Semantic Structural Graph Summaries for Evolving and Distributed Graphs

Dissertation zur Erlangung des Doktorgrades

Dr. rer. nat.

der Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
der Universität Ulm

vorgelegt von

Till Blume

aus Hannover

Institut für Datenbanken und Informationssysteme
Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Universität Ulm

2022
Amtierender Dekan
Maurits Ortmanns, Prof. Dr.-Ing.

Gutachter
Ansgar Scherp, Prof. Dr.-Ing
David Richerby, PhD
Ralf Schenkel, Prof. Dr.-Ing

Tag der Promotion
11. März 2022
Semantic Structural Graph Summaries for Evolving and Distributed Graphs

Dissertation zur Erlangung des Doktorgrades

Dr. rer. nat.

der Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
der Universität Ulm

vorgelegt von

Till Blume

aus Hannover

Institut für Datenbanken und Informationssysteme
Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Universität Ulm

2022
Abstract

Representing data as a graph is increasingly popular since graphs allow a more efficient and a more flexible implementation of certain applications compared to traditional relational databases. In general, such data graphs contain labeled vertices and edges, which describe the data and their relationships. In huge, heterogeneous data graphs several tasks are computationally expensive, such as cardinality computations for queries, data exploration, data visualization, vocabulary term recommendations, and related entity retrieval. Thus, often structural graph summaries are used to efficiently implement these tasks. Structural graph summaries are condensed representations of graphs such that the vertices are partitioned based on equivalent subgraphs. To determine equivalent subgraphs, only structural features are used, e.g., specific labels denoting data types. Semantic structural graph summaries extend structural graph summaries by supporting the use of concepts from ontologies as labels for vertices and edges. 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 only. Despite the different purposes and concepts followed by the existing graph summaries, we find common patterns in the captured graph structures. We abstract from these patterns and provide for the first time a formally defined common model FLUID (FLexible graph sUmmarIes for Data graphs). FLUID allows to quickly define, adapt, and compare different graph summaries for different purposes and datasets. To this end, FLUID provides features of structural summarization based on equivalence relations such as distinction of types and properties, the direction of edges, bisimulation, and semantic inference. FLUID covers all existing structural graph summary models found in the related work and beyond. Based on an extensive analysis of existing structural graph summary models and algorithms, we develop the formal model FLUID, a sequential batch algorithm for static graphs, and a parallel incremental algorithm that can also be applied to evolving graphs. The algorithms are embedded in the formal model, their computational complexity is thoroughly analyzed, and their practically demonstrated in extensive empirical evaluations.
Most of the results presented in this thesis are based on the following peer-reviewed publications.


In addition to the most relevant publications above, the research presented in this thesis contributed to the following peer-reviewed publications.


Acknowledgements

I want to express my gratitude to my supervisor, Ansgar Scherp, for giving me the opportunity to pursue my research, the guidance, and the continuous support. Likewise, I thank David Richerby for always engaging in discussions and pondering ideas with me. I would also like to express my sincere gratitude to my co-authors and reviewers for their insightful comments and suggestions.

My gratitude extends to the fantastic group at the ZBW in Kiel, with whom I had countless exciting discussions, collaborations, and excursions. I had the pleasure to travel to many overwhelming places and meet countless people who inspired my work and I feel very fortunate for this opportunity.

Finally, I want to mention that writing this thesis would not have been possible without the support of my family and friends. In particular, I want to thank my parents, Anke and Jürgen, and my brother Kai. Your advice and encouragement has allowed me to focus and never lose sight of the really important matters. I also want to specifically thank my friends Malte and Lars, who were always there to discuss strange research ideas even at nightly hours.
# Contents

## 1 Introduction

1.1 Applications .................................................. 3  
1.2 Problem Statements ........................................... 6  
1.3 Contributions .................................................. 9  
1.4 Outline .......................................................... 10  

## 2 Data Graph Models

2.1 RDF Graphs .................................................... 15  
2.2 Labeled Property Graphs ..................................... 17  
2.3 Transformation of Graph Models ............................. 19  
2.4 Discussion ...................................................... 22  
2.5 Summary ........................................................ 23  

## 3 Related Work

3.1 Structural Graph Summary Models ............................ 26  
3.2 Incremental Graph Summarization Algorithms .............. 32  
3.3 Summary ........................................................ 35
8 Empirical Evaluation of the Incremental Graph Summarization  
8.1 General Experimental Apparatus ........................................ 104  
8.2 Experiment 1: Update Performance .................................... 106  
8.3 Experiment 2: Parallelization Performance ........................ 113  
8.4 Experiment 3: Memory Consumption ................................. 119  
8.5 Summary ................................................................. 123  

9 Conclusion and Future Work  
9.1 Conclusion .............................................................. 125  
9.2 Future Work ........................................................... 127  

References  
Curriculum Vitae  
Acronyms  
Index
1

Introduction

The concept of graph databases has been known in computer science since at least 1969 [8]. Nowadays, representing data as a graph is increasingly popular since graphs allow a more efficient and a more flexible implementation of certain applications compared to relational databases [37, 51]. Relational databases operate on tabular data models and employ strict data schemas [46]. This means, typically you first define the data schema, i.e., tables, columns headers, primary keys, and foreign key relationships etc. Subsequently, the data is added to the corresponding tables, i.e., data rows are created.

Graph data models are easily expandable, which makes them suitable for complex and irregular data [37]. In general, a graph consists of vertices and edges that connect vertices [18]. A data graph is a graph where vertices and/or edges are labeled. Graph databases use vertices to represent entities such as people or places and edges to represent relationships between entities [51, 69]. Graph data models allow to flexibly add new vertex labels and edge labels, i.e., adapt to new entities and relationships. This flexibility regarding the data schema means that the inherent structure of the data may be unclear at first or evolve during the lifetime of a graph database.
Graph summarization facilitates the identification of meaning and structure in data graphs [61]. Thus, graph summaries can be used, amongst other applications, to discover the data schema of an existing graph database [23]. Different attempts have been made to classify the different graph summarization approaches [19, 23, 56, 61]. In this work, we focus on structural graph summaries, i.e., graph summaries that precisely capture specific structural features of the data graph [23].

Structural graph summaries are condensed representations of graphs such that a set of chosen (structural) features of the graph summary are equivalent to the features in the original graph. To achieve this, structural graph summaries partition vertices based on the equivalence of the subgraphs centered on them. To determine subgraph equivalences, only structural features are used, such as specific combinations of labels. Structural graph summaries do not include statistical approaches such as sampling or pattern-mining approaches [23]. Other graph summarizing approaches focus on “neighborhood preserving” compression, i.e., where plain unlabeled graphs are summarized into super graphs that (approximately) preserve the number of neighbors each vertex has [57, 83]. We focus on structural graph summaries since many real-life applications require exact matches [36].

Semantic structural graph summaries extend structural graph summaries by supporting the use of concepts from ontologies as labels for vertices and edges [23, 58, 84]. Ontologies model hierarchical relationships between concepts, i.e., relationships between types and properties with which the vertices and edges are labeled, e.g., the type Proceedings is a subtype of the type Book [1]. Following this subtype relation allows us to infer the type Book for all vertices that are labeled with the type Proceedings.

Structural graph summaries are often an order of magnitude smaller than the input graph [23]. As graph summaries share the structural features observed in the input graph, they are also often used as indices to search for equivalent subgraphs [23, 42]. In the example depicted in Figure 1.1, 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 [68], data exploration [10, 64, 74, 84], data visualization [39], vocabulary term recommendations [80], and related entity retrieval [25]. Next, we describe three applications in more detail and use them throughout the thesis to motivate certain design decisions. Subsequently, we describe open problems regarding structural graph summaries and the contributions of this work to close them.
1.1 Applications

Many different structural graph summaries have been developed to solve different tasks [10, 21, 25, 27, 40, 54, 58, 64, 65, 68, 80, 81, 84, 88]. In the following, we describe three common applications of structural (semantic) graph summaries in detail. These applications are typical applications for managed graph database systems but also for the decentralized Semantic Web [1].

1.1.1 Semantic Entity Retrieval

One application of structural graph summaries is to find semantically related entities in the Semantic Web [25]. Entities on the Semantic Web are represented using vertices and edges and typically stored as a Resource Description Framework (RDF) graph [1]. Each entity is identifiable by an International Resource Identifier (IRI) [33], i.e., a vertex. One approach to determine semantically related entities is to summarize those entities that share a common semantic representation, i.e., which RDF types and RDF properties are used to describe the entities. If two entities share the same set of RDF types and RDF properties, we can say they are semantically related [25]. Structural graph summaries summarize vertices, and therefore entities, based on such structural subgraph features (compare example presented in Figure 1.1). We can solve the task of finding semantically related entities by memorizing the vertex identifiers of each summarized vertex in the computed structural graph summary. Using the graph summary as an index allows constant-time retrieval of related vertices.

Figure 1.1: An example data graph and part of its graph summary that uses the SchemEX graph summary model [58]. The graph summary is equivalent to the data graph regarding vertex label and edge label up to a distance of one vertex-hop but uses different vertex identities.

1. Applications

Many different structural graph summaries have been developed to solve different tasks [10, 21, 25, 27, 40, 54, 58, 64, 65, 68, 80, 81, 84, 88]. In the following, we describe three common applications of structural (semantic) graph summaries in detail. These applications are typical applications for managed graph database systems but also for the decentralized Semantic Web [1].

1.1.1 Semantic Entity Retrieval

One application of structural graph summaries is to find semantically related entities in the Semantic Web [25]. Entities on the Semantic Web are represented using vertices and edges and typically stored as a Resource Description Framework (RDF) graph [1]. Each entity is identifiable by an International Resource Identifier (IRI) [33], i.e., a vertex. One approach to determine semantically related entities is to summarize those entities that share a common semantic representation, i.e., which RDF types and RDF properties are used to describe the entities. If two entities share the same set of RDF types and RDF properties, we can say they are semantically related [25]. Structural graph summaries summarize vertices, and therefore entities, based on such structural subgraph features (compare example presented in Figure 1.1). We can solve the task of finding semantically related entities by memorizing the vertex identifiers of each summarized vertex in the computed structural graph summary. Using the graph summary as an index allows constant-time retrieval of related vertices.

1. Applications

Many different structural graph summaries have been developed to solve different tasks [10, 21, 25, 27, 40, 54, 58, 64, 65, 68, 80, 81, 84, 88]. In the following, we describe three common applications of structural (semantic) graph summaries in detail. These applications are typical applications for managed graph database systems but also for the decentralized Semantic Web [1].

1.1.1 Semantic Entity Retrieval

One application of structural graph summaries is to find semantically related entities in the Semantic Web [25]. Entities on the Semantic Web are represented using vertices and edges and typically stored as a Resource Description Framework (RDF) graph [1]. Each entity is identifiable by an International Resource Identifier (IRI) [33], i.e., a vertex. One approach to determine semantically related entities is to summarize those entities that share a common semantic representation, i.e., which RDF types and RDF properties are used to describe the entities. If two entities share the same set of RDF types and RDF properties, we can say they are semantically related [25]. Structural graph summaries summarize vertices, and therefore entities, based on such structural subgraph features (compare example presented in Figure 1.1). We can solve the task of finding semantically related entities by memorizing the vertex identifiers of each summarized vertex in the computed structural graph summary. Using the graph summary as an index allows constant-time retrieval of related vertices.

1. Applications

Many different structural graph summaries have been developed to solve different tasks [10, 21, 25, 27, 40, 54, 58, 64, 65, 68, 80, 81, 84, 88]. In the following, we describe three common applications of structural (semantic) graph summaries in detail. These applications are typical applications for managed graph database systems but also for the decentralized Semantic Web [1].

1.1.1 Semantic Entity Retrieval

One application of structural graph summaries is to find semantically related entities in the Semantic Web [25]. Entities on the Semantic Web are represented using vertices and edges and typically stored as a Resource Description Framework (RDF) graph [1]. Each entity is identifiable by an International Resource Identifier (IRI) [33], i.e., a vertex. One approach to determine semantically related entities is to summarize those entities that share a common semantic representation, i.e., which RDF types and RDF properties are used to describe the entities. If two entities share the same set of RDF types and RDF properties, we can say they are semantically related [25]. Structural graph summaries summarize vertices, and therefore entities, based on such structural subgraph features (compare example presented in Figure 1.1). We can solve the task of finding semantically related entities by memorizing the vertex identifiers of each summarized vertex in the computed structural graph summary. Using the graph summary as an index allows constant-time retrieval of related vertices.

1. Applications

Many different structural graph summaries have been developed to solve different tasks [10, 21, 25, 27, 40, 54, 58, 64, 65, 68, 80, 81, 84, 88]. In the following, we describe three common applications of structural (semantic) graph summaries in detail. These applications are typical applications for managed graph database systems but also for the decentralized Semantic Web [1].

1.1.1 Semantic Entity Retrieval

One application of structural graph summaries is to find semantically related entities in the Semantic Web [25]. Entities on the Semantic Web are represented using vertices and edges and typically stored as a Resource Description Framework (RDF) graph [1]. Each entity is identifiable by an International Resource Identifier (IRI) [33], i.e., a vertex. One approach to determine semantically related entities is to summarize those entities that share a common semantic representation, i.e., which RDF types and RDF properties are used to describe the entities. If two entities share the same set of RDF types and RDF properties, we can say they are semantically related [25]. Structural graph summaries summarize vertices, and therefore entities, based on such structural subgraph features (compare example presented in Figure 1.1). We can solve the task of finding semantically related entities by memorizing the vertex identifiers of each summarized vertex in the computed structural graph summary. Using the graph summary as an index allows constant-time retrieval of related vertices.

1. Applications

Many different structural graph summaries have been developed to solve different tasks [10, 21, 25, 27, 40, 54, 58, 64, 65, 68, 80, 81, 84, 88]. In the following, we describe three common applications of structural (semantic) graph summaries in detail. These applications are typical applications for managed graph database systems but also for the decentralized Semantic Web [1].

1.1.1 Semantic Entity Retrieval

One application of structural graph summaries is to find semantically related entities in the Semantic Web [25]. Entities on the Semantic Web are represented using vertices and edges and typically stored as a Resource Description Framework (RDF) graph [1]. Each entity is identifiable by an International Resource Identifier (IRI) [33], i.e., a vertex. One approach to determine semantically related entities is to summarize those entities that share a common semantic representation, i.e., which RDF types and RDF properties are used to describe the entities. If two entities share the same set of RDF types and RDF properties, we can say they are semantically related [25]. Structural graph summaries summarize vertices, and therefore entities, based on such structural subgraph features (compare example presented in Figure 1.1). We can solve the task of finding semantically related entities by memorizing the vertex identifiers of each summarized vertex in the computed structural graph summary. Using the graph summary as an index allows constant-time retrieval of related vertices.
since all semantically related vertices are summarized together, i. e., have the same index key. For evolving databases, however, it is important to update such an index efficiently. Recomputing an index based on a graph summary from scratch when the graph database changes may decrease the overall performance of an implementation.

1.1.2 Cardinality Computation

Cardinality computation is often desired in databases [68, 77]. For graph databases, graph summaries can be used as an index to look-up how many vertices will be returned by certain queries. For example in Figure 1.1, we can query the graph summary for all vertices labeled with the type "Book." If the graph summary is smaller than the database, the query yields results faster on the graph summary than on the database. This task is related to the finding semantically related entities task described above. However, for this purpose, we only need to memorize the number of summarized vertices rather than memorizing all vertex identifiers. For very large databases, storing, e. g., a 64-bit numeric value, instead of billions of arbitrary length identifiers (see IRI [33]) greatly reduces the space consumption. A graph summary that memorizes the number of summarized vertices enables fast implementations of query size estimation [68]. Analogously to the semantically related entities task above, expensive recomputation of the graph summary from scratch when the database changes can yield an unwanted performance overhead that potentially reduces the overall performance of the implementation.

Furthermore, memorizing the number of summarized vertices can be used as an indicator to assess the data completeness in knowledge bases [77]. Knowledge bases often have a data schema or ontology that defines how entities should be modeled. Structural graph summaries help to determine how many entities strictly follow the schema, match the schema only partially, or even contradict the schema. The number of entities that have missing information according to the schema is one indicator of the completeness of a knowledge base and, thus, contributes to assess the overall quality of a knowledge base. It is often desired to evaluate the evolution of data quality over time [77]. To trace the evolution of data completeness, an incremental update mechanism is needed.
1.1.3 Data Source Search

Data search is still considered an open but important problem [24]. We consider the task of data source search, where one needs to find (sub)graphs in the Semantic Web that match a given schema structure [42]. Structural graph summaries can be used as an index that memorizes the location of summarized vertices on the Web. This is illustrated in Figure 1.2. Given a SPARQL (SPARQL Protocol and RDF Query Language) [47] query as shown in Listing 1.1, such an index can return Uniform Resource Locators (URLs) of data sources ?ds that contain bibliographic metadata.

Listing 1.1: Structural query to find data sources ?ds containing information about proceedings that have a title and an author that is a person.

```
SELECT ?ds WHERE {
  ?ds rdfs:typeof Proceedings ;
  _:_author _:Person ;
  _:_title ?_ .
}
```

First, a structural graph summary is queried to identify relevant data sources matching this query. Then, the data sources are accessed to download the graphs matching the query. Search systems like LODatio [42], LODeX [10], Loupe [64], and LODatlas [74] rely on structural graph summaries to offer a search for relevant data sources or exploration of data sources.

To implement this task, we need to memorize the identifiers of the locations where each summarized vertex appeared. As the data on the Web changes [52], the summaries need to be updated as well.

Figure 1.2: Finding data sources on the Web using an index based on graph summaries. A structural query is executed over an index to identify relevant data sources (1). Subsequently, the data sources are accessed to retrieve actual vertices (2).
In contrast to the previous two tasks, for data source search we neither memorize vertex identifiers nor the number of summarized vertices but only their location. Note that in the Semantic Web, often multiple vertices are defined in a single data source but also single vertices are defined across multiple data sources [52]. However, for the Semantic Web, we can expect the number of memorized location identifiers to be less than the number of vertex identifiers in the semantic entity retrieval task [52].

1.2 Problem Statements

Structural graph summarization is the task of finding a condensed representation $SG$ (Summary Graph) of an input graph $G$ such that a set of chosen (structural) features are equivalent in $SG$ and the original graph [23]. Intuitively, structural graph summarization means that we can conduct specific tasks – e.g., counting the vertices with a specific type label – directly on $G$ or, alternatively, obtain the same information from the structural graph summary $SG$. The fundamental idea of structural graph summaries is that the task can be completed much faster on the graph summary than on the original graph.

1.2.1 Defining Structural Graph Summary Models

Many different structural graph summaries have been developed for different purposes, capturing different structural features of graphs [10, 21, 25, 27, 40, 54, 58, 64, 65, 68, 80, 81, 84, 88]. The problem with this plethora of existing summary models is that each model defines its own data structure that is designed for solving only a specific task and is evaluated using different metrics [10, 25, 42, 64, 68, 74, 80, 84]. Our hypothesis is that there is no structural graph summary model that fits all tasks and that the performance depends on the specific types of queries and characteristics of the datasets. So far only very limited work has been done on understanding the behavior of graph summary models in different application contexts. With each graph summary model evaluated independently in a specific context, it remains difficult to judge whether a model generalizes well to another task or not. In other words, it is not known which graph summary model can be used for which contexts, tasks, and datasets. To tackle this, we need a single, generalizable framework to compute structural graph summaries [22, 76, 88].

All structural graph summaries essentially partition vertices based on equivalent subgraphs. Equivalence relations describe any graph partitioning in a formal way [35]. Thus, we can define structural graph summary models using equivalence relations [22, 76, 88]. The problem is that for any
finite set, there are exponentially many possible equivalence relations. More precisely, the number of equivalence relations is denoted by the Bell Number \[26\]. We argue that for structural graph summaries, not all partitions are useful and that a subset of all possible equivalence relations is sufficient. Restricting the number of possible equivalence relations has the advantage that it allows to define a single parameterized algorithm to compute all of them. To the best of our knowledge, no such algorithm exists yet. Such a parameterized algorithm is the foundation for a single, generalizable framework to compute structural graph summaries.

**Example 1.** The data graph shown in the left of Figure 1.1 contains two graphs \(G_1\) and \(G_2\). Both graphs contain vertices labeled Book, Subject, and Person and edges labeled topic and author. One can define a graph summary using an equivalence relation \(\sim\) that summarizes vertices in the data graph 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 and a new subgraph in the graph summary is created. For instance, since \(v_1\) and \(v_7\) are equivalent under \(\sim\) they are summarized by the same subgraph in the graph summary \((v_1)\). This part of the graph summary is shown in the right of Figure 1.1. It preserves the information about the combinations of graph label (Book \(\rightarrow\) topic \(\rightarrow\) Subject and Book \(\rightarrow\) author \(\rightarrow\) Person) found in the data graph. Note that the data graph can, in principle, be distributed, e. g., on the Web where \(G_1\) and \(G_2\) would be hosted on different Web servers. In Figure 1.1, this is indicated by labeling the graphs with two sources, “X” for \(G_1\) and “Y” for \(G_2\).

### 1.2.2 Updating Structural Graph Summaries for Evolving Graphs

When the data graph evolves over time, it is often prohibitively expensive to recompute the structural graph summary from scratch. In particular, when only few changes occur in relation to the overall size of the data graph. Thus, an algorithm is needed that can efficiently update computed structural graph summaries. However, existing incremental algorithms have several limitations.

For example, some algorithms incrementally process parts of the input graph \([7,39,49,58]\). This means, however, that these algorithms do not consider modifications and deletions, which we can expect in graphs that evolve over time. Other approaches often assume the availability of a change log that explicitly details the added and removed edges of the graph \([67,95]\). Especially for graphs on the
Figure 1.3: Evolving data graph and part of its graph summary that uses the SchemEX model [58].

Semantic Web, updates often do not provide a reliable change log to the previous version [31, 52]. This could be overcome by storing local copies of the data graphs [31] but this is impractical for huge graphs like the Semantic Web [31, 78]. So far, to the best of our knowledge, there is no suitable solution to update only these parts of the graph summary that need to be updated, without having a change log and without losing the high compression of structural graph summaries by storing local copies of the data graph.

Example 2. This example is based on Example 1. The data graph at time $t$ shown in the top-left of Figure 1.3 shows the same data graph as described in Example 1. At time $t+1$, shown in the bottom-left of Figure 1.3, the data graph has changed. While vertex $v8$ was labeled Person at time $t$, it is labeled Agent at time $t+1$. This means, the structural graph summary needs to be updated. Since $v1$ and $v7$ are no longer equivalent under $\sim$ they are now summarized by different subgraphs in the graph summary.
1.3 Contributions

In this thesis, we present our formal common model FLUID (FLExible graph sUmmarIes for Data graphs) to quickly define, adapt, and compare different graph summaries for (semantic) graphs and two algorithms to compute structural graph summaries defined with FLUID over static graphs and evolving graphs.

We analyze existing structural graph summaries and identify commonly used, generic features. Based on our analysis, we develop a small set of operators to define structural summarization based on equivalence relations, i.e., FLUID. FLUID supports features such as distinction of types and properties, the direction of edges, bisimulation, and inference. We show how FLUID covers all existing structural graph summary models found in the related work and beyond. Furthermore, based on FLUID, we develop a parameterized implementation that covers all FLUID features. Based on this parameterized implementation of FLUID, we propose two algorithms to compute structural graph summaries defined with FLUID over static and evolving graphs.

For static graphs, we developed a sequential batch algorithm. We show that our algorithm computes all graph summaries defined with FLUID in the worst case in $O(n^2)$ w.r.t. $n$ edges in the data graph, if semantic inferencing is used. For all other graph summaries that do not use semantic inferencing, we can compute structural graph summaries in essentially linear-time using hash maps. Furthermore, we analyze large-scale graphs obtained from the Semantic Web with billions of edges and find indications that even using semantic inferencing, the average complexity is bounded by $\Theta(n)$. We evaluate the compression ratio and summarization ratio of seven carefully selected, representative graph summary models computed with our sequential batch algorithm. We show that although structural graph summaries are usually an order of magnitude smaller than the input graph, there are huge variations between the different models. In addition, we evaluate a stream-based implementation of the sequential batch algorithm. The stream-based approach is designed to scale to graphs of arbitrary sizes by observing the graph over a stream of edges with fixed window size. Inherently, this approach introduces inaccuracies in the graph summary computation. Thus, we evaluate the approximation quality of the graph summaries using F1-score. Again, we observe huge variations in F1-scores between the different models. However, we find statistical significant correlations between between compression ratio, summarization ratio, and approximation quality.

For evolving graphs, we propose a parallel incremental algorithm. We show that all FLUID graph summaries can be updated in $O(\Delta \cdot d^k)$, where $\Delta$ is the number of additions, deletions, and modifications in the input graph, $d$ is the maximum degree of the input graph, and $k$ is the maximum
radius of the neighborhoods considered. We empirically evaluate the performance of our parallel algorithm on synthetic benchmark datasets and real-world datasets. Our experiments show that for commonly used graph summary models and datasets, the parallel incremental summarization algorithm almost always outperforms their parallel batch counterpart, even when about 50% of the graph database changes. Furthermore, the incremental summarization algorithm outperforms the batch summarization algorithm when using fewer cores, despite that the batch computation benefits more from parallelization than the incremental algorithm. Moreover, using the real-world datasets, we find that using four cores, the incremental algorithm is on average 5 to 44 times faster while only producing a memory overhead of 8% (±1%).

In summary, based on an extensive analysis of existing structural graph summary models and algorithms, we developed the formal model FLUID, a sequential batch algorithm for static graphs, and a parallel incremental algorithm that can also be applied to evolving graphs. The algorithms are embedded in the formal model, their computational complexity is thoroughly analyzed, and their practicality is demonstrated in extensive empirical evaluations.

Furthermore, we developed an extension to the existing data source search engine LODatio [42] that allows the usage of different structural graph summaries. Based on this, we developed the IMPULSE (Integrate Public Metadata Underneath professional Library SErvices) framework.* Using IMPULSE, we integrated millions of publicly available bibliographic metadata records into the EU H2020 funded MOVING platform [89]. We also used IMPULSE to integrate publicly available metadata about legal documents into the open source platform Open Legal Data [70].

1.4 Outline

The remainder of this thesis is structured as follows: Next, we formalize the two commonly used graph models Labeled Property Graph (LPG) and Resource Description Framework (RDF). We show how to transform LPGs and RDF graphs into each other, thus, bridging the two worlds of industry standard graph databases and the Semantic Web.

In Chapter 3, we describe related works. First, we analyze existing (semantic) structural graph summaries to identify common features. Second, we discuss existing algorithms to incrementally update (semantic) structural graph summaries. In Chapter 4, we introduce the formal model FLUID.

*Source code available on GitHub: https://github.com/t-blume/impulse
We define FLUID’s basic building blocks, i.e., schema elements and parameterizations as well as payload elements. We summarize FLUID by defining all graph summaries analyzed in Chapter 3 using FLUID. Finally, we define a generic data structure based on the LPG graph model to store all structural graph summaries defined with FLUID.

In Chapter 5, we present our single, parameterized, sequential algorithm to compute structural graph summaries defined with FLUID. We analyze the computational complexity of our algorithm regarding space and time. We pay particular attention to the impact of each parameterization on the overall complexity. In Chapter 6, we empirically evaluate the sequential algorithm using representative structural graph summaries. In three sets of experiments, we evaluate the compression and summarization ratios, the quality of a stream-based implementation, and the impact of the inferencing parameterization on the graph summaries’ size.

In Chapter 7, we present our single, parameterized, parallel, and incremental algorithm. First, we present and analyze the algorithm when used to compute graph summaries in batch mode over static graphs. Second, we present our incremental update mechanism for evolving graphs and analyze its update complexity. In Chapter 8, we empirically evaluate the parallel, incremental algorithm using representative structural graph summaries. In three sets of experiments, we compare the performance of the incremental algorithm compared to the batch algorithm, we evaluate the impact of the parallelization on the overall performance, and we evaluate the memory consumption.

Finally, in Chapter 9, we conclude by summarizing the main contributions of this thesis and discuss possible future works.
Data graph models are essentially a set of data structure types used to define data graphs. Data graph models in combination with operators and integrity rules make up the graph database model [3]. There are currently two major competing data graph models, namely the Resource Description Framework (RDF) and Labeled Property Graphs (LPG). RDF graphs are directed, edge-labeled multigraphs where a subset of vertices are treated differently, i.e., blank nodes and literals [51]. LPGs do not have special vertices like blank nodes and literals but allow so-called key–value properties (key–value pairs) to be added to each vertex and edge. Intuitively, RDF represents all information as edges, e.g., properties of vertices are edges in the graph and types of vertices are linked via an edge labeled \texttt{rdf:type}. In contrast, LPGs allow vertex labels (types) and, furthermore, property information to be not only edge labels but also key–value pairs attached to vertices and also edges. A comprehensive
Table 2.1: Overview of the most important graph notations. Note that the symbol $G$ to denote graphs is overloaded since often both, LPGs and RDF graphs can be used. In any other case, the alternative symbol is used. Analogously to $G$, the General symbols are also applicable for RDF and LPG graphs.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDB</td>
<td>Labeled Property Graph database $(V, E, G, \ell_G)$</td>
</tr>
<tr>
<td>$V, E$</td>
<td>Vertices and edges of a GDB</td>
</tr>
<tr>
<td>$G$</td>
<td>Multi-set ${G_1, \ldots, G_n}$ of graphs in GDB</td>
</tr>
<tr>
<td>$G \in G$ or $G_{LPG}$</td>
<td>LP Graph $G = (V_G, E_G, \ell_V, \ell_E, \ell_P)$</td>
</tr>
<tr>
<td>$V_G \subseteq V$</td>
<td>Vertices appearing in graph $G$</td>
</tr>
<tr>
<td>$E_G \subseteq E$</td>
<td>Edges appearing in graph $G$</td>
</tr>
<tr>
<td>$\ell_V : V \rightarrow P(\Sigma_{V})$</td>
<td>Labeling function for all vertices in GDB</td>
</tr>
<tr>
<td>$\ell_E : E \rightarrow P(\Sigma_{E})$</td>
<td>Labeling function for all edges in GDB</td>
</tr>
<tr>
<td>$\ell_P : V \cup E \rightarrow P(\Sigma_{Key} \times \Sigma_{Value})$</td>
<td>Property function for all vertices and edges in GDB</td>
</tr>
<tr>
<td>$\ell_G : G \rightarrow \Sigma_{G}$</td>
<td>Labeling function for all graphs in GDB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RDF</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$ or $G_{RDF}$</td>
<td>RDF Graph $G \subseteq (V_I \cup V_B) \times P \times (V_I \cup V_B \cup L)$</td>
</tr>
<tr>
<td>$G$ or $G_{N,RDF}$</td>
<td>Named RDF Graph $G \subseteq (V_I \cup V_B) \times P \times (V_I \cup V_B \cup L) \times (V_I \cup V_B)$</td>
</tr>
<tr>
<td>$V_I$</td>
<td>The set of IRIs</td>
</tr>
<tr>
<td>$V_B$</td>
<td>The set of blank nodes</td>
</tr>
<tr>
<td>$P \subseteq V_I$</td>
<td>The set of predicates</td>
</tr>
<tr>
<td>$L$</td>
<td>The set of literals</td>
</tr>
<tr>
<td>$V_C \subseteq V_I$</td>
<td>The set of all RDF types</td>
</tr>
<tr>
<td>$(s, p, a) \in G$</td>
<td>A subject-predicate-object triple</td>
</tr>
<tr>
<td>$(s, p, o, d) \in G$</td>
<td>A subject-predicate-object-source quad (for Named RDF graphs)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mapping</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$V \subseteq V_I \cup V_B$</td>
<td>The set of vertices in an LPG is a subset of IRIs and blank nodes</td>
</tr>
<tr>
<td>$\Sigma_{V} = V_C$</td>
<td>LPG vertex labels are RDF types</td>
</tr>
<tr>
<td>$\Sigma_{E} \cup \Sigma_{P} = P \setminus {\text{rdf\text{:type}}}$</td>
<td>LPG edge labels are the set of RDF properties</td>
</tr>
<tr>
<td>$\Sigma_{Value} = L$</td>
<td>The set of LPG property values is the set of RDF literals</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>General</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma : V \rightarrow P(V)$</td>
<td>The set of $v$’s neighbors</td>
</tr>
<tr>
<td>$\Gamma^+ : V \rightarrow P(V)$</td>
<td>The set of $v$’s outgoing neighbors</td>
</tr>
<tr>
<td>$\Gamma^- : V \rightarrow P(V)$</td>
<td>The set of $v$’s incoming neighbors</td>
</tr>
<tr>
<td>$d, d^+, d^-$</td>
<td>degree, outdegree, and indegree in G</td>
</tr>
<tr>
<td>$\ell_T : V \rightarrow P(V_C)$</td>
<td>The type set of a vertex</td>
</tr>
<tr>
<td>$\ell_P : V \rightarrow P(P)$</td>
<td>The property set of a vertex (not including $\text{rdf\text{:type}}$)</td>
</tr>
<tr>
<td>$\ell_P : V \rightarrow P(P)$</td>
<td>The incoming property set of a vertex (not including $\text{rdf\text{:type}}$)</td>
</tr>
</tbody>
</table>

Overview of popular graph models with accompanying examples highlighting advantages and disadvantages is presented by Hogan et al. \cite{51}. Next, we define and discuss these two graph models in detail. We then show that both graph models can be used interchangeably. This allows us to use the term data graph, which subsumes RDF graphs and LPGs, for the remainder of this thesis.
RDF graphs are widely adopted on the Web because they are standardized by the World Wide Web Consortium (W3C) [28]. Conceptually, an RDF graph is an edge-labeled directed graph. An RDF graph [28] is a set of triples \((s, p, o)\), with a subject \(s\), predicate \(p\), and object \(o\). Each triple denotes a directed edge from the subject vertex \(s\) to the object vertex \(o\), the edge being labeled with the predicate \(p\).

RDF graphs distinguish three kinds of vertices, namely International Resource Identifiers (IRIs) [33], blank nodes, and literals, each of which has a different role.

Formally, an RDF graph is defined as \(G \subseteq (V_I \cup V_B) \times P \times (V_I \cup V_B \cup L)\), where \(V_I\) denotes the set of IRIs, \(V_B\) the set of blank nodes, \(P \subseteq V_I\) the set of predicates (also identified by IRIs), and \(L\) the set of literals (represented by strings) [28]. IRIs conceptually correspond to real-world entities and are globally unique: an IRI may be included in more than one RDF graph, but this corresponds to stating different facts about the same real-world entity. In contrast, blank nodes are only locally defined, within the scope of a specific RDF graph, to serve special data modeling tasks. Through skolemization, blank nodes can be turned into Skolem IRIs, which are globally unique [28, Section 3.5]. Literal vertices are finite strings of characters from a finite alphabet such as Unicode [28]. Thus, two literal vertices which are term-equal (i.e., are the same string) [28, Section 3.3] are the same vertex. Although IRIs, blank nodes, and literals have different roles in RDF, this distinction is not relevant in this work and we treat them equally as vertices.

Predicates \(p \in P\) act as edge labels. The RDF standard includes the predicate \(\text{rdf}:\text{type}\), which is used to simulate vertex labels: the triple \((s, \text{rdf}:\text{type}, o)\) denotes the vertex \(s\) having label \(o\). Such vertex labels are called RDF types. This indirect representation of vertex labels is a design decision of RDF and contrasts with the direct use of vertex labels in labeled property graphs [51]. Edges in an RDF graph which are not labeled with \(\text{rdf}:\text{type}\) are called RDF properties.

We define the set \(\ell_T(s) := \{o \in V_I \mid (s, \text{rdf}:\text{type}, o) \in G\}\) as the type set of a vertex \(s\) in an RDF graph [43]. We define the set \(\ell_P(s) := \{p \in P \mid (s, p, o) \in G\text{ for some }o\text{ and }p \neq \text{rdf}:\text{type}\}\) as the property set of a vertex \(s\) in an RDF graph [43]. We write \(V_C\) for the set of all RDF types, i.e., the set \(\{o \in V_I \mid (s, \text{rdf}:\text{type}, o) \in G\text{ for some }s\}\). Furthermore, we write \(\Gamma^+(s) = \{o \mid (s, p, o) \in G\text{ for some }p\}\) for the set of outgoing neighbors of \(s\) in the RDF graph \(G\) and \(\Gamma^-(o) = \{s \mid (s, p, o) \in G\text{ for some }p\}\) for the set of \(o\)’s incoming neighbors. Finally, we define \(\ell_p^-(o) := \{p \mid (s, p, o) \in G\text{ for some }s\text{ and }p \neq \text{rdf}:\text{type}\}\) as the incoming property set.
An example RDF graph is shown in Figure 2.1. On the left-hand side, the RDF graph is shown as set of triples. On the right-hand side, it is depicted as a graph. The vertex $v_1$ has the type set $\ell_T(v_1) = \{\text{Proceedings}\}$ and the property set $\ell_P(v_1) = \{\text{author, title}\}$. The vertex $v_2$ has the type set $\ell_T(v_2) = \{\text{Person}\}$ and the property set $\ell_P(v_2) = \{\text{name}\}$. But, $v_1$ has predicates $\{\text{author, title, rdf:type}\}$ and outgoing neighbors $\Gamma^+(v_1) = \{\text{Proceedings, v2, "Graph Database"}\}$.

A special characteristic of RDF graphs is their support for semantic labels, which allows the inference of implicit information. Such semantic labels are from ontologies, where semantic relationships between types and properties are denoted, e.g., with predicates from the RDF Schema vocabulary. RDF Schema (RDFS) and its entailment rules are standardized by the W3C [20]. A comprehensive overview of these rules is presented in [1].

For example, the semantics of a triple $(p, \text{rdfs:subPropertyOf}, p') \in G$ states that for any subject vertex $s$ with $(s, p, o) \in G$, we can infer existence of the additional triple $(s, p', o)$. This means, when using RDF Schema inference, each vertex may have more types and properties in its type set and property set, respectively. Finally, RDF supports the definition of named graphs [28]. Following the W3C standard [28], each named graph is a pair consisting of the graph name, which is an IRI, and an RDF graph as defined above.* Following Harth et al. [48], we formalize named graphs by extending each triple $(s, p, o)$ to a tuple $((s, p, o), d)$ or a quad $(s, p, o, d)$, where $d$ denotes the name of the data source from which the triple originated [96]. In this work, the data source $d$ of a tuple is only used as payload and not to determine the equivalence of vertices.

*Formally, blank nodes are also allowed as graph names but this is of limited practical use, since blank nodes cannot be used as URLs.
2.2 Labeled Property Graphs

Labeled Property Graphs (LPGs) are often used in graph databases [59] and multi-modal databases [69], as all kinds of information – including complete documents – can be attached as key–value properties [37].

There exist a variety of formal definitions of LPGs, which emphasize different aspects [4, 25]. In general, LPGs are defined as graphs in which vertices and edges are labeled and can have key–value properties. Thus, we define an LPG as $G_{\text{LPG}} = (V, E, \ell_V, \ell_E, \ell_P)$ with vertices $V$, edges $E$, and labeling functions $\ell_V, \ell_E, \ell_P$. Furthermore, we define finite alphabets $\Sigma_V$ (vertex labels), $\Sigma_E$ (edge labels), $\Sigma_{\text{Key}}$ (property keys), and $\Sigma_{\text{Value}}$ (property values).

We define labeling functions $\ell_V, \ell_E$ for vertices and edges. The first function $\ell_V: V \to \mathcal{P}(\Sigma_V)$ maps each vertex to zero or more labels from the finite alphabet $\Sigma_V$. The second function $\ell_E: E \to \mathcal{P}(\Sigma_E)$ maps each edge to zero or more labels from the finite alphabet $\Sigma_E$. Furthermore, $\ell_P: V \cup E \to \mathcal{P}(\Sigma_{\text{Key}} \times \Sigma_{\text{Value}})$ maps each vertex $v$ and each edge $e$ to zero or more key–value pairs $\Sigma_{\text{Key}} \times \Sigma_{\text{Value}}$, i. e., the properties.

In our LPG definition, 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 $\ell_E(v, w) = \ell_E(w, v)$.

Vertex labels are often used to represent type information and edge labels to denote specific relationships between vertices. Information about a specific relationship can be added via key–value properties. Key–value properties can also be added to vertices. This can be used to state a property of a vertex $v$ as an alternative to creating a new vertex $w$ and an edge relating $w$ to $v$ [51]. For example, in Figure 2.2, we represent the same graph from Figure 2.1 as LPG. We can simulate RDF literals by creating vertices $v_3, v_4$ and labeling them with their literal values as key–value properties. This way, with the exception of the rdf:type predicates, Figure 2.1 and Figure 2.2 have the same edges. Alternatively, one could add the title “Graph Database” and the name “Max Power” as a key–value properties to vertex $v_1$ and $v_2$, respectively. This way, the LPG contains even fewer vertices and edges.

Notably, the definition of LPGs deliberately allows to define vertices without edges but no edges without vertices. Vertices can be labeled and can have key–value properties. Thus, they can add valuable information to a graph even without incoming or outgoing edges. In contrast, edges between two vertices imply the existence of these vertices, thus, they also have to exist.
LPGs cover a wide range of aspects since all kind of information can be attached as key–value properties [37]. In particular, provenance information can be attached as key–value properties to all vertices and edges. However, when working with physically and logically distributed graphs, precise statements about the origin of certain information are desirable, e. g., the location inside a distributed database. To formalize this distributed aspect of LPGs, we introduce the notion of Labeled Property Graph Database (GDB). We define a GDB as a multiset of labeled property graphs with shared vertices and edges. Multisets of graphs allow data replication, which is particularly important when graph databases are distributed without a centralized management such as in the Semantic Web.

Formally, we define a graph database $GDB = (V, E, G, ℓ_G)$, where $V$ is a set of vertices, $E \subseteq V \times V$ is the set of edges, $G = \{G_1, \ldots, G_n\}$ is a multiset of LPGs, and $ℓ_G$ is a labeling functions for graphs. Each labeled property graph $G \in G$ is a tuple: $G = (V_G, E_G, ℓ_V, ℓ_E, ℓ_P)$ with $V_G \subseteq V$ and $E_G \subseteq E$. We define a fourth labeling function $ℓ_G : G \rightarrow \Sigma_G$ that maps each instance of each graph $G \in G$ to a single label from the finite alphabet $\Sigma_G$. This graph label is the name of the graph.

The sets $V, E$ are not multisets, i. e., all vertices and edges are uniquely identified within the GDB, e. g., by IRIs on the Web. The graphs $G \in G$ 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 $V_{G_i} \cap V_{G_j} \neq \emptyset$. This is a design decision that allows vertices and edges in a graph $G$ to be implicitly labeled with the graph label of $ℓ_G(G)$. If vertices and edges appear in multiple graphs, consequently they have multiple graph labels. In principal, a GDB can be used exactly like an LPG, as presented above. In this case, vertex labels denote types, edge labels classify relationships, and key–value properties are used for information only relevant to a single vertex or edge, i. e., when adding an
additional vertex or edge adds no additional meaning to the graph. When we use GDBs, we label the
different graphs contained in the multiset \( G \). Following our implied semantics, labeling \( G \) is equivalent
to adding the property “graph name” to all vertices and edges contained in this graph \( G \).

Thus, GDBs offer a shortcut to define this concept of labeling (or naming) LPGs to ease notation
when general statements about a set of vertices and edges are made. Note, though, that an LPG with
a “graph name” is not formally equivalent to an GDB, as it allows inconsistencies that cannot occur
in GDBs. For example, in a GDB, for every edge \((x, y)\) in, say, \( G_1 \), the vertices \( x \) and \( y \) must also be
in \( G_1 \). However, an LPG could give \( x, y \) and \((x, y)\) arbitrary values of the graph name property. Thus,
GDBs are more than a shortcut but formalize the distributed aspect of LPGs.

2.3 Transformation of Graph Models

In this section, we discuss how RDF graphs and LPGs can be transformed into each other. To this
end, we first describe how the different sets of vertices, edges, and label sets relate to each other. Subse-
quently, we provide simple algorithms to transform RDF graphs into LPGs, and vice versa. These
algorithms demonstrate that transforming graphs between the two formats is possible without losing
information. This allows us to use both graph models interchangeably.

2.3.1 Mapping of RDF and LPG

One important aspect of our proposed transformation is that the number of vertices and edges changes
when transforming RDF graphs to LPGs, or vice versa. We use LPGs that have vertices labeled with
types and edges labeled with predicates. In RDF, each type is represented as vertex \( t \in V_C \) and all
predicates \( p \in P \) can also be vertices (see Section 2.1). Therefore, the set of vertices \( V \) in an LPG is
only a subset of all vertices in an RDF graph. Thus, we can state that \( V \subseteq V_I \cup V_B \). As described in
Section 2.1, the set of vertices in an RDF graph is the union of IRIs \( V_I \), blank nodes \( V_B \), and literals
\( L \). Note that \( V \neq V_I \setminus (P \cup V_C) \cup V_B \) since RDF allows predicates \( P \) and types \( V_C \) as subjects and
objects of RDF triples. In these cases, the predicate or type is treated as a vertex.\(^1\)

\(^1\) Exception: although \( o \) is object of \( (s, rdftype, o) \), this statement does not require a vertex \( o \) in the LPG.
Regarding vertex label, we can say that all vertex labels are RDF types and that all RDF types are vertex labels. Thus, we have $\Sigma_V = V_C$. Note that inferencing does not contradict this statement.

As visualized in Figure 2.2, we store RDF literals as key–value properties in an LPG. More precisely, RDF literals are the possible values for LPG properties, i.e., $\Sigma_{\text{Value}} = L$.

All RDF predicates except $\text{rdf}: \text{type}$ are used as edge labels or as keys for the key–value properties in the LPG. Thus, we can state that $\Sigma_E \cup \Sigma_{\text{Key}} = P \setminus \{\text{rdf}: \text{type}\}$. In the following paragraphs, we discuss this transformation in more detail.

### 2.3.2 Transforming RDF graphs to LPGs

In Algorithm 1, we present a simple algorithm to transform RDF graphs into LPGs. The algorithm includes steps to handle named RDF graphs and GDBs.

#### Algorithm 1: Parse RDF to LPG

1. function `CONVERTRDFtOLPG(RDF)`
2. returns `GDB`
3. create `GDB` $\leftarrow (V = \emptyset, E = \emptyset, G = \emptyset, \ell_G)$;
4. forall $(s, p, o, d) \in \text{RDF}$ do
5.   get or create a graph $G$ from $G$ with $\ell_G(G) = d$;
6.   if $p = \text{rdf}: \text{type}$ then
7.     $V_G \leftarrow V_G \cup \{s\}$; /* add vertex $s$ to $V_G$ */
8.     $\ell_V(s) \leftarrow \ell_V(s) \cup \{o\}$; /* add $o$ to the set of labels $\ell_V(s)$ */
9.   else
10.      $V_G \leftarrow V_G \cup \{s, o\}$; /* add vertices $s$ and $o$ to $V_G$ */
11.      $E_G \leftarrow E_G \cup \{(s, o)\}$; /* add edge $(s, o)$ to $E_G$ */
12.      $\ell_E((s, o)) \leftarrow \ell_E((s, o)) \cup \{p\}$; /* add label $p$ to $\ell_E((s, o))$ */
13.      if $\text{IsLiteral}(o)$ then
14.        $\ell_P(o) \leftarrow \{(\text{literal}, o)\}$; /* add property $(\text{literal}, o)$ to $o$ */
15.   $V \leftarrow \bigcup_{G \in G} V_G$;
16.   $E \leftarrow \bigcup_{G \in G} E_G$;
17. return `GDB`;

In Algorithm 1 Line 4, we loop over all quads in the input RDF graph. For each data source IRI $d$, we create a named graph $G$ and add it to $G$. In case the predicate $p$ of the quad is the $\text{rdf}: \text{type}$ predicate, we add the label $o$ to the set of labels of $s$. We ensure the existence of vertex $s$ in Line 7.
In the case where the predicate \( p \) is not \( \text{rdf:type} \), we create a relationship denoted by a labeled edge between \( s \) and \( o \). In Line 11, we create the edge \((s, o)\), if not already created. In Line 12, we add the label \( p \) to the edge’s label set. We ensure the existence of vertex \( s \) and vertex \( o \) in Line 10. If \( o \) is an RDF literal, we set the key-value property \((\text{literal}, o)\) of \( o \) (Line 14).

### 2.3.3 Transforming LPGs to RDF graphs

Intuitively, transforming an LPG into an RDF graph is the inverse of Algorithm 1. In the pseudocode presented in Algorithm 2, we assume that all vertices, labels, and property keys in the LPG are valid IRIs. For any other LPG, IRIs need to be generated accordingly, e.g., by using a custom domain prefix and a suitable hash function applied on the identifiers of vertices, labels, and property keys.

**Algorithm 2: Parse LPG to RDF**

```plaintext
function ConvertLPGToRDF(GDB = (V, E, G)) returns RDF
RDF ← ∅;
forall G ∈ G do
  forall v ∈ V_G do
   forall l ∈ ℓ_P(v) do
      RDF ← RDF ∪ \{(v, \text{rdf:type}, l, ℓ_G(G))\}; /* ensure literal is not used in combination with other keys */
      forall (key, value) ∈ ℓ_P(v) with key ≠ literal do
        RDF ← RDF ∪ \{(v, key, value, ℓ_G(G))\};
   forall (v1, v2) ∈ E_G do /* if there are key-value properties on an edge, introduce an */
      if ℓ_E((v1, v2)) ≠ ∅ then
        forall l ∈ ℓ_E((v1, v2)) do
          x ← new vertex IRI;
          RDF ← RDF ∪ \{(v1, l, x, ℓ_G(G))\};
          RDF ← RDF ∪ \{(x, l, v2, ℓ_G(G))\};
          forall (key, value) ∈ ℓ_P((v1, v2)) do
            RDF ← RDF ∪ \{(x, key, value, ℓ_G(G))\};
        else
          forall l ∈ ℓ_E((v1, v2)) do
            RDF ← RDF ∪ \{(v1, l, v2, ℓ_G(G))\};
      return RDF;
```

21
In Line 4, we iterate over all Labeled Property Graphs \( G \) contained in the multiset \( G \) and add quads to the originally empty RDF graph. All added quads use the name of \( G \) as data source IRI \( d \). For each \( G \), we iterate over all vertices in Line 5 and over all edges in Line 10. Thus, we traverse all vertices \( V \) and edges \( E \) (compare LPG definition in Section 2.2).

For each vertex label (Line 6), we add a quad using \( \text{rdf:type} \) as predicate to denote type information in Line 7. Furthermore, for each key-value property of each vertex \( v \), we add quads with the \textit{key} as predicate and the \textit{value} as object (see Line 9). Note that we assume \textit{key} to be a valid IRI. The \textit{value} may be either an IRI or an RDF literal.

Similarly, we handle all edge labels. However, one main difference between RDF and LPG is the lack of key-value properties in RDF. In our algorithm, key-value properties of vertices can be transformed to edges with literals as objects (Line 9). However, the transformation of key-value properties of edges is not straightforward as RDF has no concept to describe such relationships. To this end, we use a workaround by adding an intermediate vertex \( x \) (following \([51]\)). In the case that an edge has key-value properties, we create two quads with the edge label as predicate in Lines 14 and 15. Instead of directly connecting the vertices \( v_1 \) and \( v_2 \), we connect them over the intermediate vertex \( x \). This allows us to attach all key-value properties of the LPG edge to the intermediate vertex \( x \) (Line 17). We can guarantee that transforming an LPG to an RDF graph and back again results in the same LPG, if we implement an escaping mechanism to correctly transform all artificially added intermediate vertices \( x \). To reduce the number of newly introduced vertices, we use this work-around only when edges have key-value properties. In the absence of such properties, we connect \( v_1 \) and \( v_2 \) directly in Line 20. In the case that we use literal key-value properties, the vertex \( v_2 \) in Lines 15 and 20 will be an RDF literal. Thus, we successfully transform all information included in the LPG to an RDF graph.

### 2.4 Discussion

Both graph models, Labeled Property Graphs (LPGs) and RDF, can be used to model the same information as graphs. On a theoretical level, it is a matter of preference which formalization is used to describe graph data. On a practical level, it is a matter of representation and compression. RDF data can be easily serialized since all information is encoded as triples or quads (directed, edge-labeled multigraph). The ability to serialize and, thus, easily distribute and share RDF graphs has been a key feature
and motivation for the design of the data graph model for the Semantic Web. However, querying the data is sometimes “hideous” since multiple edges and vertices need to be queried in expensive union operations in order to collect relevant information [73]. Furthermore, information is often attached using blank nodes, which is another hop for the graph query.

In contrast, depending on the actual implementation, LPGs allow a more compressed representation of graph data. Properties with literal values can be stored as key–value pairs inside a vertex and even inside an edge. A work-around for properties of edges can be quite bulky (see already the simple example in Section 2.3). Storing RDF types as labels of vertices is a design decision we made to reduce the overall number of vertices and ease querying. For some use cases, it may be desired to keep the RDF-like representation of types as separate vertices. However, it is not possible to store vertex labels in an RDF graph since only edge labels are supported. Thus, in general, LPGs offer more flexibility for data architects in how they model their data. This is a possible explanation for why LPGs are supported by modern graph databases and are popular in industry and industry related-research [87]. For example, Apache Spark GraphX [5] and Pregel [62] operate on LPGs rather than on RDF graphs. In contrast, RDF is a widely used and easy to serialize standard, thus, enabling easy exchange of graph data, especially the triple/quad syntax. This observation is not surprising since RDF was developed for this purpose.

In conclusion, it seems reasonable to use databases based on the LPG model and share the graph data with the RDF model. Of course, various other factors can be taken into account including personal preference. Often, native RDF databases (so-called triple stores) have advantages when solely working with RDF, e.g., optimized SPARQL query execution [72] and native inferencing support. However, since both formats can be easily transformed into each other, as shown in Section 2.3, research from both worlds can and should be taken into account when working with graph data.

2.5 Summary

We define structural graph summaries as equivalence relations over data graphs. As data graphs, we work with Labeled Property Graphs (LPGs) and Resource Descriptions Framework (RDF) graphs. We presented simple algorithms to transform LPGs into RDF graphs, and vice versa. Thus, without loss of generality, LPGs and RDF can be used interchangeably.
What is the meaning of the "but" in "We extend the idea of the schema graph from ABSTAT but include all RDFS properties"? The right syntax would be "We extend the idea of the schema graph from ABSTAT with ..."
3.1 Structural Graph Summary Models

In the following, we analyze existing (semantic) structural graph summaries with respect to the captured schema structure, i.e., what features of the input graph are used to summarize vertices. Existing surveys about graph summaries cover a wider range of approaches and, thus, lack this level of granularity [19, 23, 56, 61]. Table 3.1 shows a cross-table of each analyzed structural graph summary model and its features. In total, we analyzed 19 graph summary models and identified 12 different features. We organize the structural graph summary models and the features into different groups. We distinguish features that only use triple information (triple features), features that define how features of multiple vertices are combined (subgraph features), and features that define explicit semantic rules such as joining and inference (semantic rule features). Each group of features adds another level of complexity, i.e., intuitively, the computational complexity grows when features of different groups are used. There is no single graph summary model that supports all features. However, we see common combinations of features. In the following, we discuss the graph summary models along the identified features shown in Table 3.1 from left to right.

3.1.1 Triple Features

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

Property sets. The most commonly used feature in structural graph summaries is using properties to compute the schema of vertices. More specifically, for each vertex \( s \) in the data graph the property set \( \ell_P(s) \) is compared. Campinas et al. [21] proposed so-called “Attribute-based Collections” (we sometimes refer to it simply as Attribute Collection), a graph summary that relies solely on property sets to compute the schema structure of vertices. If two vertices \( s, s' \) share the same property set, i.e., \( \ell_P(s) = \ell_P(s') \), they are considered equivalent, thus, are summarized together.
Table 3.1: Structural graph summary models found in the literature and what features they use (X) and do not use (-) to capture the schema structure of vertices. The features are grouped by features that use triple information only (triple features), features that define how features of multiple vertices are combined (subgraph features), and features that define explicit semantic rules (semantic rule features).

<table>
<thead>
<tr>
<th>Graph summary</th>
<th>Feature</th>
<th>Triple features</th>
<th>Subgraph features</th>
<th>Semantic rule features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Property sets</td>
<td>Type sets</td>
<td>Label sets</td>
</tr>
<tr>
<td>Simple</td>
<td>Attribute Collection [21]</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Class Collection [21]</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Characteristic Sets [68]</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>SemSets [25]</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Complex</td>
<td>SchemEX [58]</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ABSTAT [84]</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>LODex [10]</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Loupe [64]</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>TermPicker [80]</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Weak Summary [40]</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Strong Summary [40]</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Typed Weak Summary [40]</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Typed Strong Summary [40]</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Tran et al. [88]</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>A(κ)-index [54]</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>T-index [65]</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Consens et al. [27]</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Schätzle et al. [81]</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
**Type sets.** Another commonly used feature is using the vertices’ types to compute the schema. Here, for each vertex $s$ in the data graph, the type set $\ell_T(s)$ is compared. If two vertices $s, s'$ share the same type set, i.e., $\ell_T(s) = \ell_T(s')$, they are considered equivalent. Campinas et al. [21] proposed a second graph summary, called “Class-based Collections” (we sometimes refer to it simply as Class Collection), which uses only the vertices’ type sets to compute the schema. Both graph summaries, Attribute-based Collections and Class-based Collections, were developed to enhance SPARQL query formulations by providing recommendations [21].

**Label sets.** Tran et al. [88] proposed the feature of label parameterization for graph summaries. With the label parameterization, only a subset of all edge labels are used to compute the schema. More precisely, one defines a set of predicates $P$, the so-called label set, which are ignored when determining the equivalence of vertices. Tran et al.’s graph summary combines property sets $\ell_P(s)$ with label sets. Furthermore, they combine this with k-bisimulation (see below).

**Neighbor vertex ID.** The final triple feature is using the identity of outgoing neighbors $\Gamma^+(s)$ to determine the equivalence of vertices. It appears that no existing graph summary summarizes vertices solely by comparing the neighbor identities. However, SemSets [25] summarize vertices that share the same outgoing predicates, which are linked to the same vertices. To check if two vertices $s, s'$ are equivalent under SemSets, all triples where $s$ or $s'$ are the subject vertices are compared. For each triple $(s, p, o) \in G$ there has to be a triple $(s', p, o) \in G$, and vice versa. Thus, they combine neighbor vertex identifiers $\Gamma^+(s)$ with predicate paths (see below). SemSets were developed to discover semantically similar sets of vertices in graphs and to use these vertex sets to improve keyword-based ad-hoc retrieval.

**3.1.2 Subgraph Features**

The neighbor vertex identifier is the most direct approach to incorporate neighbor information that leads to a wider range of summary models that consider neighbor information, e.g., vertices in $\Gamma^+(s)$. We classify features as subgraph features when they combine triple features of multiple vertices.
Neighbor triple. SchemEX [58], SchemEX+U+I [15], ABSTAT [84], LODeX [10], and Loupe [64] summarize vertices \(s\) and \(s'\) based on a common type set and common properties linking to vertices with the same type sets.

This means, in order to compute the schema of one vertex \(s\), also the type sets of outgoing neighbors \(\Gamma^+(s)\) are required to be equivalent, i. e., we compare neighbor triples. In contrast to SemSets [25], these approaches use not the neighbor vertex identifiers \(\Gamma^+(s)\) but the type set \(\ell_T(o)\) for each \(o \in \Gamma^+(s)\). SchemEX [58], SchemEX+U+I [15], ABSTAT [84], LODeX [10], and Loupe [64] combine type sets \(\ell_T(s)\), property sets \(\ell_P(s)\), and neighbor type sets \(\ell_T(o)\) using predicate paths (see below). The summary models were developed for data source search and exploration.

Predicate path. As indicated in the previous two features, almost all analyzed graph summaries that use neighbor information combine the schema structures using predicate paths, i. e., they compare which predicates link to which neighbors. A predicate path compares via which path of predicates a vertex is connected to neighboring vertices’ schema structures. For example, SchemEX [58], SchemEX+U+I [15], ABSTAT [84], LODeX [10], and Loupe [64] consider which property links to which type set. TermPicker [80] follows a different strategy to integrate the schema of neighboring vertices. TermPicker summarizes vertices \(s\) based on having the same type set \(\ell_T(s)\), the same property set \(\ell_P(s)\), and the same “aggregated” type set \(\bigcup \ell_T(o)\) of all \(o \in \Gamma^+(s)\). Consequently, TermPicker’s graph summaries compress all type sets of all neighbors into a single type set. Thus, TermPicker’s graph summaries do not contain information about which specific property linked to which neighbor.

\((k-)\)bisimulation. Many graph summaries compute the schema of vertices by taking into account the schema of neighbors over multiple hops [54, 58, 76, 88]. This is commonly defined as a bisimulation. Bisimulation operates on state transition systems and defines an equivalence relation over states [79]. Two states are equivalent (or bisimilar) if they change into equivalent states with the same type of transition. Interpreting a labeled graph as a representation of a state transition system allows us to apply bisimulation on graph data to discover structurally equivalent parts.

In practice, graph summaries usually define a stratified \(k\)-bisimulation [54, 58, 76, 88]. A stratified bisimulation restricts the maximum path length to \(k\) edges in the connected subgraph. This increases the chance that two vertices are considered equivalent. An efficient \(k\)-bisimulation algorithm
was proposed by Kaushik et al. [54] but there are also efficient implementations specialized to their corresponding graph summary, e. g., by Konrath et al. [58] and Goasdoué et al. [40]. Tran et al. [88] propose, in addition to the label parameterization feature described above, also the height parameterization feature, formulating \( k \)-bisimulation as feature for graph summaries.

Some graph summaries combine the feature of using only incoming or only outgoing properties with the \( k \)-bisimulation feature [27, 65, 81]. This is referred to as backward \( k \)-bisimulation and forward \( k \)-bisimulation, respectively [40]. Milo and Suciu [65] developed the so-called T-index to support path queries in semi-structured databases. This summarizes vertices \( s \) based on the common set of incoming property-paths, i. e., they use \( k \)-bisimulation only on incoming property sets \( \ell^{-}_p(s) \). Consens et al. [27] propose a structural graph summary model to support navigational SPARQL queries, so-called Extended Property Paths (EPPs). They summarize vertices \( s \) based on the common set of outgoing property-paths, i. e., they use \( k \)-bisimulation only on outgoing property sets \( \ell_p(s) \). In addition, for each hop, the type sets \( \ell_T(s) \) have to be equivalent. Schätzle et al. [81] developed a similar approach like Consens et al. [27]. The difference is that Schätzle et al. do not consider type sets, but object equivalences (neighbor vertex IDs). Furthermore, they do not distinguish between types and properties. Thus, they compute the bisimulation over all predicates \( p \in P \). This means, for each hop the same vertex is connected over the same predicate. For \( k = 1 \), Schätzle et al.’s summary model is equivalent to the SemSets [25] model.

**Incoming property sets.** Incoming property sets are frequently used in combination with \( k \)-bisimulation. It appears no graph summary summarizes solely based on incoming property sets \( \ell^{-}_p(s) \). But, Characteristic Sets [68] summarize two vertices \( s, s' \) that have the same outgoing property sets \( \ell_p(s) = \ell_p(s') \) and the same incoming property sets \( \ell^{-}_p(s) = \ell^{-}_p(s') \). Characteristic Sets were designed for cardinality estimations of queries in RDF databases. Analogously, Goasdoué et al. [40] define the Strong Summary. The Strong Summary summarizes vertices \( s \) and \( s' \) if they have the same property sets \( \ell_p(s) = \ell_p(s') \) and the same incoming property set \( \ell^{-}_p(s) = \ell^{-}_p(s') \). Note, Characteristic Sets and the Strong Summary are equivalent. Furthermore, they propose the Typed Strong Summary, which summarizes vertices \( s \) and \( s' \) based on two conditions: (1) if they have empty type sets \( \ell_T(s) = \ell_T(s') = \emptyset \) and they have the same property sets \( \ell_p(s) = \ell_p(s') \) and the same incoming property set \( \ell^{-}_p(s) = \ell^{-}_p(s') \) or (2) if they have the same non-empty type sets.
Both the Strong Summary and the Typed Strong Summary also allow $k$-bisimulation and semantic rule features such as taking so-called “related properties” into account and exploit RDF Schema inferencing (see below).

3.1.3 Semantic Rule Features in Graph Summaries

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

**OR combination.** Goasdoué et al. [40] define the Weak Summary using an “or-like” combination. In the Weak Summary, two vertices $s$ and $s'$ are equivalent if they have the same property set $\ell_p(s) = \ell_p(s')$ or the same incoming property set $\ell^-_p(s) = \ell^-_p(s')$ (or both). Analogously to the Typed Strong Summary, they define the Typed Weak Summary, which relaxes the first condition of the equivalence of the property set and the incoming property set using the “OR” combination of the Weak Summary [40].

**Related properties.** Goasdoué et al. [40] also propose to include property relations. Two properties $p$ and $p'$ are source-related if they co-occur in any property set $\ell_p(s)$ of any vertex $s$ and they are target-related if they co-occur in any incoming property set $\ell^-_p(s)$ of any vertex $s$. They developed this feature to generate comprehensible graph visualizations.

**RDF Schema.** Several semantic structural graph summaries use RDF Schema inferencing to enhance their summaries. ABSTAT [84] exploits RDF Schema type hierarchies to compute so-called minimal patterns. They select the minimal number of types, i.e., they only keep the most specific types from the RDF Schema type hierarchy. Goasdoué et al. [40] exploit RDF Schema type hierarchies, property hierarchies, and RDF Schema domain and RDF Schema range. With domain and range, types for the subject vertex and the object vertex can be inferred. They also propose a so-called shortcut for inferencing, which improved the time needed to perform the inferencing in graph summaries by up to 94% [40].
OWL SameAs. SchemEX+U+I [15] also uses the full RDF Schema inferencing but also exploits the semantics of the owl:sameAs property. This property is part of W3C’s Web Ontology Language (OWL) [63], which is heavily used in the context of RDF graphs. The owl:sameAs property defines an equivalence relation [63, Section 4.2], intended to identify vertices that represent the same real-world entity. To compute the schema structure of one vertex \( v \), the schema structures of all vertices \( v' \) in the weakly connected components in an owl:sameAs-labeled subgraph of \( G \) are merged (see Ding et al. [30] for details on owl:sameAs networks). SchemEX+U+I was developed for a data search task.

3.2 Incremental Graph Summarization Algorithms

Structural graph summaries partition a graph based on features of vertices or edges. In this section we analyze existing algorithms to compute such structural graph summaries. We also discuss selected incremental algorithms of the related research fields of subgraph indices, graph compression, and schema discovery.

Incremental Structural Graph Summarization. Existing structural graph summarization algorithms are often designed and/or evaluated using static graphs only [10, 25, 61, 68, 80, 84]. Only few structural graph summaries are designed for evolving graphs and they have limitations [39, 58]. So far, to the best of our knowledge, there is no suitable solution to update only these parts of the graph summary that need to be updated, without having a change log and without losing the high compression of structural graph summaries by storing local copies of the data graph.

Konrath et al. [58] compute their graph summary over a stream of vertex-edge-vertex triples, i.e., they can deal with the addition of new vertices and edges to the graph. However, they cannot deal with the deletion of vertices or edges or the modification of their labels. Similarly, Goasdoué et al. [39] support to iteratively compute their structural graph summaries, which means they also handle only additions. Thus, both approaches are not suitable to update structural summaries of evolving graphs.
Incremental Subgraph Indices. 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 [45]. We focus on incremental subgraph indices as they most closely relate to structural graph summaries. Yuan et al. [94] 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 [95] and by Kansal and Spezzano in 2017 [53]. 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.

Qiao et al. [75] propose an approach to compute an index of isomorphic subgraphs in an unlabeled and undirected graph $G$. The goal is to find the set of subgraphs in $G$ that are isomorphic to a given query pattern. The result is a compression of the original graph that is suitable to answer, e.g., cardinality queries regarding subgraphs. Their work does yet not support changes in the graph.

The algorithm of Fan et al. [36] 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.

Min et al. [66] propose an algorithm for continuous subgraph matching using an auxiliary data structure, which stores the intermediate results between a query graph and a dynamic data graph. As data graphs, they consider undirected graphs where only vertices are labeled. Dynamic graphs are updated through a sequence of edge insertions and edge deletions.

The TipTap [67] algorithm computes approximations of the frequent $k$-vertex subgraphs w.r.t. a given threshold in evolving graphs. TipTap’s purpose is to count the number of occurrences of different subgraphs in large, evolving graphs. The evolution of graphs is modeled as a stream of updates on an existing graph.

Duong et al. [32] propose a streaming algorithm using approximate pattern matching to determine subgraph isomorphisms. They employ $k$-bisimulation to determine equivalent subgraphs and store them in an index. However, this index is computed offline for a static graph only and their algorithm considers a stream of graph queries as input. Thus, not the data graph is evolving but the set of queries.
Incremental Graph Compression. There are also works on graph compression, which use a technique called “corrections” \cite{57, 83} to provide a concise representation of a graph. A correction is a set of edges that are to be added or removed from the compressed graph to reconstruct the original graph \cite{83}. Incremental variants of corrections for graph compression also exist, where the changes are stored as sets of corrections \cite{57}.

Incremental Schema Discovery. Another area related to our work is incremental schema discovery from large datasets in NoSQL databases. Wang et al. \cite{90} propose an approach for an incremental discovery of attribute-based schemas from JSON documents. The schema is stored in a tree-like data structure, following the nesting of JSON objects. The attributes are considered per document only, i.e., there are no cross-document connections of attributes. The algorithm incrementally adds information to the schema, when more and more documents are processed. Baazizi et al. \cite{7} also compute schema from JSON objects with focus on optional and mandatory fields.

In addition to document oriented formats like JSON, schema discovery is also used for graph data. For example, XStruct \cite{49} 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 \cite{55} 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.

Incremental Graph Mining. Tesseract \cite{11} is a distributed framework for executing general graph mining algorithms on evolving graphs implemented using Apache Spark Structured Streaming. They follow a vertex centric approach to distribute updates to different workers following the assumption that, in general, changes effect only local graphs, thus, only few duplicate updates need to be detected. Tesseract supports $k$-clique enumeration, graph keyword search, motif counting, and frequent subgraph mining.
3.3 Summary

There exists a large variety of structural graph summary models that use different combinations of features. However, none of the existing approaches covers all features. In the past, the idea of a single, generalizable framework based on equivalence relations to compute structural graph summaries has been discussed a few times [22, 76, 88]. In particular, the concept of bisimulation has been used to define a single, adaptive framework for graph summaries based on equivalence relations [76, 88]. However, none of the existing approaches define a language to define these equivalence relations, which also allows a single, parameterized algorithm to compute them. The proposed features and the corresponding algorithms only implement a subset of features, as shown in Table 3.1. Thus, a language for semantic structured graph summarization is needed that incorporates all of the different features and that allows us to flexibly define (semantic) structural graph summaries.

Furthermore, existing graph summarization 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.
The authors manage to convey a quite complex model, through formal descriptions and examples, in a way that makes the topic easy to follow.

R1, GvDB, 2018

4

The Common Model FLUID

In this chapter, we formally define FLUID. We proposed FLUID initially in the peer-reviewed workshop paper [13]. This chapter is based on the further developed peer-reviewed journal article, which includes all formal definitions presented in this chapter [17].

First, we define the fundamental concepts of structural graph summarization based on equivalence relations in Section 4.1. Subsequently, define the building blocks of FLUID as equivalence relations, i.e., the schema elements and their parameterizations. An overview of the elements and parameterizations of FLUID is given in Table 4.1.

In Section 4.2, we define three simple schema elements. Based on simple schema elements, we define complex schema elements. Second, we define in Section 4.3 six parameterizations to further specialize the schema elements. Third, we define three payload elements in Section 4.4. We summarize FLUID in Section 4.5, by showing how all graph summary models analyzed in Section 3.1 are defined with FLUID. Finally, we define a generic data structure based on the LPG model to store all structural graph summaries defined with FLUID.
4.1 Structural Graph Summarization

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

**Definition 1.** [17] An equivalence relation on a set \( X \) (e.g., a set of graph vertices) is a subset \( \text{EQR} \subseteq X \times X \) that is reflexive, symmetric, and transitive. When \( (x, y) \in \text{EQR} \), we say that \( x \) is equivalent to \( y \) and write \( x \sim y \). For any \( y \in X \), the set \( \{x \in X \mid x \sim y \} \) is called the equivalence class of \( y \), typically denoted by \( [y]_{\text{EQR}} \). We denote by \( \top \) the tautology equivalence relation \( X \times X \), in which all elements of \( X \) are equivalent. We denote by \( \text{id} \) the identity equivalence relation \( \{(x, x) \mid x \in X\} \), in which no two distinct elements of \( X \) are equivalent.

**Remark 1.** For a given equivalence relation, any two equivalence classes either are disjoint or equal, so equivalence relations on a set \( X \) correspond precisely to partitions (decompositions) of \( X \) [35]. Furthermore, the intersection of two equivalence relations over \( X \) is also an equivalence relation.

Note that the intersection of equivalence relations results in an equivalence relation, but the union of equivalence relations does not. To demonstrate this, let us consider the equivalence relations \( \sim_A = \{(1, 1), (2, 2), (1, 2), (2, 1), (3, 3)\} \) and \( \sim_B = \{(1, 1), (2, 2), (3, 3), (2, 3), (3, 2)\} \). The union of \( \sim_A \) and \( \sim_B \) is not transitive, as it contains \((1, 2)\) and \((2, 3)\) but not \((1, 3)\). Thus, to combine equivalence relations in an “or-like” fashion to give a new equivalence relation, we define an extended union operator \( \cup_{\text{ex}} \).
Definition 2. [17] Let \( \sim_A \) and \( \sim_B \) be equivalence relations over the same set. We define the extended union \( \sim_A \cup \text{ex} \sim_B \) to be the transitive closure of \( \sim_A \cup \sim_B \). That is, \( \sim_A \cup \text{ex} \sim_B \) is the unique equivalence relation \( \sim_{\text{ex}} \) such that \( \sim_A \cup \sim_B \subseteq \sim_{\text{ex}} \) and for every equivalence relation \( \sim' \) such that \( \sim_A \cup \sim_B \subseteq \sim' \), we have \( \sim_{\text{ex}} \subseteq \sim' \).

For FLUID, we define equivalence relations over the set of vertices appearing in the graph. For LPGs, this problem is straightforward since we use distinct sets of vertices \( V \) and edges \( E \). However, RDF graphs are defined as sets of triples \( (s, p, o) \). Fortunately, since each triple has exactly one subject vertex \( s \), an equivalence relation \( \text{EQR}_v \) over vertices induces an equivalence relation \( \text{EQR}_t \) over the triples, given by \( ((s, p, o), (s', p', o')) \in \text{EQR}_t \) iff \( (s, s') \in \text{EQR}_v \). Any partitioning of \( G \)'s vertices into disjoint subsets can be expressed as an equivalence relation over vertices in the data graph \( G \). Thus, we can describe graph summarization using an equivalence relation \( \text{EQR} \). When two vertices are placed into the same partition, we say they are summarized. The graph summary for \( G \) with respect to \( \text{EQR} \) is a labeled graph \( SG \).

Each equivalence class \( [y]_{\text{EQR}} \) is represented in \( SG \) in a so-called vertex summary. The vertex summary \( \text{vs}_{y} \) is a subgraph in \( SG \) that is equivalent to the subgraphs of all summarized vertices \( v \in [y]_{\text{EQR}} \) in \( G \) under \( \text{EQR} \). For vertex summaries, we distinguish primary vertices, which are equivalence classes of \( \text{EQR} \), and secondary vertices, which are equivalence classes of the relations from which \( \text{EQR} \) is defined.* Furthermore, we can attach payload [42] to each primary vertex in a vertex summary \( \text{vs} \subseteq SG \). The payload can be tailored for different purposes, e.g., to contain the number of summarized vertices. For FLUID, we define a set of payload elements \( \text{PAY} \) to implement a specific task. Payload elements map vertex summaries to payload. Formally, we can now denote a (semantic) structural graph summary model as a 3-tuple of a data graph \( G \), an equivalence relation \( \text{EQR} \), and a set of payload elements \( \text{PAY} \).

Definition 3. [17] A structural graph summary model is a tuple \( (G, \text{EQR}, \text{PAY}) \), where \( G \) is the data graph, \( \text{EQR} \) is an equivalence relation over vertices in \( G \), and \( \text{PAY} \) is a set of payload elements.

*Please see Section 4.6 for the formal definition of the FLUID data structure, i.e., the vertex summaries build from primary vertices \( r_i \) and secondary vertices \( s_j \).
Table 4.1: Overview of FLUID’s schema elements and parameterizations.

<table>
<thead>
<tr>
<th>Schema Element (SE)</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Schema Element (SSE)</td>
<td>Triple-based summarization of vertices</td>
<td>Definitions 4 to 6, Figures 4.1b to 4.1d</td>
</tr>
<tr>
<td>Complex Schema Element (CSE)</td>
<td>Summarization of vertices using combinations of Schema Elements (SEs)</td>
<td>Definition 7, Figure 4.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameterizations</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label parameterization $lp(SSE, P_l)$ (short: $SSE_{P_l}$)</td>
<td>SSE ignores existence of predicates not in $P_l$</td>
<td>Definition 8</td>
</tr>
<tr>
<td>Set parameterization $sp(SSE, S)$ (short: $SSE</td>
<td>S$)</td>
<td>Application of SSE is restricted to vertices containing only specified predicates and/or objects</td>
</tr>
<tr>
<td>Chaining parameterization $cp(CSE, k)$ (short: $CSE^k$)</td>
<td>Recursively apply CSE for each connected vertex up to $k$ hops</td>
<td>Definition 10</td>
</tr>
<tr>
<td>Direction parameterization $dp(SE, \delta)$ (short: $\delta$-SE)</td>
<td>Summarize vertices following the direction parameter $\delta = {o, i, b}$ based on outgoing predicates (o), incoming predicates (i), or outgoing and incoming predicates (b)</td>
<td>Definition 11</td>
</tr>
<tr>
<td>Inference parameterization $op(G, VG)$ (short: $G_{VG}$)</td>
<td>Include ontology reasoning by inferring additional triples from a vocabulary graph $VG$, e. g., using $VG_{RDFS}$</td>
<td>Definition 13</td>
</tr>
<tr>
<td>Instance parameterization $ip(SE, \Delta)$ (short: $SE[\Delta]$)</td>
<td>Unions of explicitly equivalent vertices</td>
<td>Definition 14</td>
</tr>
</tbody>
</table>

4.2 Schema Elements

We distinguish simple schema elements and complex schema elements. Simple schema elements enable triple-based vertex summarization (compare triple features in Table 3.1). Intuitively, this means they summarize vertices without considering any kind of neighbor information, i. e., they capture the “local” schema of vertices. The term local refers to triples of subject vertex, edge label (predicate), and object vertex, i. e., $(s, p, o)$ from the RDF graph (see Section 2.1). Local in this sense means that equivalence of two vertices $s, s'$ depends only on the edges where $s$ or $s'$ is the source vertex, i. e., the subject of the triples. In contrast, complex schema elements allow us to capture the schema beyond the scope of a single vertex (compare subgraph features in Table 3.1). In the following, we formally define three
simple schema elements as well as the complex schema element using the RDF graph model. Recall from Section 2.3 that RDF graphs and LPGs can be transformed into each other. In Section 4.5, we show how FLUID’s schema elements are applied on LPGs.

4.2.1 Simple Schema Elements

Simple schema elements summarize vertices $s, s'$ by comparing all triples $(s, p, o) \in G$ and all triples $(s', p', o') \in G$. We define three simple schema elements: predicate-object cluster (POC), predicate cluster (PC), and object cluster (OC). Each simple schema element compares vertices following a different strategy, i.e., compare only the predicates, only the objects, or both (see analysis of triple features in Section 3.1.1). Figure 4.1 gives an example RDF graph and the corresponding vertex summaries using simple schema elements.

**Definition 4.** [17] The **Predicate–Object Cluster** (POC) partitions the data graph by comparing the triples based on common predicates linking to common objects. For two vertices $s, s'$, the equivalence relation POC holds true, iff for each triple $(s, p, o) \in G$ there is a triple $(s', p, o) \in G$, and vice versa.

Figure 4.1b shows two vertex summaries that summarize $v_1$ and $v_2$, respectively, following the predicate-object cluster POC equivalence.

**Definition 5.** [17] The **Predicate Cluster** (PC) partitions the data graph by common predicates of vertices: $(s, s') \in PC$ iff, for each triple $(s, p, o) \in G$, there is a triple $(s', p, o') \in G$ for some $o'$, and vice versa.

Note that the predicate cluster is not equivalent to saying $\ell_p(s) = \ell_p(s')$ due to the special treatment of the RDF predicate $\text{rdf}:\text{type}$ (see Section 2.1). As shown in Figure 4.1c, the predicate cluster includes the $\text{rdf}:\text{type}$ predicate. However, the $\text{rdf}:\text{type}$ predicate is explicitly excluded from property sets. Rather, $\text{ rdf}:\text{type}$ defines type sets (see again Section 2.1).

**Definition 6.** [17] The **Object Cluster** (OC) partitions the data graph by common objects of vertices: $(s, s') \in OC$ iff, for each triple $(s, p, o) \in G$, there is a triple $(s', p', o) \in G$ for some $p'$, and vice versa.
Note that POC ≠ PC ∩ OC. Two instances are equivalent under PC ∩ OC if they contain the same predicates and the same objects, whereas POC requires the same predicate–object pairs. For example, consider the novel *David Copperfield* by Charles Dickens and a hypothetical biography *Charles Dickens* by the illusionist David Copperfield. These two books are equivalent under PC ∩ OC because each has properties “author” and “title” and each has objects “David Copperfield” and “Charles Dickens”. However, they are not equivalent under POC because, e. g., one has “Charles Dickens” as author and the other does not. This acknowledges the predicate path feature in Table 3.1.

### 4.2.2 Complex Schema Elements

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

**Definition 7.** [17] A **Complex Schema Element** is a 3-tuple $CSE := (\sim_s, \sim_p, \sim_o)$, where $\sim_s, \sim_p,$ and $\sim_o$ are equivalence relations. Two vertices $s, s'$ are equivalent under this $CSE$ iff, for every triple $(s, p, o) \in G$, there is a triple $(s', p', o') \in G$ such that $s \sim_s s', p \sim_p p', o \sim_o o'$, and vice versa.

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

**Example 3.** Let $CSE-1 = (\top, id, id)$, where $\top$ denotes the tautological equivalence relation, in which all vertices are equivalent and id denotes the identity equivalence relation. $CSE-1$ considers vertices $s$ and $s'$ equivalent iff, for every triple $(s, p, o) \in G$, there is a triple $(s', p', o') \in G$ such that $p' = p$ and $o' = o$. In other words, it is identical to the simple schema element POC. Similarly to the previous
example, PC is identical to the CSE $(T, id, T)$ and OC is the CSE $(T, T, id)$. Formally, we may regard the simple schema elements as abbreviations for the corresponding CSEs; however, we implement them separately for efficiency.

Example 4. Let $CSE_2 = (T, id, PC)$. We can see that this is a relaxation of $CSE_1$ from Example 3, in that $CSE_1$ requires objects to be identical, whereas $CSE_2$ only requires objects to have the same set of predicates. In full, $CSE_2$ declares vertices $s$ and $s'$ to be equivalent iff for every triple $(s, p, o) \in G$, there is a triple $(s', p', o') \in G$ such that $p' = p$, and $o'$ and $o$, in turn, have the same predicates, and vice versa. That is, for every predicate $p$ that links $s$ to a neighbor $o$, $p$ also links $s'$ to a neighbor with the same set of predicates as $o$, and vice versa. This means that $CSE_2$ summarizes vertices using, in part, the schema of neighboring vertices $\Gamma^+(s)$. In Figure 4.2, we illustrate the vertex summaries for the example data graph visualized in Figure 4.1a according to the $CSE_2$ equivalence relation.

4.3 Parameterizations of Schema Elements

Simple and complex schema elements are the basic building blocks of FLUID. FLUID provides six parameterizations to specialize the schema elements’ behavior. For example, a parameterization can be applied that ignores a certain set of predicates or constructs predicate paths with a specific length. In the following sections, we define six parameterizations, which are again motivated by the analysis of existing graph summaries in Section 3.1. The first two parameterizations address the triple features identified in Section 3.1.1, the next two address the subgraph features from Section 3.1.2, and the final two address the semantic rule features from Section 3.1.3.

4.3.1 Label Parameterization

The label parameterization can be used to restrict schema elements to consider only specific predicates. This allows us to define, e.g., the type cluster $OC_{type}$, an object cluster that only compares objects connected over the predicate $rdf:type$. Hence, $OC_{type}$ compares vertices based on type sets.
Definition 8. [17] The label parameterization is a function $lp(SSE, P_r)$, which takes as input a simple schema element $SSE$ and a set of predicates $P_r \subseteq P$ and returns a schema element $SE_{P_r}$. The returned schema element $SE_{P_r}$ is defined analogously to $SE$ (Definitions 4, 5, and 6) but each existential and universal quantifier is restricted to tuples $(s, p, o)$ with $p \in P_r$. In detail, $lp(SE, P_r)$ defines vertices $s$ and $s'$ to be equivalent iff, for every triple $(s, p, o) \in G$ with $p \in P_r$, there is a triple $(s', p', o') \in G$ with $p' \in P_r$ such that:

- $SE = POC$ and $p = p'$ and $o = o'$; or
- $SE = PC$ and $p = p'$; or
- $SE = OC$ and $o = o'$.

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

4.3.2 Set Parameterization

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


**Definition 9.** [17] The set parameterization is a function \( sp(SSE, S) \), which takes as input a simple schema element \( SSE \) and a set of IRIs \( S \subseteq V_I \) and returns a simple schema element \( SSE|_S \). \( SSE|_S \) defines an equivalence relation \( EQR \) such that \((s, s') \in EQR\) if\( \)ff

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

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

**Example 5.** Let us apply the set parameterization to the type cluster \( OC_{\text{type}} \) using the empty set \( \emptyset \). The set-parameterized type cluster \( OC_{\text{type}}|_\emptyset \) splits the data graph into at most two equivalence classes. The first equivalence class (vertex summary) contains all vertices that have empty type sets (no type information provided in the data graph) and the second equivalence class (vertex summary) contains all vertices that have at least one type, if any such vertices exist.

**Example 6.** Let us apply the set parameterization to the property cluster \( PC_{\text{rel}} \) using the set \( S = \{p_1, p_2\} \). The set-parameterized property cluster \( PC_{\text{rel}}|_S \) compares vertices \( v, v' \) based on the same property set only if the property set is a subset of \( S \). Any vertex \( v \) where there is a \( p \in \ell_P(v) \) that is not in \( S \) will be summarized by the same vertex summary. Also, the corner case \( \ell_P(v) = \emptyset \) will be summarized by that same vertex summary. The remaining vertices are summarized based on the
equivalence of their property sets. Thus, there can be no more than four vertex summaries for any input graph, i.e., the ones corresponding to property sets \{p_1\}, \{p_2\}, and \{p_1, p_2\}, and the one that summarizes all other vertices.

The set parameterization can be used like a filter. When such a set \(S\) of predicates and/or objects is provided, the graph summary may contain a vertex summary that summarizes all vertices that have predicates and/or objects not in \(S\). More importantly, it contains vertex summaries with exactly these predicates and/or objects. Subsequently, one could either filter out these vertex summaries or all other vertex summaries. In Example 5, we may only want to visualize all vertices with a non-empty type set. In Example 6, we may only want to visualize all vertices that exclusively use properties \(p_1\) and/or \(p_2\).

### 4.3.3 Chaining Parameterization

Complex schema elements take the schema of two directly connected vertices into account. The chaining of \(k\) complex schema elements extends this to vertices within distance \(k\). As discussed in Section 3.1, this corresponds to a stratified \(k\)-bisimulation. The chained complex schema element is denoted by \(CSE^k\).

**Definition 10.** [17] The chaining parameterization is a function \(cp(CSE, k)\), which takes a complex schema element \(CSE := (\sim', \sim^p, \sim^o)\) and a chaining parameter \(k \in \mathbb{N}\) as input and returns an equivalence relation \(CSE^k\) that corresponds to recursively applying \(CSE\) to a distance of \(k\) hops. \(CSE^k\) is defined inductively as follows:

\[
CSE^0 = \sim' \\
CSE^1 = (\sim', \sim^p, \sim^o) \\
CSE^{k+1} = (\sim', \sim^p, CSE^k) \quad \text{for} \ k \geq 1.
\]

**Example 7.** Recall \(CSE-2 := (\top, \text{id}, \text{PC})\) from Example 4 and denote by \(\sim_2\) the equivalence relation it defines. Consider the chained complex schema element \(CSE-3 := cp(CSE-2, 2)\). By Definition 10, this corresponds to the CSE \((\top, \text{id}, \sim_2)\). Let \(CSE-3\) define the equivalence relation \(\sim_3\) and consider vertices \(s\) and \(s'\). We have \(s \sim_3 s'\) iff the following conditions are met.
• $s$ and $s'$ must have the same predicates, because predicate equivalence is defined by equality using the identity equivalence $\text{id}$.

• For every neighbor $o \in \Gamma^+(s)$ via any predicate, there is a neighbor $o' \in \Gamma^+(s')$ via the same predicate, such that the vertices $o$ and $o'$ are equivalent under CSE-2. That is, for each predicate $p$ that links $o$ to a neighbor with some predicates, the predicate $p$ also links $o'$ to a neighbor with the same predicates.

• Vice versa, for every neighbor $o'$ of $s'$, a similarly corresponding neighbor $o$ of $s$ exists.

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

### 4.3.4 Direction Parameterization

The direction parameterization $dp$ is applied on schema elements to use only outgoing predicates (parameter $\delta = o$), only incoming predicates ($\delta = i$), or both ($\delta = b$). Our schema elements OC, PC, POC, and CSE only take outgoing predicates into account. Schema structures like Characteristic Sets [68] use also incoming predicates. To address incoming predicates, a parameterized version of the three simple schema elements can be defined by using the incoming triples $(x, p, v) \in G$ with the summarized vertex $v$ in object position.

![Figure 4.3](image)

**Figure 4.3:** Following the b-PC equivalence relation, the two vertex summaries identified by the primary vertices $r_1$ and $r_2$ summarize the vertices $v_1$ and $v_2$ from Figure 4.1a, respectively.
Definition 11. [17] The direction parameterization is a function $dp(SE, \delta)$, which takes as input a schema element $SE$ and one direction parameter $\delta \in \{i, o, b\}$ and returns a schema element $\delta$-$SE$. Direction $i$ uses only incoming predicates, $o$ uses only outgoing predicates, and $b$ uses both, incoming and outgoing predicates. The returned schema element $\delta$-$SE$ is a modification of $SE$ in which all assertions made by $SE$ about the triples are applied on the incoming edges only, the outgoing edges only, or on both. This means that $o$-$SE = SE$, $i$-$SE$ is $SE$ modified to use incoming predicates instead of outgoing and $b$-$SE = i$-$SE \cap o$-$SE$.

Example 8. The bidirectional predicate cluster $b$-PC partitions the data graph by comparing the triples based on common incoming and outgoing predicates. Thus, the equivalence relation $b$-PC holds true, iff for each triple $(s_1, p_1, o_1) \in G$ there exists a triple $(s_2, p_1, o_2) \in G$, and vice versa, and for each triple $(s_3, p_3, s_1) \in G$ there exists a triple $(s_4, p_3, s_2) \in G$, and vice versa. The bidirectional predicate cluster $b$-PC is visualized in Figure 4.3.

4.3.5 Inference Parameterization

Some graphs contain semantic information, e.g., ontologies described with the RDF Schema (RDFS) vocabulary which, like RDF, is standardized by the W3C [20]. The inference parameterization is our first parameterization to include the semantics of the data graph in the structural graph summary and also falls under our category of semantic rule features (see Section 3.1.3). The inference parameterization $op$ is applied on the graph $G$ and enables ontology reasoning using a vocabulary graph $VG$. Applying RDFS inferencing means that a vocabulary graph $VG_{RDFS}$ is constructed from the data graph $G$, which stores all hierarchical dependencies between types and properties denoted by RDFS properties found in the data graph $G$. To construct the RDFS vocabulary graph, we first extract all triples containing RDFS vocabulary terms, namely all properties $P_{RDFS} = \{rdfs:subClassOf, rdfs:subPropertyOf, rdfs:range, rdfs:domain\}$. Next, we add the corresponding hierarchical dependencies of $rdfs:subClassOf$ and $rdfs:subPropertyOf$ to the vocabulary graph with further cross connections regarding $rdfs:range$ and $rdfs:domain$. An example of such a vocabulary graph is illustrated in Figure 4.4.

Definition 12. [17] An RDFS vocabulary graph is an edge-labeled directed multigraph $VG_{RDFS} \subseteq (V_C \cup P) \times P_{RDFS} \times (V_C \cup P)$. The set of vertices is the union of types $V_C$ and predicates $P$ in $G$. The edges are labeled with predicates $p \in P_{RDFS}$.
With hierarchical dependencies of types and predicates represented using our vocabulary graph, additional triples can be inferred using our inference parameterization. For a given triple in the data graph $G$, we can look up the vertex representing the used predicates or type in $VG$. Starting from this vertex in $VG$, the information needed for inference is contained in a polytree with the vertex itself as root.† The entailment rules for $VG_{\text{RDFS}}$ are defined by the W3C [20]. A comprehensive overview of these rules is presented in [1].

**Definition 13.** [17] The inference parameterization is a function $op(G, VG)$ which takes any data graph $G$ and a vocabulary graph $VG$ as input and, based on the entailment rules defined in $VG$, returns a data graph $G_{VG}$, which also includes all inferred triples.

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

FLUID’s formalization does not pose any restrictions on when the actual inference happens. In general, we can distinguish inference inside the structural graph summary and outside the structural graph summary. Inference inside means that all inferrable information is added to the original graph and the graph summary is computed for the new data graph. However, the order of these two is interchangeable. Inferring on the data graph and then summarizing can be equivalent to summarizing the data graph and then inferring on the graph summary [38, 40, 60]. The latter one then may require multiple iterations over the graph summary.

†A polytree is an orientation of an undirected tree.
Inference outside means that we compute the graph summary for the original graph and keep the constructed vocabulary graph. When we query the graph summary to fulfill our task, we can infer only the additional information that affects the query result, e.g., by generating additional queries based on entailment rules defined in the vocabulary graph. The result sets for inferring inside and outside are equivalent. This decision affects the build-time and the size of the computed structural graph summary as well as time to compute query results on the structural graph summary. We discuss practical implications of this decision in the empirical analysis in Section 6.4.

4.3.6 Instance Parameterization

In RDF graphs, vertices and all of their outgoing triples are commonly referred to as RDF instances. Referencing this terminology, we define the instance parameterization $ip$. This allows us to define rules to join sets of outgoing triples of different vertices, e.g., vertices linked with $owl:sameAs$. The instance parameterization is the final parameterization regarding the rule features (see Section 3.1.3).

Definition 14. [17] The instance parameterization $ip$ is a function $ip(SE, \Delta)$, which extends any schema element $SE$ to additionally take the schema of all equivalent vertices into account, following the instance equivalence relation $\Delta$. The returned schema element $SE[\Delta]$ is an extension of $SE$, which restricts the triples to be in $[\sigma]_\Delta$. Thus, $ip$ merges vertices that are equivalent under $\Delta$ and then applies the schema element $SE$.

In the following, we give two definitions of such an instance equivalence relation parameter $\Delta$, namely SameAs instances $\sigma$ and related property instances $\rho$.

**SameAs instances $\sigma$.** In the context of the Semantic Web, the $owl:sameAs$ property is of particular interest since $(s, owl:sameAs, s')$ explicitly states the equivalence of the entities identified by the vertices $s$ and $s'$. To take this information into account, we use the notion of **SameAs instances** $[\sigma]_\sigma$, which are equivalence classes of vertices or, equivalently, sets of vertices. Two vertices $s$ and $s'$ are equivalent according to the equivalence relation $\sigma$, iff there is a property-path in the data graph $G$ labeled $owl:sameAs$ from $s$ to $s'$. Note that the property-path is independent of the direction of the $owl:sameAs$ relation. When summarizing vertices using the SameAs instance parameterization, we take the schema information from all equivalent vertices into account.
Figure 4.5: Sample data graph containing three vertices $v_1$, $v_2$, and $v_3$. The object cluster assigns $v_1$, $v_2$, and $v_3$ to different vertex summaries. The instance parameterized object cluster using SameAs instances $[v]$, summarizes $v_1$, $v_2$, and $v_3$ into a single vertex summary.

Example 9. See Figure 4.5. According to the object cluster definition, the vertices $v_1$, $v_2$, and $v_3$ are not equivalent since $v_1$ has the object “Book”, $v_2$ has the object “Proceedings”, and $v_3$ has the objects “Book” and “Proceedings”. Merging $v_1$ and $v_2$ to a SameAs instance $[v]$ leads to the equivalence of all three vertices $v_1$, $v_2$, and $v_3$.

Related property instances $\rho$. Merging vertices also allows modeling property cliques [40]. Property cliques transitively check the co-occurrence of properties and summarize all vertices that have at least one property in common. We can define an instance equivalence relation $\rho$ that leads to equivalent vertices if they share at least one property. We call the equivalence classes $[v]_\rho$ related property instances. We distinguish source related properties and target related properties [40]. Schema elements parameterized with related property instances that consider outgoing properties use source related properties. For schema elements using outgoing properties, two vertices $s$ and $s'$ are equivalent according to the equivalence relation $\rho$, iff there exists vertex $s'' \in [s]_\rho$ such that there are triples $(s'', p, o) \in G$ and $(s', p, o) \in G$ with $p \neq rdf\!\: type$, or, similarly for some $s''' \in [s']_\rho$.

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

Note that $\rho$ takes transitively co-occurring properties into account. Thus, $\rho$ may summarize two vertices that do not share any property in the data graph.
4.4 Payload Elements

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

When computing the graph summary \( SG \), information about the summarized vertices can be attached to vertex summaries by using the notion of payload \([42]\). The payload contains information about the actual data, e.g., number of vertices summarized or a reference to their data source. Our intention is to make payloads as flexible as possible so we do not make any restrictions on what can be attached as payload and we allow multiple different kinds of payload to be attached to a single vertex summary. To this end, we define \( PAY \) to be a set of payload elements. Payload elements map vertex summaries to a payload, i.e., information extracted from the summarized vertices. They are functions that define what information is attached. For example, one payload element may attach the data sources of summarized vertices to vertex summaries while another payload element attaches the number of summarized vertices to vertex summaries. An example is illustrated in Figure 4.6. The payload \( dsp_1 \) contains data source IRIs that were extracted from the vertices summarized by the vertex summary represented by \( r_1 \). The payload \( vcp_1 \) contains an integer value representing the number of vertices summarized by \( r_1 \).

4.4.1 Semantic Entity Retrieval

One purpose of structural graph summaries is to find semantically related vertices (see Section 1.1.1). To implement semantic entity retrieval, we need to build an index for vertices based on their schema structure. To this end, we need to be able to store the identifiers of summarized vertices as payload of vertex summaries.

Definition 15. [17] Consider a graph summary of some graph, generated from an equivalence relation \( EQR \). The **vertex identity payload** is the set of identifiers (IRIs) of the vertices summarized by each vertex summary \( vs \). The vertex identity payload element \( vip \) is a function that takes a vertex summary \( vs_v \) of a vertex \( v \) as input and returns the set \( [v]_{EQR} \) of summarized vertices. We define the vertex identity payload as \( vip(vs_v) := [v]_{EQR} \).
The vertex summary identified by the primary vertex $r_1$ summarizes vertices with the property set \{author, title\}. The data source payload $dsp_1$ contains information about the data sources and the vertex count payload element $vcp_1$ contains the number of summarized vertices.

### 4.4.2 Cardinality Computation

Another purpose of structural graph summaries is cardinality computation in databases. As described in Section 1.1.2, this allows among other applications to implement query size estimation and assessment of data completeness. To implement this, we only need the number of summarized vertices and not the vertex identifiers. Thus, we can define another payload element, which maps only the integer value denoting the number of summarized vertices.

**Definition 16.** [17] The **vertex count payload** provides the number of vertices summarized by the vertex summary. It is computed by the vertex count payload element $vcp$, which takes a vertex summary $vs_v$ as input and returns an integer denoting the number of summarized vertices. Thus, formally $vcp$ is a function defined as $vcp(vs_v) := |vvp(vs_v)|$.

### 4.4.3 Data Source Search

Our third considered application of structural graph summaries is to use them for data source search (see Section 1.1.3). To implement this task, we define our third payload element, i.e., the data source payload element $dsp$. The data source payload element $dsp$ maps a vertex summary to the set of data source IRIs of all summarized vertices. This payload is only valid for named graphs, i.e., where each triple $(s, p, o)$ is extended to a tuple $((s, p, o), d)$ (see Chapter 2).
Definition 17. [17] The data source payload contains the locations of the vertices summarized by the vertex summary. The payload element $dsp$ is a function that takes a vertex summary $vs_v$ as input and returns all data source URLs $d$ for which there is tuple $((s, p, o), d)$ in $G$ for some vertex $s$ that is summarized by the vertex summary $vs_v$. Formally, the data source payload is defined as:

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

4.5 Defining Graph Summary Models with FLUID

In this section, we review all existing works analyzed in Section 3.1 and show how to define them using FLUID. Thus, we demonstrate that FLUID can indeed be used to model all the existing (semantic) structural graph summaries in Table 3.1. We briefly describe their definitions using FLUID, which are shown in Table 4.2. Note that the summaries defined in the related work were defined for specific tasks (e.g., query size estimation) so, here, we use the payload appropriate to that task. We could, of course, adapt the summaries to other tasks by using other payloads.

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

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

For the second group of complex graph summaries, we use the complex schema element to combine equivalence relations. To define SchemEX, we combine the type cluster $OC_{type}$ and $id_{rel}$, the label-parameterized identity equivalence using all predicates except $\text{rdf:type}$ (i.e., only properties), in a
Table 4.2: All graph summary models from the related work (see Table 3.1) defined using the FLUID language.

<table>
<thead>
<tr>
<th>Graph Summary Model</th>
<th>FLUID definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Attribute Collection</td>
<td>((G, PC_{rel}, {\text{vip}}))</td>
</tr>
<tr>
<td>Class Collection</td>
<td>((G, OC_{\text{type}}, {\text{vip}}))</td>
</tr>
<tr>
<td>Characteristic Sets</td>
<td>((G, b-PC, {\text{vcp}}))</td>
</tr>
<tr>
<td>SemSets</td>
<td>((G, POC, {\text{vip}}))</td>
</tr>
<tr>
<td>LODex [10]</td>
<td>((G, (OC_{\text{type}}, \text{id}<em>{rel}, OC</em>{\text{type}}), {\text{vip}}))</td>
</tr>
<tr>
<td>Loupe [64]</td>
<td>((G, (OC_{\text{type}}, \text{id}<em>{rel}, OC</em>{\text{type}}), {\text{vip}}))</td>
</tr>
<tr>
<td>SchemEX [58]</td>
<td>((G, (OC_{\text{type}}, \text{id}<em>{rel}, OC</em>{\text{type}}), {\text{dp}}))</td>
</tr>
<tr>
<td>SchemEX+U+I [15]</td>
<td>((G_{\Gamma_{\text{RDFS}}}, (OC_{\text{type}}, \text{id}<em>{rel}, OC</em>{\text{type}}){\sigma}, {\text{dp}})))</td>
</tr>
<tr>
<td>ABSTAT [84]</td>
<td>((G_{\Gamma_{\text{RDFS}}}, (OC_{\text{type}}, \text{id}<em>{rel}, OC</em>{\text{type}}), {\text{vip}}))</td>
</tr>
<tr>
<td>TermPicker [85]</td>
<td>((G, (OC_{\text{type}} \cap \text{PC}<em>{rel}, \top, OC</em>{\text{type}}), {\text{vip}}))</td>
</tr>
<tr>
<td>Weak Summary [40]</td>
<td>((G_{\Gamma_{\text{RDFS}}}, \text{i-PC}<em>{rel}{\rho} \cup \exists \text{PC}</em>{rel}{\rho}, {\text{vip}}))</td>
</tr>
<tr>
<td>Strong Summary [40]</td>
<td>((G_{\Gamma_{\text{RDFS}}}, \text{b-PC}_{rel}{\rho}, {\text{vip}}))</td>
</tr>
<tr>
<td>Typed Weak Summary [40]</td>
<td>((G_{\Gamma_{\text{RDFS}}}, (OC_{\text{type}} \cap \exists \text{i-PC}<em>{rel}{\rho}) \cup \exists \text{OC}</em>{\text{type}}{\rho}, {\text{vip}}))</td>
</tr>
<tr>
<td>Typed Strong Summary [40]</td>
<td>((G_{\Gamma_{\text{RDFS}}}, (OC_{\text{type}} \cap \exists \text{b-PC}<em>{rel}{\rho}) \cup \exists \text{OC}</em>{\text{type}}{\rho}, {\text{vip}}))</td>
</tr>
<tr>
<td>Tran et al. [88]</td>
<td>((G, (\top, \text{id}_{L}, \top)^k, {\text{vip}}))</td>
</tr>
<tr>
<td>A(k)-index [54]</td>
<td>((G, b-(OC_{\text{type}}, \text{id}<em>{rel}, OC</em>{\text{type}})^k, {\text{vip}}))</td>
</tr>
<tr>
<td>T-index [65]</td>
<td>((G, (i-PC, \top, \top)^k, {\text{vip}}))</td>
</tr>
<tr>
<td>Consens et al. [27]</td>
<td>((G, (OC_{\text{type}}, \text{id}<em>{rel}, OC</em>{\text{type}})^k, {\text{vip}}))</td>
</tr>
<tr>
<td>Schätzle et al. [81]</td>
<td>((G, (OC, \text{id}, OC)^k, {\text{vip}}))</td>
</tr>
</tbody>
</table>

Complex schema element. Since complex schema elements support predicate paths by default, they capture which predicate linked to which type set of which neighbor. As payload, we use the data sources of summarized vertices, i.e., the data source payload element \(\text{dp}\). For SchemEX+U+I, we additionally include SameAs instances \(\Delta = \sigma\) and an RDFS vocabulary graph \(V_{\text{RDFS}}\) on top of SchemEX. ABSTAT, LODex, and Loupe define the same graph summary as SchemEX, except that they use the vertex identity payload instead of the data source payload and ABSTAT uses RDF Schema inference on the data graph. To define TermPicker, we use the intersection of the label-parameterized object cluster \(OC_{\text{type}}\) and the label-parameterized property cluster \(PC_{rel}\). The existence of any predicate equivalence \(\sim \rho \neq \top\) in a complex schema element enables predicate paths, capturing which
predicate links to which type set of a neighbor. TermPicker defines a graph summary that does not use predicate paths, so we use the tautology equivalence $\top$ as predicate equivalence. This way, no predicate paths are taken into account and all types of all neighbors are aggregated.

The next four summary models are proposed by Goasdoué et al. [40]. All of their summary models use the instance parameterization with related property instances $\Delta = \rho$. The Weak Summary summarizes vertices based on incoming properties or outgoing properties or both. Thus, we define it with directed property clusters $i$-PC$_{rel}$ and $o$-PC$_{rel}$ combined using the extended union operator $\cup_{ex}$. The Strong Summary summarizes vertices based on incoming properties and outgoing properties. Thus, we define it with a bidirectional property cluster b-PC$_{rel}$. The Typed Weak Summary summarizes vertices based on the Weak Summary only if the vertices have empty type sets. Otherwise, they are summarized based on having the same type sets. We express this using the set parameterization applied on the type cluster $OC_{type}$ with $S = \emptyset$. $OC_{type}|\emptyset$ partitions the vertices into those with no types and those with types in the data graph. $OC_{type}|V_C$ partitions the vertices into those with no types and those with the same type sets in the data graph. Analogously, we define the Typed Strong Summary, which summarizes based on the Strong Summary only if the vertices have empty type sets.

The final five approaches use $k$-bisimulation and can be defined using the chaining parameterization. For the label- and height-parameterized graph summary model proposed by Tran et al. [88], we use the complex schema element that only compares outgoing predicates included in the label set $L$ up to a maximum hop length of $k$. The parameters $L$ and $k$ are user defined. Kaushik et al. [54] define their $A(k)$-index using forward and backward bisimulation, i.e., it follows incoming and outgoing predicate paths up to a maximum hop length of $k$. We define this using the bidirectional predicate cluster chained in a complex schema element. Analogously, we define the T-index proposed by Milo and Suciu [65] for incoming predicates only. Consens et al. [27] summarize vertices based on outgoing property paths only and also compare the type sets of each vertex. The graph summary proposed by Schätzle et al. [81] is analogous to the previous one, but compares the vertices’ identities instead of their type sets.

4.6 Data Structure of FLUID Graph Summaries

In this section, we define our data structure to store all graph summaries defined using FLUID. Note that we define the data structure without loss of generality as LPG. As shown in Section 2.3, any LPG graph can be transformed into an RDF graph, and vice versa.
Let $GDB = (V, E, G, \ell_G)$ be a graph database with label functions $\ell_V$, $\ell_E$, and $\ell_P$ and let $\sim$ be an equivalence relation over $V$. The **graph summary** for $GDB$ with respect to $\sim$ is a labeled graph $SG = (V_{vs} \cup V_{pe}, E_{vs} \cup E_{pe}, \ell_V, \ell_E, \ell_P)$, where $E_{vs} \subseteq V_{vs} \times V_{vs}$ and $E_{pe} \subseteq V_{vs} \times V_{pe}$.

Here, the subscript “vs” denotes “vertex summary”. The subgraph $VS = (V_{vs}, E_{vs}, \ell_V, \ell_E, \ell_P)$ contains the schema information about the GDB according to the model used for $\sim$, as introduced in Section 4.1 (compare Figure 4.7 upper half). Thus, the vertices $V_{vs}$ and edges $E_{vs}$ are those shown in the graph summary $SG$ in Figure 1.3. The subscript “pe” denotes “payload elements”. The graph $PG = (V_{vs} \cup V_{pe}, E_{pe}, \ell_V, \ell_E, \ell_P)$ connects the schema to the payload, i.e., each edge $(v, w) \in E_{pe}$ connects a vertex $v$ in $VS$ to a vertex $w$ (the payload element) that contains $v$’s payload information (compare Figure 4.7 bottom half). $SG$ 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.

**Example 10.** Consider the GDB at time $t$ in Figure 1.3. The graph summary $SG$ is defined as the SchemEX graph summary model. We denote the equivalence relation of the SchemEX graph summary model as $\sim$. SchemEX summarizes vertices that have the same types (vertex labels) and that have the same set of properties (outgoing edge labels) that are connecting a neighbor that has the same

![Diagram](image)

**Figure 4.7:** A structural graph summary contains the schema structure of summarized vertices (top) and contains payload information about the summarized vertices (bottom). The schema structure of vertices is represented using vertex summaries identified by a single primary vertex, e.g., $r_1$, and zero ore more secondary vertices, e.g., $s_2$ and $s_3$.  

58
types (vertex labels). SchemEX is defined as $CSE(OC_{type}, id_{rel}, OC_{type})$, where $OC_{type}$ (the “same type” equivalence) has classes $\{v1, v7\}$, $\{v2, v8\}$ and $\{v3, v9\}$. The identifiers of the vertices in $SG$ are $r1 = (\{v1, v7\}, \sim), r2 = (\{v2, v8\}, OC_{type})$ and $r3 = (\{v3, v9\}, OC_{type})$. Thus, the vertices $r1$ and $r2$ are primary, while $r2$ and $r3$ are secondary.

For each $v \in V$ (of the GDB) and equivalence relation $\sim$ (defined by simple or complex schema elements), we define the local vertex summary $vs_{\sim}(v)$ 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 $id = \{(v, v) \mid v \in V\}$ and $T = V \times V$. For any vertex $v \in V$, $vs_{id}(v)$ is the graph with the single vertex $\{(v, id)\}$, 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 $vs_{id}(v)$ 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 $(T; id, id)$, respectively. So, let $\sim$ be the equivalence relation defined by the CSE $(\bar{\iota}, \bar{p}, \bar{\odot})$ and let $v \in V$. Let $\Gamma^v = \{[w]_{\bar{\iota}} \mid w \in \Gamma(v)\}$ be the set of $\bar{\iota}$-equivalence classes of $v$’s neighbors. The primary vertex of $vs_{\sim}(v)$ is $([v], \bar{\iota})$. For each equivalence class $C \in \Gamma^v$, $vs_{\sim}(v)$ has a subgraph $vs_{\bar{\iota}}([w_C])$, where $w_C$ is an arbitrary vertex in $C$. Now, let $B_{\bar{p}}^C = \{[(v, w)]_{\bar{p}} \mid (v, w) \in E$ and $w \in C\}$; i.e., if $v$ has neighbors in $\bar{\iota}$-class $C$, then $B_{\bar{p}}^C$ is the set of $\bar{p}$-equivalence classes of the edges linking $v$ with a vertex $w \in C$. For each class $\beta \in B_{\bar{p}}^C$, $vs_{\sim}(v)$ contains an edge labeled $\beta$ from its primary vertex to the primary vertex of $vs_{\bar{\iota}}([w_C])$. Distinct vertex summaries may share vertices, which compresses the graph summary, since data is reused.

**Theorem 1.** [17] Let $GDB = (V, E, G, \ell_G)$ 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 $\nu_{\sim}(v)$ 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 $\nu_{\sim}(v)$ has degree more than $d$. $\nu_{\sim}(v)$ has depth $k$, so it contains at most $\sum_{i=0}^{k} d^i = O(d^k)$ vertices.

In Theorem 1, we show that the size of a single vertex summary can be bounded by the maximum degree in the input graph and the chaining parameter $k$. Thus, 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.

### 4.7 Summary

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 \cite{23}. For example, vertices in a data graph might be summarized if they share the same type set. 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 data graph 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 vertex identifiers for semantic entity retrieval described in Section 1.1.1, the number of summarized vertices for cardinality computation described in Section 1.1.2, or the data sources for data source search described in Section 1.1.3. This information about the summarized vertices is called the payload.

Such structural graph summaries 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. All combinations of the elements following this grammar are valid structural graph summary models based on equivalence relations. In this chapter, we defined FLUID for RDF graphs. As shown in Section 2.3, RDF graphs can be transformed into LPGs, and vice versa. Thus, FLUID can also be used to summarize LPGs. Following our own transformation for RDF graphs into LPGs from Section 2.3, we summarize below FLUID’s schema elements and parameterizations, but applied to LPGs. Thus, showing that indeed FLUID can be applied on RDF graphs and LPGs.

Let $GDB = (V, E, \mathcal{G}, \ell_G)$ be a label property graph database with label functions $\ell_V, \ell_E, \ell_P$. The three Simple Schema Elements (SSEs) summarize vertices $v$ using only $\ell_V(v), \ell_E(v, w)$, and/or
vertex identifiers \( w \) with \( w \in \Gamma(v) \). \textbf{Object Cluster} (OC) compares types and vertex identifiers of all neighboring vertices: two vertices \( v \) and \( v' \) are equivalent iff \( \ell_V(v) = \ell_V(v') \) and \( \Gamma(v) = \Gamma(v') \).

\textbf{Predicate Cluster} (PC) compares edge labels: \( v \) and \( v' \) are equivalent iff (i) their vertex label sets are both empty or both non-empty and (ii) for all outgoing edge labels \( \ell_E(v, w) \) there is an identical edge label \( \ell_E(v', w') \) for \( v' \), and vice versa. The condition on the existence of vertex label results from the use of the \texttt{rdf:type} predicate in RDF graphs. \textbf{Predicate-Object Cluster} (POC) combines PC and OC: \( v \) and \( v' \) are equivalent iff, \( \ell_V(v) = \ell_V(v') \) and 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. Complex Schema Elements (CSEs) use information beyond this local schema structure. CSEs are defined using a tuple of three equivalence relations, i.e., \( \text{CSE} := (\sim^s, \sim^p, \sim^o) \). \( \sim^s \) 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.

\textbf{Parameterizations} further specify the simple and complex schema elements. 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. Note that while in RDF graphs types are attached to vertices with edges labeled \texttt{rdf:type}, they are represented as vertex label in LPGs. We defined the type cluster \( \text{OC}_{\text{type}} \) as the label parameterized Object Cluster with \( P_l = \{ \text{rdf:type} \} \). Thus, \( \text{OC}_{\text{type}} \) compares vertex label sets: two vertices \( v \) and \( v' \) are equivalent iff \( \ell_V(v) = \ell_V(v') \). Similarly, we defined the property cluster \( \text{PC}_{\text{rel}} \) as the label parameterized Predicate Cluster with \( P_l = P \setminus \{ \text{rdf:type} \} \). Thus, \( \text{PC}_{\text{rel}} \) compares edge labels: \( v \) and \( v' \) are equivalent iff, for all outgoing edge labels \( \ell_E(v, w) \) there is an identical edge label \( \ell_E(v', w') \) for \( v' \), and vice versa.

The remaining parameterizations do not require particular attention to be applied to LPGs. The chaining parameterization has a parameter \( k \) that limits the maximum radius of the considered subgraphs for CSEs to \( k \) and is denoted as \( \text{CSE}_k \). 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 \texttt{owl:sameAs}.
In summary, we define graph summary models to be 3-tuples consisting of a data graph $G$, an equivalence relation $EQR$, and payload elements PAY. We introduced three simple and one complex schema elements as well as six parameterizations to define graph summaries. We demonstrated that these elements and parameterizations can be flexibly combined to define existing (semantic) structural graph summaries. All analyzed graph summaries discussed in Section 3.1 can be expressed using FLUID. We can also adapt existing graph summaries to new tasks. The simplest modification is changing the payload elements, which is easily done, but has a big impact on the size of graph summary and on which tasks can be fulfilled. For example, attaching the number of summarized vertices requires less space than attaching identifiers of summarized vertices. Furthermore, we can adapt the captured schema structure by filtering out specific types and properties or enabling inference on semantic graphs. For example, we can define a new graph summary that only uses outgoing properties and uses inference: PC-Inferred := $(G_{VGRDFS}, (\top, id, id), \{dp\})$.

There are various new possibilities to combine FLUID’s schema elements and parameterizations to define existing and new (semantic) structural graph summaries. It can be shown that every combination of FLUID’s schema elements and parameterizations actually defines a partition over the data graph and thus, is a valid graph summary. All schema elements and parameterizations are defined as equivalence relations. The intersection and the extended union of such equivalence relations is again an equivalence relation. Expressing (semantic) structural graph summaries with only a handful elements and parameterizations allows us to define a generic algorithm to compute (semantic) structural graph summaries. We present this algorithm and its computational complexity in the next section.
In this chapter, we introduce our algorithm to compute structural graph summaries. We proposed our algorithm initially in the peer-reviewed workshop paper [13]. This chapter is based on the further developed peer-reviewed journal article [17]. We propose a single, parameterized algorithm, which can compute all structural graph summaries defined with FLUID. To compute a structural graph summary, we need to summarize vertices to vertex summaries, partitioning the data graph into disjoint subsets of vertices. This is a version of the set union problem, for which Tarjan’s algorithm has been proven to be asymptotically optimal [86]. This algorithm is based on operations “make-set” and “find” and takes time $\Theta(n \cdot \alpha(n, n))$ for a sequence of $n$ of these operations, where $\alpha$ is the inverse of Ackermann’s function. Although unbounded, $\alpha$ grows extremely slowly and it is generally accepted that $\alpha(n, n) \leq 4$ for all practically possible inputs [86]. We therefore refer to the running time of $\Theta(n \cdot \alpha(n, n))$ as being “essentially linear time”. This means, the lower bound for the summary computation is essentially linear time since we cannot be faster than Tarjan’s algorithm.
Algorithm

FLUID describes the rules of how to combine schema elements, parameterizations, and payload elements. To compute structural graph summaries, we propose an algorithm based on hash maps. We assume that we can implement make-set and find operations in constant time (amortized) using hash maps, using hashes that are long enough to avoid collisions \[58\]. Our algorithm is presented in Algorithm 3. To simplify the code, we only show an excerpt of the complete algorithm. Furthermore, we define a set of globally accessible hash maps in the data structure GlobalVariables to ease readability.

In the first pass over the data graph \(G\) (Line 6 to Line 16), we iterate once over all triples to prepare our data structures. Each triple using a property \(p \in P_{\text{RDFS}}\) as predicate is added to the vocabulary graph \(V_G_{\text{RDFS}}\). The remaining triples are added to the InMap and the OutMap. The InMap stores the triples with the subject \(s\) as key and the OutMap stores triples with the object \(o\) as key. This allows us to quickly retrieve all incoming and outgoing triples for each vertex in \(G\). In case we are using SameAs instances, we construct sets of equivalent vertices following the owl:sameAs property in Lines 13 and 14. In case we are using related property instances, we construct sets of equivalent vertices following any outgoing or incoming property in Lines 15 and 16, respectively.

In the second pass (Line 17 to Line 21), we infer incoming and outgoing triples according to the constructed Vocabulary Graph \(V_G_{\text{RDFS}}\). The inferred incoming and outgoing triples are stored in the hash maps InfInMap and InfOutMap, respectively. In the third pass, the schema for each vertex is computed. For each (parameterized) schema element used to define the equivalence relation EQR, we extract the schema and compute the vertex summaries. These vertex summaries are stored in SchemaMap and identified by their primary vertices (see Section 4.6). Using the primary vertices, the summaries will be combined to form the vertex summary \(vs\) for EQR (compare Example 4 on Page 44). The primary vertices of all combined vertex summaries will become secondary vertices of the resulting summary \(vs\). Finally, for each payload element in PAY, we compute the payload for each vertex \(v\). The resulting vertex summary \(vs\) for each \(v\) along with the payloads are added to the summary graph \(SG\) in Line 26.

Example 11. Suppose we want to extract the schema according to a predicate cluster (PC) for a vertex \(s\) and there are two triples \((s, p_1, o_1)\) and \((s, p_2, o_2)\) in \(G\) with \(s\) as subject. We extract the predicate for each triple and construct the corresponding set, i.e., \(\{p_1, p_2\}\). We compute the hash value of the predicate set and store it in the schema hash map SchemaMap. The payload, e.g., the number of
summarized vertices, is stored as value. For another vertex \( s' \) with \((s', p_1, o_3), (s', p_2, o_4) \in G\), the same predicate set is extracted, so we compute the same hash value. When we update the SchemaMap, we update the payload. In this example, we increase the vertex counter by one. Consequently, the resulting vertex summary defined as PC summarizes both vertices and only stores the defined payload.

**Algorithm 3:** Sequential, parameterized algorithm to compute structural graph summaries defined as equivalence relation EQR with the FLUID language [17]

```plaintext
function ComputeGraphSummary(G, EQR, PAY)
returns graph summary SG

struct GlobalVariables = {
    OutMap ← ∅,
    InMap ← ∅,
    InfOutMap ← ∅,
    InfInMap ← ∅,
    SameAsInstanceMap ← ∅;
    SrcRelatedMap ← ∅,
    TrgRelatedMap ← ∅,
    SchemaMap ← ∅;
}

forall (s, p, o) ∈ G do
    if p ∈ PRDFS then
        /* Add triple to vocabulary graph if it is an RDF Schema triple. */
        VG.AddElement((s, p, o));
    else
        /* Index triples by subject vertex s. */
        OutMap.Merge(s, {(s, p, o)});
        /* Index triples by object vertex o. */
        InMap.Merge(o, {(s, p, o)});
        /* In case SameAs instances are used */
        if p = owl:sameAs then
            SameAsInstanceMap.Merge(s, {o});
            SameAsInstanceMap.Merge(o, {s});
        /* In case property-related instances are used */
        SrcRelatedMap.Merge(p, {s});
        TrgRelatedMap.Merge(p, {o});
        /* Prepare for inferencing (see Section 4.3.5) */
        if G uses RDF Schema inferencing then
            forall (v, OUT) ∈ OutMap do
                InfOutMap.Put(v, VG.InferOntologyInformation(OUT));
           forall (v, IN) ∈ InMap do
                InfInMap.Put(v, VG.InferOntologyInformation(OUT));
            /* Only consider vertices that are the subject of at least one triple */
            forall v ∈ OutMap.Keys() do
                us ← ExtractSchema(v, EQR);
            forall pay ∈ PAY do
                pi ← pay.ExtractPayload(v);
                SG.AddElement(us, ∪ pi);
            return SG;
```

65
One can see that the computationally expensive task is the extraction of the schema of each vertex, e. g., extracting the property set in Example 11. However, our algorithm benefits from the fact that every schema element in FLUID is defined as equivalence relation. In particular, for one vertex \( s \) there is exactly one schema structure according to any one (parameterized) schema element, e. g., one predicate cluster (PC) and one label-parameterized object cluster (OC\_{\text{type}}) etc. We define complex graph summaries by combining simple schema elements with complex schema elements, where we can include neighboring vertices in the schema structure. For example, SchemEX uses type sets of neighboring vertices to compute the schema of the actual vertex. Therefore, one may assume that depending on the in-degree of a vertex, we have to extract the schema more than once. Instead, we use a hash map to store, for each vertex in the data graph, all computed vertex summaries identified by secondary vertices. This avoids the expensive task of extracting the schema of vertices more than once.

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

### 5.2 Complexity Analysis

We analyze the complexity of computing (semantic) structural graph summaries defined with FLUID. This analysis was originally published in [17]. In order to estimate the computational complexity, we conduct a space and time analysis of the computation process for graph summaries defined with FLUID. For this analysis, we assume that hash maps have amortized constant-time read and write access and that hashes are long enough to avoid collisions when hashing, e. g., vertices’ property sets and type sets [58]. Goasdoué et al. [40] justify this assumption, via their hypothesis (\( \star \)), and provide an alternative data structure using Tarjan’s algorithm instead of hash maps. In Section 8.1.1, we find evidence that supports their hypothesis, i. e., that the numbers of incoming and outgoing predicates for each vertex are, in practice, bounded.

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

5.2.1 Schema Elements

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

The data graph $G$ contains $n$ triples. Simple schema elements determine whether two vertices $s, s' \in G$ are equivalent by considering only triples that have $s$ or $s'$ as subject. To compute PC, OC, or POC, we need to compute, for each vertex $s \in G$, the set $\{p | (s, p, o) \in G \text{ for some } o\}$, $\{o | (s, p, o) \in G \text{ for some } p\}$ or $\{(p, o) | (s, p, o) \in G\}$, respectively. All of these sets can be computed in a single scan of the graph. A hash for each set can be computed by combining the hashes of its elements, e.g., using sums, multiplications, XOR, or techniques based on symmetric polynomials [9]. The equivalence relations corresponding to the SSEs can be computed by comparing these hashes.

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

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

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

5.2.3 Set Parameterization

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

5.2.4 Chaining Parameterization

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

5.2.5 Direction Parameterization

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

For the inference parameterization, we have to consider the space and time required to build both the summary graph \(SG\) and the vocabulary graph \(VG\). As defined in Section 4.3.5, the vocabulary graph is constructed by adding types and properties to it, if there exists a triple in \(G\) using a predicate in \(PRDFS\) (Algorithm 3, Line 8). In the input data graph \(G\) of size \(n\), we find \(r\) schema triples that use a property in \(PRDFS\), with \(r \leq n\). Constructing the vocabulary graph is done analogously by updating hash maps. Thus, constructing the vocabulary graph changes neither the build-time nor the space complexity.

However, inferring information can change the overall complexity. For our analysis, we distinguish the two cases of adding inferrable information inside the structural graph summary and adding inferrable information outside of the structural graph summary (see Section 4.3.5).

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

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

Example 12. Consider an RDF graph with the triple set

\[
\{(s_1, p_1, o_1), (s_2, p_1, o_2), \ldots, (s_{n/2}, p_1, o_{n/2}),
(p_1, \text{rdfs:subPropertyOf}, p_2),
(p_2, \text{rdfs:subPropertyOf}, p_3),
\ldots,
(p_{n/2-1}, \text{rdfs:subPropertyOf}, p_{n/2})\} ,
\]

where we assume \(n\) to be even. Inference on this graph results in the graph containing the \(n^2/4\) triples \(\{(s_i, p_j, o_i) \mid 1 \leq i, j \leq n/2\}\). This shows that inference can produce a quadratic blow-up in the number of tuples in a graph.
In the next chapter, we analyze large real-world semantic graphs obtained from the Semantic Web. Our analysis suggests that adding inferrable information inside the structural graph summary may be infeasible in some cases. In these cases, adding inferrable information should be done outside of the structural graph summary or after the summary computation, as discussed in Section 4.3.5.

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

5.2.7 Instance Parameterization

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

As described in Section 5.1, two vertices can be aggregated using the instance parameterization, in principal, in amortized constant time using hash maps (Algorithm 3, Lines 13 to 16). To handle transitivity, we have to recursively access the hash map SameAsInstanceMap (for SameAs instances) or SrcRelatedMap and TrgRelatedMap (for related property instances). Transitivity means here that, for each equivalent vertex $v'$ of some vertex $v$, we use the hash maps to look up the set of equivalent vertices of $v'$. The transitive closure only needs to be computed once.

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

Graph summaries defined with FLUID can be computed in linear time and space with respect to the number of triples $n$, unless the inference parameterization is used, in which case the running time and space usage may be quadratic in $n$. For the inference parameterization, we run our own analyses on large real-world datasets to justify the assumption of essential linear runtime. These analyses can be found in Section 6.4.
From many described approaches, 7 approaches have been reasonably selected for the experiments. Although 7 approaches is already quite many, looking at the number of available approaches, many more approaches could be additionally chosen [for] the experiments (future work).

R2, DEXA, 2020

6

Empirical Evaluation of the Sequential Algorithm

In the previous chapters, we discussed a variety of structural graph summaries, presented our formal model FLUID to define them, our generic data structure to store them, and our parameterized algorithm to compute them. In this chapter, we use this as foundation to compute a representative sample of structural graph summaries. We evaluate the computed structural graph summaries in the context of the data source search task described in Section 1.1.3. As a reminder, the task of data source search is to find (sub)graphs on the Web that match a given schema structure.

As discussed in Chapter 3, different structural graph summaries have been developed for different tasks such as query size estimation [68], vocabulary terms recommendation [80], and related entity retrieval [23]. The problem is that all graph summaries were designed, implemented, and evaluated for their individual task only, using different queries, datasets, and metrics (compare problem statement in Section 1.2.1). We use FLUID and the parameterized algorithm to fill this gap. We conduct an
extensive empirical evaluation of representative graph summary models. Based on the discussion of related works, we choose six graph summary models with unique feature combinations, which were originally developed for different tasks, to understand and compare their behavior for the data search task. We empirically investigate the behavior of each selected graph summary model in three variants, where we use the chaining parameterization to summarize based on subgraphs with a maximum radius of 0, 1, and 2. The empirical evaluation consists of three sets of experiments.

In a first set of experiments, we analyze the relative size of the computed graph summary compared to the original dataset (compression ratio) and the number of vertex summaries in the graph summary compared to the number of summarized vertices in the dataset (summarization ratio). As a reminder, we use our vertex summaries as generic data structure to store information about the summarized vertices in the graph summary (see Section 4.6).

The second set of experiments quantifies the quality of a stream-based computation of the graph summary for large datasets obtained from the Semantic Web. The stream-based approach is designed to scale to graphs of arbitrary sizes by observing the graph over a stream of edges with fixed window size. Inherently, this approach introduces inaccuracies in the graph summary computation by potentially extracting incomplete schema structures due to limited window size [58].

Finally, we evaluate the impact of the RDF Schema inferencing parameterization on structural graph summaries. To this end, we analyze four large semantic graphs obtained from the Semantic Web w.r.t. RDF Schema predicates.

The experiments are published in peer-reviewed conference and journal articles [15, 17, 82]. The remainder of this chapter is structured as follows: First, we introduce the general experimental apparatus, i.e., the datasets and the structural graph summary models. Subsequently, we present the three experiments outlined above as well as their results, in Sections 6.2 to 6.4. In Section 6.5, we discuss the results of all three empirical evaluations. Finally, we summarize the main findings of this chapter.

6.1 General Experimental Apparatus

All three experiments are conducted using the same datasets and the same graph summary models. We first describe the datasets and subsequently the graph summary models. The information specific to each experiment is described in the respective sections.
6.1.1 Datasets

In this section, we describe four real-world datasets obtained from the Semantic Web. The datasets have different characteristics resulting from different crawling strategies. In particular, we selected datasets with different sizes (number of triples \( n \)), obtained in different years, and containing multiple different data providers, e.g., DBpedia*, Wikidata†, and others.

TimBL-11M. The first dataset is called TimBL-11M and contains about 11 million triples [58]. The crawl was conducted in 2011 with a breadth-first search starting from the single URL of Tim Berners-Lee’s FOAF profile. The TimBL-11M dataset is a graph with 673 thousand subject vertices distributed over 18 thousand data sources. Each subject vertex has on average 13 outgoing properties (±275), 3 incoming properties (±848), 3 types (±40), and is defined in 3 data sources (±40). The latter means that, on average, triples with a common subject vertex can be found in three distinct data sources, which makes the challenge for data search specifically hard. Overall, 3, 919 unique properties and 2, 738 unique RDF types appear in the TimBL-11M dataset.

DyLDO-127M. The second dataset is called DyLDO-127M and contains about 127 million triples [52]. The Dynamic Linked Data Observatory (DyLDO) provides regular snapshots of the Semantic Web. In contrast to TimBL-11M, which uses only a single seed URL, the DyLDO populates their datasets by crawling in breadth-first search about 95 thousand representative seed URLs of the Semantic Web [52]. The crawling of these seed URLs is restricted to a crawling depth of two hops. Note that a crawling is done over data source URLs, i.e., paths in the graph can be longer than 2 hops. We use the first snapshot of the DyLDO dataset, as it is with 127M triples the largest one. The DyLDO-127M dataset contains 7 million subject vertices distributed over 154 thousand data sources. Each subject vertex has on average 17 outgoing properties (±6503), 7 incoming properties (±635), a single type (±17), and is defined in 2 data sources (±17). Overall, 15 thousand unique properties and 31 thousand unique types appear in the DyLDO-127M dataset.

---

* DBpedia is a crowd-sourced community effort to extract structured content from the information created in various Wikimedia projects. This structured information resembles an open knowledge graph (OKG) which is available for everyone on the Web.” – [https://www.dbpedia.org/](https://www.dbpedia.org/), last accessed: November 17, 2022

† Wikidata is a free and open knowledge base that can be read and edited by both humans and machines. Wikidata acts as central storage for the structured data of its Wikimedia sister projects including Wikipedia, Wikivoyage, Wiktionary, Wikisource, and others.” – [https://www.wikidata.org/wiki/Wikidata:Main_Page](https://www.wikidata.org/wiki/Wikidata:Main_Page), last accessed: November 17, 2022

75
BTC-2B. The third dataset is the Billion Triple Challenge 2019 dataset (BTC-2B), which contains about 2 billion triples \[50\]. The BTC-2B dataset was crawled breadth-first in January 2019 starting from 450 seed URLs taken from the DyLDO-127M dataset \[50\].

Laundromat-38B. To the best of our knowledge, the largest collections of Linked Open Data (LOD) is the LOD Laundromat dataset (Laundromat-38B) \[78\]. It contains more than 38 billion triples and combines various other data sources into one single dataset. In contrast to the previous datasets, the LOD Laundromat dataset is not a single crawled dataset but contains different crawled datasets, database exports, and user uploads.

6.1.2 Structural Graph Summaries

We select six representative models based on our analysis in Section 3.1. These models are Characteristic Sets, Weak Summary, SemSets, TermPicker, SchemEX, and SchemEX+U+I. For each of the six selected models, we apply the \(k\)-chaining parameterization (see Definition 10 on Page 47). Reasonable values for \(k\) are 0, 1, and 2 \[88\]. Thus, in total, we compare 18 unique graph summary models.

Characteristic Sets \[68\] uses property sets and incoming property sets of each vertex only, while most other graph summary models also take type sets into account. When parameterizing Characteristic Sets using the chaining parameterization (short: chaining Characteristic Sets), the resulting graph summary is equivalent to the T-index \[65\]. The Weak Summary \[40\] is similar to Characteristic Sets, but is unique in using the \(\cup_{ex}\) combination to combine property sets and incoming property sets. SemSets \[25\] uses outgoing properties and object equivalences. When chaining SemSets, the resulting graph summaries are equivalent to the k-bisimulation approach proposed by Schätzle et al. \[81\]. TermPicker \[80\] used sets of neighbors instead of predicate paths to integrate neighbor type information. Thus, chaining TermPicker does not result in a bisimulation. SchemEX \[58\] shares the same base schema structure with LODex, Loupe, and ABSTAT (see Table 3.1). Furthermore, chaining SchemEX results in the same graph summaries using k-bisimulation as the one proposed by Consens et al. \[27\]. Finally, we select SchemEX+U+I since it uses \texttt{owl:sameAs} and full RDFS reasoning.

6.2 Experiment 1: Compression and Summarization

In this experiment, we are interested in how well the graph summary models of varying complexity summarize real-world data obtained from the Semantic Web. This experiment was originally published in \[15\].
6.2.1 Procedure

We use two datasets crawled from the Semantic Web, i.e., TimBL-11M and DyLDO-127M (see Section 8.1.1). With 11M triples and 127M triples, both datasets are reasonably large and still allow us to compute an exact graph summary. To compute exact graph summaries, we need to load the complete graph into the main memory at once (more details on this in Section 6.3). No pre-processing was conducted on the datasets except removing triples that do not follow the W3C standards.

We evaluate the size of the computed selected graph summaries over the two datasets. The size of a graph summary refers to the number of triples when stored as an RDF graph. We compare the number of triples in the dataset to the number of triples in the graph summary (compression ratio). Furthermore, we compare the number of vertices in the dataset to the number of vertex summaries in the graph summary (summarization ratio). This ratio gives an idea of how well the defined schema structure can summarize vertices on the Semantic Web. In contrast to the compression ratio, the summarization ratio is independent of the used graph model. This mean, when working with LPGs instead of RDF graphs (or a mix of both), the summarization ratio remains unchanged.

6.2.2 Results

The results of the experiments regarding the compression ratio and summarization ratio are documented in Table 6.1. As one can see, there is a huge variety in terms of how graph summaries compress and summarize the data. For the TimBL-11M dataset, SemSets’ compression ratio (with $k = 1$) is about 10 times lower than all other graph summaries except for Weak Summary (only about 5 times lower). For the DyLDO-127M dataset, SemSets’ compression ratio (with $k = 1$) is up to 75 times lower. Additionally, there is no increase in graph summary size from $k = 1$ to $k = 2$, but a more than ten times increase from $k = 0$ to $k = 1$. A similar increase appears for the summarization ratio. SemSets is the only graph summary that uses neighbor information but not neighbor type sets, i.e., they compare the object IRI $o$ of each $(s, p, o)$ triple. In contrast, the other graph summaries either ignore objects or consider their type sets only. SemSets has a summarization ratio of $4 - 5$, i.e., on average $4 - 5$ vertices share the same schema structure. The smallest graph summary, Characteristic Sets, has a summarization ratio of 306.1, i.e., about 300 vertices share the same schema structure. A
we use TimBL-11M (top) and DyLDO-127M (bottom).

### Table 6.1: Results from the analysis of the compression ratio and summarization ratio of the six selected graph summary models (with chaining parameter $k \in \{0, 1, 2\}$). $\#t$ is the number of triples in millions (M) in the graph summary and in brackets below the ratio compared to the number of triples in the dataset (compression ratio). $\#vs$ is the number of vertex summaries in thousands (T) in the graph summary and in brackets below the ratio compared to the number of vertices in the dataset (summarization ratio). As datasets, we use TimBL-11M (top) and DyLDO-127M (bottom).

<table>
<thead>
<tr>
<th>Model</th>
<th>$k = 0$</th>
<th>$k = 1$</th>
<th>$k = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$#t$</td>
<td>$#vs$</td>
<td>$#t$</td>
</tr>
<tr>
<td>Character-istic Sets</td>
<td>na</td>
<td>na</td>
<td>0.7M</td>
</tr>
<tr>
<td>Weak Summary</td>
<td>na</td>
<td>na</td>
<td>1.9M</td>
</tr>
<tr>
<td>SemSets</td>
<td>0.3M</td>
<td>2.8T</td>
<td>7.6M</td>
</tr>
<tr>
<td>SchemEX</td>
<td>0.3M</td>
<td>2.8T</td>
<td>0.8M</td>
</tr>
<tr>
<td>TermPicker</td>
<td>0.3M</td>
<td>2.8T</td>
<td>0.7M</td>
</tr>
<tr>
<td>SchemEX +U+I</td>
<td>0.4M</td>
<td>3.1T</td>
<td>0.8M</td>
</tr>
<tr>
<td>Character-istic Sets</td>
<td>na</td>
<td>na</td>
<td>0.6M</td>
</tr>
<tr>
<td>Weak Summary</td>
<td>na</td>
<td>na</td>
<td>14.8M</td>
</tr>
<tr>
<td>SemSets</td>
<td>4.1M</td>
<td>46.6T</td>
<td>45.3M</td>
</tr>
<tr>
<td>SchemEX</td>
<td>4.1M</td>
<td>46.6T</td>
<td>15.7M</td>
</tr>
<tr>
<td>TermPicker</td>
<td>4.1M</td>
<td>46.6T</td>
<td>11.1M</td>
</tr>
<tr>
<td>SchemEX +U+I</td>
<td>8.5M</td>
<td>53.0T</td>
<td>19.9M</td>
</tr>
</tbody>
</table>

Notable exception is the Weak Summary, which shows the most condensed summarization (summarization ratio of up to 69,000). However, the combination of either incoming or outgoing related properties in Weak Summaries leads to a considerably large compression ratio. Weak Summary graph summaries are more than twice the size of Characteristic Sets graph summaries.
When considering the semantics of RDFS and owl:sameAs in SchemEX+U+I, the graph summary size increases compared to SchemEX by about 3% more triples. Despite being a larger graph summary in terms of the number of triples, for $k = 1$ fewer vertex summaries are computed when including the semantics of RDFS and owl:sameAs. For $k = 0$ and $k = 2$, SchemEX+U+I requires more vertex summaries than SchemEX to summarize the vertices.

In summary, including the semantics of owl:sameAs and RDF Schema increases the size of the graph summary. However, it can reduce the number of vertex summaries. Furthermore, using weak equivalences leads to a handful of vertex summaries, which summarize all vertices.

### 6.3 Experiment 2: Stream-based Summary Computation

In this experiment, we are interested in how well queries of varying complexity can be supported by the graph summaries when the graph summary is computed over a stream of graph edges. This experiment was originally published in [15].

#### 6.3.1 Procedure

Motivated from stream-databases, the idea is to consider the triples in the datasets as a stream that is observed in fixed sized windows. In our experiment, we use windows of size 1000 (1T), 100,000 (100T), and 200,000 (200T). Using a fixed window size allows scaling the computation to in principle arbitrary sized input graphs [58]. However, the approach produces approximation errors since only a fraction of the data graph is kept simultaneously in the main memory, while the remainder is not yet known or inaccessible. Thus, we potentially extract incomplete schema structures.

Note that to realize a stream-based implementation, we implemented Algorithm 3 such that only one pass over the dataset is needed by implementing the loop over the triples (Algorithm 3 Line 6) and the loop over aggregated sets of triples (Algorithm 3 Line 22) as consecutive pipeline steps. Regarding the graph summary model SchemEX+U+I, we evaluate two variants in this experiment: The RDFS inferencing requires an additional data structure during the computation process, the so-called vocabulary graph (see Section 4.3.5) This vocabulary graph is constructed from the triples using RDFS range, domain, subclassOf, or subPropertyOf. With the domain, range, and hierarchical types/properties information, we infer additional types and properties for the remaining vertices (Algorithm 3 Line 17). In SchemEX+U+oI, the RDFS information is extracted and inferred on-the-fly.
Here, we construct the vocabulary graph simultaneously to the graph summary computation as an
additional pipeline step. The advantage is that still only one pass over the dataset is needed. However,
since the vocabulary graph is built while the graph summary is computed information may be missing
for the inferencing. In SchemEX+U+pI, we first extract all RDFS information in a pre-processing step
to construct the vocabulary graph. The advantage is that the inferencing of triples is conducted on
the complete vocabulary graph only. The drawback is that two passes over the dataset are needed.

**Queries.** A central challenge for this experiment is the choice of queries to be executed over the
graph summaries. Here, we follow the work by Konrath et al. [58], who conducted a data-driven
query generation for the evaluation of approximate graph graph summaries. This means the queries
are generated from the actual vertices in the datasets, i.e., their combination of types and proper-
ties. We distinguish two types of queries, simple queries (SQ) and complex queries (CQ). Simple
queries search for vertices that have a common type set (or in the case of SemSets a common set
of objects). Analyses of existing query logs show that most SPARQL queries in search systems are
simple queries [6]. In contrast, complex queries search for vertices that match the complete schema
structure defined by the specific graph summary model, e.g., include property paths over 2 hops
for Characteristic Sets with \( k = 2 \).

**Metric.** We execute the simple and complex queries on the graph summary computed with
fixed window size and on the gold standard graph summary. For our data search task, the results
of the queries are the two sets \( D_{gold} \) and \( D_{window} \), which contain the corresponding data source
URLs. Following Konrath et al. [58], the approximation quality is measured by comparing \( D_{gold} \)
and \( D_{window} \) using the F1-score.

### 6.3.2 Results

Figure 6.1 shows the approximation quality in terms of F1-score for the selected graph summary
models. For graph summaries with a chaining parameter \( k = 0 \), the simple queries and the complex
queries are alike. Moreover, Characteristic Sets and Weak Summary do not use type information (or
object information). Thus, simple queries are not available for these graph summary models.

From the results of our experiment, we can state that simple queries consistently show higher F1-
scores than complex queries. TermPicker and Weak Summary are the only graph summaries that have
Figure 6.1: F1-score for simple queries (SQ) and complex queries (CQ) and for window sizes (1T, 100T, 200T). The left column shows the values for the TimBL-11M dataset and the right column the DyLDO-127M dataset, respectively. The influence of the chaining parameter $k \in \{0, 1, 2\}$ can be seen in the rows from top to bottom.
a higher F1-score on the DyLDO-127M dataset than on the TimBL-11M dataset. As described in Chapter 3, TermPicker is the only graph summary not using the predicate path feature. This restriction is the only difference in the schema structure compared to SchemEX. Still, TermPicker has a 50% lower F1-score than SchemEX on the TimBL-11M dataset.

Regarding the complex queries, the F1-scores of Weak Summary are the highest in the experiments with $k = 1$ and 2. For $k = 2$, Weak Summary has between .12 and .54 higher F1-scores compared to the other graph summary models. SemSets only have a small drop in F1-score from $k = 1$ to $k = 2$. SchemEX+U+oI consistently has a lower F1-score than SchemEX. For SchemEX+U+pI, we extracted the RDF Schema information in a pre-processing step. Compared to SchemEX+U+oI, SchemEX+U+pI has consistently higher F1-scores. Furthermore, for window sizes 100T and 200T, SchemEX+U+pI has higher F1-scores than SchemEX (except for $k = 2$ on the TimBL-11M dataset).

We also observe an influence of the characteristics of the crawled dataset on the approximation quality. All graph summaries have on average a .15 lower F1-score on the DyLDO-127M dataset compared to the TimBL-11M dataset. In particular, simple queries achieve much lower F1-scores. On average, simple queries have .25 lower F1-scores and complex queries have .04 lower F1-scores on the DyLDO-127M dataset compared to the TimBL-11M dataset. Furthermore, larger window sizes consistently improve F1-scores. In contrast, on-the-fly inferencing lowered the F1-scores in our experiment compared to no inferencing.

6.4 Experiment 3: RDF Schema Inference Parameterization

In this experiment, we are interested in the impact of the RDF Schema inferencing on the computation time and graph summary size independent of a specific graph summary model. This experiment was originally published in [17].

6.4.1 Procedure

In this experiment, we analyze all four datasets with respect to RDF Schema inference parameterization. In the top half of Table 6.2, we present the size of the computed vocabulary graph $V_{GRDFS}$ for each dataset. Furthermore, we distinguish between property vertices, i.e., vertices representing
a property $p$ used in the data graph $G$, and type vertices. As described in Section 4.3.5, properties and types used in the data graph $G$ are only added to the vocabulary graph $V_{G_{RDFS}}$ if there is a triple about this property or type with a property $p_r \in P_{RDFS}$. For each such property $p_r$, we also counted the number of (hierarchical) relationships expressed, namely for $rdfs:subPropertyOf$, $rdfs:subClassOf$, $rdfs:domain$, and $rdfs:range$.

In the lower half of the table, we present data on the impact of RDFS inference on each dataset. First, we counted the number of properties $p$ and the number of types $t$ in the datasets and distinguished whether they are represented as vertex in the vocabulary graph $V_{G_{RDFS}}$, i.e., have inferrable information, or not. Note that we counted how many triples use the property $p$ as predicate, i.e., $(s, p, o)$, or label a vertex with the type $t$ using $rdf:type$ as predicate, i.e., $(s, rdf:type, t)$. This way, we can use these numbers to estimate an upper bound for the size of the structural graph summary. Finally, we counted the number of additional properties and types that can be added to the dataset based on the inference rules in $V_{G_{RDFS}}$. The increase factor reflects for properties, types, and both, how much additional information is added. These factors denote the influence of the inference parameterization on the build-time and space complexity of structural graph summaries for the analyzed datasets. Note that while the factor denotes the increased build-time, the storage space for the structural graph summary might be significantly lower. As described in Chapter 7, structural graph summaries summarize the data graph, i.e., common schema structures are only stored once in the summary graph.

### 6.4.2 Results

From the results of our analysis (Table 6.2), we can state that the size of the computed vocabulary graph $V_{G_{RDFS}}$ is only a small fraction of the input data graph $G$. On average, the vocabulary graph is only about 5% of the data graph in terms of number of triples. In contrast, adding the inferrable information increases the size of the data graph by a factor of between 2 and 20. Notably, the largest dataset (Laundromat-38B) also has by far the largest increase factor.

Another main result is that over all datasets, $rdfs:subClassOf$ hierarchies are considerably longer than $rdfs:subPropertyOf$ hierarchies. While on average 1.04 (max 304) additional properties can be inferred, on average 13.64 (max 3641) additional types can be inferred. This is also reflected in the number of $rdfs:subClassOf$ triples in the vocabulary graphs. Over all four vocabulary graphs, $rdfs:subClassOf$ triples make up 95% to 99% of all triples in the vocabulary graph.
Table 6.2: Dataset analysis with rounded number to the nearest thousand T, million M, or billion B.

<table>
<thead>
<tr>
<th>Vocabulary Graph $\mathcal{VG}_{\text{RDFS}}$</th>
<th>TimBL-11M</th>
<th>DyLDO-127M</th>
<th>BTC-2B</th>
<th>Laundromat-38B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property vertices ( rdfs:\text{subPropertyOf} ) triples</td>
<td>12T</td>
<td>31T</td>
<td>26T</td>
<td>330T</td>
</tr>
<tr>
<td>Type vertices ( rdfs:\text{subClassOf} ) triples</td>
<td>141T</td>
<td>312T</td>
<td>339T</td>
<td>3.7M</td>
</tr>
<tr>
<td>( rdfs:\text{domain} ) triples</td>
<td>10T</td>
<td>39T</td>
<td>38T</td>
<td>211T</td>
</tr>
<tr>
<td>( rdfs:\text{range} ) triples</td>
<td>10T</td>
<td>19T</td>
<td>16T</td>
<td>196T</td>
</tr>
<tr>
<td>Total vertices (</td>
<td>V</td>
<td>)</td>
<td>153T</td>
<td>343T</td>
</tr>
<tr>
<td>Total triples (</td>
<td>E</td>
<td>)</td>
<td>673T (6%)</td>
<td>3M (2%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset and Inference</th>
<th>TimBL-11M</th>
<th>DyLDO-127M</th>
<th>BTC-2B</th>
<th>Laundromat-38B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties ( p \notin \mathcal{VG}_{\text{RDFS}} )</td>
<td>889T (10%)</td>
<td>75M (66%)</td>
<td>98M (5%)</td>
<td>12B (42%)</td>
</tr>
<tr>
<td>Properties ( p \in \mathcal{VG}_{\text{RDFS}} )</td>
<td>8M (90%)</td>
<td>39M (34%)</td>
<td>1.9B (95%)</td>
<td>16B (58%)</td>
</tr>
<tr>
<td>Properties in dataset</td>
<td>9M</td>
<td>114M</td>
<td>2.1B</td>
<td>28B</td>
</tr>
<tr>
<td>Added through inference</td>
<td>7M</td>
<td>33M</td>
<td>703M</td>
<td>50B</td>
</tr>
<tr>
<td>Total number of properties</td>
<td>17M</td>
<td>148M</td>
<td>2.7B</td>
<td>78B</td>
</tr>
<tr>
<td>Types ( t \notin \mathcal{VG}_{\text{RDFS}} )</td>
<td>181T (9%)</td>
<td>2.6M (20%)</td>
<td>20M (22%)</td>
<td>938M (23%)</td>
</tr>
<tr>
<td>Types ( t \in \mathcal{VG}_{\text{RDFS}} )</td>
<td>2M (91%)</td>
<td>10M (80%)</td>
<td>71M (78%)</td>
<td>3B (77%)</td>
</tr>
<tr>
<td>Types in dataset</td>
<td>2M</td>
<td>13M</td>
<td>92M</td>
<td>4B</td>
</tr>
<tr>
<td>Added through inference</td>
<td>17M</td>
<td>120M</td>
<td>2.4B</td>
<td>646B</td>
</tr>
<tr>
<td>Total number of types</td>
<td>19M</td>
<td>134M</td>
<td>2.5B</td>
<td>650B</td>
</tr>
<tr>
<td>Increase factor properties</td>
<td>1.9</td>
<td>1.3</td>
<td>1.3</td>
<td>2.7</td>
</tr>
<tr>
<td>Increase factor types</td>
<td>9.2</td>
<td>10.5</td>
<td>27.5</td>
<td>160.3</td>
</tr>
<tr>
<td>Increase factor total</td>
<td><strong>3.3</strong></td>
<td><strong>2.2</strong></td>
<td><strong>2.5</strong></td>
<td><strong>22.2</strong></td>
</tr>
</tbody>
</table>

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

6.5 Discussion of Experiments

Key insights from our experiments are: (1) graph summary models perform very differently in terms of compression ratio, summarization ratio, and approximation quality depending on the queries as well.
as the characteristic of the dataset. (2) The approximation quality of a graph summary computed in a stream-based approach depends on three factors: First, we observe an influence of the characteristics of the crawled dataset. Second, simple queries consistently outperform complex queries. Third, a larger window size typically improves the approximation quality only marginally.

Regarding the first insight, we conducted a detailed analysis to understand the relationship between compression ratio and summarization ratio. We computed the Spearman correlation coefficient for the \( n = 32 \) graph summary reported in Table 6.1. Results of the Spearman correlation indicated that there was a significant relationship between compression ratio and summarization ratio, \( r_s(30) = .64, p < .0001 \). Furthermore, there is a significant correlation between summarization ratio and approximation quality of a stream-based computation approach. We computed the Pearson and Spearman correlation coefficient for the three cache sizes \( 1\T, 100\T, \) and \( 200\T \). We compared the reported F1-scores for the complex queries (for \( k = 0 \), we used the simple queries) for each cache size (Figure 6.1) to the summarization ratio of the corresponding gold standard graph summary (Table 6.1). For all three cache sizes, we found a correlation coefficient with \( p < .05 \) (see Table 6.3).

From the statistical analysis, we can see that a higher summarization ratio leads to a higher F1-score. This means graph summaries that summarize well, i.e., summarize many vertices to the same vertex summary, can be computed with high accuracy in a stream-based approach. When we compute correct vertex summaries in the stream-based approach, for graph summary models with a high summarization ratio, we assign more vertices to the correct vertex summary than for graph summary models with a low summarization ratio.

Note that the extreme summarization ratio of Weak Property Cliques also produces the highest F1-scores. This can point to an explanation for the observed correlation. With only a handful of vertex summaries in the graph summary (see Table 6.1), it is more likely that a vertex is summarized by the correct vertex summary. This results in an overall higher F1-score.

Table 6.3: Results of correlation analysis between summarization ratio and approximation quality (F1-score). Pearson and Spearman coefficients and respective p-values for \( n = 32 \) graph summaries (Table 6.1) for three cache sizes with a degree of freedom \( df = 30 \).

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>1\T</th>
<th>100\T</th>
<th>200\T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pearson</td>
<td>.16</td>
<td>.17</td>
<td>.152</td>
</tr>
<tr>
<td>Spearman</td>
<td>.73</td>
<td>.74</td>
<td>.74</td>
</tr>
<tr>
<td></td>
<td>( p &lt; .04 )</td>
<td>( p &lt; .04 )</td>
<td>( p &lt; .05 )</td>
</tr>
<tr>
<td></td>
<td>( p &lt; .0001 )</td>
<td>( p &lt; .0001 )</td>
<td>( p &lt; .0001 )</td>
</tr>
</tbody>
</table>
We also observe an influence of the characteristics of how the data has been crawled. First, all graph summaries have on average a .15 lower F1-score on the DyLDO-127M dataset compared to the TimBL-11M dataset. We explain this observation by the different crawling strategies. The TimBL-11M dataset was crawled starting from a single seed URL. In contrast, the DyLDO-127M dataset was crawled using more than 95,000 seed URLs from 652 unique pay-level domains [52]. Furthermore, the crawling depth is limited to two hops. Because of this difference in the crawling strategy, vertices have different characteristics in both datasets. First, the DyLDO-127M dataset contains nearly 4 times more unique properties and about 11 times more unique types than the TimBL-11M dataset (see Section 8.1.1). Moreover, the TimBL-11M contains fewer data sources and vertices are defined in fewer data sources than in the DyLDO-127M dataset. This could be one possible explanation for the overall better performance on the TimBL-11M dataset. The dataset characteristic also influences the size of the graph summary. On average, the compression ratio of graph summaries computed for the TimBL-11M dataset is 14.9% and for the DyLDO-127M dataset, it is 10.5%. Additionally, vertices in the DyLDO-127M dataset have more variety in the number of outgoing properties, but less variety in the number of types. However, the graph summaries using types (SchemEX, TermPicker, SchemEX+U+I) consistently achieve better compression and summarization ratios on the TimBL-11M dataset. The evaluated graph summaries not using types (Characteristic Sets, Weak Summary, SemSemts) achieve better compression and summarization ratios on the DyLDO-127M dataset. Thus, the complexity of the combination of type sets and properties seems to be predominately impacted by the number of properties rather than the number of types.

Finally, we observe that inferencing RDF Schema information on-the-fly (SchemEX+U+ol) leads to lower F1-scores than inferencing in a pre-processing step (SchemEX+U+pI). For SchemEX+U+ol, the vocabulary graph information used for inferencing is incomplete until the last triple using an RDFS property is processed. Thus, for SchemEX+U+ol inferencing is another source for approximation errors. However, while including the semantics of owl:sameAs and RDFS increases the size of the graph summary, it reduced the number of vertex summaries in some experiments, i.e., it achieves a better summarization ratio.

Our results regarding the RDF Schema inferencing (Table 6.2) indicate a considerable impact on the size of the data graph when using RDF Schema inference. For three out of four datasets, the amount of data increases by a factor of about two to three. A notable exception is here the Laundromat-38B dataset. Despite having a comparably small vocabulary graph (less than 1% of
more than 640 billion additional types can be added. This means, after inference, about 160 times more types appear in the graph. The Laundromat-38B is the only dataset in our analysis that is an aggregation of user-uploaded datasets. In contrast, the other datasets are crawled from the Web using different strategies.

Moreover, the vocabulary graphs \( V_{GRDFS} \) comprise only a small fraction of the data graph, thus, allowing a compact representation of RDF Schema information. In our analysis, we found that for larger datasets, the size decreases compared to the dataset size. Therefore, it appears practical to add inferrable information outside of the structural graph summary for large semantic graphs, i.e., store structural graph summary and vocabulary graph separately. However, Goasdoué et al. [40] also propose an optimization technique to improve the performance of inference, which reduced the time needed to perform inference in graph summaries by up to 94%.

As noted in Section 4.3.5, inferring on the data graph and then summarizing can be equivalent to summarizing the data graph and then inferring on the graph summary [38, 40, 60].

One should also note that the structural graph summary is designed to be orders of magnitude smaller than the original graph. It remains to be evaluated if the graph summaries grow by the same factor as the dataset, when RDF Schema inference is used. In Experiment 1, we evaluated the size of SchemEX and SchemEX+U+I, where SchemEX+U+I extends SchemEX with RDF Schema and owl:sameAs. As one can see in Table 6.1, we have found a notably smaller increase in size than one would expect from Table 6.2. For the TimBL-11M dataset, the structural graph summaries increase on average by a factor of 1.2 (instead of 3.3) and for the DyLDO-127M dataset, the structural graph summaries increase on average by factor of 1.5 (instead of 2.2). Thus, the size of the structural graph summaries does not necessarily grow by the same factor as the data graph grows. We expect that in most cases the increase factor of the data graph will be an upper bound for the increase factor of the semantic structural graph summary.

### 6.6 Summary

Our empirical evaluations reveal huge variations in compression ratio, summarization ratio, and approximation quality for different graph summary models, queries, and datasets. This supports our hypothesis that there is no single graph summary model that equally fits all tasks and that the performance of the summary model depends on the specific types of queries and characteristics of the datasets. However, we observed meaningful correlations in the results that help to determine the right
graph summary model for a given task, type of query, and dataset. Furthermore, our RDF Schema inferencing analysis suggests that adding inferrable information inside the structural graph summary may be infeasible in some cases. In these cases, adding inferrable information should be done outside of the structural graph summary or as a post-processing step. However, the experiments also indicate that the increase factor of the data graph size due to RDF Schema inferencing defines only an upper bound of the increase factor of the semantic structural graph summary.
Incremental and Parallel Graph Summarization

Structural graph summaries can be used to implement various tasks, when they summarize the input graph precisely. When the input graph changes, it is often prohibitively expensive to recompute the structural graph summary from scratch. Thus, an update algorithm is needed (compare problem statement in Section 1.2.2). Existing incremental algorithms have several limitations, e.g., they often assume that changes explicitly detail the added and removed edges of the graph database, i.e., a change log [67, 95]. For Web graphs, accessing remote graphs is costly and subject to network variations [31]. Furthermore, updates on these graphs usually do not provide a change log to the previous version and storing local copies of all graphs is infeasible as Web graphs are too large [78].

In this chapter, we describe our incremental algorithm to compute structural graph summaries and to update them when the input graph evolves over time. Our incremental algorithm can automatically detect changes in the data graph, i.e., it works without a provided change log. This detection
of changes, 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 [86] and implementing the make-set phase following a vertex-centric programming model [62, 85] The vertex-centric programming model is inherently iterative, synchronous, and deterministic [34]. The programming model achieves parallelism very similar to MapReduce [29], while it is specifically aiming for graph data [34]. Distribution-related details are hidden behind an abstract API [62]. As vertex-centric programming model, we rely on Pregel [62].

Moreover, analogously to our sequential algorithm presented in Chapter 5, our incremental algorithm is not designed for a specific graph summary model, but can handle arbitrary models defined in our formal language FLUID. Our parallel incremental algorithm scales through its distributed design. In contrast to the stream-based approach, it does not introduce approximation errors. We proposed our algorithm initially in the peer-reviewed workshop paper [14]. This chapter is based on the further developed peer-reviewed conference article [16].

7.1 Parallel Algorithm

In this section, we describe our parallel incremental algorithm. In the following subsection, we describe the algorithm to compute structural graph summaries in parallel and in a distributed environment, when used on static graphs. Subsequently, we introduce the extension that allows incremental updates of computed graph summaries for evolving graphs.

7.1.1 Outline of the Parallel Algorithm

Our algorithm is inspired by the two-phase approach of Tarjan’s algorithm for the set union problem [86].

Phase 1 (make-set): 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 and is depicted as Phase 1 in Figure 7.1.
Phase 2: Find and Merge

Phase 1: Make-set

Phase 0: Partitioning

Figure 7.1: Phase 0: Distribute parts of the data graph to the computing nodes. Phase 1: incrementally compute vertex summaries $v_s$. Phase 2: merge equivalent vertex summaries, i.e., remove redundant information provided by $r_4$, $r_5$, and $r_6$.

Phase 2 (find and merge): Find the vertex summaries that have the same schema structure and merge them. This refers to the find 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 (see Phase 2 in Figure 7.1).

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 (see Phase 0 in Figure 7.1). Thus, the equivalence defined by simple schema elements (SSEs) can be computed for each vertex independently.

Example Graph Summary: To construct a graph summary, we first define an equivalence relation $\sim$, i.e., we define the schema structure we want to capture. SchemEX defines a schema structure that summarizes vertices that have a common set of labels and a common set of edge labels that link to vertices with again a common set of labels (see example in Chapter 1). Using FLUID, SchemEX is formally defined with two simple schema elements ($\text{OC}_{\text{type}}$) and the identity equivalence $\text{id}$ on properties combined in a complex schema element $\sim\text{SchemEX} := (\text{OC}_{\text{type}}, \text{id}_{\text{rel}}, \text{OC}_{\text{type}})$ (compare Section 4.5. To compute the SchemEX graph summary, we need to extract the information required to check for the SchemEX equivalence relation from each vertex $v$ in the graph database GDB. For SchemEX, this means all vertex label (types) and all edge label (properties). For example, we extract the two labels \{Book, Proceedings\} from a vertex $v$. The complex schema element also requires us to
compare the vertex labels of all neighbors \( w \in \Gamma(v) \). Thus, we have to exchange information between vertices. In the SchemEX case, this is the label sets of vertices \( w \) to all vertices \( v \), where there is an edge \((v, w) \in E \). Finally, we combine this neighbor information with the local vertex information into a vertex summary \( \mathcal{w} \) for all vertices \( v \) in the graph database. Thus, completing the first phase as described above. Note that, when we apply the \( k \)-chaining parameter, we must repeat this step of exchanging and combining information between vertices up to \( k \)-times.

7.1.2 Parallel Algorithm

We support the parallel computation of all graph summary models defined in Chapter 4. This is achieved by using a parameterized implementation of the simple and complex schema elements. The pseudocode of the summarization algorithm is presented in Algorithm 4. In Line 7, the extraction of the schema for each vertex \( v \) in the graph database begins. In parallel, for each \( v \), the local vertex schemas 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 \( \sim \) and \( \sim^o \) equivalence relations (see Lines 8 and 9) 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). In Line 10, each vertex \( v \) receives the schema (according to the object equivalence relation \( \sim^o \)) of all its neighbors. Likewise, Line 12, collects neighbors’ schemas and constructs the data structure defined in Section 4.6. When we use the \( k \)-chaining parameterization, this step of sending and aggregating information is done \( k \) times (Lines 15 to 17) and a vertex accesses the schema information from vertices up to distance \( k \). Line 18 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 \( \mathcal{w} \) and payload vertex \( \mathcal{p} \) are stored in a centralized managed data structure (Line 19), e.g., a graph database, where the find and merge phase is implemented. When the same vertex summary \( \mathcal{w} \) is computed for multiple vertices, it is only stored once and the payload vertices 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 in Line 3. Following our discussion in Section 4.3.5 and our results of Section 6.4, the inference parameterization is a post-processing step.

Note that the `ParallelSummarize` and `FindAndMerge` functions allow a graph summary `SG` to be passed as a parameter. Passing an empty graph summary `SG` corresponds to batch computation; for incremental computation, the previous graph summary is passed.

### Algorithm 4: Parameterized Graph Summarization [16]

```plaintext
function ParallelSummarize(GDB, SG, (∼₁, ∼₉, ∼₉)ₖ)
Input: graph database GDB = (V, E, G, ℓ₆)
Input: graph summary SG
/* SG can be empty */
Input: equivalence relation (∼₁, ∼₉, ∼₉)ₖ
returns graph summary SG
if Instance Parameter S is used then
  forall (v, w) ∈ E do
    if ℓ₆(v, w) ∈ S then
      MergeVertices(v, w);
  forall v ∈ V do in parallel
    vs ← ExtractSimpleVertexSchema(v, E, ∼₁);
    tmp ← ExtractSimpleVertexSchema(v, E, ∼₉);
    /* send information relevant for ∼₉ to incoming neighbors */
    forall w ∈ V : (w, v) ∈ E do
      w.Inbox ← (tmp, 1);
    /* merge information of neighbors and construct complex vertex summaries */
    forall (tmp(vs), r) ∈ v.Inbox do
      t ← ExtractSimpleEdgeSchema((v, w), ∼₉);
      vs.Neighbor₉ ← (t, tmp(vs));
      /* k-chaining repeats send and merge k-times */
      if r < k then
        forall w ∈ V : (w, v) ∈ E do
          w.Inbox ← (vs, r + 1);
      pc ← ExtractPayload(v);
      SG ← FindAndMerge(SG, vs, pc, v);
  return SG;
```

93
7.1.3 Complexity of Parallel Summarization

The complexity of the parallel summarization algorithm is the same as the complexity of the sequential algorithm analyzed in Section 5.2. Again, this algorithm is inspired by the two-phase approach of Tarjan’s algorithm for the set union problem [86]. Phase 1 partitions the set of \( n \) vertices using \( n \) \textit{make-set} operations and Phase 2 uses some number \( m \leq n \) of \textit{find} operations. The summary computation can be done in essentially linear time. As a reminder, the complexity analysis in Section 5.2 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 Algorithm 4. We consider chaining parameterization in detail in Section 7.2.3.

Typical payload functions extract information from a single vertex, e. g., counting or storing the source of a data graph [23]. These functions run in time \( O(1) \) since only a single vertex is needed to extract the payload (see definition of payload in Section 4.4). When we find and merge the vertex summaries, the payload is merged in time \( O(1) \) as well.

7.2 Incremental and Parallel Algorithm

In the previous section, we described and analyzed our graph summarization algorithm. With no further alteration, the algorithm can be used to compute graph summaries defined with FLUID as a batch on static graphs. In this section, we present additions to this algorithm in order to perform incremental updates on the graph summary when the input graph evolves. Our incremental graph summarization algorithm can be still executed in parallel and in a distributed environment.

7.2.1 Outline of the Incremental Algorithm

For the incremental algorithm, we adapt the find and merge 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 \textit{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 (change type ADD-SG), a new vertex is observed with a known schema (change type ADD-PE), a known vertex is observed with a changed schema (change type MOD-SG), a known vertex is observed with changed “payload-relevant information” (change type MOD-PE), a vertex with its schema and payload information no longer exist (change type MOD-PE), and no more vertices with a specific schema structure exist (change type DEL-SG).

Figure 7.2: 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 \( pe \).

To check if vertices have changed, we use an additional data structure called the VertexUpdateHashIndex. The VertexUpdateHashIndex is a three-layered data structure, as depicted in Figure 7.2. 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 \( pe \). 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. However, maintaining data consistency in the VertexUpdateHashIndex is a core problem, which is solved by our algorithm.
7.2.2 Incremental Algorithm

The incremental graph summarization algorithm contains two extensions to the parallel graph summarization algorithm, Algorithm 4. First, we replace the FindAndMerge function with an incremental version called IncrementalFindAndMerge, which is shown in Algorithm 5. This handles additions and modifications. Second, we add a loop to handle deletions from the graph database, i.e., Algorithm 6.

Algorithm 5: Incremental FindAndMerge Algorithm [16]

```plaintext
function IncrementalFindAndMerge(SG, vs, pe, v)
    Input: graph summary SG
    Input: vertex summary vs
    Input: payload element pe
    Input: vertex v
    returns updated graph summary SG

    if VertexHashIndex.ContainsLink(v) then
        /* Modifications are handled as delete + add operation */
        id_prev ← VertexHashIndex.GetLink(v);
        vs_prev ← SG.GetElement(id_prev);
        if vs_prev ≠ vs then
            VertexHashIndex.RemoveLink(v);
            if |VertexHashIndex.GetLinks(vs_prev)| ≤ 0 then
                SG.RemoveElement(vs_prev);
            end if
        end if
    end if
    if not VertexHashIndex.ContainsLink(v) then
        VertexHashIndex.AddLink(v, Hash(vs));
    end if
    if SG.ContainsElement(vs) then
        SG.UpdatePayload(vs, pe);
    else
        SG.AddElement(vs, pe);
    end if
    return SG;
```

Line 3 of Algorithm 5 checks if the vertex v is already in the VertexUpdateHashIndex. If it is, we retrieve its existing vertex summary \( v_{\text{prev}} \) (Lines 4 and 5). If the current vertex summary \( v_s \) differs from \( v_{\text{prev}} \), then v’s schema has changed (MOD-SG) so we delete the link between v and \( v_{\text{prev}} \) in the VertexUpdateHashIndex (Line 7). If this was the last link for \( v_{\text{prev}} \), then \( v_{\text{prev}} \) 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. This reflects the case that the payload has changed (MOD-PE), e.g., the source graph label has changed. 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). The pseudocode for deletions is presented in Algorithm 6. All vertices that are no longer in the GDB are deleted from the VertexUpdateHashIndex (Algorithm 6 Line 1). 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). If a vertex summary no longer summarizes vertices in the GDB, the vertex summary is deleted (Algorithm 6 Line 5). Note that deleting a vertex summary means deleting at least the primary vertex and its edges. The secondary vertices may still be part of other vertex summaries as well.

**Algorithm 6:** Delete all vertices not in the current version of the GDB.

```plaintext
/* Assumption: all vertices not in the current GDB got deleted */
/* Incremental updates on $SG$ for deletions. */
1 forall $v \in SG$.GetSummarizedVertices(): $v \notin V$ do
2     $vs \leftarrow SG$.GetLink($v$);
3     SG.RemoveLink($v$);
4     if $|SG$.GetLinks($vs$)| $\leq$ 0 then
5         /* Delete vertex summaries that no longer summarize GDB vertices. */
6             SG.RemoveElement($vs$);
```

We now prove correctness of our incremental algorithm. Fix a summary model $\sim$. We use Algorithm 4 with the incremental find and merge implementation of Algorithm 5 and the deletion routine of Algorithm 6 to define the function $\text{Incr}(G, SG) = \text{ParallelSummarize}(G, SG, \sim)$, corresponding to incremental summarization of the graph $G$ given an existing summary $SG$. We also define the function $\text{Batch}(G) = \text{ParallelSummarize}(G, \emptyset, \sim)$, corresponding to batch computation of a summary of $G$ without a pre-existing summary. Our intended application of the following theorem is that $GDB_1$ and $GDB_2$ are snapshots of the same database at different times but the proof is applicable for arbitrary GDBs.

**Theorem 2.** Fix a graph summary model $\sim$. Let $V$ be a set of vertices and let $GDB_1$ and $GDB_2$ be any two GDBs. Then $\text{Incr}(GDB_2, \text{Batch}(GDB_1)) = \text{Batch}(GDB_2)$. 97
Proof. For $i \in \{1, 2\}$, let $GDB_i = (V_i, E_i, G_i, \ell_{G_i})$. Let $BSG_1 = \text{Batch}(GDB_1)$, $BSG_2 = \text{Batch}(GDB_2)$ and $ISG = \text{Incr}(GDB_2, BSG_1)$. Note that $BSG_1$ contains summaries for exactly all the vertices of $GDB_1$, i.e., exactly all the vertices in $V_1$.

We must show that $ISG = BSG_2$. To do this, we consider separately the vertices in $V_2 \setminus V_1$ (new vertices, via ADD-SG and ADD-PE), vertices in $V_2 \cap V_1$ (unchanged vertices and those modified via MOD-SG and MOD-PE) and vertices in $V_1 \setminus V_2$ (those deleted via DEL-SG and DEL-PE). We describe how the computations $\text{Incr}(GDB_2, BSG_1)$ and $\text{Batch}(GDB_2)$ process vertices in these three classes.

Consider a vertex $v \in V_2 \setminus V_1$. Since $v \notin V_1$, $v$ is not in the VertexUpdateHashIndex when we begin to compute $\text{Incr}(GDB_2, BSG_1)$. We add it and its summary $vs$ to the VertexUpdateHashIndex (Line 11). In the case of ADD-PE, $vs$ is already contained in $BSG_1$ so we just update its payload in $ISG$ (Line 13); for ADD-SG, $vs$ is not in $BSG_1$ so we add it to $ISG$ (Line 15). When computing $\text{Batch}(GDB_2)$, we compute the same summary $vs$. If $v$ is the first vertex we have seen with this summary, we add $vs$ to $BSG_2$ with the appropriate payload and link $v$ to it in the VertexUpdateHashIndex; otherwise, $vs$ is already in $BSG_2$ and we update its payload and link $v$ to it.

Now, consider a vertex $v \in V_1 \cap V_2$. When we begin to compute $\text{Incr}(GDB_2, BSG_1)$, $v$ is summarized in $BSG_1$ so it is contained in the VertexUpdateHashIndex. We retrieve its existing vertex summary $vs_{prev}$ (Lines 4 and 5) and compare this to the new vertex summary $vs$. If the vertex summary has changed (MOD-SG), we disconnect $v$ from $vs_{prev}$ in the VertexUpdateHashIndex, delete $vs_{prev}$ from $ISG$ if it no longer summarizes any vertices and then link $v$ to $vs$ (Lines 6–11). When we compute $\text{Batch}(GDB_2)$, $v$ is processed in the same way as in the previous case: if the summary $vs$ is already present in $BSG_2$, its payload is updated; if $vs$ is not already present, it is added.

Finally, consider a vertex $v \in V_1 \setminus V_2$. Again, $v$ is summarized in $BSG_1$ so it is contained in the VertexUpdateHashIndex when Algorithm 4 begins. $v$ is not processed by Algorithm 4, because $v \notin V_2$, so it remains in the VertexUpdateHashIndex when Algorithm 4 completes. However, we then run Algorithm 6. This removes the link in the VertexUpdateHashIndex from $v$ to its summary $vs$ (DEL-PE) and, if $vs$ no longer summarizes any vertices, we delete $vs$ from $ISG$, too (DEL-SG). When computing $\text{Batch}(GDB_2)$, $v$ is not processed because it is not in $V_2$. Therefore, $v$ is not summarized in $BSG_2$.

Thus, we have constructed summary graphs $ISG$ and $BSG_2$. Every vertex in $V_2$ has the same summary in $ISG$ as it does in $BSG_2$ and every vertex not in $V_2$ is not summarized in either $ISG$ or $BSG_2$. Therefore, $ISG = BSG_2$, as claimed. \qed
The following corollary shows that, if we incrementally compute a sequence of summaries over an evolving graph, the resulting summaries are the ones we would obtain by just running the batch algorithm at each version of the evolving graph.

**Corollary 1.** Let $V$ be a set of vertices and let $GDB_1, GDB_2, \ldots$ be GDBs. Let $SG_1 = \text{Batch}(GDB_1)$ and, for all $k > 1$, let $SG_k = \text{Incr}(GDB_k, SG_{k-1})$. Then, for all $k \geq 1$, $SG_k = \text{Batch}(GDB_k)$.

**Proof.** The case $k = 1$ holds by hypothesis. Suppose that $SG_k = \text{Batch}(GDB_k)$. Then,

$$SG_{k+1} = \text{Incr}(GDB_{k+1}, SG_k)$$

$$= \text{Incr}(GDB_{k+1}, \text{Batch}(GDB_k))$$

$$= \text{Batch}(GDB_{k+1}),$$

where the first equality is defined in the corollary’s statement, the second is by the inductive hypothesis and the third is by Theorem 2.

---

**7.2.3 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. This analysis was originally published in [16].

**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 $SG$ with $d$ as the maximum degree in the data graph and the chaining parameter $k$. As discussed in Section 4.6, 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 \( v' \) at time \( t \), but already summarized \( v \) with a different vertex summary \( v_s \) at time \( t-1 \), we have to modify the graph summary \( SG \). Transforming a vertex summary \( v_s \) to another vertex summary \( v'_s \) means in the worst case deleting all vertices and edges in \( v_s \) and adding all vertices and edges in \( v'_s \). This occurs when the schema of \( v \) has entirely changed from \( t-1 \) to \( t \). Thus, modifications to \( v'_s \) 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, \( v'_s \) 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 \( v_s \) 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 \( v_s \) 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 Section 4.6). Thus, there are as many vertex summaries \( v_s \) in \( SG \) that may need to be updated as there are partitions in the GDB. For each \([v]_\sim\), there is exactly one entry \( \text{hash}(v_s) \) stored in the L1 layer of the VertexUpdateHashIndex. For each vertex \( v \) in the GDB, there is a \( \text{hash}(v) \) 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 \( v_s \), but require up to two updates on the VertexUpdateHashIndex. We observe a new vertex \( v \) that is summarized by an existing vertex summary \( v_s \subseteq SG \) (ADD-PE). Thus, no updates on \( SG \) are needed (the vertex’s schema is already “known”) and only a single operation on the VertexUpdateHashIndex is required to add \( v \) to L2 (and connect that vertex to L1). A vertex \( v \) which is summarized by a vertex summary \( v_s \) is deleted from the GDB but there are other vertices \( v' \) in the GDB that are summarized by the same \( v_s \) (DEL-PE). In this case, no update on \( SG \) is required and one operation on the VertexUpdateHashIndex is
required to delete $hash(v)$ from $L_2$. The vertex $v$ is observed at time $t$ with the vertex summary $vs$ and same vertex is already summarized by same vertex summary $vs$ at the previous time $t - 1$ (MOD-PE). Thus, no update on $SG$ no update on the VertexUpdateHashIndex are required.

**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 $L_3$ 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 $\Delta$ 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 radius 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.

### 7.3 Summary

We presented a parallel and distributed graph summarization algorithm for structural graph summaries defined with FLUID. The algorithm can compute structural graph summaries for static graphs and scales without approximation errors through its vertex-centric design. Moreover, using our proposed VertexUpdateHashIndex, the algorithm supports incrementally updating structural graph summaries when the data graphs evolves over time. Using this algorithm, all graph summaries defined with FLUID can be updated in time $O(\Delta \cdot d^k)$, where $\Delta$ is the number of additions, deletions, and modifications to the input graph, $d$ is its maximum degree, and $k$ is the maximum radius in the subgraphs considered.
Empirical Evaluation of the Incremental Graph Summarization

In this chapter, we empirically evaluate the time and space requirements of the graph summarization algorithms presented in Chapter 7 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. We run three sets of experiments. First, for each version of the graph database, we analyze the costs of computing a new graph summary from scratch (batch-based computation) compared to incrementally updating an existing graph summary from a previous version. Second, we evaluate the impact of the parallelization on the overall performance. Third, we evaluate the memory consumption of our algorithm. The results of the first set of experiments are published in a peer-reviewed conference article [16].
8.1 General Experimental Apparatus

All experiments are conducted using the same datasets and the same graph summary models. We first describe the datasets and subsequently the graph summary models. The information specific to each experiment is described in the respected sections.

8.1.1 Datasets

For this experiment, we need data sets that range across multiple versions. 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, which renders the data search useless. We use the real-world datasets to implement the data search task since they contain multiple source graphs.

_LUBM100:_ The Lehigh University Benchmark (LUBM) generates benchmark datasets containing people working at universities [44]. 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 11 M edges. Over all versions, the mean degree is 5.1 (±0.1).

_BSBM:_ The Berlin SPARQL Benchmark (BSBM) is a suite of benchmarks built around an e-commerce use case [12]. 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 mean degree is 10.9 (±0.2).

_DyLDO:_ The Dynamic Linked Data Observatory (DyLDO) provides regular crawls of the Web of Data [52]. The crawls started from about 95,000 representative seed URLs (source label of graphs). There are two variants of this dataset: The dataset containing only the graphs identified by the seed
URLs is referred to as **DyLDO-core**. It contains the 95,000 different graphs obtained from the seed URLs, stored in the multi-set $G$ of the GDB. The extended crawl (including the core) is referred to as **DyLDO-ext**. Starting from the seed URLs, 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. Over all weekly crawls, the mean degree is $4.8 \pm 0.5$. 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, which contain 7–10 M vertices and 84–106 M edges. The mean degree is $10.8 \pm 0.7$.

### 8.1.2 Structural Graph Summaries

Based on our discussion in Section 3.1, we choose three summary models for our experiments, namely SchemEX [58], Attribute Collection [21], and Class Collection [21]. These three summary models are representatives of the most common structural features, i.e., types and properties. Class Collection summarizes vertices that share the same label. They are defined as the label parameterized object cluster $OC_{type}$ (see Table 4.2). Attribute Collection summarizes vertices that share the same labels of outgoing edges. They are defined as property cluster $PC_{rel}$ (see Table 4.2). SchemEX is a combination of Class and Attribute Collection, i.e., two vertices have the same label, have edges with the same label, and neighbors with the same label. This is defined as CSE ($OC_{type}$, $id_{rel}$, $OC_{type}$) (see Table 4.2).

### 8.1.3 Implementation Details and Test System

We implement our algorithm in Scala 2.12.8 and Java 11.0.5 using Apache Spark GraphX 2.4.3 [5, 41]. We use a single Spark context on a dedicated server using at most 20 cores and 200 GB heap space. The datasets are read in parallel as RDF graphs in gzipped n-triple files and mapped to GraphX’s internal label property graph data structure using Algorithm 1 from Chapter 2. The mapped graph is partitioned using random vertex cut. The number of partitions is equal to the number of available cores. This concludes the data preparation stages.

Phase 1 of our algorithm (Make-set phase) is implemented using GraphX’s neighborhood aggregating feature in combination with the Pregel API [5, 62]. Thus, after Phase 1, each vertex $v$ in the data graph has its vertex summary stored as attribute. For Phase 2, we write the vertex summaries to
the OrientDB 3.0.25 graph database [69]. The database maintains unique hash indices to quickly access the vertex summaries. For each partition of the graph, a thread pool is initialized that allows to write the vertex summaries to the database in parallel. To avoid unnecessary synchronizations, we used non-transactional communications. This means, we used an optimistic approach where we assume that most of the times there will be no update conflict. If a vertex summary already exists in the graph database, only the payload is updated. In case of an update conflict that cannot be resolved by OrientDB’s graph repair feature*, we roll back the failed transaction and retry it until it is successful with a small random delay between 0 and 10ms. Note, due to Spark’s parallelization optimization, computation steps are not strictly ordered, i.e., where possible, following “stages” are executed while the previous ones are still running.

The VertexHashIndex is implemented as Java Object using HashMaps and is persisted as serialized, gzipped object. To minimize synchronization overhead, we manage the access using read and write access restrictions and manage them separately for layer L1 and L2. To increase efficiency, we implemented L3 as part of L2. While L2 and L3 are logically different layers, accessing L2 most often means accessing L3 afterwards. Thus, we implemented them together.

We run OrientDB in a Docker container with at most 20 GB heap space and 100 GB memory mapped files. All execution times are measured using the built in logging feature of Apache Spark. Our test system is a server equipped with 2 × Intel Xeon CPU E5-2690 v2 at 3.00GHz with 10 cores and 20 threads each. Furthermore, each CPU has access to 12 × 16GB Samsung DDR3 memory at 1600MHz, resulting in a total of 384GB main memory.

8.2 Experiment 1: Update Performance

In this experiment, we are interested a direct comparison of performance between our batch algorithm and our incremental algorithm. This experiment was originally published in [16].

8.2.1 Procedure

We compute the structural graph summary for each version of the GDB. The batch algorithm recomputes the graph summary from scratch for each GDB version. The incremental algorithm detects the changes in the GDB and only performs the necessary updates on the graph summary. We measure the performance of the algorithms using multiple metrics.

*Graph-Consistency in OrientDB: https://orientdb.org/docs/3.0.x/java/Graph-Consistency.html
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 summarization algorithms. Regarding size, we first count the number of vertices $|V|$ and edges $|E|$ in GDB and the number of vertices $|V_v|$ and edges $|E_v|$ 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.

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 schemas 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 (see Section 7.2.3). 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.

8.2.2 Results

The experimental results are visualized in Figures 8.1 and 8.2. Each plot shows on the horizontal axis the database versions over time and on the vertical axis the metrics. From top to bottom, the rows of Figures 8.1 and 8.2 show the metrics size, update complexity, and performance. Figure 8.1 shows the results on the benchmark datasets, i.e., LUBM100 (left) and BSBM (right), and Figure 8.2 shows the results on the real-world datasets, i.e., DyLDO-core (left) and DyLDO-ext (right). Unless specified otherwise, we refer in the text to mean and standard deviation values computed over all version of a GDB. The sub figures in the top rows of Figures 8.1 and 8.2 show the size of the GDB and the size of the graph summary $SG$ for each version of the database. The summarization ratio is the quotient of the number of vertices in the GDB and the number of vertices in the graph summary. Intuitively, the distance between the colored dotted lines and the grey dotted line indicates the summarization ratio, where an intersection of both lines indicates a summarization ratio of 1.0. Note that, for Attribute Collections (AttrColl) and Class Collections (ClassColl), the number of summary vertices $|V_v|$ is identical to the number of vertex summaries since all summary vertices are primary vertices. For SchemEX, secondary vertices are needed to model the schema structure.
Figure 8.1: Results on the synthetic benchmark datasets for SchemEX (blue), Attribute Collection (green), and Class Collection (red). Each column is a different dataset and each row a different metric.
Figure 8.2: Results on the real-world datasets for SchemEX (blue), Attribute Collection (green), and Class Collection (red). Each column is a different dataset and each row a different metric.
However, over all experiments and datasets, these secondary vertices make up only about 4% of all vertices. Thus, we do not display primary vertices separately in Figures 8.1 and 8.2. The sub figures in the center rows show the vertex change percentage and the graph summary update percentage. The sub figures on the bottom rows show the average execution times of three consecutive repetitions of the graph summary computation.

Comparing the size of the graph summaries ($|V_{ci}|$, $|E_{ci}|$) and the graph database ($|V|$, $|E|$) (Figures 8.1 and 8.2 top rows), the graph summaries are orders of magnitude smaller (consistently over all experiments). Also, the Class Collections (abbreviated ClassColl) 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 $10^4$ fewer vertices, followed by Class Collections with a factor of $10^3$ fewer vertices. SchemEX has the lowest summarization ratio, i.e., the greatest number of vertex summaries are needed to partition the vertices. Still, SchemEX has in the order of a factor of $10^2$ fewer vertices than the GDB. For Class Collections, no edges are needed to represent the vertex summaries. Thus, for Class Collections, in each version the number of edges in the graph summary $|E_{ci}|$ is zero, which further reduces the overall size of the graph summary. Comparing the changes in GDB and the updates on SG (Figures 8.1 and 8.2 middle rows), we see that over all datasets, the number of updates on the graph summary is orders of magnitude smaller than the number of changes in the graph database. On average, there are $12,967 (\pm 18,583)$ more changes than updates for the Class Collections, $19,721 (\pm 30,157)$ more for the Attribute Collection, and $901 (\pm 1,737)$ more for SchemEX.

Furthermore, the incremental algorithm usually computes Class Collections the fastest with an average of $27 (\pm 33)$ minutes and SchemEX the slowest with an average of $42 (\pm 95)$ minutes (Figures 8.1 and 8.2 bottom rows). In relation to the batch counterpart, these numbers are a relative speed up of $1.8 (\pm 0.7)$ for SchemEX, $1.8 (\pm 0.9)$ for Attribute Collection, and $3.7 (\pm 2.7)$ for Class Collection.

### 8.2.3 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 ($10 \times \text{LUBM100}, 40 \times \text{BSBM}, 49 \times \text{DyLDO-core}, 5 \times \text{DyLDO-ext}$). 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. A detailed
evaluation of the performance metrics shows a strong monotonic correlation between the schema computation (Phase 1) and the number of edges in the graph database, \( r_s(311) = 0.867, p < .0001. \)

This substantiates our theoretical complexity analysis. As described in Section 7.2.2, 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.

If many updates are required, they seem to be more expensive on the BSBM dataset than on the DyLDO-core dataset since the batch algorithm outperforms the incremental algorithms during periods of high amount of updates (percentage) only on the BSBM dataset. The average vertex degree is more than two times higher on the BSBM dataset. Since the update complexity depends on the degree of the vertex, this could be a reasonable explanation for this.

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. All three graph summary models produce 10 – 100 times larger graph summaries on the DyLDO-core dataset than on similarly sized BSBM dataset version 20. 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\% (±15.64\%).

Compared to the other experiments, the performances of the batch computations of Class Collections on the two DyLDO datasets have an exceptional curve. The batch computation takes twice as long for the Class Collection on the DyLDO-core than for Attribute Collection and SchemEX. We noticed this in several runs and on a different test system. Further investigation revealed that the

---

1Since our experimental data does not follow a normal distribution but a skewed distribution (D’Agostino’s K-squared test for edges: \( K^2 = 315.3483, p < 0.0001 \)), we calculate the Spearman rank-order correlation coefficient.
cause for this salient observation is rather simple: There are a few “hot” vertex summaries that sum-
marize most of the vertices when using Class Collection. In particular, the specific vertex summary that
summarizes all vertices in the GDB that have no type information stands out. These are those vertices
that have an empty label set $\ell_V$. During the find and merge phase, the payload information of different
vertices that are summarized by the same vertex summary $v$ is merged (ADD-PE). Although merging
payload information in our experiment is done in constant time (merging sets of source graph label),
merging basically all payload information into few centrally stored payload elements cannot be done
in parallel. This means, merging the payload of the Class Collection on the DyLDO-core dataset
requires an exceptional number of synchronizations. For example, the vertex summary that summa-
rizes vertices with no type information in the Class Collection needs on average $12,828 (\pm 4,148)$
synchronizations for each version update. In contrast, for Attribute Collection and SchemEX, on
average $122 (\pm 81)$ synchronizations are needed for all vertex summaries.

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 $L_3$ is needed. One might consider this as a design flaw in the batch algorithm, which deter-
mines 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 excep-
tions, outperforms the batch counterpart.

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 hetero-
genity, 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 [23]. 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 (see Table 3.1). Thus, the results capture the performance of our graph summarization algo-
rithm for a wide, representative range of experimental settings. Beyond the three representative graph
summary models used in the evaluation, our incremental algorithm is suitable for other summary
models reported in the literature, which is possible due to our FLUID language [17].
8.3 Experiment 2: Parallelization Performance

In this experiment, we are interested in how much performance our algorithms gain through parallelization.

8.3.1 Procedure

We systematically analyze the impact of the number of cores available for the graph summary computation. To this end, we repeat Experiment 1, but we limit the number of available cores for the computation to 1, 2, 4, 8, and 16 cores. Furthermore, we focus on two datasets, namely the synthetic BSBM dataset and the real-world DyLDO-core dataset. We designed the evolution of the BSBM benchmark dataset in such a way that the size grows linearly for the first 20 version and shrinks linearly for the remaining 20 version. Thus, we can easily see any non-linear performance changes. As a representative real-world dataset, we use the DyLDO-core dataset. As shown in Experiment 1, real-world datasets have different characteristics compared to the benchmark datasets. Compared to DyLDO-full, DyLDO-core has still a considerably small size, which allows computing all graph summaries even with few cores in reasonable times. Analogously to Experiment 1, the runtime performance of the algorithm 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. Note, the batch computations require significantly more time when running on few cores.

For this experiment, we focus on the real-word DyLDO-core dataset. We observed in Experiment 1 that the graph summary computation on real-world datasets is considerably more complex than on synthetically generated datasets, which leads to significant differences between the evaluated graph summary models. From the results of Experiment 1, we know that overall, the performance of Attribute Collection, Class Collection, and SchemEX differs only marginally on the BSBM dataset. Given these two observations from Experiment 1, we can reasonably assume that for our three graph summary models, the impact of using a different number of cores will also differ only marginally. Thus, for the BSBM dataset, we only report the evaluation results of Attribute Collection.

To compare the performance results over multiple versions of a dataset that evolves over time, we aggregate the speed-up for each version. Inspired by macro and micro F1-scores, we report the macro and micro average speed-ups in Table 8.1. For the macro average speed-up, we compute the speed-up for each version and then calculate the mean and the standard deviation. For the micro average speed-up, we sum-up the execution times for each version and then calculate the mean speed-up.
8.3.2 Results

Figure 8.3: Performance of the batch computation (left) and the incremental computation (right) of Attribute Collection on the BSBM dataset. The horizontal axis shows the GDB version over time, the vertical axis the duration of the summary computation, and the depth axis the number of available cores. The three contour lines show the run with the longest duration (red), the shortest duration (blue), and the median duration (green).

The main results of Experiment 2 are shown in Figures 8.3 and 8.4. Figure 8.3 shows the results of Attribute Collection on the BSBM dataset and Figure 8.4 shows the results of Attribute Collection, Class Collection, and SchemEX on the DyLDO-core dataset. The subfigures on the left show the performance of the batch computation and the subfigures on the right show the performance of the incremental computation.

For both datasets, our main observation is that increasing the number of available cores improves the performance of the batch computation and the incremental computation. Remarkable, however, is that the single core runs on the real-world DyLDO-core dataset of the incremental computation are competitive to the 16 core runs of the batch computation, often even outperforming the batch computation. The Attribute Collections are computed incrementally between 1.2 and 2.4 (mean: 1.8 ± 0.3) times faster using only one core than the batch computation with 16 cores on the DyLDO-core dataset. For SchemEX, the incremental computation is between 0.5 and 2.0 (mean: 1.6 ± 0.3) times faster. Notably for SchemEX, only two single core incremental computations were slower than the 16 core batch computations, i.e., version 46 (speed-up of 0.5) and 47 (speed-up of 0.6).
Figure 8.4: Performance of the batch computation (left) and the incremental computation (right) on the DyLDO-core dataset. The horizontal axis shows the GDB version over time, the vertical axis the duration of the summary computation, and the depth axis the number of available cores. The three contour lines show the run with the longest duration (red), the shortest duration (blue), and the median duration (green).
Table 8.1: The macro and micro average speed-ups comparing the batch computation and the incremental computation for different numbers of cores. The incremental computation outperforms the batch computation with a speed-up value greater than 1.

<table>
<thead>
<tr>
<th>Model</th>
<th>Metric</th>
<th>1 core</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
<th>16 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSBM</td>
<td>Macro Avg.</td>
<td>1.44 ± 0.05</td>
<td>1.83 ± 0.31</td>
<td>1.46 ± 0.08</td>
<td>1.16 ± 0.05</td>
<td>1.07 ± 0.04</td>
</tr>
<tr>
<td></td>
<td>Micro Avg.</td>
<td>1.44</td>
<td>1.86</td>
<td>1.48</td>
<td>1.17</td>
<td>1.07</td>
</tr>
<tr>
<td>DyLDO-core</td>
<td>Macro Avg.</td>
<td>9.35 ± 1.81</td>
<td>13.48 ± 3.21</td>
<td>7.62 ± 1.33</td>
<td>4.02 ± 0.56</td>
<td>3.71 ± 0.66</td>
</tr>
<tr>
<td></td>
<td>Micro Avg.</td>
<td>8.98</td>
<td>12.44</td>
<td>7.38</td>
<td>3.89</td>
<td>3.48</td>
</tr>
<tr>
<td>ClassColl</td>
<td>Macro Avg.</td>
<td>64.52 ± 7.7</td>
<td>77.82 ± 20.29</td>
<td>44.08 ± 4.45</td>
<td>21.48 ± 2.23</td>
<td>14.06 ± 1.4</td>
</tr>
<tr>
<td></td>
<td>Micro Avg.</td>
<td>65.23</td>
<td>77.87</td>
<td>44.32</td>
<td>21.48</td>
<td>14.16</td>
</tr>
<tr>
<td>SchemEX</td>
<td>Macro Avg.</td>
<td>6.26 ± 1.23</td>
<td>8.43 ± 2.14</td>
<td>5.49 ± 1.01</td>
<td>3.07 ± 0.47</td>
<td>3.26 ± 0.68</td>
</tr>
<tr>
<td></td>
<td>Micro Avg.</td>
<td>5.71</td>
<td>7.55</td>
<td>5.07</td>
<td>2.92</td>
<td>3.00</td>
</tr>
</tbody>
</table>

discussed in Experiment 1, here about 50% of the data graph changed. For Class Collections, the speed-up is even higher than for Attribute Collection. Class Collections are computed incrementally between 4.9 and 6.9 (mean: 5.7 ± 0.5) times faster using only one core than the batch computation with 16 cores on the DyLDO-core dataset.

For the BSBM benchmark dataset, the four core batch runs outperform the single-core incremental runs. Beyond four cores, the incremental computation are within a margin of error of 30 seconds as fast as the batch runs.

Furthermore, we observe that in general, adding more cores benefits the batch computation more than the incremental computation. On the BSBM dataset version 20, i.e., the largest version, the single core batch computation of Attribute Collection takes 57 minutes and the 16 core batch computation takes 22 minutes. This is a speed-up of about 2.6. For the same BSBM dataset version, the incremental computation time for Attribute Collection drops from 40 minutes (single core) to 20 minutes (16 cores). This is a speed-up of 2.0.

The difference between batch computation and incremental computation is more apparent on the DyLDO-core dataset. Here, the single core batch computations of Attribute Collection take 157 to 338 minutes (mean: 237 ± 60) and the 16 core batch computations only take 37 to 51 minutes (mean: 44 ± 4). This means, the speed-up gained for the batch computation of Attribute Collection from
Table 8.2: The mean of the relative speed-up gains when doubling the number of cores used for the batch computation and the incremental computation. Ideal speed-up of 2.0 means that doubling the number of cores reduces the computation time to a half.

<table>
<thead>
<tr>
<th>Model</th>
<th>Model</th>
<th>Algorithm</th>
<th>1 → 2 cores</th>
<th>2 → 4 cores</th>
<th>4 → 8 cores</th>
<th>8 → 16 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>111 → 222</td>
<td>222 → 444</td>
<td>444 → 888</td>
<td>888 → 1666</td>
</tr>
<tr>
<td>BSBM</td>
<td>AttrColl</td>
<td>batch</td>
<td>1.25 ± 0.20</td>
<td>1.75 ± 0.27</td>
<td>1.24 ± 0.05</td>
<td>0.98 ± 0.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>incremental</td>
<td>1.55 ± 0.09</td>
<td>1.40 ± 0.06</td>
<td>0.99 ± 0.02</td>
<td>0.91 ± 0.03</td>
</tr>
<tr>
<td>DyLDO-core</td>
<td>AttrColl</td>
<td>batch</td>
<td>1.00 ± 0.12</td>
<td>2.60 ± 0.26</td>
<td>1.90 ± 0.11</td>
<td>1.09 ± 0.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>incremental</td>
<td>1.42 ± 0.10</td>
<td>1.50 ± 0.17</td>
<td>1.01 ± 0.04</td>
<td>0.99 ± 0.04</td>
</tr>
<tr>
<td>ClassColl</td>
<td>batch</td>
<td>1.36 ± 0.40</td>
<td>2.55 ± 0.64</td>
<td>2.15 ± 0.37</td>
<td>1.64 ± 0.13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>incremental</td>
<td>1.54 ± 0.08</td>
<td>1.44 ± 0.09</td>
<td>1.03 ± 0.02</td>
<td>1.07 ± 0.03</td>
<td></td>
</tr>
<tr>
<td>SchemEX</td>
<td>batch</td>
<td>1.02 ± 0.20</td>
<td>2.40 ± 0.34</td>
<td>1.78 ± 0.09</td>
<td>0.94 ± 0.08</td>
<td></td>
</tr>
<tr>
<td></td>
<td>incremental</td>
<td>1.33 ± 0.10</td>
<td>1.57 ± 0.17</td>
<td>1.01 ± 0.08</td>
<td>0.99 ± 0.05</td>
<td></td>
</tr>
</tbody>
</table>

1 to 16 cores is between 3.8 and 7.4 (mean: 5.4 ± 1.2). For SchemEX, we observe a similar speed-up behavior. The single core batch computation of SchemEX takes 131 to 215 minutes (mean: 167 ± 27) and the 16 core batch computations only take 35 to 49 minutes (mean: 42 ± 3). For the extreme case of the Class Collection, the single core batch computation takes 879 to 2430 minutes (mean: 1463 ± 398) and the 16 core takes 84 to 199 minutes (mean: 129 ± 30). This leads to a speed-up for the batch computation of Class Collection from 1 to 16 cores between 10.4 and 12.7 (mean: 11.2 ± 0.6).

In contrast, excluding the first initial summary computation, the single core incremental computations of all three summary models take 17 to 90 minutes (mean: 25 ± 7) and the 16 core incremental computations take 7 to 33 minutes (mean: 11 ± 3). The computation of the first graph summary is essentially a batch computation since no prior graph summary can be exploited. For the incremental computation, the speed-up from one core to 16 cores is between 1.9 and 2.7 (mean: 2.2 ± 0.1). In general, incrementally, SchemEX is computed fastest and Attribute Collections slowest on the DyLDO-core dataset.

The mean speed-up for each increase in number of cores is presented in Table 8.2. On both datasets, we observe that the performance of the incremental computation greatly improves when adding up to 4 cores. The performance of the incremental computation increases only marginally or even decreases when adding more than 4 cores. For the batch computation, we still notice large
improvements of the performance when adding up to 8 cores, in particular on the real-world DyLDO-core dataset. Class Collection is the only evaluated summary model that scales beyond 8 cores. On the real-world dataset DyLDO-core, over all analyzed numbers of cores, we have an average speed-up of incremental algorithm over the batch algorithm for Attribute Collection between 5.5 and 13.3 (mean: 9.5 ± 1.6), for Class Collection between 51.5 and 85.9 (mean: 64.3 ± 7.7), and for SchemEX between 2.0 and 7.4 (mean: 6.3 ± 1.1). Overall, on the real-world dataset DyLDO-core, over all analyzed numbers of cores, we have a macro average speed-up between 3.0 and 77.8 (mean: 19 ± 3.3) for the incremental algorithm.

8.3.3 Discussion

The key insight from the experiment on the parallelization of our algorithm is that the incremental computation outperforms the batch computation for 1, 2, 4, 8, and 16 cores. In addition, on the DyLDO-core dataset, the 16 core batch computation is slower than the single core incremental computation.\(^2\) This is remarkable since we also observe that adding more cores benefits the batch computation more than the incremental computation. To explain this, we evaluate the runtime of all phases during the computation. We find that the find and merge phase takes a significant portion of runtime (see Table 8.3). The incremental computation gains its performance boost by reducing the number of find and merge operations. As described in our experimental apparatus (Section 8.1), the find and merge phase is done in the graph database containing the graph summary. To write and update the graph summary, we use non-transactional, optimistic operations. This reduces the synchronization overhead and allows fast parallelization. The batch computation outputs the full graph summary for each version of the input graph database. This explains two observations: First, having multiple cores available greatly benefits the parallel access to the graph summary, i.e., reduces the time required to find and merge vertex summaries in the graph summary. Second, since the incremental computation avoids unnecessary find and merge operations, a reduction for the find and merge phase runtime has a lower impact on the overall runtime. It remains to be evaluated if different database implementations can significantly boost the overall runtime. However, database access is required for both, the incremental algorithm and the batch algorithm. Thus, a faster database implementation will have a positive impact on both algorithms as well. Furthermore, the incremental algorithm avoids unnecessary database access and avoiding the database at all is faster than any database access.

\(^2\) except SchemEX for the two DyLDO-core versions 46 and 47
Table 8.3: The percentage of the overall runtime taken by the find and merge phase of the batch computation and the incremental computation.

<table>
<thead>
<tr>
<th>Model</th>
<th>Algorithm</th>
<th>Find and Merge Phase for Number of Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>BSBM</td>
<td>batch</td>
<td>36% ± 1%</td>
</tr>
<tr>
<td></td>
<td>incremental</td>
<td>7% ± 1%</td>
</tr>
<tr>
<td>AttrColl</td>
<td>batch</td>
<td>36% ± 1%</td>
</tr>
<tr>
<td></td>
<td>incremental</td>
<td>7% ± 1%</td>
</tr>
<tr>
<td>DynDO-core</td>
<td>batch</td>
<td>91% ± 1%</td>
</tr>
<tr>
<td></td>
<td>incremental</td>
<td>17% ± 12%</td>
</tr>
<tr>
<td>ClassColl</td>
<td>batch</td>
<td>99% ± 1%</td>
</tr>
<tr>
<td></td>
<td>incremental</td>
<td>6% ± 1%</td>
</tr>
<tr>
<td>SchemEX</td>
<td>batch</td>
<td>88% ± 1%</td>
</tr>
<tr>
<td></td>
<td>incremental</td>
<td>25% ± 15%</td>
</tr>
</tbody>
</table>

The make set phase is the same for the incremental and the batch computation. The parallelization of this step is done by Apache Spark. Apache Spark also de-serializes the gzipped input files. Based on existing studies of graph processing frameworks, we assume that for larger datasets and more complex graph summaries, e.g., using the $k$-chaining parameterization, the multi-core performance scales beyond 4 cores [2]. Apache Spark is a state-of-the-art processing framework [3]. However, optimizing Apache Spark is out of the scope of this thesis. In this experiment, we focus on the parallelization of our proposed algorithms.

8.4 Experiment 3: Memory Consumption

In this experiment, we are interested in the memory consumption of our algorithms and the memory overhead induced by the VertexUpdateHashIndex used by the incremental algorithm.

8.4.1 Procedure

To analyze the memory consumption of our proposed algorithm, we log the memory consumption of Apache Spark. More precisely, we enable Spark’s metrics log to estimate the size of the GraphX RDDs
Figure 8.5: Results of the experimental evaluation of memory consumption on the BSBM dataset (left) and the DyLDO-core dataset (right). The dash-dotted grey lines depict the raw dataset size as RDF n-triple text file. The black lines depict the maximum allocated memory of Spark’s GraphX RDDs containing the input graph and all vertex summaries of the graph summary before the find and merge phase (equal for all three summary models on each dataset). The colored solid lines depict the mean allocated memory of Spark for the respective summary models and the colored dotted lines the memory allocated by the VertexUpdateHashIndex (VHI) after the graph summary is computed.

that contain the in-memory representation of the input graph and the graph summary. Furthermore, we use a Java Instrumentation Agent to log the actual in-memory size of the VertexUpdateHashIndex. To this end, we log the sum of all shallow sizes of each referenced object in the VertexUpdateHashIndex. To exclude the possibility of negative impacts on the runtime, we logged the memory consumption in a separate set of experiments from Experiment 1 and 2. Analogously to Experiment 2, we use the BSBM and DyLDO-core dataset. Based on the discussion in Section 8.3.3, we use 4 cores for all runs. Partitioning and distributing graph data during processing potentially creates data overheads due to data redundancy. Since Spark does not use more than 4 partitions for our datasets, adding more cores does not benefit the performance of the make-set phase (Phase 1). Reducing the number of cores means unnecessarily long runtimes. Thus, 4 cores is a reasonable choice for this experiment.

8.4.2 Results

The results of this experiment are shown in Figure 8.5. Notice that while the memory consumption of the summary computation is the memory usage of the complete run logged by Spark (labeled Spark Memory), the size of the final VertexUpdateHashIndex (labeled VHI) is calculated via a Java Instrumentation Agent.
Our main observation in this experiment is that the memory consumption scales linear with the input size for all three summary models. Furthermore, the memory overhead due to the VertexUpdateHashIndex is only a fraction of the overall memory consumption. For the BSBM dataset, we find a memory overhead of about 2% (±0.06). For the DyLDO-core dataset, we find a memory overhead between 6% and 11% (mean: 8% ± 0.68).

Another interesting observation in this experiment is that the maximum memory allocated for the summary computation is exactly the same for all three summary models for both datasets, respectively (Figure 8.5, black lines). Only the mean memory allocated by Spark differs between the summary models (Figure 8.5, colored solid lines). We also see that the graph summary computation on the synthetic benchmark dataset BSBM does require a comparable amount of memory to the graph summary computation on the real-world dataset DyLDO-core. Comparing the maximum allocated memory, on the BSBM dataset, we need to allocate on average 4.23 (±0.06) times more memory than to store the raw dataset on disk. On the DyLDO-core dataset, this factor is on average 4.43 (±0.11).

Second, for each dataset, the size of the VertexUpdateHashIndex of each graph summary model differs only marginally. On the BSBM dataset, the VertexUpdateHashIndex of SchemEX is between 2 and 326 megabytes (mean: 160 ± 97 megabytes), the one of Attribute Collection is between 2 and 324 megabytes (mean: 159 ± 97 megabytes), and the one of Class Collection is between 2 and 324 megabytes (mean: 159 ± 97 megabytes). On the DyLDO-core dataset, the VertexUpdateHashIndex of SchemEX is between 569 and 984 megabytes (mean: 759 ± 145 megabytes), the one of Attribute Collection is between 561 and 976 megabytes (mean: 755 ± 146 megabytes), and the one of Class Collection is between 559 and 974 megabytes (mean: 750 ± 145 megabytes).

When we also consider the results of Experiment 1, which show a strong monotonic correlation between the VertexUpdateHashIndex size and the number of vertices in the graph database ($r_s(311) = 0.908, p < .0001$). Thus, L2 of the VertexUpdateHashIndex dominates the overall size. The number of edges is not as important for the size of the VertexUpdateHashIndex since edges are reflected by the vertex summaries ($r_s(311) = 0.661, p < .0001$), but still relevant.

While the difference in size of the VertexUpdateHashIndex between the summary models on a fixed dataset is only a few megabytes, the difference between the datasets is considerably large. The number of vertices of the 20th version of the BSBM dataset is comparable to the 36th version of the DyLDO-core dataset. Version 20 of the BSBM dataset contains with 1,286,066 vertices the most vertices of all BSBM version. Version 36 of the DyLDO-core dataset contains with 1,503,466 vertices the fewest vertices of all DyLDO-core version.
We observe for all three summary models that the VertexUpdateHashIndex in the 36th version of the DyLDO-core dataset is about 2-times larger than the VertexUpdateHashIndex in the 20th version of the BSBM dataset. On average, the size of the VertexUpdateHashIndex in the 36th version of the DyLDO-core dataset is 602 (±5) megabytes. The VertexUpdateHashIndex in the 20th version of the BSBM dataset requires about 324 (±1) megabytes of memory.

8.4.3 Discussion

From the experiment on the memory consumption of the VertexUpdateHashIndex, we can state that the memory overhead required for maintaining this additional data structure is only a fraction of the memory consumption of the graph summary computation. In particular, considering the significant performance benefits found in Experiments 1 and 2 for the incremental algorithm, the memory overhead induced by the VertexUpdateHashIndex of less than 10% on the real-world dataset seems negligible. On the synthetic benchmark dataset, this overhead is only 2%. Since the maximum Spark memory consumption for the graph summary computation is roughly the same, the main varying factor is the absolute size of the VertexUpdateHashIndex.

Furthermore, we can state that the number of vertices in the dataset is mostly responsible for the size of the VertexUpdateHashIndex, i.e., the memory overhead. Second in relevance is the number of primary vertices in the computed graph summary, i.e., how many different schema structures appear in the input dataset. We noticed a nearly 2-times increase of the VertexUpdateHashIndex for the real-world DyLDO-core dataset over the synthetic BSBM dataset in two versions of each dataset, where the difference in number of vertices is less than 15%. References to each vertex are stored in L2. Thus, the payload (L3) and/or the references to vertex summaries (L1) make up the remainder.

Regarding L1, the final graph summary is orders of magnitude larger on the DyLDO-core dataset than on the BSBM dataset. For example, the Attribute Collection in the 20th version of the BSBM dataset contains about $10^2$ vertices, while it contains about $2 \cdot 10^4$ vertices in the 36th version of the DyLDO-core dataset (see Experiment 1, Figures 8.1 and 8.2). The VertexUpdateHashIndex stores a reference to each primary vertex in the graph summary. Thus, the size of the VertexUpdateHashIndex grows w.r.t. the number of primary vertices in the computed graph summary.

Regarding L3, note that we store the data source payload for the DyLDO-core dataset and the vertex count payload on the BSBM dataset, which also contributes to an overall larger VertexUpdateHashIndex on the DyLDO-core dataset. However, we evaluated the impact of the data source payload
by measuring the size of the VertexUpdateHashIndex without any payload information taken into account (neither vertex counts on the BSBM dataset nor the data source on the DyLDO-core dataset). On average, the sizes of the VertexUpdateHashIndex decreased on both datasets by about 25% to 30%, which still results in an about 2-times larger VertexUpdateHashIndex for the DyLDO-core dataset.

Another observation is that, in our experiments, we needed about 4 to 5 times more memory than the raw dataset size. Note that during the graph summary computation, Spark stores the original graph and the corresponding vertex summary for each vertex in memory. It remains to be evaluated if a different graph processing framework can further compress this information to reduce the overall memory consumption, e.g., Bogel [2, 93].

8.5 Summary

In Chapter 7, we presented our incremental graph summarization algorithm for structural graph summaries and analyzed its complexity. In this chapter, we 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. Furthermore, the incremental summarization algorithm outperforms the batch summarization algorithm even when using fewer cores, despite that the batch computation benefits more from parallelization than the incremental algorithm. Finally, we analyzed the memory overhead of the incremental algorithm induced by the VertexUpdateHashIndex. Using the real-world dataset DyLDO-core, we found that using four cores, the incremental algorithm is on average 5 to 44 times faster while only producing a memory overhead of 8% (±1%).
In this final chapter, we summarize the main contributions of this work and present interesting follow-up research topics.

9.1 Conclusion

In this thesis, we focused on semantic structural graph summarization and incremental updates of such graph summaries for evolving graphs. 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. Semantic structural graph summaries extend structural graph summaries by supporting the use of concepts from ontologies as labels for vertices and edges. We motivated the practical benefits of (semantic) structural graph summaries by describing three applications in detail, i.e., semantic entity retrieval (Section 1.1.1), cardinality computation (Section 1.1.2), and data source search (Section 1.1.3).
We defined structural graph summary models as tuples of data graph $G$, equivalence relation $EQR$, and payload elements $PAY$. As data graphs $G$, we consider the two commonly used graph models Labeled Property Graphs (LPGs) and the Resource Description Framework (RDF). As payload elements $PAY$, we defined three frequently used payloads: First, the vertex identity payload, which is needed to implement the semantic entity retrieval application. Second, the vertex count payload, which is needed to implement the cardinality computation application. Third, the data source payload, which is needed to implement the data source search application. To define equivalence relations suitable for structural graph summarization, we introduced the formal model FLUID (FLexible graph sUmmarIes for Data graphs). FLUID is a formal common model to define structural graph summaries, in particular, the equivalence relations $EQR$ for the graph summary models.

Based on an extensive analysis of existing (semantic) structural graph summaries, we introduced three simple and one complex schema elements as well as six parameterizations to define $EQR$. We demonstrated that these elements and parameterizations can be flexibly combined to define existing (semantic) structural graph summaries. All analyzed graph summaries discussed in Section 3.1 can be expressed using FLUID. FLUID also allows adapting existing graph summaries to new tasks. The simplest modification is changing the payload elements, which is easily done, but has a big impact on the size of graph summary and on which tasks can be fulfilled. Other modifications include filtering specific types and properties or enabling inference on semantic graphs.

Expressing (semantic) structural graph summaries with only a handful elements and parameterizations allowed us to define a generic algorithm to compute (semantic) structural graph summaries. Our complexity analysis concluded that graph summaries defined with FLUID can be computed in essentially linear time and space with respect to the number of vertices and edges in the input graph, unless the inference parameterization is used, in which case the running time and space usage may be quadratic. Our analysis of large scale datasets obtained from the Semantic Web shows that often the running time remains essentially linear.

Furthermore, we presented two variants of our parameterized algorithm to compute all structural graph summaries defined with FLUID. First, we empirically evaluated a stream-based implementation of the sequential algorithm for static graphs using 18 graph summary models. Our empirical evaluations revealed huge variations in compression ratio, summarization ratio, and approximation quality for different graph summary models, queries, and datasets. This supports our hypothesis that
there is no single graph summary model that equally fits all tasks and that the performance of the summary model depends on the specific types of queries and characteristics of the datasets. However, we observed meaningful correlations in the results that help to determine the right graph summary model for a given task, type of query, and dataset.

Second, we evaluated the parallel, incremental algorithm for evolving graphs using three graph summary models. Our complexity analysis of the incremental update algorithm concluded that all graph summaries defined with FLUID can be updated in time $O(\Delta \cdot d^k)$, where $\Delta$ is the number of additions, deletions, and modifications to the input graph, $d$ is its maximum degree, and $k$ is the maximum radius of the neighborhoods considered. We empirically evaluated the incremental algorithm regarding size, update complexity, performance, and memory consumption. Our experimental results showed that the incremental summarization algorithm almost always outperforms its batch counterpart, even when about 50% of the graph database changes. Furthermore, the incremental summarization algorithm outperforms the batch summarization algorithm when using fewer cores, despite that the batch computation benefits more from parallelization than the incremental algorithm. Moreover, using the real-world datasets, we found that using four cores, the incremental algorithm is on average 5 to 44 times faster while only producing a memory overhead of 8% (±1%).

In conclusion, based on an extensive analysis of existing structural graph summary models and algorithms, we developed the formal model FLUID, a sequential batch algorithm for static graphs, and a parallel incremental algorithm that can also be applied to evolving graphs. The algorithms are embedded in the formal model, their computational complexity was thoroughly analyzed, and their practicality is demonstrated in extensive empirical evaluations. We published all source code and experimental data under an open source license on GitHub.*

### 9.2 Future Work

In this work, we offer a flexible solution to define and to efficiently compute and update different graph summaries tailored for different applications. A reasonable next step is to implement our framework into existing applications based on graph summaries, thus, offering different search experiences. In particular, following our findings in Chapter 6 regarding the data search scenario, we believe that

*Stream-based FLUID implementation: https://github.com/t-blume/fluid-framework; Parallel Incremental FLUID implementation: https://github.com/t-blume/fluid-spark; Raw data, statistical analyses, and plots: https://github.com/t-blume/visualize-experiments
implementing different graph summaries in a search engine for data sources can yield interesting results. In our ongoing work, we integrated a selection of graph summaries into the search engine LODatio \[42\]. Based on this implementation, we developed IMPULSE.\(^{†}\) IMPULSE searches for data sources containing bibliographic metadata and automatically harvests the metadata, maps the metadata to an internal data schema, and integrates the metadata into an existing metadata database. Thus, IMPULSE could be a valuable contribution towards advanced dataset search \[24\]. However, a systematic evaluation of the impact of the choice of the structural graph summary model on the amount and quality of the harvested data remains future work.

Furthermore, we defined the graph summary \(SG\) as centralized Labeled Property Graph (LPG). One possible extension is to model the graph summary again as a distributed LPG to support distributed databases. This way, the algorithm could be parallelized and distributed even more and the synchronization effort reduced even further. Following the ideas of Tesseract \[11\], in general, changes effect only local graphs, thus, only few duplicate updates need to be detected. This means, our vertex centric incremental algorithm could be implemented in a truly distributed environment, i.e., where workers and database instances reside on different machines.

Moreover, FLUID was developed for structural graph summaries based on equivalence relations only. Relaxing the criteria of using equivalence relations would allow for a larger variety of graph models, e.g., approximate summarization algorithms \[11, 67\]. For applications that do not rely on 100% accuracy, using Bloom filter to further compress the VertexUpdateHashIndex is a reasonable next step as well \[91, 92\]. A Bloom filter is a space-efficient probabilistic data structure that can be used to test if an element is contained in a specific set. Bloom filters guarantee that there are no false negative results, i.e., if a vertex is not summarized by a specific vertex summary. But, false positives are possible, i.e., a vertex may or may not be summarized by a specific vertex summary. This means that for a vertex that was summarized previously, we cannot determine with absolute certainty, which vertex summary summarized it previously. Thus, we may believe a vertex has changed although in fact, it did not.

Finally, we see that using graph summaries to generate neural network architectures as an interesting future work. We pursue the idea of training the neural network model on the graph summary and, subsequently, apply the inference on the original graph. Since the graph summaries are magnitudes smaller, training is much more efficient. Thus, training computationally expensive models like Transformer models is feasible without using large-scale high-performance computing cluster.

\(^{†}\)Source Code available on GitHub: https://github.com/t-blume/impulse
References


Till Blume - Curriculum Vitae

Year of Birth  1988

Place of Birth  Hannover, Germany

Citizenship  German

Education

since 02/2021  PhD student at the University of Ulm with Prof. Scherp.
07/2016 – 11/2020  PhD student at the University of Kiel with Prof. Scherp.
10/2013 – 06/2016  Master of Science from the University of Kiel, thesis with Prof. Luttenberger.
09/2009 – 10/2013  Bachelor of Science from the University of Kiel, thesis with Prof. Jansen.

Work Experience

since 05/2020  Data Architect at Ernst & Young GmbH WPG – R&D Germany.
11/2018 – 04/2020  Research Assistant at the University of Kiel.
since 02/2018  Co-founder of Open Legal Data.
05/2016 – 09/2019  Research Assistant at the Leibniz Information Centre for Economics.
03/2012 – 08/2015  Student Assistant at the University of Kiel.
Till Blume - Curriculum Vitae

Extracurricular activity

10/2018 – 03/2019  Open Legal Data is funded by the Federal Ministry of Education and Research.
10/2017 – 09/2019  Elected PhD Representative at the Leibniz Information Centre for Economics.
06/2017 – 09/2019  Server administrator of Prof. Scherp’s research group.
2016              2nd place at the student competition of the 2016 European Semantic Web Conference summer school.
09/2013 – 02/2014  Erasmus exchange with the Institut polytechnique de Grenoble, France.
2009              Scholarship for the University of Kiel.

Publications

2021  Till Blume, David Richerby, Ansgar Scherp: FLUID: A common model for semantic structural graph summaries based on equivalence relations, Theoretical Computer Science (TCS).
2018 Till Blume, Ansgar Scherp: Towards Flexible Indices for Distributed Graph Data: The Formal Schema-level Index Model FLuID, GI-Workshop on Foundations of Databases (GvDB).

Acronyms

**B** Billion.

**CQ** Complex Query.

**CSE** Complex Schema Element.

**FLUID** FLexible graph sUmmarIes for Data graphs.

**GDB** Labeled Property Graph Database.

**IRI** International Resource Identifier.

**LOD** Linked Open Data.

**LPG** Labeled Property Graph.

**M** Million.

**OC** Object Cluster.

**OWL** Web Ontology Language.

**PC** Predicate Cluster.

**POC** Predicate–Object Cluster.

**RDF** Resource Description Framework.

**RDFS** RDF Schema.

**SE** Schema Element.
SG  Summary Graph.

SPARQL  SPARQL Protocol and RDF Query Language.

SQ  Simple Query.

SSE  Simple Schema Element.

T  Thousand.

URL  Uniform Resource Locator.

## Index

<table>
<thead>
<tr>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Approximation Quality</td>
<td>80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch Computation</td>
<td>90, 93</td>
</tr>
<tr>
<td>Bisimulation</td>
<td>29, 47</td>
</tr>
<tr>
<td>Blank Node</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality Computation</td>
<td>4, 54</td>
</tr>
<tr>
<td>Chaining Parameterization</td>
<td>40, 47, 61</td>
</tr>
<tr>
<td>Changes</td>
<td>107</td>
</tr>
<tr>
<td>Complex Query</td>
<td>80</td>
</tr>
<tr>
<td>Complex Schema Element</td>
<td>42, 43, 61</td>
</tr>
<tr>
<td>Compression Ratio</td>
<td>77</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Graph</td>
<td>13</td>
</tr>
<tr>
<td>Data Source</td>
<td>16</td>
</tr>
<tr>
<td>Data Source Payload</td>
<td>55</td>
</tr>
<tr>
<td>Data Source Search</td>
<td>5, 54</td>
</tr>
<tr>
<td>Direction Parameterization</td>
<td>40, 49, 61</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge Label</td>
<td>17</td>
</tr>
<tr>
<td>Entity</td>
<td>15</td>
</tr>
<tr>
<td>Equivalence Class</td>
<td>38</td>
</tr>
<tr>
<td>Equivalence Relation</td>
<td>38</td>
</tr>
<tr>
<td>Extended Union</td>
<td>39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
<td>38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph Label</td>
<td>18</td>
</tr>
<tr>
<td>Graph Name</td>
<td>16, 18</td>
</tr>
<tr>
<td>Graph Summary</td>
<td>38, 58</td>
</tr>
<tr>
<td>Graph Summary Model</td>
<td>38, 60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity Equivalence</td>
<td>38</td>
</tr>
<tr>
<td>Incoming Neighbor</td>
<td>15</td>
</tr>
<tr>
<td>Incoming Property Set</td>
<td>15, 30</td>
</tr>
<tr>
<td>Incremental Computation</td>
<td>94, 96</td>
</tr>
<tr>
<td>Incremental Graph Compression</td>
<td>34</td>
</tr>
<tr>
<td>Incremental Graph Summarization</td>
<td>32</td>
</tr>
<tr>
<td>Incremental Schema Discovery</td>
<td>34</td>
</tr>
<tr>
<td>Incremental Structural Graph Summarization</td>
<td>7</td>
</tr>
<tr>
<td>Incremental Subgraph Index</td>
<td>33</td>
</tr>
<tr>
<td>Inference</td>
<td>16</td>
</tr>
<tr>
<td>Inference Parameterization</td>
<td>40, 50, 61</td>
</tr>
<tr>
<td>Instance Parameterization</td>
<td>40, 51, 61</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>K</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Key–Value Property</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Label Parameterization</td>
<td>40, 45, 61</td>
</tr>
<tr>
<td>Labeled Property Graph</td>
<td>17</td>
</tr>
<tr>
<td>Labeled Property Graph Database</td>
<td>18</td>
</tr>
<tr>
<td>Literal</td>
<td>15</td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>Named Graph</td>
<td></td>
</tr>
<tr>
<td>Neighbor</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Object Cluster</td>
<td></td>
<td>41, 61</td>
</tr>
<tr>
<td>Ontology Inference</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>OR Combination</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Outgoing Neighbor</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>OWL SameAs</td>
<td></td>
<td>32, 51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterization</td>
<td></td>
<td>44, 61</td>
</tr>
<tr>
<td>Payload</td>
<td></td>
<td>39, 53, 60</td>
</tr>
<tr>
<td>Payload Element</td>
<td></td>
<td>39, 53, 58</td>
</tr>
<tr>
<td>Performance</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>Predicate</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Predicate Cluster</td>
<td></td>
<td>41, 61</td>
</tr>
<tr>
<td>Predicate Path</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>Predicate–Object Cluster</td>
<td></td>
<td>41, 61</td>
</tr>
<tr>
<td>Primary Vertex</td>
<td></td>
<td>39, 58</td>
</tr>
<tr>
<td>Property Cluster</td>
<td></td>
<td>45, 61</td>
</tr>
<tr>
<td>Property Key</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>Property Set</td>
<td></td>
<td>15, 26</td>
</tr>
<tr>
<td>Property Value</td>
<td></td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Quad</td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RDF Graph</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>RDF Property</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>RDF Schema</td>
<td></td>
<td>16, 31, 49</td>
</tr>
<tr>
<td>RDF Type</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Related Property</td>
<td></td>
<td>31, 52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SameAs</td>
<td></td>
<td>32, 51</td>
</tr>
</tbody>
</table>

| Schema Element    |                              | 40, 60 |
| Schema Structure  |                              | 38, 58 |
| Secondary Vertex  |                              | 39, 58 |
| Semantic Entity Retrieval |                    | 3, 53 |
| Semantic Label    |                              | 16  |
| Semantic Rule Feature |                        | 31  |
| Sequential Processing |                        | 64  |
| Set Parameterization |                        | 40, 46, 61 |
| Simple Query      |                              | 80  |
| Simple Schema Element |                        | 41, 61 |
| Size              |                              | 107 |
| Skolemization     |                              | 15  |
| Stream Processing |                              | 79  |
| Structural Graph Summarization |                  | 6, 38 |
| Structural Graph Summary |                    | 38  |
| Structural Graph Summary Model |                | 6, 26, 55 |
| Subgraph Feature  |                              | 28  |
| Subject           |                              | 15  |
| Summarization Ratio |                        | 77, 107 |
| Summary Graph     |                              | 58  |
| Tautology Equivalence |                    | 38  |
| Triple            |                              | 15  |
| Triple Feature    |                              | 26  |
| Type Cluster      |                              | 45, 61 |
| Type Set          |                              | 15, 27 |

<table>
<thead>
<tr>
<th>U</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Update Complexity</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>Updates</td>
<td></td>
<td>107</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Count Payload</td>
<td></td>
<td>54</td>
</tr>
<tr>
<td>Vertex Identity Payload</td>
<td></td>
<td>53</td>
</tr>
<tr>
<td>Vertex Label</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>Vertex Summary</td>
<td></td>
<td>39, 58, 59</td>
</tr>
<tr>
<td>Vertex–Centric Programming</td>
<td></td>
<td>90</td>
</tr>
<tr>
<td>Vocabulary Graph</td>
<td></td>
<td>49</td>
</tr>
</tbody>
</table>