Blume, Till; Scherp, Ansgar

Conference Paper

Towards an Incremental Schema-level Index for Distributed Linked Open Data Graphs

Suggested Citation: Blume, Till; Scherp, Ansgar (2018) : Towards an Incremental Schema-level Index for Distributed Linked Open Data Graphs, In: Proceedings of the Conference "Lernen, Wissen, Daten, Analysen", LWDA 2018, Mannheim, Germany, August 22-24, 2018, RWTH, Aachen

This Version is available at:
http://hdl.handle.net/11108/374

Towards an Incremental Schema-level Index for Distributed Linked Open Data Graphs

Till Blume[1,2] and Ansgar Scherp[3]

[1] Kiel University, Germany
[2] ZBW – Leibniz Information Centre for Economics, Kiel, Germany
[3] University of Stirling, Scotland UK
tbl@informatik.uni-kiel.de
ansgar.scherp@stir.ac.uk

Abstract Semi-structured, schema-free data formats are used in many applications because their flexibility enables simple data exchange. Especially graph data formats like RDF have become well established in the Web of Data. For the Web of Data, it is known that data instances are not only added, changed, and removed regularly, but that their schemas are also subject to enormous changes over time. Unfortunately, the collection, indexing, and analysis of the evolution of data schemas on the web is still in its infancy. To enable a detailed analysis of the evolution of Linked Open Data, we lay the foundation for the implementation of incremental schema-level indices for the Web of Data. Unlike existing schema-level indices, incremental schema-level indices have an efficient update mechanism to avoid costly recomputations of the entire index. This enables us to monitor changes to data instances at schema-level, trace changes, and ultimately provide an always up-to-date schema-level index for the Web of Data. In this paper, we analyze in detail the challenges of updating arbitrary schema-level indices for the Web of Data. To this end, we extend our previously developed meta model FLuID. In addition, we outline an algorithm for performing the updates.

Keywords: Incremental schema-level index · Schema computation · LOD

1 Introduction

The Web of Data comprises a huge amount of interlinked data instances (Linked Data) in a standard format, e. g., RDF. However, since the Web of Data is by its nature a decentralized database, without a central authority responsible for data management, different data schemas are used. In addition, data instances on the Web are not just regularly added, modified, and removed [10], but their schema is also subject to enormous changes over time [4]. Consequently, data-driven applications that aim to make use of the Web of Data render useless if they rely on, e. g., outdated data caches [3]. The analysis of the evolution of data and data schemas can bring enormous advantages for the understanding of the data [13] and help to keep local copies of the data up-to-date [3]. Many research works

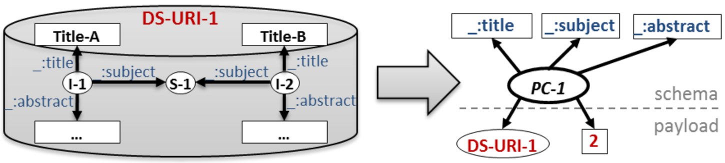

Figure 1. Two instances (I-1,I-2) using the same three properties can be summarized by one Property Cluster (PC-1). PC-1 provides schema information (properties) and payload information, e. g., their data source URI and number of summarized instances.

analyze the (co-)evolution of data [5, 9, 14], but few analyze the (co-)evolution of Linked Open Data to understand dependencies of change behavior such as frequently used vocabularies, network links, and other (co-)evolution patterns of data instances [4,13]. In order to perform large scale analyses on the Web of Data, efficient access to the dynamic data is required, with incremental indices playing a decisive role. Efficient access to data on the Web can be achieved with crawling and indexing. Crawling the data produces a data stream that can only be indexed with an incremental algorithm, which needs to provide efficient updates of the index instead of costly re-computations of the entire index.

So far, there exist only incremental instance-level indices but no incremental schema-level indices. Instance-level indices allow to quickly retrieve nodes or answer questions about reachability, distance, or shortest path [16]. In contrast, schema level indices (SLI) index the schema computed from the data [2, 6]. For example, if there are instances in a data source, each described with three properties (see Fig. 1), an SLI can summarize these instances. An SLI stores only the combination of the three properties (schema information) and links them to the corresponding data source (payload information). The payload is needed to use an SLI in a concrete application context such as for data search [7], where the aim is to find data sources relevant to a user's query. There are numerous different variations of SLIs that index different schema information and store different payload information [2, 6]. However, there is no incremental schema-level index that allows the analysis of the evolution of data schemas on the Web.

In the related field of schema discovery in NoSQL databases, there are incremental algorithms to discover hidden schemas in the data [1,18]. However, these are limited to document-oriented formats such as JSON and are designed for closed database systems. Therefore, they cannot be applied to an incremental schema-level index for open, distributed graph databases where data instances are stored decentralized. In such an open system like the Web of Data, the index computation must be able to process incomplete data due to the size of the Web and the high dynamics. Let us assume that a data source DS-URI-2 is crawled at a certain point in time after data source DS-URI-1 has been indexed (Fig. 1). If the data source DS-URI-2 contains instances with the same schema as the instances in DS-URI-1, the payload of PC-1 must be updated. The crawler can also visit the data source DS-URI-1 again and provide changed information. This can trigger different types of updates. For example, if only one of two instances in Fig. 1 disappears, we have to update the payload with the new number of summarized instances (1 Instance). If both instances disappear we must also remove the data source and consider removing the schema element (PC-1) from the SLI since a matching query would not return any payload. If, however, only the instance information changes, e. g., the title of I-2 changes to "Title-C", no update (neither on schema nor on payload) is required.

For the Web of Data, we make certain assumptions about the data graph. First, we consider the crawling strategy as a black box. This means that all data sources in the Web of Data are (re-)visited eventually by the crawler at an unknown point in time. Second, crawler download the entire data source for a given data source URI. Thus, we assume that all statements fetched via one data source URI come next to each other in the data stream. Third, each data source can contain statements about instances stored in other data sources. Our experiments suggest that almost one third of the instances are distributively defined. Thus, instances can be defined in a truly distributed setting.

With this work, we want to lay the foundation to close the gap between continuously crawling data from the Web and analyzing the data schemas. We do not want to develop a customized incremental SLI, but rather create the basis for updates to any SLI. To this end, we examine all possible update operations on SLIs. For this purpose, we extend our previously developed formal model FLuID to describe and implement any SLI using only 11 building blocks [2]. For updates at schema level, we need to consider several cases: Instances with new schemas are observed (SE_{new}), instances with a known schema are added (PE_{add}), schemas of known instances are changed (SE_{mod}), not all instances with a known schema are deleted (PE_{del}), and all instances with a known schema are deleted (SE_{del}).

The remainder is structured as follows: In Sect. 2, we discuss related works. In Sect. 3, we outline FLuID's building blocks and present how SLIs can be computed for a given data graph. Subsequently, we analyze the problem of updating SLIs in accordance with the FLuID model in Sect. 4. Finally, we give a brief outlook on our incremental algorithm, before we conclude the paper.

2 Related Work

The approaches for indexing graph data can be divided in instance-level indices and schema-level indices. Instance-level indices store information about the actual data instances and their statements [8]. They support queries for specific resources such as finding all persons with the surname "Müller". Schema-level indices provide a concise summary of the data instances by memorizing the schema defined by the instances' statements. Schema-level indices satisfy information needs like "Find all data sources with information about persons who are actors and American presidents" [2, 6]. There exist various variants of schema-level indices (SLIs) [2,6,17], which greatly differ in what kind of schema is captured. For example, TermPicker [17] summarizes instances based on a common set of types, a common set of properties, and a common set of types of all property-linked resources. However, there exists no incremental schema-level index. Therefore, we discuss existing incremental instance-level indices and discuss incremental schema discovery algorithms in NoSQL databases.

There are few incremental instance-level indices that allow adding new data instances on the fly and seamlessly incorporating it into the the index [19, 20]. Yuan et. al. [19] propose a subgraph mining algorithm to mine frequent and discriminative features in subgraphs in order to optimize a computed index and regroup subgraphs based on newly added features. Thereby the algorithm minimizes the number of index lookups for a given type of query. The idea of subgraph mining for instance-level indices has been further improved by various works [11, 20]. Some works investigate the efficient subgraph matching problem on unlabeled and undirected graphs [15]. Qiao et al. [15] propose to compress the data graph G using a pattern graph P by extracting isomorphic sub-graphs. They propose interesting techniques and address the problem of graphs too large for the main memory. However, it needs to be evaluated whether they can also be applied on labeled directed graphs such as the Web of Data.

Wang et al. [18] propose a schema management framework for JSON document stores. They propose to store the schema in tree-like structures, which can be computed using each instance only once. However, their approach has several limitations. First, they assume that instances are always complete when the schema is computed. Second, their approach is only applicable for schema structures that consider properties used by the specific instance only and no information beyond that is needed. Baazizi et al. [1] compute schemas for JSON documents to present information about optional fields and mandatory fields. However, incrementally discovering hidden schemas is not the same as computing exact schemas given by the data, they are limited to document orientated formats like JSON, and are designed for closed database systems.

In summary, all related approaches assume that the data is accessible at any time and that data instances are defined in a single data source, i. e., are not distributed. The only approach handling decentralized graph data is proposed by Konrath et al. [12]. The authors assume that the data is provided by a crawler and apply the idea of a stream-based index computation over a window of the data. Their computation approach, however, does not support incremental updates.

3 Basic building blocks of Schema-level Indices

In our previous work, we developed the FLuID model, which is able to define arbitrary schema-level indices as combination of equivalence relations [2]. Formally, a schema-level index is a 3-tuple $(G, \mathrm{EQR}, \mathrm{PAY})$, where G is the data graph which is indexed, EQR is an equivalence relation over instances in G, and PAY is an n-tuple of payload functions, which map instance information to equivalence classes in EQR. These equivalence relations can be defined using paramaterized simple and complex schema elements [2]. They basically define how the schema is computed from the data graph, e. g., which types and properties are taken into account. In the following, we introduce common notions, define the data graph, and highlight the relevant aspects of FLuID's schema elements, their parameterizations, and the payload elements [2].

A data graph G is defined by $G \subseteq V_{UB} \times P \times (V_{UB} \cup L)$, where V_{UB} denotes the set of URIs and blank nodes, P the set of properties, and L the set of literals. A triple is a statement about a resource $s \in V_{UB} \cup P$ in the form of a subject-predicate-object expression $(s, p, o) \in G$. We define instances $I_s \subseteq G$ to be a set of triples, where each triple shares a common subject URI s. The properties P can be divided into disjoint subsets $P = P_{type} \dot{\cup} P_{rel}$, where P_{type} contains the properties denoting type information and P_{rel} contains the properties linking instances in the data graph. We say the instance I_s with the subject URI s is of type c if there exists a triple $(s, p_t, c) \in G$, with $p_t \in P_{type}$. In the context of RDF, P_{type} contains only *rdf:type* and P_{rel} all $p \in P \setminus P_{type}$. Furthermore, we denote with $\delta^+(I_s)$ the outdegree of an instance I_s and with $\delta^-(I_s)$ the indegree.

3.1 Index Definition with FluID

Below, we briefly describe FLuID's building blocks and subsequently describe, how FLuID-indices can be computed.

Simple Schema Elements summarize instances $I_1 \in G$ and $I_2 \in G$ by comparing all triples $(s_1, p_1, o_1) \in I_1$ to the triples $(s_2, p_2, o_2) \in I_2$. For simple schema elements, we distinguish property cluster (PC), object cluster (OC), and property-object cluster (POC). Each simple schema element compares triples following a different pattern, i. e., comparing only the properties, only the objects, or the combination of both. Furthermore, there are undirected versions of the three simple schema elements, where additionally the incoming triples $(x, p, i) \in G$ with i as the subject of the instance in object position are considered.

Complex Schema Elements partition the data graph by summarizing instances based on the three given equivalence relations $\sim^s$, $\sim^p$, and $\sim^o$. Therefore, they can be defined as 3-tuple $CSE := (\sim^s, \sim^p, \sim^o)$. While the simple schema elements summarize instances by comparing triples using the identity equivalence "=", the complex schema elements compare triples using arbitrary equivalences $\sim^s$, $\sim^p$, and $\sim^o$, e. g., simple schema elements. Thus, they can be considered as containers to combine any number of simple schema elements.

Parameterizations further specify our schema elements. There are four parameterizations defined in FLuID. Chaining parameterization determines the size of the considered sub-graph of a complex schema element CSE of up to k-hops, and is denoted by CSE_k. Label parameterization allows restricting the SLI to consider only specific properties and can be used to define type cluster OC_{type}, where the object cluster only compares objects connected over the properties in P_{type}. Inference parameterization Φ is applied on the data graph G and enables ontology reasoning using a schema graph SG. In practice this means that a schema graph SG_{RDFS} is constructed on-the-fly (or in a pre-processing step), which stores all hierarchical dependencies between types and properties denoted by RDFS properties found in the data graph G. Instance parameterization σ allows merging equivalent instances, e. g., instances linked with *owl:sameAs*.

The label, inference, and instance parameterizations pose further restrictions or relaxations on the comparison of triples when computing the schema elements. Thus, they change the number of considered triples for each schema element. For example, the label parameterization allows ignoring a certain set of properties to determine the equivalence of two instances. The chaining parameterizations, however, affects the number of neighboring instances that also need to be equivalent according to the complex schema element. Two instances I_1 and I_2 are considered equivalent by CSE_k, if for each neighboring instances of I_1 in the data graph for each distance up to k, there is a neighboring instance of I_2 in the same distance that is considered equivalent by CSE.

Payload Elements are attached to schema elements and contain information about the summarized instances, e. g., their data source or the number of summarized instances. The payload is needed to implement a concrete application, e. g., a search engine [7]. For each type of payload, a mapping function needs to be defined. One such function is the data source mapping function ds, which maps a schema element to the data sources of the summarized instances. The function ds takes a schema element EQR as input and returns all sources, with $ds(EQR) := \bigcup_{I \in EQR} context(I)$, with the function $context : \mathcal{P}(G) \rightarrow \mathcal{P}(V_U)$ returning all data sources of an instance I.

3.2 Index Construction with FLuID

In this section, we present our approach to compute schema-level indices modeled with FLuID. Susequently, we extend this to an incremental indexing approach.

Computing an SLI can be described as a function $SC : (G, \text{EQR}, \text{PAY}) \rightarrow sli$, which computes according to the equivalence relations given in EQR, and the n payload functions PAY the concrete schema-level index sli for a data graph G. Since the equivalence relations given in EQR can be defined with schema elements, we can treat EQR as a tuple of schema elements. In the following, we denote schema elements in EQR as abstract schema elements. For example, the SLI TermPicker [17] defines the schema structure that summarizes instances that have a common set of types, a common set of properties, and a common set of combined types of all neighboring instances. With the help of FLuID, this can be expressed with four abstract schema elements:

$$\text{EQR}_{\text{TermPicker}} := (OC_{type} \cap PC_{rel}, T, OC_{type}), \qquad (1)$$

A complex schema element wraps the label parameterized object cluster using the set P_{type} denoted as OC_{type}, the label parameterized property cluster using the set P_{rel} denoted as PC_{rel}, and an arbitrary tautology denote as T, which considers everything equivalent.

To compute a concrete schema-level index sli, the abstract schema elements are used as “blueprints” to compute (instantiate) schema elements for the instances

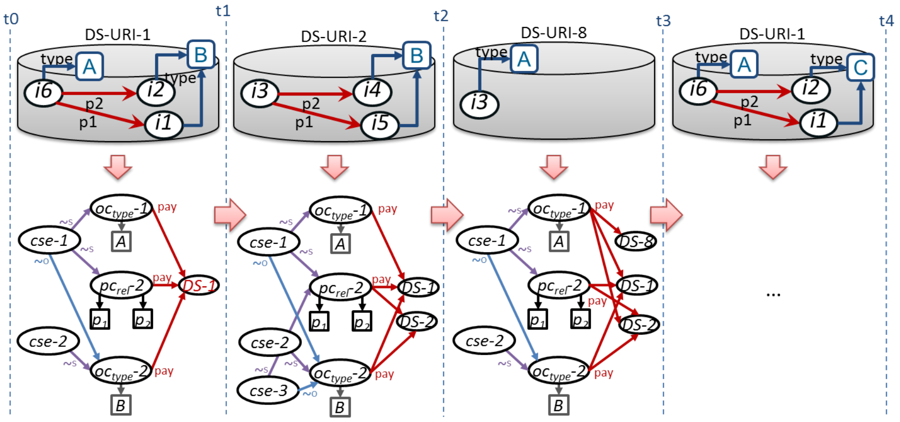

Figure 2. Computing and updating a schema-level index over time: $[t0, t1]$: initially computing *sli* for DS-URI-1. $[t1, t3]$: adding new information from DS-URI-2 and DS-URI-8 to *sli*. $[t3, t4]$: updating the type information in DS-URI-1.

in the data graph G. This can, for example, be done, by extracting all properties of an instance to form a property cluster. For instances using the same set of properties, the same schema element is computed. More specifically, instantiating schema elements means creating a schema element $se\text{-}i \in sli$ with a certain set of specific attributes, e. g., the three properties {_:title, _:subject, _:abstract}.

There is a significant difference between computing simple schema elements and computing complex schema elements. The FLuID model defines simple schema elements as low-level building blocks that summarize instances by comparing triples using the identity equivalence ($=$) for properties and/or objects. Thus, simple schema elements can always be computed directly from the data without dependencies to other schema elements. In contrast, complex schema elements are deliberately designed to have dependencies either on complex schema elements or on simple schema element. To explain this in more detail, we look at the example shown in Fig. 2. When we start with an empty *sli* (interval $[t0, t1]$), to compute the schema of instances, we first need to compute the sub-schema structures (i. e., the simple schema elements oc_{type}-1, oc_{type}-2, pc_{rel}-1) according to the abstract schema definition (blueprint) given in Eq. (1). We can compute the complex schema elements *cse*-1 and *cse*-2 based on the simple schema elements. To compute the complex schema element for instance i6, we need to know the type information about instances i2 and i1 (oc_{type}-2).

The advantage of this feature is that with increasing complexity of the abstract schema definition, the computation costs increase only linearly since the index reuses already computed simple schema elements for several complex schema elements [2]. In general this means, we have to instantiate at most all abstract schema elements for each instance $I_1 \in G$. We denote with ς the number of abstract simple schema elements and with τ the number of abstract complex schema elements. Please note that the chaining parameterization is defined so that the pattern of an abstract complex schema element is applied recursively k times. When chaining a CSE k times, comparable to unrolling a loop, we can simply assume that we have to calculate $k \times \tau$ many abstract complex schema elements. However, no new abstract simple schema element is required, so the number ς is not affected.

The payload information needs to be computed for a data graph G according to the payload functions PAY. Computing the payload elements follows the same principal as computing the schema elements. Thus, we omit the details here. Please note, in the following, we consider a computed *sli* for a data graph G again as a graph. However, other data structures can be used as well.

To summarize, the total number of instances in G can be bounded by the number of nodes in the data graph G, with $|G| = v$. When computing an schema-level index, we have to compute $\alpha \leq \varsigma \times v$ many simple schema elements and $\beta \leq \tau \times v$ many complex schema elements. Furthermore, we need to consider p many payload mapping functions instantiating $\gamma \leq p \times v$ many payload elements for the data graph G. Thus, we know that $\alpha + \beta \leq v \times (\varsigma + \tau \times k)$ schema elements are instantiated in the final schema-level index. With a data graph where no two instances can be summarized, we end up with $\alpha + \beta = v \times (\varsigma + \tau \times k)$.

4 Update Operations on an Index built with FLuID

As described in the previous sections, computing an SLI means extracting schema and payload information from the data to compute schema and payload elements. In the following, we analyze the different cases of updates with respect to the expected complexity. We compare the costs implied with our incremental index to computing the index from scratch.

4.1 Updating Schema Elements

There are six cases of updates possible for an SLI: a new instance is observed with a new schema (SE_{new}), a new instance is observed with a known schema (PE_{add}), a known instance is observed with a changed schema (SE_{mod}), a known instance is observed with only changed instance information (PE_{mod}), a instance with its schema and payload information no longer exists (PE_{del}), and no more instance with a specific schema exists in the data graph (SE_{del}). Please note, we denote with the schema of an instance the complete schema defined by all abstract schema elements in the index definition, e. g., the complex schema element. Subschemas are smaller parts of the schema, e. g., simple schema elements combined within a complex schema element.

New Data Instances Adding an instance to G means adding a set of triples (s_1, p_1, s_2) about a resource s_1 when there where no triple about this resource in G before, neither with s_1 as subject nor with s_1 as object. Due to our blackbox crawling scenario, we cannot assume any order in the data. Thus, we may obeserve statements with s_1 as object before we observe statements with s_1 as subject. In this case, we consider s_1 as instance with an empty schema and no payload.

For new instances in G, we consider two cases. First, if the instance is observed with an entirely new schema (SE_{new}), all $\varsigma + \tau \times k$ (sub-) schema elements need to be computed and added to the index. However, it is possible that some sub-schema elements are already known even if the schema of the instance is new. For example, the specific combination of properties (pc-1) is known but not the specific combination of types (tc-8) and therefore also not the complex combination cse-3 using pc-1 and tc-8. This means, adding an instance with a new schema to G (SE_{new}) can require adding between 1 and $\varsigma + \tau \times k$ schema elements to the index.

Second, if the instance is observed with a known schema (PE_{add}), also all sub-schema elements have to already exist in the index. Therefore, only the payload of existing schema elements may needs to be updated.

Modifying Known Data Instances Modifying instances means adding and/or removing a set of triples $(s_1, p_1, s_2) \in G$ about the resource s_1 when there existed triples about this resource in G before. When the modifications are only on instance-level (PE_{mod}), this requires no change on the schema elements but the payload may needs to be updated. However, the modifications can be on schema-level (SE_{mod}), e. g., new properties are added or types are changed. When we modify an instance on schema-level, analogously to adding an instance, between 1 and $\varsigma + \tau \times k$ schema-elements in the index may need to be updated or created. Furthermore, it is possible that for another instance I_2 a triple $(s_2, p_1, s_1) \in G$ was observed before, where s_1, s_2 are the subject URIs of I_1, I_2 respectively. This means, for each such instance I_2 in G, we may need to update the schema elements as well. Thus, for each incoming edge to I_1 in G, up to τ many complex schema elements require an update. When we apply the chaining parameterization, the individual indegree of each neighbor up to a distance of k hops needs to be considered. To ease notation, we denote with $\delta^-(I_1^{max_k})$ the maximum indegree of all instances within a k-hop distance from instance I_1. We can then simplify the estimation to $\delta^-(I_1^{max_k})^k \times \tau$ effected schema elements. Thus, SE_{mod} requires a maximum of $\varsigma + \tau \times k + \delta^-(I_1^{max_k})^k \times \tau$ updates on schema-level. Since ς and τ are fixed before computing the index, the only variable factor depending on the data is the indegree δ^- of the instances. Please note, for existing SLIs we know that $\tau \leq 1$ and $\varsigma \leq 3$ [2]. Furthermore, our results discussed in Sect. 5 indicate that the indegree is on average only 5.

Deleting Known Data Instances Deletion can be seen as an extreme case of modification where all instance information is deleted, i. e., all triples about the resource. Due to the decentrality of the Web of Data, there can still be links to that instance in the data graph. Thus, the URI of the instance can still appear in the data graph, but only on object position of a triple.

First, the deletion can mean that there are no further data instances in the data graph G with that schema (SE_{del}). This means, we basically have to revert

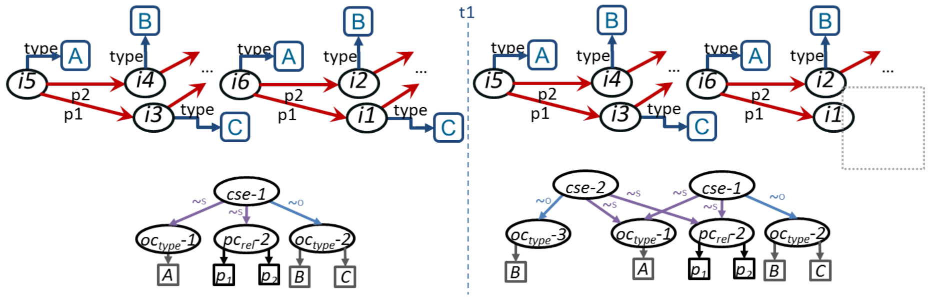

Figure 3. Deletion of the information about instance $i1$ effects the complex schema of the instance $i6$, which has a property linking to $i1$.

all changes that where needed to add a new instance with a new schema (SE_{new}) and we have to delete between 1 and $\varsigma + \tau \times k$ schema elements from the index. Furthermore, deleting single schema elements may require an update of depending schema-elements when (chained) complex schema elements are used analogously to the modification SE_{mod}. Considering Fig. 2, the schema is computed based on types extracted from instances linked via properties. This means, when we remove instance i4 and i5 after $t4$, updates on depending schema elements are necessary. We delete the (sub-) schema elements of cse-2 and also have to update the schema element cse-1 since it can no longer reference oc_{type}-2. Thus, analogously to the modification, we have to consider the indegrees of neighboring instances and possibly update up to $\delta^{-}(I_1^{max_k})^k \times \tau$ complex schema elements.

Second, the deletion can mean that there are still instances in the data graph G with the same schema as the deleted instance (PE_{del}). This means, also all sub-schema elements have to remain in the index. Thus, there is no deletion of schema information required and only the payload of existing schema elements may needs to be updated. For this case, although no schema element is deleted, we may still have to update $\delta^{-}(I_1^{max_k})^k \times \tau$ complex schema elements. Let us consider the example illustrated in Fig. 3. When we compute indices using (chained) complex schema elements, e. g., TermPicker, we do not delete even a sub-schema element but we rather need to add schema elements cse-2 and oc_{type}-3. Thus, up to $\delta^{-}(I_1^{max_k})^k \times \tau + \varsigma$ schema elements need an update.

4.2 Updating Payload Elements

With p many payload functions, up to p new payload-elements need to be instantiated for each new instance I_1. Making general assumptions about the payload is not possible since there can be a unlimited number of different payload functions. Thus, we focus on two often used payload functions, the data source function and the instance count function. Adding or deleting an instance always triggers an update on the instance count payload. Modifying instances does not trigger a change on the instance counts. All three kinds of instance-level change can trigger an update of the data source payload if a new data source is added or an existing instance is moved to another data source.

5 Discussion

As we discussed in the previous sections, to update schema-level indices some information about the data graph needs to be preserved to coordinate the right update operation to the correct schema elements. Thus, we propose to store two kinds of link information, instance URI to schema element and instance URI to data source URI. Furthermore, if complex schema elements are used, we need to memorize a subset of incoming edges for each instance. The amount of storage space required to store all these links for the Web of Data may be undesirably large. We plan to conduct detailed analyses using real-world datasets crawled from the Web of Data. From our analysis in Sect. 4, we know that one major factor is the indegree of instances in the data graph. Our preliminary results for the first four snapshots crawled by the Dynamic Linked Data Observatory (DyLDO)[4] indicate large variety in the link distribution and instance distribution. About 28% of the data instances are split over 2 to $5,800$ (average 2.2) data sources. Furthermore, about one third of the instances have dependencies on 1 to $172,000$ instances. However, on average instances only have dependencies on 5 instances. Storing all links sums up to about 33% of the number of links in the data graph. These first results suggest that implementing an efficient incremental schema-level index using a complex schema element for the Web requires storing significantly more data. Thus, we will also analyze possible optimization and approximation strategies.

6 Conclusion and Outlook

We reviewed existing works on incremental indices and showed that there is a clear gap for incremental schema-level indices. Our analysis of update operations on the FLuID model shows that the update performance of schema-level indices mainly depends on the indegree of changed data instances and on the number of complex schema elements used. All known schema-level indices, however, need only at most one complex schema element. Furthermore, our preliminary results suggest that to update an incremental index for the Web of Data defined with a complex schema element, we need to memorize about an additional 33% of data graph size. A detailed evaluation of such an incremental SLI is needed to measure the impact on the performance when computing indices incrementally.

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

[4] http://km.aifb.kit.edu/projects/dyldo/

References

1. Baazizi, M.A., Ben Lahmar, H., Colazzo, D., Ghelli, G., Sartiani, C.: Schema inference for massive JSON datasets. In: EDBT. pp. 222–233. OpenProceedings.org (2017)
2. Blume, T., Scherp, A.: Towards flexible indices for distributed graph data: The formal schema-level index model FLuID. In: Foundations of Databases. CEUR-WS.org (2018)
3. Dividino, R.Q., Gottron, T., Scherp, A.: Strategies for efficiently keeping local linked open data caches up-to-date. In: ISWC. Springer (2015)
4. Dividino, R.Q., Gottron, T., Scherp, A., Gröner, G.: From changes to dynamics: Dynamics analysis of linked open data sources. In: PROFILES@ESWC. CEUR-WS.org (2014)
5. Fan, Q., Zhang, D., Wu, H., Tan, K.: A general and parallel platform for mining co-movement patterns over large-scale trajectories. PVLDB 10(4), 313–324 (2016)
6. Gómez, S.N., Etcheverry, L., Marotta, A., Consens, M.P.: Findings from two decades of research on schema discovery using a systematic literature review. In: AMW. CEUR-WS.org (2018)
7. Gottron, T., Scherp, A., Krayer, B., Peters, A.: LODatio: using a schema-level index to support users infinding relevant sources of linked data. In: K-CAP. ACM (2013)
8. Hose, K., Schenkel, R., Theobald, M., Weikum, G.: Database foundations for scalable RDF processing. In: Reasoning Web. - Int. Summer School. Springer (2011)
9. Hu, C., Cao, H., Ke, C.: Detecting influence relationships from graphs. In: SDM. SIAM (2014)
10. Käfer, T., Abdelrahman, A., Umbrich, J., O'Byrne, P., Hogan, A.: Observing linked data dynamics. In: ESWC. Springer (2013)
11. Kansal, A., Spezzano, F.: A scalable graph-coarsening based index for dynamic graph databases. In: CIKM. ACM (2017)
12. Konrath, M., Gottron, T., Staab, S., Scherp, A.: SchemEX - efficient construction of a data catalogue by stream-based indexing of Linked Data. J. Web Sem. 16, 52–58 (2012)
13. Nishioka, C., Scherp, A.: Temporal patterns and periodicity of entity dynamics in the linked open data cloud. In: K-CAP. ACM (2015)
14. Ohsaka, N., Akiba, T., Yoshida, Y., Kawarabayashi, K.: Dynamic influence analysis in evolving networks. PVLDB 9(12), 1077–1088 (2016)
15. Qiao, M., Zhang, H., Cheng, H.: Subgraph matching: on compression and computation. PVLDB 11(2), 176–188 (2017)
16. Sakr, S., Al-Naymat, G.: Graph indexing and querying: a review. J. of Web Inf. Sys. 6(2), 101–120 (2010)
17. Schaible, J., Gottron, T., Scherp, A.: TermPicker: Enabling the reuse of vocabulary terms by exploiting data from the Linked Open Data cloud. In: ESWC. Springer (2016)
18. Wang, L., Zhang, S., Shi, J., Jiao, L., Hassanzadeh, O., Zou, J., Wangz, C.: Schema management for document stores. VLDB 8(9), 922–933 (2015)
19. Yuan, D., Mitra, P., Yu, H., Giles, C.L.: Iterative graph feature mining for graph indexing. In: ICDE. IEEE Computer Society (2012)
20. Yuan, D., Mitra, P., Yu, H., Giles, C.L.: Updating graph indices with a one-pass algorithm. In: SIGMOD. ACM (2015)