Incremental and Parallel Computation of Structural Graph Summaries for Evolving Graphs

Till Blume
Kiel University
Germany
tbl@informatik.uni-kiel.de

David Richerby
University of Essex
Colchester, UK
david.richerby@essex.ac.uk

Ansgar Scherp
Ulm University
Germany
ansgar.scherp@uni-ulm.de

ABSTRACT

Graph summarization is the task of finding condensed representations of graphs such that a chosen set of (structural) subgraph features in the graph summary are equivalent to the input graph. Existing graph summarization algorithms are tailored to specific graph summary models, only support one-time batch computation, are designed and implemented for a specific task, or evaluated using static graphs. Our novel, incremental, parallel algorithm addresses all these shortcomings. We support various structural graph summary models defined in our formal language FLUID. All graph summaries defined with FLUID can be updated in time $O(\Delta \cdot d^k)$, where Δ is the number of additions, deletions, and modifications to the input graph, d is its maximum degree, and k is the maximum distance in the subgraphs considered. We empirically evaluate the performance of our algorithm on benchmark and real-world datasets. Our experiments show that, for commonly used summary models and datasets, the incremental summarization algorithm almost always outperforms their batch counterpart, even when about 50% of the graph database changes. The source code and the experimental results are openly available for reproducibility and extensibility.

ACM Reference Format:
Till Blume, David Richerby, and Ansgar Scherp. 2020. Incremental and Parallel Computation of Structural Graph Summaries for Evolving Graphs. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3340531.3411878

1 INTRODUCTION

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 the equivalence of subgraphs. To determine subgraph equivalences, only structural features are used, such as specific labels and data types. 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 [5, 7].

CIKM '20, October 19–23, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3411878

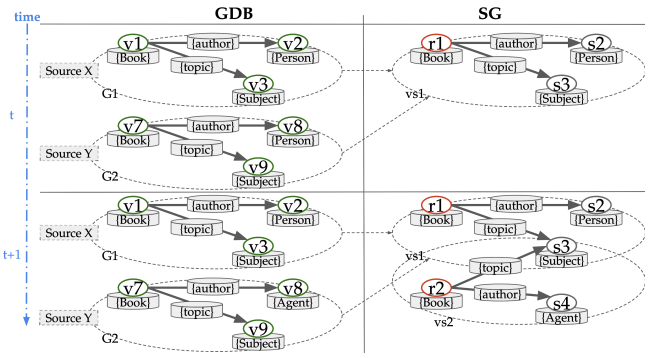


Figure 1: Evolving graph database (GDB) and part of its graph summary (SG) that uses the SchemEX model [18].

Illustrative example: The graph database (GDB) at time t shown in the top-left of Fig. 1 contains two graphs $G1$ and $G2$. Both graphs contain vertices labeled Book, Subject, and Person and edges labeled topic and author. One can define a graph summary (SG) using an equivalence relation $\sim$ that summarizes vertices in the GDB that have the same label and are connected to vertices with the same label by edges with the same label. If two equivalent (sub)graphs are found, the redundant information is removed, giving a condensed representation in SG. This graph summary SG is shown in the top right of Fig. 1. It preserves the information about the combinations of graph label (Book–topic→Subject and Book–author→Person) found in the GDB at time t. Note that the GDB can, in principle, be distributed, e. g., on the Web where $G1$ and $G2$ would be hosted on different Web servers. In Fig. 1, this is indicated by labeling the graphs with two source URIs, "X" for $G1$ and "Y" for $G2$. Structural graph summaries are often an order of magnitude smaller than the input graph [7]. As graph summaries share the structural features observed in the input graph, they often serve as indices to search for equivalent subgraphs [7, 11]. In the example above, the graph summary can answer data search questions like: "Where can one find graphs on the Web with vertices of type Book that are connected to vertices with the type Person by an edge labeled author?" or "How many vertices have outgoing edges labeled title, author, and abstract?". Other applications of structural graph summaries include cardinality computations for queries on graphs [21], data exploration [1, 20, 22, 25], data visualization [10], vocabulary term recommendations [24], and related entity retrieval [8]. Next, we describe the tasks of data search and cardinality computation. We use these tasks to motivate the need for incremental computation of structural graph summaries.

The task of *data search* is to find (sub)graphs on the Web that match a given schema structure [11]. First, a structural graph summary is queried to identify relevant data sources matching this query. Then, the data source URIs are accessed to download the graphs matching the query. To implement data search, structural graph summaries need to memorize schema structure and location of vertices on the Web. As the data on the Web changes [15], the summaries need to be updated as well. The task of *cardinality computation* is to calculate how many (sub)graphs match a given schema structure. This enables, e. g., query size estimation [21] or assessment of completeness in knowledge bases [23]. To implement this task, one needs to memorize the number of summarized vertices for each schema structure. As the GDB changes, also the numbers in the summary need to be updated.

When the input graph changes, it is often prohibitively expensive to recompute the structural graph summary from scratch. Thus, an update algorithm is needed. Our novel incremental algorithm to compute structural graph summaries automatically detects changes in the data graph. This, however, takes time linear in the size of the input graph. Furthermore, our algorithm is designed to allow parallel execution in a distributed system architecture. We achieve this by following the idea of Tarjan's two-phase algorithm for the *set union problem* [27] and implementing the make-set phase following the signal and collect programming model [26]. Moreover, in contrast to existing solutions, our algorithm is not designed for a specific graph summary model, but can handle arbitrary models defined in our formal language FLUID [4]. In summary, our contributions are as follows:

- A parallel algorithm to incrementally compute and update structural graph summaries defined as equivalence relations following our formal language.
- Detailed theoretical complexity analysis of our incremental algorithm showing that all graph summaries defined in the formal language can be updated in time $O(\Delta \cdot d^k)$.
- Empirical analyses on benchmark and real-world datasets show that our incremental algorithm outperforms a batch computation even when about 50% of the graph changes.

The paper is structured as follows. Next, we discuss related works. In Sect. 3, we define graph databases, graph summary models, and our data structure for graph summaries as well as its data complexity. In Sect. 4, we present the parallel summarization algorithm and discuss its computational complexity. In Sect. 5, we present the extension to incrementally update graph summaries and analyze the update complexity. In Sect. 6, we describe the experimental apparatus. Finally, we present and discuss the experimental results.

2 RELATED WORK

Structural graph summaries partition a graph based on features of vertices or edges. There are a variety of structural graph summary models in the literature, which capture different structural features such as edge labels [6], vertex labels [6], incoming vs. outgoing edges [21], combinations of such features for different subgraphs [10, 18], and others. A detailed survey is provided by Čebiric et al. [7]. Of particular interest are the works by Goasdoué et al. [10] and Konrath et al. [18] who compute graph summaries of a stream of vertex-edge-vertex triples, i. e., they can deal with the addition of new vertices and edges to the graph. However, neither approach deals with the deletion of vertices or edges or the modification of their labels. Thus, they cannot update the structural summaries of evolving graphs. Existing structural graph summaries have in common that they are designed only for a single purpose and cannot be easily adapted and extended to different tasks.

Besides structural graph summaries that abstract from a graph based on common vertex or edge features, there are also general purpose graph database indices. Commonly, graph databases use path indices, tree indices, and subgraph indices [13]. We focus on incremental subgraph indices as they most closely relate to structural graph summaries. Yuan et al. [29] propose an index based on mining frequent and discriminative features in subgraphs. The algorithm minimizes the number of index lookups for a given query. It regroups subgraphs based on newly added features. The runtime performance was improved by the same authors in 2015 [30] and by Kansal and Spezzano in 2017 [16]. However, mining frequent features in subgraphs only optimizes the index for lookup operations for commonly used queries. It does not compute a comprehensive graph summary along specified structural features. The algorithm of Fan et al. [9] can deal with graph changes for the subgraph isomorphism problem. Their incremental computation of an index for isomorphic subgraphs is closely related to structural graph summarization, but it differs in that the graph pattern p is an input to the algorithm, not the output. The goal of structural graph summarization is to compute, based on a summary model that specifies the features, the set of common graph patterns that occur in a graph.

Another area related to our work is incremental schema discovery from large datasets in NoSQL databases. For example, XStruct [14] follows a heuristic approach to incrementally extract the XML schema of XML documents. However, such schema discovery approaches cannot deal with modifications or deletions of nodes in the XML tree. Other schema discovery approaches focus on generating (probabilistic) dataset descriptions. Kellou-Menouer and Kedad [17] apply density-based hierarchical clustering on vertex and edge labels in a graph database. This computes profiles that can be used to visualize the schema of the graph. The term "incremental" for the related schema discovery algorithms refers in their work to the concept of incrementally processing large documents, not considering modifications and deletions. Thus, they are not designed to update the discovered schema for evolving graphs.

In conclusion, existing algorithms are either not designed for evolving graphs but support structural graph summaries, or are incremental algorithms for graph database indices but not designed to support structural summaries. They are designed for a single task only, and the provided solution cannot be easily adapted or extended to other graph summary models or tasks.

3 FOUNDATIONS

We define graph databases, how to model structural graph summaries, and our data structure for structural graph summaries.

3.1 Graph Databases

We define a graph database as multiset of labeled property graphs with shared vertices and edges. Multisets of graphs allow data replication. This is important when graph databases are distributed

without centralized management such as in the Web of Data, where graphs are stored and maintained on nodes controlled by various parties. To deal with such decentralized storage, graphs contain data provenance information, i. e., their location on the Web.

Formally, we define a graph database $GDB = (V, E, \mathcal{G})$, where V is a set of vertices, $E \subseteq V \times V$ is the set of edges, and $\mathcal{G} = \{G_1, \ldots, G_n\}$ is a multiset of graphs. Each graph $G \in \mathcal{G}$ is a tuple: $G = (V_G, E_G)$ with $V_G \subseteq V$ and $E_G \subseteq E$. The sets V, E are not multisets, i. e., all vertices and edges are uniquely identified within the GDB, e. g., by URIs on the Web. The graphs $G \in \mathcal{G}$ are not necessarily connected, i. e., they may contain multiple disjoint components. Two graphs G_i and G_j with $i \neq j$ can have common vertices, i. e., it is possible that $V_{G_i} \cap V_{G_j} \neq \emptyset$.

We define two labeling functions for our structural graph summaries. The first function $\ell_V : V \rightarrow \mathcal{P}(\Sigma_V)$ maps each vertex to zero or more labels from the finite alphabet Σ_V (also referred to as types). The second function $\ell_E : E \rightarrow \mathcal{P}(\Sigma_E)$ maps each edge to zero or more labels from the finite alphabet Σ_E (also referred to as properties). In addition, we assume that each instance of each graph $G \in \mathcal{G}$ is labeled with a source label, e. g., the URI of the document containing G, serving as provenance information.

We assume that edges are directed, i. e., $(v, w) \neq (w, v)$ for all distinct $v, w \in V$. We can represent the undirected edge vw by the pair of directed edges (v, w) and (w, v). In a directed graph, we might have edges (v, w) and (w, v) and, in this case, we might have $\ell_E(v, w) \neq \ell_E(w, v)$. However, in an undirected graph (where all edges are undirected), we will always have $\ell_E(v, w) = \ell_E(w, v)$. Furthermore, we write $\Gamma(v) = \{w \mid (v, w) \in E\}$ for the set of neighbors of v in the graph database GDB.

3.2 Structural Graph Summary Models

A structural graph summary is a condensed representation of a graph such that a set of chosen (structural) features are equivalent in the graph summary and the original graph [7]. For example, vertices in graph database (GDB) might be summarized if they share the same label. How the graph is summarized can be expressed and formally defined as an equivalence relation. We call this definition the **graph summary model**. The **graph summary** of a GDB is computed following a specific graph summary model and can be used to implement a variety of tasks as described in the introduction. For different tasks, different features of the summarized vertices are of interest, e. g., the number of summarized vertices (cardinality computation) or the source labels (data search). This information about the summarized vertices is called the **payload**.

Graph Summary Model: Any equivalence relation over the vertices V of a graph G defines a partition of G, e. g., two vertices are equivalent under $\sim$ if they have the same labels (types). In this case, for any $v \in V$, the equivalence class $[v]_\sim$ contains only vertices with the same label as v. Existing graph summaries often define equivalence relations over vertices using combinations of vertex labels (types) and edge labels (properties) [4]. We call these combinations of types and properties the **schema structure** of vertices. Such schema structures can be flexibly defined in our formal language FLUID using three simple and one complex **schema elements** along with conjunction, disjunction and a set of parameterizations [4]. We now summarize the relevant concepts of FLUID.

The three **Simple Schema Elements** (SSEs) summarize vertices v using only $\ell_V(v)$, $\ell_E(v, w)$, and/or vertex identifiers w with $w \in \Gamma(v)$. **Object Cluster** (OC) compares vertex identifiers of all neighboring vertices: two vertices v and v' are equivalent iff $\Gamma(v) = \Gamma(v')$ (and vice versa). **Property Cluster** (PC) compares edge labels: v and v' are equivalent iff, for all outgoing edge labels (properties) $\ell_E(v, w)$ there is an identical property $\ell_E(v', w')$ for v'(and vice versa). **Property-Object Cluster** (POC) combines PC and OC: v and v' are equivalent iff, for all neighbors $w \in \Gamma(v)$ there is a neighbor $w' \in \Gamma(v')$ with the same vertex identifier, and with $\ell_E(v, w) = \ell_E(v', w')$ (and vice versa). SSEs are called "simple" because equivalence of vertices can be computed just by comparing the vertices and their outgoing edges. In contrast, **Complex Schema Elements** (CSEs) use information beyond this local schema structure. CSEs are defined using a tuple of three equivalence relations, i. e., $CSE := (\sim^s, \sim^p, \sim^o)$. $\sim^s$ defines the local schema structure of the vertex v. $\sim^o$ defines the local schema structure of neighbors $w \in \Gamma(v)$. Intuitively, $\sim^p$ defines how the local schema structures of v and w are connected.

Parameterizations further specify the simple and complex schema elements. The chaining parameterization has a parameter k that limits the maximum distance of the considered subgraphs for *CSEs* to k and is denoted as CSE_k. For example, CSE_2 is equivalent to a nested complex schema element, i. e., $CSE := (\sim^s, \sim^p, (\sim^s, \sim^p, \sim^o))$. The label parameterization restricts the edges considered for the summaries to edges with labels defined in a given set P_l. Intuitively, all edges with labels not in P_l do not change the summarization. This can, for example, be used to consider types in RDF graphs, since they are represented as vertex identifiers and attached to vertices with edges labeled `rdf:type`. As using the types of vertices (i. e., vertex label in our GDB definition) is a commonly used feature, we denote the OC with the label parameterization $P_l = \{\texttt{rdf:type}\}$ as OC_{type}. The set parameterization has as parameter a set of labels or vertex identifiers S. It forces, in addition to the equivalence of vertex and/or edge label, that all labels are also contained in S. The direction parameterization allows to consider only outgoing edges, incoming edges, or both. The inference parameterization enables ontology reasoning using a vocabulary graph. The vocabulary graph stores all hierarchical dependencies between vertex labels (types) and edge labels (properties) denoted by ontologies present in the graph database. The instance parameterization allows vertices to be merged when they are labeled as equivalent, e. g., vertices linked with `owl:sameAs`.

3.3 Data Structure of Graph Summaries

Let $GDB = (V, E, \mathcal{G})$ be a graph database with label functions ℓ_V and ℓ_E, and let $\sim$ be an equivalence relation over V. The **graph summary** for GDB with respect to $\sim$ is a labeled graph $SG = (V_{\text{vs}} \cup V_{\text{pe}}, E_{\text{vs}} \cup E_{\text{pe}})$, where $E_{\text{vs}} \subseteq V_{\text{vs}} \times V_{\text{vs}}$ and $E_{\text{pe}} \subseteq V_{\text{vs}} \times V_{\text{pe}}$. Here, the subscript "vs" denotes "vertex summary". The subgraph $VS = (V_{\text{vs}}, E_{\text{vs}})$ contains the *schema* information about the GDB according to the model used for $\sim$, as introduced in Sect. 3.2. Thus, the vertices V_{vs} and edges E_{vs} are those shown in the graph summary SG in Fig. 1. The subscript "pe" denotes "payload elements". The graph $PG = (V_{\text{vs}} \cup V_{\text{pe}}, E_{\text{pe}})$ connects the schema to the payload, i. e., each edge $(v, w) \in E_{\text{pe}}$ connects a vertex v in VS to a vertex w (the

payload element) that contains v's payload information. The *SG* is the union of the vertex summaries VS and their payload PG.

Each vertex in V_{vs} has as its identifier a pair (C, R), where R is an equivalence relation over V, the vertices of the GDB being summarized, and C is one of R's equivalence classes. The edges in E_{vs} correspond to the edges in the GDB through which the equivalence relations are defined. We further divide V_{vs} into *primary vertices*, which are equivalence classes of $\sim$, and *secondary vertices*, which are equivalence classes of the relations from which $\sim$ is defined.

For each $v \in V$ (of the GDB) and equivalence relation $\sim$ (defined by simple or complex schema elements), we define the *local* summary graph $vs_{\sim}(v)$ by induction on the structure of the schema elements. This local summary graph is computed for each $v \in V$ and is called the **vertex summary**.

To serve as *base-cases for the inductive definition* of vertex summaries, we define equivalence relations $\text{id} = \{(v, v) \mid v \in V\}$ and $T = V \times V$. For any vertex $v \in V$, $vs_{\text{id}}(v)$ is the graph with the single vertex $(\{v\}, \text{id})$, which is the primary vertex, and no edges; similarly, $vs_T(v)$ has the single (primary) vertex (V, T) and no edges. Note that $vs_T(v)$ is identical for every $v \in V$, i. e., all vertices are summarized by the same vertex summary, but $vs_{\text{id}}(v)$ is distinct for every $v \in V$, i. e., each vertex summary summarizes one vertex.

For the *inductive step*, we define the vertex summaries for CSEs. This implicitly includes SSEs since, although SSEs are implemented separately for efficiency, the SSEs OC, PC and POC are equivalent to the CSEs (T, T, id), (T, id, T) and $(T, \text{id}, \text{id})$, respectively. So, let $\sim$ be the equivalence relation defined by the CSE $(\tilde{s}, \tilde{p}, \tilde{o})$ and let $v \in V$. Let $\Gamma^o = \{[w]_{\tilde{o}} \mid w \in \Gamma(v)\}$ be the set of $\tilde{o}$-equivalence classes of v's neighbors. The primary vertex of $vs_{\sim}(v)$ is $([v]_{\tilde{s}}, \tilde{s})$. For each equivalence class $C \in \Gamma^o$, $vs_{\sim}(v)$ has a subgraph $vs_{\tilde{o}}(w_C)$, where w_C is an arbitrary vertex in C. Now, let $B^p_C = \{[(v, w)]_{\tilde{p}} \mid (v, w) \in E \text{ and } w \in C\}$; i. e., if v has neighbors in $\tilde{o}$-class C, then B^p_C is the set of $\tilde{p}$-equivalence classes of the edges linking v with a vertex $w \in C$. For each class $\beta \in B^p_C$, $vs_{\sim}(v)$ contains an edge labeled β from its primary vertex to the primary vertex of $vs_{\tilde{o}}(w_C)$.

Theorem 3.1. *Let $GDB = (V, E, \mathcal{G})$ be a graph database with maximum degree at most $d > 1$, and let $\sim$ be an equivalence relation on V defined by nesting CSEs to depth k. For every $v \in V$, $vs_{\sim}(v)$ is a tree (possibly with parallel edges) with $O(d^k)$ vertices.*

Proof. That $vs_{\sim}(v)$ is a tree follows from the definition: the base cases are one-vertex trees and the inductive steps cannot create cycles. Any vertex in V has at most d neighbors, so is adjacent to at most d equivalence classes. Therefore, no vertex in $vs_{\sim}(v)$ has degree more than d. $vs_{\sim}(v)$ has depth k, so it contains at most $\sum_{i=0}^{k} d^i = O(d^k)$ vertices. □

In principle, a single vertex summary in a graph summary may be bigger than original GDB. However, this requires the use of highly nested CSEs on small GDBs, which is unlikely in practice.

4 PARALLEL GRAPH SUMMARIZATION

Our algorithm to compute graph summaries can be executed in parallel and in a distributed system architecture. In the subsequent section, we introduce the extension to this algorithm that allows incremental updates of computed graph summaries.

4.1 Outline of the Parallel Algorithm

Our algorithm is inspired by the two-phase approach of Tarjan's algorithm for the set union problem [27]. *Phase 1 (schema computation):* Compute for each vertex v the corresponding vertex summary vs. This corresponds to the make-set operation and generates a vertex summary vs for each vertex. *Phase 2 (find and merge):* Find the vertex summaries that have the same schema structure and merge them. This refers to the find-set operation in Tarjan's algorithm. When all vertex summaries with the same schema structure are found and merged, we successfully partitioned the vertices of a GDB into disjoint subsets. One fundamental idea of our algorithm is that to achieve high parallelism, we can partition the graph database for the computation process in such a way that all vertices with their label set and their (outgoing) edges including their label are in a single partition Thus, the equivalence defined by simple schema elements (SSEs) can be computed for each vertex independently.

4.2 Parallel Algorithm

We support the parallel computation of all graph summary models defined in Sect. 3.2. This is achieved by using a parameterized implementation of the simple and complex schema elements. The pseudo code of the summarization algorithm is presented in Alg. 1. In Line 3, the extraction of the schema for each vertex v in the graph database begins. In parallel, for each v, the local vertex schemata are extracted as defined by the simple schema elements of the graph summary model provided as input. This simple schema extraction is applied using both the $\sim^s$ and $\sim^o$ equivalence relations (see Lines 4 and 5) of the graph summary model.

The locally computed vertex schema is exchanged between the vertices to construct the complex schema information (as defined by the graph summary model). We use the common technique of signal and collect [26]. In Line 6, each vertex v receives as signals the schema (according to the object equivalence relation $\sim^o$) of all its neighbors. Likewise, Line 8, collects neighbors' schemas and constructs the data structure defined in Sect. 3.3. When we use the k-chaining parameterization, this step of sending and aggregating information is done k times (Lines 11 to 13) and a vertex accesses the schema information from vertices up to distance k. Line 14 extracts the payload information from the vertex v. Example payload functions are counting the number of vertices (increasing a counter) or memorizing the source label of v. The final vertex summary vs and payload element *pe* are stored in a centralized managed data structure (Line 15), e. g., a graph database, where the find and merge phase is implemented. When the same vertex summary vs is computed for multiple vertices, it is only stored once and the payload elements are merged, e. g., the number of summarized vertices is increased or the corresponding source labels are added.

The direction parameterization only changes how the graph is traversed but not the algorithm itself, so it is not shown. The label and set parameterizations are omitted here since they require only a lookup in the corresponding parameter set. The instance parameterization is a pre-processing step, i. e., all vertices connected by an edge with a specific label, e. g., `owl:sameAs`, are merged. Following Liebig et al. [19], the inference parameterization is a post-processing step, since inference on a single vertex summary vs is equivalent to inference on all summarized subgraphs.

Algorithm 1: Parameterized Graph Summarization

```
1  function ParallelSummarize(GDB, SG, (~_s, ~_p, ~_o)_k)
2      returns graph summary SG
3      forall v ∈ V do in parallel
4          vs ← ExtractSimpleVertexSchema(v, E, ~_s);
5          tmp ← ExtractSimpleVertexSchema(v, E, ~_o);
           /* signal information relevant for ~_o to
              incoming neighbors                          */
6          forall w ∈ V : (w, v) ∈ E do
7              w.Inbox ← (tmp, 1);
           /* collect information of neighbors and
              construct complex vertex summaries          */
8          forall (tmp_vs', r) ∈ v.Inbox do
9              t ← ExtractSimpleEdgeSchema((v, w), ~_p);
10             vs.Neighbor_w ← (t, tmp_vs');
               /* k-chaining repeats signal and
                  collect k-times                         */
11             if r < k then
12                 forall w ∈ V : (w, v) ∈ E do
13                     w.Inbox ← (vs, r + 1);
14         pe ← ExtractPayload(v);
15         SG ← FindAndMerge(SG, vs, pe, v);
16     return SG;
```

4.3 Complexity of Parallel Summarization

Since our algorithm is inspired by the two-phase approach of Tarjan's algorithm for the set union problem [27], we briefly review its complexity. Phase 1 partitions the set of n vertices using n *make-set* operations and phase 2 uses some number $m \leq n$ of find operations. For this problem, the worst-case complexity is proven to be $O(n + m \cdot \alpha(m + n, n))$, where α is the functional inverse of Ackermann's function [27]. It is generally accepted that in practice $\alpha \leq 4$ holds true [27]. Thus, the summary computation can be done in essentially linear time. A detailed discussion on the impact of the different parameterizations can be found in [4]. It concludes that only the chaining parameterization and inference parameterization have an impact on the worst case complexity, where the inference parameterization applied on the graph database produced a worst-case complexity of $O(n^2)$. Since we implement the inference parameterization as a post-processing step, this has no impact on Alg. 1. We consider chaining parameterization in detail in Sect. 5.3.

Typical payload functions extract information from a single vertex, e. g., counting or storing the source of a data graph [7]. These functions run in time $O(1)$ since only a single vertex is needed to extract the payload. When we find and merge the schema elements, the payload is merged in time $O(1)$ as well.

5 INCREMENTAL GRAPH SUMMARIZATION

The parallel graph summarization algorithm presented above can be used for batch computation of graph summaries. In this section, we extend this algorithm to allow incremental updates on a non-empty graph summary SG given as input to the algorithm.

5.1 Outline of the Incremental Algorithm

For the incremental algorithm, we adapt the FindAndMerge phase. In the batch algorithm, all vertex summaries vs are computed, found, and merged. In the incremental algorithm, only vertex summaries of vertices with changed information are found and merged. This avoids unnecessary and costly operations. However, if there is no change log available, for each vertex v in the graph database the make-set operation needs to be executed, i. e., the new vertex summary vs needs to be extracted. When a change log is provided, vertex summaries only for changed vertices need to be extracted.

There are six changes in a graph database that could require updates in a structural graph summary: a new vertex is observed with a new schema (ADD-SG), a new vertex is observed with a known schema (ADD-PE), a known vertex is observed with a changed schema (MOD-SG), a known vertex is observed with changed "payload-relevant information" (MOD-PE), a vertex with its schema and payload information no longer exist (DEL-PE), and no more vertices with a specific schema structure exist (DEL-SG).

To check if vertices have changed, we use an additional data structure called VertexUpdateHashIndex. This allows us to trace links between vertices v in the graph database, vertex summaries vs in the graph summary, and payload elements pe in the graph summary. Intuitively, in the find and merge phase, we look in the VertexUpdateHashIndex to see if the vertex summary and payload element for a vertex contain information that requires an update to the graph summary. If no update is required, we can skip the vertex. When accessing the VertexUpdateHashIndex is faster than actually finding and merging vertex summaries, we decrease the computation time. We implement the VertexUpdateHashIndex as a three-layered unique hash index with cross-links between the layers. Since only hashes and cross-links are stored, updates on the secondary data structure are faster than updates on the graph summary. The unique hash indices ensure that there is at most one entry for all referenced vertices in each layer.

5.2 Incremental Algorithm

The incremental graph summarization algorithm contains two extensions to the parallel graph summarization algorithm, Alg. 1. First, we replace the FindAndMerge function with an incremental version called IncrementalFindAndMerge, which is shown in Alg. 2. This handles additions and modifications. Second, we add a loop to handle deletions from the graph database.

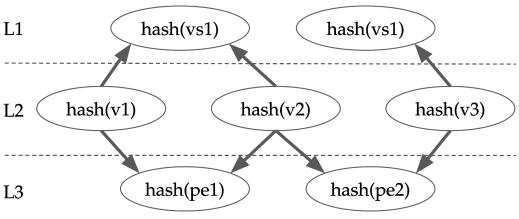


Figure 2: The VertexUpdateHashIndex is a three-layered data structure. L1 is a unique hash index for the vertex summaries vs, L2 is a unique hash index for vertices v, and L3 is a unique hash index for payload vertices pe.

Algorithm 2: Incremental FindAndMerge Algorithm.

```
1  function IncrementalFindAndMerge(SG, vs, pe, v)
2      returns updated graph summary SG
3      if VertexHashIndex.ContainsLink(v) then
4          id_prev ← VertexHashIndex.GetLink(v);
5          vs_prev ← SG.GetElement(id_prev);
6          if vs_prev ≠ vs then
7              VertexHashIndex.RemoveLink(v);
8              if |VertexHashIndex.GetLinks(vs_prev)| ≤ 0
                 then
9                  SG.RemoveElement(vs_prev);
10     if not VertexHashIndex.ContainsLink(v) then
11         VertexHashIndex.AddLink(v, Hash(vs));
12     if SG.ContainsElement(vs) then
13         SG.UpdatePayload(vs, pe);
14     else
15         SG.AddElement(vs, pe);
16     return SG;
```

Line 3 of Alg. 2 checks if the vertex v is already in the VertexUpdateHashIndex. If it is, we retrieve its existing vertex summary vs_{prev} (Lines 4 and 5). If the current vertex summary vs differs from vs_{prev}, then v's schema has changed (MOD-SG) so we delete the link between v and vs_{prev} in the VertexUpdateHashIndex (Line 7). If this was the last link for vs_{prev}, then vs_{prev} no longer summarizes any vertex and it is deleted from SG in Line 9 (DEL-SG). At this point (Line 10), there are two reasons that v might not be in the VertexUpdateHashIndex: it could be a new vertex that was not in the previous version of the GDB (ADD-PE or ADD-SG) or it could be a vertex whose schema has changed since the previous version and we deleted it at Line 7 (MOD-SG). In any case, we add a link from v to its new summary vs at Line 11. After the VertexUpdateHashIndex is updated, we update the graph summary SG. If vs is already in SG, we update the payload element pe of vs in Line 13 (ADD-PE). Thus, we found and merged a vertex summary. If vs does not yet exist in SG, we add the vertex summary vs to the graph summary SG (ADD-SG).

After completing phases 1 and 2 of our incremental graph summarization algorithm, we handle deletions (DEL-PE). All vertices that are no longer in the GDB are deleted from the VertexUpdateHashIndex. Analogously to the deletion described above, deleting entries in the VertexUpdateHashIndex can trigger a deletion of a vertex summary vs in the graph summary SG (DEL-SG).

Proof of correctness (sketch): To prove that the incremental algorithm is correct, we need to show that the batch algorithm applied on the graph database at time $t+1$ (GDB_{t+1}) returns the same graph summary SG as the incremental algorithm first applied on GDB_t and then applied on GDB_{t+1}. We denote by $\emptyset$ the empty (summary) graph, by SG_{batch} the result of the batch computation ParallelSummarize($GDB_{t+1}, \emptyset, \sim$), and by SG_{incr} the result of the incremental computation ParallelSummarize(GDB_{t+1}, ParallelSummarize($GDB_t, \emptyset, \sim$), $\sim$). It can be shown by induction for all $t \in \mathbb{N}$ that $SG_{\text{batch}} \subseteq SG_{\text{incr}}$ and $SG_{\text{incr}} \subseteq SG_{\text{batch}}$, with $\subseteq$ denoting the subgraph relation. The full proof is available online [3].

5.3 Complexity of Incremental Summarization

In the following, we analyze the update complexity of all possible changes in the data graph w.r.t. the number of operations on the vertex summary, i. e., adding and/or removing vertices and/or edges. We first discuss ADD-SG, MOD-SG, and DEL-SG as they require an update on the vertex summary and possible cascading updates on other vertex summaries. Then, we discuss updates on the VertexUpdateHashIndex, which are common to all six changes. Finally, we briefly discuss payload changes.

Graph Summary Updates: Observed vertices with a new schema (ADD-SG) require at least 1 and at most $d^k + 1$ new vertices to be added to SG. As discussed in Sect. 3.3, reusing vertices and edges in the graph summary reduces the number of add operations. However, since it is a new vertex summary, at least one new vertex needs to be added, i. e., the primary vertex. Furthermore, up to d^k edges are to be added to SG, in the same way. Deleting all vertices v summarized by a vertex summary vs (DEL-SG) also requires deleting vs from SG. DEL-SG is the counterpart to ADD-SG, i. e., we have to revert all operations. Thus, DEL-SG has the same complexity as ADD-SG. When we observe a vertex v with vertex summary vs' at time t, but already summarized v with a different vertex summary vs at time $t-1$, we have to modify the graph summary SG. Transforming a vertex summary vs to another vertex summary vs' means in the worst case deleting all vertices and edges in vs and adding all vertices and edges in vs'. This occurs when the schema of v has entirely changed from $t-1$ to t. Thus, modifications to vs' are in the worst case $d^k + 1$ added vertices, d^k added edges, $d^k + 1$ deleted vertices, d^k deleted edges. In the best case, vs' already exists in SG and no updates to the graph summary are needed.

Cascading Updates: When complex schema elements are used, updates on the vertex summary vs of a vertex v can require an update on the vertex summaries of any neighboring vertex w, if $v \in \Gamma(w)$. Thus, for each incoming edge to v, up to d^- vertices need an update. Complex schema elements (CSE) correspond to a chaining-parameterization (bisimulation) of $k = 1$. For arbitrary $k \in \mathbb{N}$, updating one vertex summary vs requires up to d^{-k} additional updates. Therefore, the complexity of ADD-SG, DEL-SG, and MOD-SG is $O(d^k)$ for a single vertex update. Since k is fixed before computing the index, the only variable factor depending on the data is the maximum degree d of the vertices in the GDB.

VertexUpdateHashIndex Updates: All six changes require an update on the VertexUpdateHashIndex. Summary models defined using equivalence relations partition vertices of the GDB into disjoint subsets, i. e., the vertex summaries (see Sect. 3.2). Thus, there are as many vertex summaries vs in SG that may need to be updated as there are partitions in the GDB. For each $[v]_\sim$, there is exactly one entry $hash(vs)$ stored in the L1 layer of the VertexUpdateHashIndex. For each vertex v in the GDB, there is a $hash(v)$ stored in L2, which links to exactly one hash in L1. Thus, ADD-SG, DEL-SG, and MOD-SG require two operations on the VertexUpdateHashIndex. The remaining three changes ADD-PE, DEL-PE, and MOD-PE require no updates on the vertex summaries vs, but require up to two updates on the VertexUpdateHashIndex.

Payload Updates: All six changes possibly require an update to the payload. As discussed above, different payloads are used to implement different tasks. Thus, the number of updates depends

on what is stored as payload. For example, for data search, we memorize the source label. The payload information (source label) is stored in payload elements in the graph summary. Links to these payload elements are stored in L3 of the VertexUpdateHashIndex. In this example, we only update payload elements if a source label changed. As mentioned above, this requires at most two updates on the VertexUpdateHashIndex.

Summary: Any change in a GDB with maximum degree d requires at most $O(d^k)$ update operations on the graph summary, when the equivalence relation $\sim$ is defined using a chaining parameter of k. Thus, the overall complexity of incrementally computing and updating the graph summary with Δ changes on the GDB is bounded by $O(n + \Delta \cdot d^k)$, where the GDB has n vertices and maximum degree d, and the chaining parameter is k. From our analysis, we see three predominant factors regarding the complexity of incrementally updating structural graph summaries. First, the maximum degree d in the GDB. Second, the chaining parameterization k, i. e., the maximum distance in the matched equivalent subgraphs. Third, the number of summary vertices and the number of summary edges in SG, i. e., the schema heterogeneity, since ADD-SG and DEL-SG require in the best case to add/delete only one summary vertex and one summary edge.

6 EXPERIMENTAL APPARATUS

We empirically evaluate the time and space requirements of the presented graph summarization algorithms for different graph summary models. We run experiments on graph databases that evolve over time, i. e., we observe different versions of the graph database at different points in time. For each version of the graph database, we analyze the cost of computing a new graph summary from scratch (batch-based computation) compared to incrementally updating an existing graph summary from a previous version.

We implemented our algorithm in Scala using Apache Spark GraphX (single context with 20 cores and 200 GB heap space) and store graph summaries in an OrientDB graph database (single DB with 100 GB heap space and 20 GB memory mapped files). More details are available online [3].

We chose representative summary models for our experiments, namely SchemEX [18], Attribute Collection [6], and Type Collection [6]. Type Collection summarizes vertices that share the same label. In our formal model, they are defined as the label parameterized object cluster OC_{type} (see Sect. 3.2). Attribute Collection summarizes vertices that share the same labels of outgoing edges. Thus, they are defined as property cluster PC. SchemEX is a combination of Type and Attribute Collection, i. e., two vertices have the same label, have edges with the same label, and neighbors with the same label. Thus, they are defined as CSE (OC_{type}, id, OC_{type}). As payload, we store the source graph label or the number of summarized vertices to implement data search and cardinality computation as described in Sect. 1.

6.1 Datasets

We use two benchmark datasets (LUBM100 and BSBM) and two variants of the real-world weekly crawled DyLDO dataset. The two benchmark datasets are only suitable for the cardinality computation task since they have only one source graph.

LUBM100: The Lehigh University Benchmark (LUBM) generates benchmark datasets containing people working at universities [12]. We use the Data Generator v1.7 to generate 10 versions of a graph containing 100 universities. Thus, all versions are of similar size, but we emulate modifications by generating different vertex identifiers, i. e., each version is considered as timestamped graph. Each graph contains about 2.1 M vertices and 13 M edges. Over all versions, the avg. degree is 6.7 (standard deviation: $\sigma < 0.1$), the avg. in-degree is 6.8 ($\sigma < 0.1$), and the avg. out-degree is 5.1 ($\sigma < 0.1$).

BSBM: The Berlin SPARQL Benchmark (BSBM) is a suite of benchmarks built around an e-commerce use case [2]. We generate 21 versions of the dataset with different scale factors. The first dataset, with a scale factor of 100, contains about 7,000 vertices and 75,000 edges. We generate versions with scale factors between 2,000 and 40,000 in steps of 2,000. The largest dataset contains about 1.3 M vertices and 13 M edges. For our experiments, we first use the different versions ordered by size from small to large (version 0 to 20) to simulate a growing graph database. Subsequently, we reverse the order to emulate a shrinking graph database. Over all versions, the avg. degree is 8.1 ($\sigma = 0.5$), avg. in-degree is 4.6 ($\sigma = 0.3$), and avg. out-degree is 9.8 ($\sigma = 0.2$).

DyLDO: The Dynamic Linked Data Observatory (DyLDO) provides regular crawls of the Web of Data [15]. The crawls started from about 95,000 representative seed URIs (source label of graphs). There are two variants of this dataset: The dataset containing only the graphs identified by the seed URIs is referred to as **DyLDO-core**. It contains the 95,000 different graphs obtained from the seed URIs. The extended crawl (including the core) is referred to as **DyLDO-ext**. Starting from the seed URIs, a breadth-first search is conducted with a crawling depth of 2, i. e., recursively graphs are added that are referenced from already crawled graphs.

For DyLDO-core, we use all 50 crawls conducted between January 20, 2019 and January 12, 2020. DyLDO-core contains 2.1–3.5 M vertices and 7–13 M edges. Over all weekly crawls, the avg. degree is 3.5 ($\sigma = 0.2$), avg. in-degree is 3.0 ($\sigma = 0.2$) and avg. out-degree is 2.9 ($\sigma = 0.2$). Note that week 21 (June 16, 2019) is an anomaly as DyLDO-core contains only eight edges due to a crawling failure. Thus, we excluded weeks 21 and 22 from the results. For DyLDO-ext, we use the first 5 crawls that contain 7–10 M vertices and 84–106 M edges. The avg. degree is 11.1 ($\sigma = 0.6$), avg. in-degree is 8.0 ($\sigma = 0.5$), and avg. out-degree is 10.7 ($\sigma = 0.7$).

6.2 Metrics

We employ different metrics to evaluate the **size** of the graph summaries, the **update complexity** for the graph summaries, and the runtime **performance** of the graph summary computation algorithms. Regarding **size**, we count the number of vertices $|V|$ and edges $|E|$ in GDB and the number of vertices $|V_{vs}|$ and edges $|E_{vs}|$ in the graph summaries. Furthermore, we denote by $\frac{|V|}{|V/\sim|}$ the *summarization ratio*, i. e., the fraction of the number of all vertices V in the GDB and the number of different equivalence classes under the equivalence relation $\sim$ in the graph summary SG. The summarization ratio describes how many vertices v are on average summarized by one vertex summary vs. Higher summarization ratios indicate a low variety of schema structures. A ratio of 1 indicates that no two vertices share the same schema structure.

The size of the VertexUpdateHashIndex is measured in bytes. By design, the VertexUpdateHashIndex has an entry in L1 for each primary vertex in *SG*, an entry in L2 for each vertex in the GDB, and an entry in *L*3 for each payload element. Thus, comparing the number of entries in VertexUpdateHashIndex to vertices and edges in the GDB does not bring additional insights. The VertexUpdateHashIndex's advantage is that the information is reduced to a minimum. Thus, we determine the data overhead in terms of a *compression ratio* by comparing the size of the VertexUpdateHashIndex in bytes (serialized gzipped Java object) to the size of the GDB in bytes (gzipped n-quads files). Regarding the **update complexity**, we count the number of vertices with a changed schema (ADD-SG, DEL-SG, and MOD-SG), i. e., the relevant *changes* in the GDB. Since changed schemata require in the best case 0 updates and in the worst case $O(d^k)$ updates on the graph summary, we also count the number of *updates* on the graph summary. Finally, the runtime **performance** of the algorithms is measured by the time needed to compute a summary from scratch or, in the incremental case, updating the summary *SG* for each version of the GDB.

7 RESULTS

The experimental results are visualized in Fig. 3. Each plot shows on the x-axis the database versions over time. From left to right, the columns of Fig. 3 show the metrics **size**, **update complexity**, and **performance**. From top to bottom, we show the results for the **LUBM100**, **BSBM**, **DyLDO-core**, and **DyLDO-ext** datasets. Unless specified otherwise, we refer in the text to average and standard deviation values computed over all version of a GDB.

Comparing the size of the graph summaries ($|V_{vs}|$, $|E_{vs}|$) and the graph database ($|V|$, $|E|$) (Fig. 3 left column), the graph summaries are orders of magnitude smaller (consistently over all experiments). Also, the Type Collections (abbreviated TypeColl) and Attribute Collections (abbreviated AttrColl) have higher summarization ratios than SchemEX. This means, fewer vertex summaries are needed to partition the GDB based on vertex labels or edge labels compared to combining vertex and edge labels. Over all datasets, Attribute Collections have the highest summarization ratios and have around a factor of of 10^4 fewer vertices, followed by Type Collections with a factor of 10^3 fewer vertices. SchemEX has the lowest summarization ratio. Still, SchemEX has on the order of a factor of 10^2 fewer vertices than the GDB. Comparing the *changes* in GDB and the *updates* on SG (Fig. 3 center column), we see that over all datasets, the number of updates on the graph summary is magnitudes smaller than the number of changes in the graph database. On average, there are 12, 000 more changes than updates for the Type Collections, 20, 000 more for the Attribute Collection, and 650 more for SchemEX. Furthermore, the incremental algorithm usually computes Type Collections the fastest (avg. 28min) and SchemEX the slowest (avg. 46min) (Fig. 3 right column). In relation to the batch counterpart, these numbers are a relative speed up of 1.8 ($\sigma = 0.7$) for SchemEX, 1.8 ($\sigma = 0.9$) for Attribute Collection, and 3.7 ($\sigma = 2.7$) for Type Collection.

The plots in Fig. 3 (right column) show the better run time performance of the incremental algorithm, which is achieved by maintaining the VertexUpdateHashIndex, compared to the batch computation. The compression ratio of the VertexUpdateHashIndex is shown in Table 1. We observe huge variations in the compression for each dataset ranging from 8% up to 64% of the GDB's size.

8 DISCUSSION

In total, we have run 312 experiments, i. e., we have $n = 312$ measure points of our three graph summary models over all versions of our four datasets (10× LUBM, 40× BSBM, 49× DyLDO-core, 5× DyLDO-ext). The key insight from our experiments is that, over all summary models and datasets, the incremental algorithm is almost always faster than the batch counterpart. Furthermore, a more detailed evaluation of the performance metrics shows a significant linear correlation between the schema computation (phase 1) and the number of edges in the graph database, $r(311) = 0.945$, $p < .0001$. This substantiates our theoretical complexity analysis. As described in Sect. 5, phase 1 is identical for both algorithms.

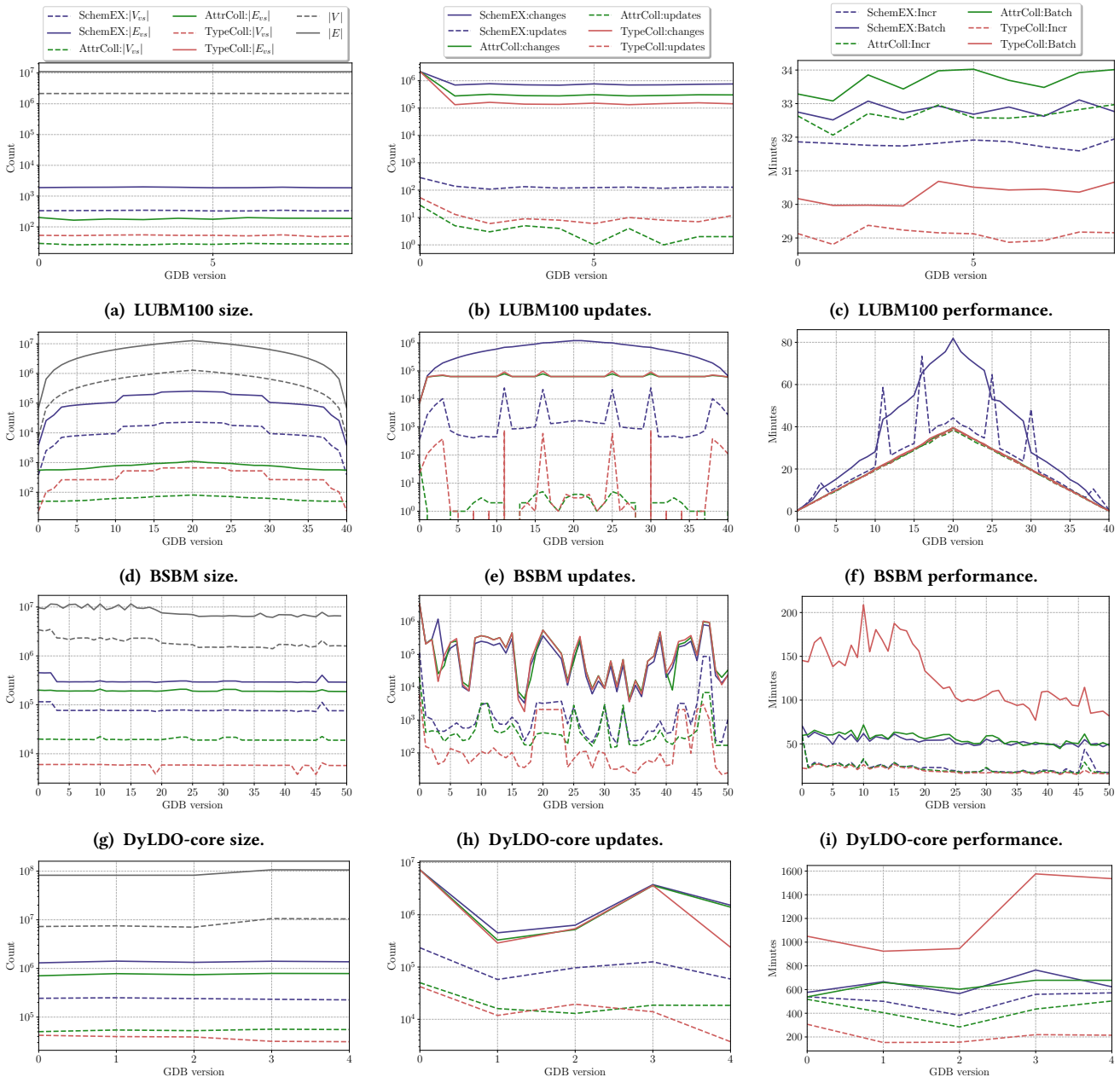
Regarding the overall runtime (phase 1 and 2), we observe that the incremental algorithm outperforms the batch variant almost always on the BSBM dataset, even though about 90% of the GDB changes in each version. Even for the DyLDO-core dataset, when about 46% of the GDB changes from version 46 to 47, the incremental algorithm is still 1.5 times faster. This can be explained by the fact that changes in the GDB do not necessarily require an update to the graph summary.

From our results, we can also state that graph summarization on benchmark datasets is easier than on real-world datasets, i. e., there are fewer vertex summaries needed to summarize a similar-sized graph database. This observation highlights the importance of using real-word datasets when evaluating graph summarization algorithms, as the observed variety of schema structures in the DyLDO datasets is not covered by existing benchmark datasets. Experiments on the BSBM dataset suggest that cascading updates due to neighbor changes have a huge impact on the performance of the incremental graph summarization algorithm. More than 99% of all vertex modifications are due to a neighbor change. Of all vertex changes, this makes up 83.88% ($\sigma = 15.64\%$).

Compared to the other experiments, the performances of the batch computations of Type Collections on the two DyLDO datasets have a exceptional trends. The batch computation takes twice as long for the Type Collection on the DyLDO-core than for the Attribute Collection and SchemEX. Further investigation revealed that there a few "hot" vertex summaries that summarize most of the vertices. During the find and merge phase, the payload information of different vertices that are summarized by the same vertex summary vs is merged (ADD-PE). Although merging payload information is done in constant time (merging sets of source graph label), merging basically all payload information into few payload elements cannot

Table 1: Relative size (in %) of the VertexUpdateHashIndex compared to the graph database (compression).

Dataset	SchemEX	AttrColl	TypeColl
LUBM100	40% ($\sigma < 1\%$)	39% ($\sigma < 1\%$)	39% ($\sigma < 1\%$)
BSBM	08% ($\sigma < 1\%$)	06% ($\sigma < 1\%$)	06% ($\sigma < 1\%$)
DyLDO-core	64% ($\sigma < 9\%$)	62% ($\sigma = 8\%$)	60% ($\sigma < 8\%$)
DyLDO-ext	24% ($\sigma < 5\%$)	23% ($\sigma = 5\%$)	23% ($\sigma < 3\%$)



(a) LUBM100 size. **(b) LUBM100 updates.** **(c) LUBM100 performance.**

(d) BSBM size. **(e) BSBM updates.** **(f) BSBM performance.**

(g) DyLDO-core size. **(h) DyLDO-core updates.** **(i) DyLDO-core performance.**

(j) DyLDO-ext. **(k) DyLDO-ext updates.** **(l) DyLDO-ext performance.**

Figure 3: Results of the experimental evaluation of three summary models: SchemEX (blue), Attribute Collection (green), and Type Collection (red). Each row is a different dataset and each column a different metric.

be done in parallel. For the incremental algorithm, this has no effect since the payload information is updated in parallel in the VertexUpdateHashIndex. For the VertexUpdateHashIndex, no additional synchronization for updating L3 is needed. One might consider this as a design flaw in the batch algorithm, which decides whether the incremental algorithm outperforms the batch algorithm or vice versa. However, first, there is no alternative to store the payload information in the graph summary since the batch algorithm requires no additional data structure. Second, the source graph payload is only used in the real-world datasets. For the benchmark datasets, no additional synchronization due to payload merges is done during the batch computation. Still, the incremental algorithm, with very few exceptions, outperforms the batch counterpart.

The evaluation of the data overhead shows a significant linear correlation between the VertexUpdateHashIndex size and the number of vertices in the graph database, $r(311) = 0.952$, $p < .0001$. Thus, L2 of the VertexUpdateHashIndex dominates the overall size. The sizes of the VertexUpdateHashIndex for the different summary models differ only marginally but noticeably. As shown in Table 1, the VertexUpdateHashIndex for SchemEX is consistently 1–4% larger than for the other two.

The number of edges is not as important for the size of the VertexUpdateHashIndex since edges are reflected by the vertex summaries. In consequence, graph databases with few vertices and more edges are compressed to a relatively lower size (BSBM, DyLDO-ext in Table 1). In light of this data overhead, future extensions may be to further compress L2 by using approximative data structures such as Bloom filters [28]. However, as pointed out by Fan et al. [9], many real-life applications require exact matches.

Overall, we evaluated more than 100 versions of four datasets (two synthetic and two real-world) each with different characteristics in terms of data change rates, types of changes, schema heterogeneity, size, and degree. Thus, we capture a wide range of characteristics of datasets suitable for graph summarization. Furthermore, we computed three representative summary models on these datasets. The selected summary models are widely used across different research areas, datasets, and tasks [7]. In addition, many summary models use the schema structure of the three evaluated summary models, though often under different names and sometimes in combination with further features [4].

We acknowledge that we did not re-evaluate the effectiveness of the summary models for their specific tasks. However, the summary models were evaluated by their respected authors. Our algorithm does not change the summary models but allows an efficient computation of the original summary models for evolving graphs. Beyond the specific graph summary models evaluated in this paper, our parameterized incremental algorithm is suitable for any other summary model defined in our formal language [4]. Our framework, the experimental apparatus, and the results are available on GitHub under an open source license [3]. Thus, we encourage further experiments using other models and datasets. For example, in our analysis, we could not find a link between graph properties such as degree, number of changes, size of the graph summary etc. that would allow a reliable prediction of the performance of the incremental graph summarization algorithm. Here, further experiments with different models and datasets are required. However, our experiments already show that for commonly used summary models, the incremental algorithm almost always outperforms the batch computation, even if about 50% of the data changes from version t of a GDB to version $t + 1$.

9 CONCLUSION

We presented an incremental graph summarization algorithm for structural graph summaries defined using equivalence relations. We analyzed the complexity of our algorithm and empirically evaluated time and space requirements for three graph summary models on two benchmark and two real-world datasets. The incremental summarization algorithm almost always outperforms its batch counterpart, even when about 50% of the graph database changes.

Acknowledgment. We would like to thank Malte Ostendorff for his valuable contributions when discussing the results.

REFERENCES

[1] F. Benedetti, S. Bergamaschi, and L. Po. 2015. Exposing the Underlying Schema of LOD Sources. In *Proc. WI-IAT 2015*. IEEE, 301–304.
[2] C. Bizer and A. Schultz. 2009. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.* 5, 2 (2009), 1–24.
[3] T. Blume. 2020. FLUID Spark (v1.2). https://github.com/t-blume/fluid-spark.
[4] T. Blume and A. Scherp. 2019. FLUID: A Common Model for Semantic Structural Graph Summaries Based on Equivalence Relations. (2019). http://arxiv.org/abs/1908.01528 arXiv CoRR:abs/1908.01528.
[5] A. Bonifati, S. Dumbrava, and H. Kondylakis. 2020. Graph Summarization. *CoRR* abs/2004.14794 (2020). https://arxiv.org/abs/2004.14794
[6] S. Campinas, T. Perry, D. Ceccarelli, R. Delbru, and G. Tummarello. 2012. Introducing RDF Graph Summary with Application to Assisted SPARQL Formulation. In *Proc. DEXA 2012*. IEEE, 261–266.
[7] Š. Čebiric, F. Goasdoué, H. Kondylakis, D. Kotzinos, I. Manolescu, G. Troullinou, and M. Zneika. 2019. Summarizing semantic graphs: a survey. *VLDB J.* 28, 3 (2019), 295–327.
[8] M. Ciglan, K. Nørvåg, and L. Hluchý. 2012. The SemSets model for ad-hoc semantic list search. In *Proc. WWW '12*. ACM, 131–140.
[9] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. 2011. Incremental graph pattern matching. In *Proc. SIGMOD 2011*. ACM, 925–936.
[10] F. Goasdoué, P. Guzewicz, and I. Manolescu. 2019. Incremental structural summarization of RDF graphs. In *Proc. EDBT 2019*. OpenProceedings.org, 566–569.
[11] T. Gottron, A. Scherp, B. Krayer, and A. Peters. 2013. LODatio: using a schema-level index to support users in finding relevant sources of linked data. In *Proc. K-CAP 2013*. ACM, 105–108.
[12] Y. Guo, Z. Pan, and J. Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.* 3, 2-3 (2005), 158–182.
[13] W.-S. Han, J. Lee, M.-D. Pham, and J. Xu Yu. 2010. iGraph: A Framework for Comparisons of Disk-Based Graph Indexing Techniques. *PVLDB* 3, 1 (2010), 449–459.
[14] J. Hegewald, F. Naumann, and M. Weis. 2006. XStruct: Efficient Schema Extraction from Multiple and Large XML Documents. In *Proc. ICDE 2006*. IEEE, 81.
[15] T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan. 2013. Observing Linked Data Dynamics. In *Proc. ESWC 2013 (LNCS)*, Vol. 7882. Springer, 213–227.
[16] A. Kansal and F. Spezzano. 2017. A Scalable Graph-Coarsening Based Index for Dynamic Graph Databases. In *Proc. CIKM 2017*. ACM, 207–216.
[17] K. Kellou-Menouer and Z. Kedad. 2015. Schema Discovery in RDF Data Sources. In *Proc. ER 2015 (LNCS)*, Vol. 9381. Springer, 481–495.
[18] M. Konrath, T. Gottron, S. Staab, and A. Scherp. 2012. SchemEX - Efficient construction of a data catalogue by stream-based indexing of linked data. *J. Web Semant.* 16 (2012), 52–58.
[19] T. Liebig, V. Vialard, M. Opitz, and S. Metzl. 2015. GraphScale: Adding Expressive Reasoning to Semantic Data Stores. In *Proc. ISWC 2015 (Posters & Demos) (CEUR Workshop Proceedings)*, Vol. 1486.
[20] N. Mihindukulasooriya, M. Poveda-Villalón, R. García-Castro, and A. Gómez-Pérez. 2015. Loupe - An Online Tool for Inspecting Datasets in the Linked Data Cloud. In *ISWC (Posters & Demos) 2015 (CEUR Workshop Proceedings)*, Vol. 1486.
[21] T. Neumann and G. Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proc. ICDE 2011*. IEEE, 984–994.
[22] E. Pietriga, H. Gözükan, C. Appert, M. Destandau, Š. Čebiric, F. Goasdoué, and I. Manolescu. 2018. Browsing Linked Data Catalogs with LODAtlas. In *Proc. ISWC 2018 (LNCS)*, Vol. 11137. Springer, 137–153.
[23] M. Rashid, G. Rizzo, N. Mihindukulasooriya, M. Torchiano, and Ó. Corcho. 2017. KBQ - A Tool for Knowledge Base Quality Assessment Using Evolution Analysis. In *Proc. K-CAP 2017 (CEUR Workshop Proceedings)*, Vol. 2065. 58–63.
[24] J. Schaible, T. Gottron, and A. Scherp. 2016. TermPicker: Enabling the Reuse of Vocabulary Terms by Exploiting Data from the Linked Open Data Cloud. In *Proc. ESWC 2016 (LNCS)*, Vol. 9678. Springer, 101–117.
[25] B. Spahiu, R. Porrini, M. Palmonari, A. Rula, and A. Maurino. 2016. ABSTAT: Ontology-Driven Linked Data Summaries with Pattern Minimalization. In *ESWC 2015 Satellite Events, Revised Selected Papers (LNCS)*, Vol. 9989. 381–395.
[26] P. Stutz, D. Strebel, and A. Bernstein. 2016. Signal/Collect12. *Semantic Web* 7, 2 (2016), 139–166.
[27] R. E. Tarjan and J. van Leeuwen. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM* 31, 2 (1984), 245–281.
[28] S. Xiong, Y. Yao, S. Li, Q. Cao, T. He, H. Qi, L. M. Tolbert, and Y. Liu. 2017. kBF: Towards Approximate and Bloom Filter based Key-Value Storage for Cloud Computing Systems. *IEEE Trans. Cloud Computing* 5, 1 (2017), 85–98.
[29] D. Yuan, P. Mitra, H. Yu, and C. L. Giles. 2012. Iterative Graph Feature Mining for Graph Indexing. In *Proc. ICDE 2012*. IEEE Computer Society, 198–209.
[30] D. Yuan, P. Mitra, H. Yu, and C. L. Giles. 2015. Updating Graph Indices with a One-Pass Algorithm. In *Proc. SIGMOD 2015*. ACM, 1903–1916.