constructed vocabulary graph. When we query the graph summary to fulfill our task, we can infer only the additional information that affects the query result, e. g., by generating additional queries based on entailment rules defined in the vocabulary graph. The result sets for inferencing inside and outside are equivalent. This decision affects the build-time and the size of the computed structural graph summary as well as time to compute query results on the structural graph summary. We discuss practical implications of this decision in the empirical analysis in Section 6.

4.2.6. Instance Parameterization

In RDF graphs, vertices and all of their outgoing triples are commonly referred to as RDF instances. Referencing this terminology, we define the instance parameterization *ip*. This allows us to define rules to join sets of outgoing triples of different vertices, e. g., vertices linked with `owl:sameAs`. The instance parameterization is the final parameterization regarding the rule features (see Section 3.3).

Definition 14. The **instance parameterization** *ip* is a function $ip(SE, \Delta)$, which extends any schema element SE to additionally take the schema of all equivalent vertices into account, following the instance equivalence relation Δ. The returned schema element $SE[\Delta]$ is an extension of SE, which restricts the triples to be in $[v]_\Delta$. Thus, *ip* merges vertices that are equivalent under Δ and then applies SE.

In the following, we give two definitions of such an instance equivalence relation parameter Δ, namely SameAs instances σ and related property instances ρ.

SameAs instances σ. In the context of Linked (Open) Data, the `owl:sameAs` property is of particular interest since $(s, \texttt{owl:sameAs}, s')$ explicitly states the equivalence of the entities identified by the vertices s and s'. To take this information into account, we use the notion of **SameAs instances** $[v]_\sigma$, which are equivalence classes of vertices or, equivalently, unions of vertices. Two vertices s and s' are equivalent according to the equivalence relation σ, iff there is a property-path in the data graph G labeled `owl:sameAs` from s to s' [3]. Note that the property-path is independent of the direction of the `owl:sameAs` relation. When summarizing vertices using the SameAs instance parameterization, we take the schema information from all equivalent vertices into account.

Example 7. See Figure 6. According to the object cluster definition, the vertices v_1, v_2, and v_3 are not equivalent since v_1 has the object "Book", v_2 has the object "Proceedings", and v_3 has the objects "Book" and "Proceedings". Merging v_1 and v_2 to a SameAs instance $[v]_\sigma$ leads to the equivalence of all three vertices v_1, v_2, and v_3.

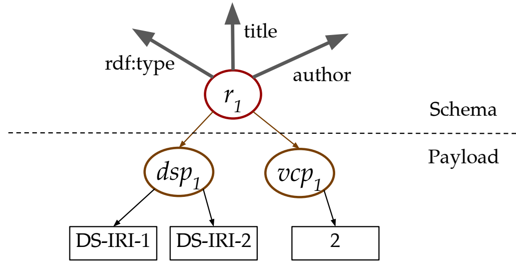


Figure 7: The vertex summary identified by the primary vertex r_1 summarizes vertices with the property set $\{$**author**, **title**, **rdf:type**$\}$. The data source payload dsp_1 contains information about the data sources and the vertex count payload element vcp_1 contains the number of summarized vertices.

Related property instances ρ. Merging vertices also allows to model property cliques [18]. Property cliques transitively check the co-occurrence of properties and summarize all vertices that have at least one property in common. We can define an instance equivalence relation ρ that leads to equivalent vertices if they share at least one property. We call the equivalence classes $[v]_\rho$ **related property instances**. We distinguish source related properties and target related properties [18]. Schema Elements parameterized with related property instances that consider outgoing properties use source related properties. For Schema Elements using outgoing properties, two vertices s and s' are equivalent according to the equivalence relation ρ, iff there exists vertex $s'' \in [s]_\rho$ such that there are triples $(s'', p, o) \in G$ and $(s', p, o) \in G$ with $p \neq \texttt{rdf:type}$, or, similarly for some $s''' \in [s']_\rho$.

Schema Elements parameterized with related property instances that consider incoming properties use target-related properties. For Schema Elements using incoming properties, two vertices s and s' are equivalent according to the equivalence relation ρ, iff there exists a vertex $s'' \in [s]_\rho$ such that there are triples $(x, p, s'') \in G$ and $(x', p, s') \in G$ and $p \neq \texttt{rdf:type}$, or similarly for some $s''' \in [s']_\rho$.

Note that ρ takes transitively co-occurring properties into account. Thus, ρ may summarize two vertices that do not share any property in the data graph.

4.3. Payload Elements

In total, FLUID provides four schema elements and six parameterizations. They are designed to capture all schema structures defined by (semantic) structural graph summaries found in the related work and beyond. In this section, we define several possible payload elements that are designed for purposes found in existing (semantic) structural graph summaries.

When computing the graph summary SG, information about the summarized vertices can be attached to vertex summaries by using the notion of payload [19]. The payload contains information about the actual data, e. g., number of vertices summarized or a reference to their data source. Our intention is to make payloads as flexible as possible so we do not make any restrictions on what can be attached as payload and we allow multiple different kinds of payload to be attached to a single vertex summary. To this end, we define PAY to be a set of payload elements. Payload elements map vertex summaries to a payload, i. e., information extracted from the summarized vertices. They are functions that define what information is attached. For example, one payload element may attach the data sources of summarized vertices to vertex summaries while another payload element attaches the number of summarized vertices to vertex summaries. An example is illustrated in Figure 7. The payload dsp_1 contains data source IRIs that were extracted from the vertices summarized by the vertex summary represented by r_1. The payload vcp_1 contains an integer value representing the number of vertices summarized by r_1.

4.3.1. Indexing Vertices

One purpose of structural graph summaries is to index vertices by their schema. Thus, we need to be able to store the identifiers of summarized vertices as payload of schema elements.

Definition 15. Consider a graph summary of some graph, generated from an equivalence relation EQR. The **vertex identity payload** is the set of identifiers (IRIs) of the vertices summarized by each vertex summary vs. The vertex identity payload element vip is a function that takes a vertex summary vs_v of a vertex v as input and returns the set $[v]_{\mathrm{EQR}}$ of summarized vertices, i. e., $vip(vs_v) := [v]_{\mathrm{EQR}}$.

4.3.2. Query Size Estimation

Another purpose of structural graph summaries is cardinality estimation for database queries [33]. To implement this task, we only need the number of summarized vertices and not the vertex identifiers. Thus, we can define another payload element, which maps only the integer value denoting the number of summarized vertices.

Definition 16. The **vertex count payload** provides the number of vertices summarized by the vertex summary. It is computed by the vertex count payload element *vcp*, which takes a vertex summary vs_v as input and returns an integer denoting the number of summarized vertices. Thus, formally *vcp* is a function defined as $vcp(vs_v) := |vip(vs_v)|$.

4.3.3. Data Source Selection

Our third payload element is the data source payload element *dsp*, which is needed to implement the data search task [19]. The data source payload element *dsp* maps a vertex summary to the set of data source IRIs of all summarized vertices. This payload is only valid for named graphs, i. e., where each triple (s, p, o) is extended to a tuple $((s, p, o), d)$ (see Section 2.1).

Definition 17. The **data source payload** contains the locations of the vertices summarized by the vertex summary. The payload element *dsp* is a function that takes a vertex summary vs_v as input and returns all data source IRIs d for which there is tuple $((s, p, o), d)$ in G for some vertex s that is summarized by the vertex summary vs_v. Formally, the data source payload is defined as:

$$dsp(vs_v) := \{d \mid \text{there is a tuple } ((s, p, o), d) \in G \text{ for some } s \in vip(vs_v)\} .$$

4.4. Defining the Existing Graph Summary Models with FLUID

In this section, we review all existing works analyzed in Section 3 and show how to define them using FLUID. Thus, we demonstrate that FLUID can indeed be used to model all the existing (semantic) structural graph summaries in Table 1. We briefly describe their definitions using FLUID, which are shown in Table 3. Note that the summaries defined in the related work were defined for specific tasks (e.g., query size estimation) so, here, we use the payload appropriate to that task. We could, of course, adapt the summaries to other tasks by using other payloads.

The first group of *simple* graph summaries are defined using simple schema elements and parameterizations of them. For Attribute-based Collection, we use the property cluster PC_{rel}, i. e., label-parameterized predicate cluster not considering the predicate `rdf:type`. Thus, we summarize vertices solely based on common outgoing properties (the property sets). For Class-based Collection, we use the type cluster OC_{type}, i. e., the label-parameterized object cluster considering only the predicate `rdf:type`. Thus, we summarize vertices solely based on common type sets. For both graph summaries, we attach identifiers of the summarized vertices as payload using the vertex identity payload element *vip*.

Characteristic Sets are defined using the bidirectional predicate cluster b-PC. This means that vertices are summarized based on having the same outgoing predicates and the same incoming predicates. As payload, we attach the number of summarized vertices using the vertex count payload element *vcp* to implement the cardinality estimation task. SemSets are defined using the predicate-object cluster POC, i. e., vertices are summarized based on having the same outgoing triples. As payload, we again attach the summarized vertices to implement related-entity search.

For the second group of *complex* graph summaries, we use the complex schema element to combine equivalence relations. To define SchemEX, we combine the type cluster OC_{type} and id_{rel}, the label-parameterized identity equivalence using all predicates except `rdf:type` (i. e., only properties), in a complex schema element. Since complex schema elements support predicate paths by default, they capture which predicate linked to which type set of which neighbor. As payload, we use the data sources of summarized vertices, i. e., the data source payload element *dsp*. For SchemEX+U+I, we additionally include SameAs instances $\Delta = \sigma$ and an RDFS vocabulary graph VG_{RDFS} on top of SchemEX. ABSTAT, LODex, and Loupe define

the same graph summary as SchemEX, except that they use the vertex identity payload instead of the data source payload and ABSTAT uses RDF Schema inference on the data graph. To define TermPicker, we use the intersection of the label-parameterized object cluster OC_{type} and the label-parameterized property cluster PC_{rel}. The existence of any predicate equivalence $\sim^p \neq \top$ in a complex schema element enables predicate paths, capturing which predicate links to which type set of a neighbor. TermPicker defines a graph summary that does not use predicate paths, so we use the tautology equivalence $\top$ as predicate equivalence. This way, no predicate paths are taken into account and all types of all neighbors are aggregated.

The next four summary models are proposed by Goasdoué et al. [18]. All of their summary models use the instance parameterization with related property instances $\Delta = \rho$. The Weak Summary summarizes vertices based on incoming properties or outgoing properties or both. Thus, we define it with directed property clusters i-PC_{rel} and o-PC_{rel} combined using the extended union operator $\cup_{\text{ex}}$. The Strong Summary summarizes vertices based on incoming properties and outgoing properties. Thus, we define it with a bidirectional property cluster b-PC_{rel}. The Typed Weak Summary summarizes vertices based on the Weak Summary only if the vertices have empty type sets. Otherwise, they are summarized based on having the same type sets. We express this using the set parameterization applied on the type cluster OC_{type} with $S = \emptyset$. $\text{OC}_{\text{type}}|_{\emptyset}$ partitions the vertices into those with no types and those with types in the data graph. $\text{OC}_{\text{type}}|_{V_C}$ partitions the vertices into those with no types and those with the same type sets in the data graph. Analogously, we define the Typed Strong Summary, which summarizes based on the Strong Summary only if the vertices have empty type sets.

The final five approaches use k-bisimulation and can be defined using the chaining parameterization. For the label- and height-parameterized graph summary model proposed by Tran et al. [43], we use the complex schema element that only compares outgoing predicates included in the label set L up to maximum hop length of k. The parameters L and k are user defined. Kaushik et al. [25] define their A(k)-index

Table 3: All graph summary models from the related work (see Table 1) defined using the FLUID language.

	Graph Summary Model	**FLUID definition**
Simple	Attribute-based Collection [8]	$(G, \text{PC}_{\text{rel}}, \{vip\})$
	Class-based Collection [8]	$(G, \text{OC}_{\text{type}}, \{vip\})$
	Characteristic Sets [33]	$(G, \text{u-PC}, \{vcp\})$
	SemSets [11]	$(G, \text{POC}, \{vip\})$
Complex	LODex [2]	$(G, (\text{OC}_{\text{type}}, \text{id}_{\text{rel}}, \text{OC}_{\text{type}}), \{vip\})$
	Loupe [31]	$(G, (\text{OC}_{\text{type}}, \text{id}_{\text{rel}}, \text{OC}_{\text{type}}), \{vip\})$
	SchemEX [27]	$(G, (\text{OC}_{\text{type}}, \text{id}_{\text{rel}}, \text{OC}_{\text{type}}), \{dsp\})$
	SchemEX+U+I [4]	$(G_{VG_{\text{RDFS}}}, (\text{OC}_{\text{type}}, \text{id}_{\text{rel}}, \text{OC}_{\text{type}})[\sigma], \{dsp\})$
	ABSTAT [41]	$(G_{VG_{\text{RDFS}}}, (\text{OC}_{\text{type}}, \text{id}_{\text{rel}}, \text{OC}_{\text{type}}), \{vip\})$
	TermPicker [39]	$(G, (\text{OC}_{\text{type}} \cap \text{PC}_{\text{rel}}, \top, \text{OC}_{\text{type}}), \{vip\})$
	Weak Summary [18]	$(G_{VG_{\text{RDFS}}}, \text{i-PC}_{\text{rel}}[\rho] \cup_{\text{ex}} \text{o-PC}_{\text{rel}}[\rho], \{vip\})$
	Strong Summary [18]	$(G_{VG_{\text{RDFS}}}, \text{b-PC}_{\text{rel}}[\rho], \{vip\})$
	Typed Weak Summary [18]	$(G_{VG_{\text{RDFS}}}, (\text{OC}_{\text{type}}\|_{\emptyset} \cap (\text{i-PC}_{\text{rel}}[\rho] \cup_{\text{ex}} \text{PC}_{\text{rel}}[\rho])) \cup_{\text{ex}} \text{OC}_{\text{type}}\|_{V_C}, \{vip\})$
	Typed Strong Summary [18]	$(G_{VG_{\text{RDFS}}}, (\text{OC}_{\text{type}}\|_{\emptyset} \cap \text{b-PC}_{\text{rel}}[\rho]) \cup_{\text{ex}} \text{OC}_{\text{type}}\|_{V_C}, \{vip\})$
	Tran et al. [43]	$(G, (\top, \text{id}_L, \top)^k, \{vip\})$
	A(k)-index [25]	$(G, (\text{u-PC}, \top, \top)^k, \{vip\})$
	T-index [32]	$(G, (\text{i-PC}, \top, \top)^k, \{vip\})$
	Consens et al. [12]	$(G, (\text{OC}_{\text{type}}, \text{id}_{\text{rel}}, \text{OC}_{\text{type}})^k, \{vip\})$
	Schätzle et al. [40]	$(G, (\text{OC}, \text{id}, \text{OC})^k, \{vip\})$

using forward and backward bisimulation, i. e., it follows incoming and outgoing predicate paths up to a maximum hop length of k. We define this using the bidirectional predicate cluster chained in a complex schema element. Analogously, we define the T-index proposed by Milo and Sucio [32] for incoming predicates only. Consens et al. [12] summarize vertices based on outgoing property paths only and also compare the type sets of each vertex. The graph summary proposed by Schätzle et al. [40] is analogous to the previous one, but compares the vertices' identities instead of their type sets.

4.5. Summary

In summary, we define graph summaries to be 3-tuples of consisting of a data graph G, an equivalence relation EQR, and payload elements PAY. We introduced three simple and one complex schema elements as well as six parameterizations to define graph summaries. We demonstrated that these elements and parameterizations can be flexibly combined to define existing (semantic) structural graph summaries. All analyzed graph summaries discussed in Section 3 can be expressed using the FLUID language. We can also adapt existing graph summaries to new tasks. The simplest modification is changing the payload elements, which is easily done, but has a big impact on the size of graph summary and on which tasks can be fulfilled. For example, attaching the number of summarized vertices requires less space than attaching identifiers of summarized vertices. Furthermore, we can adapt the captured schema structure by filtering out specific types and properties or enabling inference on semantic graphs. For example, we can define a new graph summary PC-Inferenced $:= (G_{VG_{\text{RDFS}}}, (\top, \text{id}_{\text{rel}}, \top), \{dsp\})$ that only uses outgoing properties and uses inference.

There are various new possibilities to combine FLUID's schema elements and parameterizations to define existing and new (semantic) structural graph summaries. It can be shown that every combination of FLUID's schema elements and parameterizations actually defines a partition over the data graph and thus, is a valid graph summary. All schema elements and parameterizations are defined as equivalence relations. The intersection and the extended union of such equivalence relations is again an equivalence relation.

FLUID provides a handful of elements and parameterizations to define (semantic) structural graph summary models. This allows us to define a generic algorithm to compute (semantic) structural graph summaries. We present this algorithm and analyze its computational complexity in the next section.

5. Graph Summarization Algorithm

In this section, we introduce our algorithm to compute structural graph summaries. We propose a single, parameterized algorithm, which can compute all structural graph summaries defined with FLUID. To compute a structural graph summary, we need to summarize vertices to vertex summaries, partitioning the data graph into disjoint subsets of vertices. This is a version of the *set union problem*, for which Tarjan's algorithm has been proven to be asymptotically optimal [42]. This algorithm is based on operations "make-set" and "find" and takes time $\Theta(n \cdot \alpha(n, n))$ for a sequence of n of these operations, where α is the inverse of Ackermann's function. Although unbounded, α grows extremely slowly and it is generally accepted that $\alpha(n, n) \leq 4$ for all practically possible inputs [42]. We therefore refer to the running time of $\Theta(n \cdot \alpha(n, n))$ as being "essentially linear time".

5.1. Algorithm

FLUID describes the rules of how to combine schema elements, parameterizations, and payload elements. To compute structural graph summaries, we propose an algorithm based on hash maps. We assume that we can implement make-set and find operations in constant time (amortized) using hash maps, using hashes that are long enough to avoid collisions [27]. Our algorithm is presented in Algorithm 1. To simplify the code, we only show an excerpt of the complete algorithm. Furthermore, we define a set of globally accessible hash maps in the data structure *GlobalVariables* to ease readability.

In the first pass over the data graph G (Line 13 to Line 23), we iterate once over all triples to prepare our data structures. Each triple using a property $p \in P_{\text{RDFS}}$ as predicate is added to the vocabulary graph VG_{RDFS}. The remaining triples are added to the InMap and the OutMap. The InMap stores the triples

with the subject s as key and the OutMap stores triples with the object o as key. This allows us to quickly retrieve all incoming and outgoing triples for each vertex in G. In case we are using SameAs instances, we construct sets of equivalent vertices following the `owl:sameAs` property in Lines 20 and 21. In case we are using related property instances, we construct sets of equivalent vertices following any outgoing or incoming property in Lines 22 and 23, respectively.

In the second pass (Line 24 to Line 28), we infer incoming and outgoing triples according to the constructed Vocabulary Graph VG_{RDFS}. The inferred incoming and outgoing triples are stored in the hash maps InfInMap and InfOutMap, respectively. In the third pass, the schema for each vertex is computed. For each (parameterized) schema element used to define the equivalence relation EQR, we extract the schema and compute the vertex summaries. These vertex summaries are stored in SchemaMap and identified by their primary vertices (see Section 2.2). Using the primary vertices, the summaries will be combined to form the vertex summary vs for EQR (compare Example 2). The primary vertices of all combined vertex summaries will become secondary vertices of the resulting summary vs. Finally, for each payload element in PAY, we compute the payload for each vertex v. The resulting vertex summary vs for each v along with the payloads are added to the summary graph SG in Line 33.

Algorithm 1: Sequential, parameterized algorithm to compute structural graph summaries defined as equivalence relation EQR with the FLUID language

```
1  function ComputeGraphSummary(G, EQR, PAY)
2      returns graph summary SG
3      struct GlobalVariables = {
4          OutMap ← ∅;
5          InMap ← ∅;
6          InfOutMap ← ∅;
7          InfInMap ← ∅;
8          SameAsInstanceMap ← ∅;
9          SrcRelatedMap ← ∅;
10         TrgRelatedMap ← ∅;
11         SchemaMap ← ∅;
12     };
13     forall (s, p, o) ∈ G do
14         if p ∈ P_RDFS then
               /* Add triple to vocabulary graph if it is an RDF Schema triple.  */
15             VG.AddElement((s, p, o));
16         else
               /* Index triples by subject vertex s.  */
17             OutMap.Merge(s, {(s, p, o)});
               /* Index triples by object vertex o.  */
18             InMap.Merge(o, {(s, p, o)});
               /* In case SameAs instances are used  */
19             if p = owl:sameAs then
20                 SameAsInstanceMap.Merge(s, {o}));
21                 SameAsInstanceMap.Merge(o, {s});
               /* In case property-related instances are used, check for source-related properties...  */
22             SrcRelatedMap.Merge(p, {s});
               /* ...  and check for target-related properties  */
23             TrgRelatedMap.Merge(p, {o}));
24     if G uses RDF Schema inferencing then
25         forall (v, OUT) ∈ OutMap do
               /* Prepare for inferencing (see Section 4.2.5)  */
26             InfOutMap.Put(v, VG.InferOntologyInformation(OUT));
27         forall (v, IN) ∈ InMap do
               /* Prepare for inferencing  */
28             InfInMap.Put(v, VG.InferOntologyInformation(OUT));
       /* Design decision to only consider vertices that are the subject of at least one triple  */
29     forall v ∈ OutMap.Keys() do
           /* Extract the schema according to EQR  */
30         vs ← ExtractSchema(v, EQR);
31         forall pay ∈ PAY do
32             p_i ← pay.ExtractPayload(v);
33         SG.AddElement(vs, ⋃_i p_i);
34     return SG;
```

Example 8. Suppose we want to extract the schema according to a predicate cluster (PC) for a vertex s and there are two triples (s, p_1, o_1), and (s, p_2, o_2) in G with s as subject. We extract the predicate for each triple and construct the corresponding set, i. e., $\{p_1, p_2\}$. We compute the hash value of the predicate set and store it in the schema hash map SchemaMap. The payload, e. g., the number of summarized vertices, is stored as value. For another vertex s' with $(s', p_1, o_3), (s', p_2, o_4) \in G$, the same predicate set is extracted, so we compute the same hash value. When we update the SchemaMap, we update the payload. In this example, we increase the vertex counter by one. Consequently, the resulting vertex summary defined as PC summarizes both vertices and only stores the defined payload.

One can see that the computationally expensive task is the extraction of the schema of each vertex, e. g., extracting the property set in Example 8. However, our algorithm benefits from the fact that every schema element in FLUID is defined as equivalence relation. In particular, for one vertex s there is exactly one schema structure according to any one (parameterized) schema element, e. g., one predicate cluster (PC) and one label-parameterized object cluster ($\mathrm{OC}_{\mathrm{type}}$) etc. We define complex graph summaries by combining simple schema elements with complex schema elements, where we can include neighboring vertices in the schema structure. For example, SchemEX uses type sets of neighboring vertices to the compute the schema of the actual vertex. Therefore, one may assume that depending on the in-degree of a vertex, we have to extract the schema more than once. Instead, we use a hash map to store, for each vertex in the data graph, all computed vertex summaries identified by secondary vertices. This avoids the expensive task of extracting the schema of vertices more than once.

Lemma 1. *For each vertex v in the data graph G, we need to compute the schema according to each schema element in the summary definition only once.*

5.2. Complexity Analysis

We analyze the complexity of computing (semantic) structural graph summaries defined with FLUID. In order to estimate the computational complexity, we conduct a space and time analysis of the computation process for graph summaries defined with FLUID. For this analysis, we assume that hash maps have amortized constant-time read and write access and that hashes are long enough to avoid collisions when hashing, e. g., vertices' property sets and type sets [27]. Goasdoué et al. [18] justify this assumption, via their hypothesis ($\star$), and provide an alternative data structure using Tarjan's algorithm instead of hash maps. In a previous work, we found evidence that supports their hypothesis, i. e., that the numbers of incoming and outgoing predicates for each vertex are, in practice, bounded [5].

For the space complexity of the summary, the important factor is how well the summary model summarizes the input graph, i. e., how many different vertex summaries are needed to summarize all vertices in G. The worst case is that no two vertices are summarized by the same vertex summary, i. e., all vertex summaries in the structural graph summary summarize exactly one vertex. In this worst case, each vertex summary is a new entry for our hash map. Thus, the upper bound for the time complexity and the upper bound for the space complexity are identical. In the following, we analyze in more detail the influence of FLUID's schema elements and parameterizations on the computational complexity.

5.2.1. Schema Elements

Given a FLUID definition of a graph summary, we denote by λ the number of simple schema elements and by κ the number of complex schema elements used to define the equivalence relation EQR. The parameterizations are applied to these simple and complex schema elements and pose further restrictions or relaxations.

The data graph G contains n triples. Simple schema elements determine whether two vertices $s, s' \in G$ are equivalent by considering only triples that have s or s' as subject. To compute PC, OC, or POC, we need to compute, for each vertex $s \in G$, the set $\{p \mid (s, p, o) \in G \text{ for some } o\}$, $\{o \mid (s, p, o) \in G \text{ for some } p\}$ or $\{(p, o) \mid (s, p, o) \in G\}$, respectively. All of these sets can be computed in a single scan of the graph. A hash for each set can be computed by combining the hashes of its elements, e. g., using XOR or techniques

based on symmetric polynomials [34]. The equivalence relations corresponding to the SSEs can be computed by comparing these hashes.

The complex schema element $CSE = (\sim^s, \sim^p, \sim^o)$ (Definition 7) summarizes vertices based on the equivalence relations $\sim^s$, $\sim^p$, and $\sim^o$ which are, in turn, defined by simple or complex schema elements. We first compute the vertex summaries for these equivalence relations. For each vertex $v \in G$ and for each equivalence relation, we store the primary vertex identifier of the vertex summary that summarizes v. We now compute the vertex summary for CSE in essentially the same way as we compute POC but, instead of vertex identifiers, we use primary vertices of corresponding vertex summaries. That is, for each vertex s, we construct the set $\{(r_p, r_o) \mid (s, p, o) \in G\}$, where r_p and r_o denote the primary vertices of the vertex summaries for p and o under the equivalences $\sim^p$ and $\sim^o$, respectively. Again, we hash this set by combining the hashes of its elements. The equivalence relation for CSE is now computed by comparing these hashes and checking for equivalence under $\sim^s$. To check whether $s \sim^s s'$, we check whether their vertex summaries have the same primary vertex.

Thus, without parameterizations, a FLUID expression containing λ SSEs and κ CSEs can be evaluated in time and space $\mathcal{O}((\lambda + \kappa) \cdot n)$, on an input graph with n triples.

5.2.2. Label Parameterization

The label parameterization reduces the number of considered objects and/or predicates for each simple schema element by restricting to those in the set P_r. Thus, we compute the parameterized schema element as described in Section 5.2.1 except that, for each triple $(s, p, o) \in G$, we must check whether $p \in P_r$ and/or $o \in P_r$. This requires $\Theta(n)$ membership checks, each of which takes time $\mathcal{O}(|P_r|)$. Note that $|P_r|$ depends only on the parameterized SSE being evaluated and not on the data graph. Thus, with respect to the input data graph, the parameterized SSE is evaluated in time $\mathcal{O}(n \cdot |P_r|)$, which is linear in n. The space cost remains linear in n since applying the parameterization might not change the graph summary that is produced.

5.2.3. Set Parameterization

The set parameterization differs from the label parameterization only in what is done with subject vertices that have predicates and/or objects outside the parameter set S. By the same argument as above, the running time is $\mathcal{O}(n \cdot |S|)$ (where, again, $|S|$ is independent of the input graph) and the space used is linear in n.

5.2.4. Chaining Parameterization

Let $CSE = (\sim^s, \sim^p, \sim^o)$ be a complex schema element. By Definition 10, the chaining parameterization $cp(CSE, k)$ denotes an equivalence relation defined by recursively applying CSE k times. This is simply k nested CSEs, $(\sim^s, \sim^p, (\sim^s, \sim^p, (\cdots(\sim^s, \sim^p, \sim^o)\cdots)))$, which can be evaluated in time and space $\mathcal{O}(k \cdot n)$.

5.2.5. Direction Parameterization

With the direction parameterization, incoming and/or outgoing triples can be used (Definition 11). Any incoming predicate of a vertex v is the outgoing property of another vertex v'. Thus, for bidirectional schema elements b-EQR, we may have to consider each triple twice. However, this is still linear w.r.t. to the number n of triples in the data graph.

5.2.6. Inference Parameterization

For the inference parameterization, we have to consider the space and time required to build both the summary graph SG and the vocabulary graph VG. As defined in Section 4.2.5, the vocabulary graph is constructed by adding types and properties to it, if there exists a triple in G using a predicate in P_{RDFS} (Algorithm 1, Line 15). In the input data graph G of size n, we find r schema triples that use a property in P_{RDFS}, with $r \leq n$. Constructing the vocabulary graph is done analogously by updating hash maps. Thus, constructing the vocabulary graph changes neither the build-time nor the space complexity.

However, inferring information can change the overall complexity. For our analysis, we distinguish the two cases of adding inferrable information *inside* the structural graph summary and adding inferrable information *outside* of the structural graph summary (see Section 4.2.5).

The first case of inferencing inside can have a big impact on the build-time. The complexity depends on the number t of additional triples that can be inferred for each triple in G from the r triples in the vocabulary graph with $t \leq r$. Thus, applying the inference parameterization to a FLUID definition using λ SSEs and κ CSEs and no other parameterizations defines a graph summary model that can be computed in time and space $\Theta((\lambda + \kappa) \cdot n \cdot t)$.

Note that t may itself be a function of n, in which case $t = \mathcal{O}(n)$. This means that, when using the inference parameterization, we have in the worst case running time and space usage $\mathcal{O}(n^2)$ for the case of inference inside.

Example 9. Consider an RDF graph with the triple set

$$\begin{aligned}&\{(s_1, p_1, o_1), (s_2, p_1, o_2), \ldots, (s_{n/2}, p_1, o_{n/2}),\\ &\ (p_1, \texttt{rdfs:subPropertyOf}, p_2),\\ &\ (p_2, \texttt{rdfs:subPropertyOf}, p_3),\\ &\ \vdots\\ &\ (p_{n/2-1}, \texttt{rdfs:subPropertyOf}, p_{n/2})\}\,,\end{aligned}$$

where we assume n to be even. Inference on this graph results in the graph containing the $n^2/4$ triples $\{(s_i, p_j, o_i) \mid 1 \leq i, j \leq n/2\}$. This shows that inference can produce a quadratic blow-up in the number of tuples in a graph.

In the subsequent section, we analyze large real-world semantic graphs obtained from the Linked Open Data cloud. Our analysis suggests that adding inferrable information inside the structural graph summary may be infeasible in a practical setting. In this case, adding inferrable information should be done outside of the structural graph summary.

The vocabulary graph can be implemented using hash maps, which guarantees amortized constant time for lookup and addition operations. Inference operations are linear in the number of inferrable types and properties. For static vocabulary graphs, caching can be used to improve the run time. Thus, we have the same time complexity as for the space complexity for the case of inference inside. For the second case of inference outside the structural graph summary, space and build-time complexity remain unchanged.

5.2.7. Instance Parameterization

The instance parameterization aggregates vertices to unions of vertices, e. g., SameAs instances or related property instances. The instance parameterization does not increase the space complexity since no new triples are added to the data graph. This contrasts with the inference parameterization, which potentially adds new triples. The instance parameterization can decrease the number of different vertex summaries in the graph summary.

As described in Section 5.1, two vertices can be aggregated using the instance parameterization, in principal, in amortized constant time using hash maps (Algorithm 1, Lines 20 to 23). To handle transitivity, we have to recursively access the hash map SameAsInstanceMap (for SameAs instances) or SrcRelatedMap and TrgRelatedMap (for related property instances). Transitivity means here that, for each equivalent vertex v' of some vertex v, we use the hash maps to look up the set of equivalent vertices of v'. The transitive closure only needs to be computed once.

In the worst case, every vertex is equivalent to each other. In this case, we have $\mathcal{O}(n)$ lookup operations for the first vertex v. Since all vertices are equivalent, the computation terminates. For each vertex v that is aggregated using the instance parameterization, one fewer vertex summary needs to be computed. Thus, the time complexity also does not increase.

5.3. Summary

Graph summaries defined with FLUID can be computed in linear time and space with respect to the number of triples n, unless the inference parameterization is used, in which case the running time and space usage may be quadratic in n. For the inference parameterization, we run our own analyses on large real-world datasets to justify the assumption of essential linear runtime. This analysis on very large semantic graphs appears in the next section.

6. Empirical Analysis of the Inference Parameterization

To estimate the impact of RDF Schema inference in a practical setting, we analyze four large semantic graphs obtained from the Linked Open Data cloud. The graphs have different characteristics resulting from different crawling strategies. In particular, we selected graphs with different sizes (number of triples n), obtained in different years, and containing multiple different data providers, e. g., DBpedia, Wikidata, and others. Note that we did not perform any pre-processing except removing invalid (not parsable) triples.

6.1. Datasets

The first dataset is called TimBL-11M and contains about 11 million triples [27]. The crawl was conducted in 2011 with a breadth-first search starting from the single URI of Tim Berners-Lee's FOAF profile. The second dataset is called DyLDO-127M and contains about 127 million triples [24]. The Dynamic Linked Data Observatory (DyLDO) provides regular snapshots from the Linked Open Data cloud. We use their first snapshot crawled in May 2012 starting from about $95,000$ seed URIs. This crawl was done with a breadth-first search but limited to a crawling depth of two [24]. Although there are more recent DyLDO snapshots, they are decreasing in size. Thus, we decided to take the first and largest one. The third dataset is the Billion Triple Challenge 2019 dataset (BTC-2B), which contains about 2 billion triples [22]. The BTC-2B dataset was crawled breadth-first in January 2019 starting from 450 seed URIs taken from the DyLDO dataset [22]. To the best of our knowledge, the largest collections of Linked Open Data is the LOD Laundromat dataset[5] (Laundromat-38B) [37]. It contains more than 38 billion triples and combines various other data sources into one single dataset.

6.2. Procedure

We analyzed the four datasets with respect to RDF Schema inference. In the top half of Table 4, we present the size of the computed vocabulary graph VG_{RDFS} for each dataset. Furthermore, we distinguish between property vertices, i. e., vertices representing a property p used in the data graph G, and type vertices. As described in Section 4.2.5, properties and types used in the data graph G are only added to the vocabulary graph VG_{RDFS} if there is a triple about this property or type with a property $p_r \in P_{\mathrm{RDFS}}$. For each such property p_r, we also counted the number of (hierarchical) relationships expressed, namely for `rdfs:subPropertyOf`, `rdfs:subClassOf`, `rdfs:domain`, and `rdfs:range`.

In the lower half of the table, we present data on the impact of RDFS inference on each dataset. First, we counted the number of properties p and the number of types t in the datasets and distinguished whether they are represented as vertex in the vocabulary graph VG_{RDFS}, i. e., have inferrable information, or not. Note that we counted how many triples use the property p as predicate, i. e., (s, p, o), or label a vertex with the type t using `rdf:type` as predicate, i. e., $(s, \texttt{rdf:type}, t)$. This way, we can use these numbers to estimate an upper bound for the size of the structural graph summary. Finally, we counted the number of additional properties and types that can be added to the dataset based on the inference rules in VG_{RDFS}. The increase factor reflects for properties, types, and both, how much additional information is added. These factors denote the influence of the inference parameterization on the build-time and space complexity of structural graph summaries for the analyzed datasets. Note that while the factor denotes the increased build-time, the storage space for the structural graph summary might be significantly lower. As described in Section 2.2, structural graph summaries summarize the data graph, i. e., common schema structures are only stored once in the summary graph.

[5]http://lodlaundromat.org/

Table 4: Dataset analysis with rounded number to the nearest thousand T, million M, or billion B.

	TimBL-11M	DyLDO-127M	BTC-2B	Laundromat-38B
RDFS Vocabulary Graph VG_{RDFS}				
Property vertices	12T	31T	26T	330T
`rdfs:subPropertyOf` triples	10T	39T	38T	211T
Type vertices	141T	312T	339T	3.7M
`rdfs:subClassOf` triples	644T	3M	2M	144M
`rdfs:domain` triples	10T	20T	19T	204T
`rdfs:range` triples	10T	19T	16T	196T
Total vertices $\|V\|$	153T	343T	365T	4M
Total triples $\|E\|$	673T (6%)	3M (2%)	2M (<1%)	145M (<1%)
Dataset and Inference				
Properties $p \notin VG_{\text{RDFS}}$	889T (10%)	75M (66%)	98M (5%)	12B (42%)
Properties $p \in VG_{\text{RDFS}}$	8M (90%)	39M (34%)	1.9B (95%)	16B (58%)
Properties in dataset	9M	114M	2.1B	28B
Added through inference	7M	33M	703M	50B
Total number of properties	17M	148M	2.7B	78B
Types $t \notin VG_{\text{RDFS}}$	181T (9%)	2.6M (20%)	20M (22%)	938M (23%)
Types $t \in VG_{\text{RDFS}}$	2M (91%)	10M (80%)	71M (78%)	3B (77%)
Types in dataset	2M	13M	92M	4B
Added through inference	17M	120M	2.4B	646B
Total number of types	19M	134M	2.5B	650B
Increase factor for properties	1.9	1.3	1.3	2.7
Increase factor for types	9.2	10.5	27.5	160.3
Increase factor (total)	**3.3**	**2.2**	**2.5**	**22.2**

6.3. Results

From the results of our analysis (Table 4), we can state that the size of the computed vocabulary graph VG_{RDFS} is only a small fraction of the input data graph G. On average, the vocabulary graph is only about 5% of the data graph in terms of number of triples. In contrast, adding the inferrable information increases the size of the data graph by a factor of between 2 and 20. Notably, the largest dataset (Laundromat-38B) also has by far the largest increase factor.

Another main result is that over all datasets, `rdfs:subClassOf` hierarchies are considerably longer than `rdfs:subPropertyOf` hierarchies. While on average 1.04 (max 304) additional properties can be inferred, on average 13.64 (max 3641) additional types can be inferred. This is also reflected in the number of `rdfs:subClassOf` triples in the vocabulary graphs. Over all four vocabulary graphs, `rdfs:subClassOf` triples make up 95% to 99% of all triples in the vocabulary graph.

Furthermore, we have found that in three out of four datasets, more properties have inferrable information, i. e., appear in VG_{RDFS}, than properties that do not have inferrable information. On average, 70% of the properties used in the dataset have inferrable information. For types, we found that over all four datasets, on average more than 80% have inferrable information.

6.4. Discussion

Our results indicate a considerable impact on the size of the data graph when using RDF Schema inference. For three out of four datasets, the amount of data increases by a factor of about two to three. A notable exception is here the Laundromat-38B dataset. Despite having a comparably small vocabulary graph (less than 1% of data graph size), more than 640 billion additional types can be added. This means, after inference, about 160 times more types appear in the graph. TheLaundromat-38B is the only dataset

in our analysis that is an aggregation of user-uploaded datasets. In contrast, the other datasets are crawled from the Web using different strategies.

Moreover, the vocabulary graphs VG_{RDFS} comprise only a small fraction of the data graph, thus, allowing a compact representation of RDF Schema information. In our analysis, we found that for larger datasets, the size decreases compared to the dataset size. Therefore, it appears practical to add inferrable information outside of the structural graph summary for large semantic graphs, i. e., store structural graph summary and vocabulary graph separately. However, Goasdoué et al. [18] also propose an optimization technique to improve the performance of inference, which improved the time needed to perform inference in graph summaries by up to 94%. As noted in Section 4.2.5, inferring on the data graph and then summarizing is equivalent to summarizing the data graph and then inferring on the graph summary [18, 28].

One should also note that the structural graph summary is designed to be orders of magnitude smaller than the original graph. It remains to be evaluated if the graph summaries grow by the same factor as the dataset, when RDF Schema inference is used. In a previous work, we compared the size of one graph summary with and and without inference [4]. For the structural graph summary that extends SchemEX with RDF Schema and `owl:sameAs`, we have found a notably smaller increase in size. For the TimBL-11M dataset, the structural graph summaries increase on average by a factor of 1.2 (instead of 3.3) and for the DyLDO-127M dataset, the structural graph summaries increase on average by factor of 1.5 (instead of 2.2). Thus, the size of the structural graph summaries does not necessarily grow by the same factor as the data graph grows. But, we assume that the increase factor of the data graph defines an upper bound of the increase factor of the semantic structural graph summary.

7. Conclusion

We have presented FLUID, a common model to define (semantic) structural graph summaries. We defined FLUID's building blocks, i. e., four schema elements and six parameterizations, and demonstrated that they are sufficient to define existing (semantic) structural graph summaries and beyond. Furthermore, we presented our FLUID language, which can be used to define, implement, and evaluate structural graph summaries. Moreover, graph summaries defined with FLUID can be typically computed in time and space $\Theta(n)$ w.r.t. n, the number of triples in the data graph. Expressing (semantic) structural graph summaries with only a handful of elements and parameterizations in the FLUID language allows us to implement a single, parameterized algorithm to compute (semantic) structural graph summaries. This algorithm takes as input a data graph and a graph summary model defined using the FLUID language and returns the computed structural graph summary as output. Our implementation is available under an open-source license.[6] This way, extensions of FLUID in the future are possible. FLUID computes exact structural graph summaries. In the future, we could integrate statistical approaches such as frequent pattern mining [44].

Acknowledgments

This research was co-financed by the EU H2020 project MOVING (http://www.moving-project.eu/) under contract no. 693092.

References

[1] D. Allemang and J. A. Hendler. *Semantic Web for the Working Ontologist - Effective Modeling in RDFS and OWL, Second Edition.* Morgan Kaufmann, 2011. ISBN 978-0-12-385965-5. URL http://www.elsevierdirect.com/product.jsp?isbn=9780123859655.

[2] F. Benedetti, S. Bergamaschi, and L. Po. Exposing the underlying schema of LOD sources. In *Joint IEEE/WIC/ACM WI and IAT*, pages 301–304. IEEE Computer Society, 2015. URL https://doi.org/10.1109/WI-IAT.2015.99.

[3] T. Blume and A. Scherp. Towards flexible indices for distributed graph data: The formal schema-level index model FLuID. In *30th GI-Workshop on Foundations of Databases*, CEUR Workshop Proceedings, pages 23–28. CEUR-WS.org, 2018. URL https://dbs.cs.uni-duesseldorf.de/gvdb2018/wp-content/uploads/2018/05/GvDB2018_paper_4.pdf.

[6] https://github.com/t-blume/fluid-framework

[4] T. Blume and A. Scherp. Indexing data on the web: A comparison of schema-level indices for data search. In *Database and Expert Systems Applications - 31st International Conference, DEXA 2020*, pages 277–286, 2020. URL https://doi.org/10.1007/978-3-030-59051-2_18.

[5] T. Blume, D. Richerby, and A. Scherp. Incremental and parallel computation of structural graph summaries for evolving graphs. In *29th ACM International Conference on Information and Knowledge Management (CIKM)*. ACM, 2020. URL https://doi.org/10.1145/3340531.3411878.

[6] A. Bonifati, S. Dumbrava, and H. Kondylakis. Graph summarization. *CoRR*, abs/2004.14794, 2020. URL https://arxiv.org/abs/2004.14794.

[7] D. Brickley and R. Guha. RDF Schema 1.1. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/, 2014.

[8] S. Campinas, T. Perry, D. Ceccarelli, R. Delbru, and G. Tummarello. Introducing RDF graph summary with application to assisted SPARQL formulation. In *23rd International Workshop on Database and Expert Systems Applications, DEXA 2012*, pages 261–266. IEEE Computer Society, 2012. URL https://doi.org/10.1109/DEXA.2012.38.

[9] Š. Čebirić, F. Goasdoué, and I. Manolescu. A framework for efficient representative summarization of RDF graphs. In *International Semantic Web Conference (Posters, Demos & Industry Tracks)*, volume 1963 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017. URL http://ceur-ws.org/Vol-1963/paper512.pdf.

[10] Š. Čebirić, F. Goasdoué, H. Kondylakis, D. Kotzinos, I. Manolescu, G. Troullinou, and M. Zneika. Summarizing semantic graphs: a survey. *VLDB J.*, 28(3):295–327, 2019. URL https://doi.org/10.1007/s00778-018-0528-3.

[11] M. Ciglan, K. Nørvåg, and L. Hluchý. The semsets model for ad-hoc semantic list search. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012*, pages 131–140, 2012. URL https://doi.org/10.1145/2187836.2187855.

[12] M. P. Consens, V. Fionda, S. Khatchadourian, and G. Pirrò. S+EPPs: Construct and explore bisimulation summaries, plus optimize navigational queries; all on existing SPARQL systems. *PVLDB*, 8(12):2028–2031, 2015. URL http://www.vldb.org/pvldb/vol8/p2028-consens.pdf.

[13] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts and Abstract Syntax. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225, 2014.

[14] L. Ding, J. Shinavier, Z. Shangguan, and D. L. McGuinness. SameAs networks and beyond: Analyzing deployment status and implications of owl:sameAs in Linked Data. In *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010*, pages 145–160. Springer, 2010. URL https://doi.org/10.1007/978-3-642-17746-0_10.

[15] M. Dürst and M. Suignard. RFC 3987 internationalized resource identifiers (IRIs). https://www.ietf.org/rfc/rfc3987.txt, 2005.

[16] European Mathematical Society. Equivalence relation, 2014. URL http://www.encyclopediaofmath.org/index.php?title=Equivalence_relation&oldid=35990.

[17] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *ACM SIGMOD International Conference on Management of Data, SIGMOD 2011*, pages 925–936. ACM, 2011. URL https://doi.org/10.1145/1989323.1989420.

[18] F. Goasdoué, P. Guzewicz, and I. Manolescu. RDF graph summarization for first-sight structure discovery. *VLDB J.*, 29 (5):1191–1218, 2020. URL https://doi.org/10.1007/s00778-020-00611-y.

[19] T. Gottron, A. Scherp, B. Krayer, and A. Peters. LODatio: using a schema-level index to support users in finding relevant sources of linked data. In *7th International Conference on Knowledge Capture, K-CAP 2013*, pages 105–108. ACM, 2013. URL https://doi.org/10.1145/2479832.2479841.

[20] T. Gottron, M. Knauf, and A. Scherp. Analysis of schema structures in the linked open data graph based on unique subject uris, pay-level domains, and vocabulary usage. *Distributed Parallel Databases*, 33(4):515–553, 2015. URL https://doi.org/10.1007/s10619-014-7143-0.

[21] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007*, volume 4825 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2007. URL https://doi.org/10.1007/978-3-540-76298-0_16.

[22] J. Herrera, A. Hogan, and T. Käfer. BTC-2019: the 2019 billion triple challenge dataset. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference*, pages 163–180, 2019. URL https://doi.org/10.1007/978-3-030-30796-7_11.

[23] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, J. E. L. Gayo, S. Kirrane, S. Neumaier, A. Polleres, R. Navigli, A. N. Ngomo, S. M. Rashid, A. Rula, L. Schmelzeisen, J. F. Sequeda, S. Staab, and A. Zimmermann. Knowledge graphs. *CoRR*, abs/2003.02320, 2020. URL https://arxiv.org/abs/2003.02320.

[24] T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan. Observing linked data dynamics. In *The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013*, volume 7882 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 2013. URL https://doi.org/10.1007/978-3-642-38288-8_15.

[25] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *18th International Conference on Data Engineering*, pages 129–140. IEEE Computer Society, 2002. URL https://doi.org/10.1109/ICDE.2002.994703.

[26] A. Khan, S. S. Bhowmick, and F. Bonchi. Summarizing static and dynamic big graphs. *VLDB Endowment*, 10(12): 1981–1984, 2017. URL http://www.vldb.org/pvldb/vol10/p1981-khan.pdf.

[27] M. Konrath, T. Gottron, S. Staab, and A. Scherp. SchemEX - efficient construction of a data catalogue by stream-based indexing of linked data. *J. Web Semant.*, 16:52–58, 2012. URL https://doi.org/10.1016/j.websem.2012.06.002.

[28] T. Liebig, V. Vialard, M. Opitz, and S. Metzl. Graphscale: Adding expressive reasoning to semantic data stores. In *ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC)*, volume 1486 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015. URL http://ceur-ws.org/Vol-1486/paper_117.pdf.

[29] Y. Liu, T. Safavi, A. Dighe, and D. Koutra. Graph summarization methods and applications: A survey. *ACM Comput. Surv.*, 51(3):62:1–62:34, 2018. URL https://doi.org/10.1145/3186727.

[30] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language. https://www.w3.org/TR/2004/REC-owl-features-20040210/, 2014.

[31] N. Mihindukulasooriya, M. Poveda-Villalón, R. García-Castro, and A. Gómez-Pérez. Loupe - an online tool for inspecting datasets in the linked data cloud. In *International Semantic Web Conference (Posters & Demos)*, volume 1486 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015. URL http://ceur-ws.org/Vol-1486/paper_113.pdf.

[32] T. Milo and D. Suciu. Index structures for path expressions. In C. Beeri and P. Buneman, editors, *Database Theory - ICDT '99, 7th International Conference*, volume 1540 of *Lecture Notes in Computer Science*, pages 277–295. Springer, 1999. URL https://doi.org/10.1007/3-540-49257-7_18.

[33] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *27th International Conference on Data Engineering, ICDE 2011*, pages 984–994. IEEE, 2011. URL https://doi.org/10.1109/ICDE.2011.5767868.

[34] R. O'Keefe. How to hash a set. 2017. URL https://www.preprints.org/manuscript/201710.0192.

[35] E. Pietriga, H. Gözükan, C. Appert, M. Destandau, S. Cebiric, F. Goasdoué, and I. Manolescu. Browsing linked data catalogs with lodatlas. In *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference*, pages 137–153, 2018. URL https://doi.org/10.1007/978-3-030-00668-6_9.

[36] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *ACM SIGMOD International Conference on Management of Data*, pages 134–144. ACM, 2003. URL https://doi.org/10.1145/872757.872776.

[37] L. Rietveld, W. Beek, R. Hoekstra, and S. Schlobach. Meta-data for a lot of LOD. *Semantic Web*, 8(6):1067–1080, 2017. URL https://doi.org/10.3233/SW-170256.

[38] D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, 2009. URL https://doi.org/10.1145/1516507.1516510.

[39] J. Schaible, T. Gottron, and A. Scherp. TermPicker: Enabling the reuse of vocabulary terms by exploiting data from the Linked Open Data cloud. In *he Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016*, volume 9678 of *Lecture Notes in Computer Science*, pages 101–117. Springer, 2016. URL https://doi.org/10.1007/978-3-319-34129-3_7.

[40] A. Schätzle, A. Neu, G. Lausen, and M. Przyjaciel-Zablocki. Large-scale bisimulation of RDF graphs. In *Fifth Workshop on Semantic Web Information Management, SWIM@SIGMOD Conference 2013*, pages 1:1–1:8. ACM, 2013. URL https://doi.org/10.1145/2484712.2484713.

[41] B. Spahiu, R. Porrini, M. Palmonari, A. Rula, and A. Maurino. ABSTAT: ontology-driven linked data summaries with pattern minimalization. In *The Semantic Web - ESWC 2016 Satellite Event, Revised Selected Papers*, volume 9989 of *Lecture Notes in Computer Science*, pages 381–395, 2016. URL https://doi.org/10.1007/978-3-319-47602-5_51.

[42] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984. URL http://doi.acm.org/10.1145/62.2160.

[43] T. Tran, G. Ladwig, and S. Rudolph. Managing structured and semistructured RDF data using structure indexes. *IEEE Trans. Knowl. Data Eng.*, 25(9):2076–2089, 2013. URL https://doi.org/10.1109/TKDE.2012.134.

[44] J. Völker and M. Niepert. Statistical schema induction. In *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011*, pages 124–138, 2011. URL https://doi.org/10.1007/978-3-642-21034-1_9.

[45] A. Zimmermann. RDF 1.1: On Semantics of RDF Datasets. https://www.w3.org/TR/2014/NOTE-rdf11-datasets-20140225/, 2014.