An Algorithm for Extracting Schemas from External Memory Graphs

Yoshiki Sekine
Graduate School of Library, Information and Media Studies
University of Tsukuba
1-2, Kasuga, Tsukuba
Ibaraki, 305-8550, Japan
s1211512@klis.tsukuba.ac.jp

Kosetsu Ikeda
Academic Link Center
Chiba University
1-33, Yayoicho, Inage
Chiba, 263-8522, Japan
lumely@chiba-u.jp

Nobutaka Suzuki
Faculty of Library, Information and Media Science
University of Tsukuba
1-2, Kasuga, Tsukuba
Ibaraki, 305-8550, Japan
nsuzuki@slis.tsukuba.ac.jp

ABSTRACT

In recent years, graph data is rapidly increasing and the size of graph data is drastically growing. In contrast to other databases such as relational databases and XML, most of graphs do not have their own schemas. Therefore, in many cases we cannot make use of schema in order to manage graphs effectively. If we can extract a schema from a graph efficiently, we can take advantage of the extracted schema for query optimization, structure browsing, query formulation, and so on. In this paper, we propose an algorithm for extracting a schema from a large graph. In order to handle large graphs, our algorithm is designed as an external memory algorithm. The algorithm is designed so that each file is read sequentially in most cases and very few random accesses are required for schema extraction, which makes our algorithm I/O efficient.

CCS Concepts

•**Information systems** → *Graph-based database models;*

Keywords

graph schema; schema extraction; external memory algorithm

1. INTRODUCTION

A schema of a database system provides a structural blueprint of how the database is organized, and plays an important role in managing the database system. Indeed, a schema of a database system is commonly used for optimizing queries, ensuring integrity constraints, helping users to write correct queries, and so on.

Meanwhile, in recent years graph data is rapidly increasing. In contrast to other databases such as relational databases and XML, most of graphs do not have their own schemas. Therefore, in many cases we cannot make use of schema in order to manage graphs effectively. Here, if we can extract a schema from a graph efficiently, we can take advantage of the extracted schema for query optimization [2], structure browsing and query formulation [4], and so on. For example, consider a regular path query q, and let q' be the query obtained by constructing the product automaton of q and an extracted schema. Then q' can be executed more efficiently than q. Therefore, in this paper we propose an algorithm for extracting a schema from a graph.

Most of schema extraction algorithms proposed so far are in-memory algorithms. In other words, such algorithms assume that the entire graph fits in main memory. However, the size of recent graph data is rapidly growing, and thus a number of graphs currently available are too large to fit in main memory. In order to handle such large graphs, our algorithm is designed as an external memory algorithm.

In order to deal with large graphs, our algorithm takes a two-step approach: class extraction and edge extraction. In the class extraction step, the algorithm reads a graph sequentially and extracts classes with maintaining minimum information to extract classes in main memory, and outputs (1) a class file consisting of the classes of all the nodes and (2) an edge file consisting of edges between "nodes and classes". In the edge extraction step, the algorithm extracts edges between classes by reading the two files sequentially. The point of our approach is that due to the two-step approach each file is read sequentially in most cases and very few random accesses are required for schema extraction, which makes our algorithm I/O efficient. Another major point of our algorithm is that our algorithm handles input and intermediate files by only sequential reads and external sorting, and no other special method for accessing files is required. This enables our algorithm to be implemented much easier than usual external memory algorithms, since we can use a number of high-level scripting languages (Python, Ruby, etc.) to implement our algorithm as well as "lower-level" languages such as C and C++ that are popular for implementing external memory algorithms. Experimental results suggest that our algorithm can extract schemas from large graphs efficiently and appropriately.

The rest of this paper is organized as follows. Section 2 gives preliminary definitions. Section 3 proposes an external memory algorithm for extracting a schema from a graph. Section 4 presents some experimental results. Section 5 summarizes this paper.

BigNet '16 Indianapolis, Indiana USA

DOI: 10.475/123_4

Related Work

A number of schema extraction algorithms for graphs have been proposed. DataGuide [4] extracts a schema by grouping nodes reachable from the root via the same label path into the same class. ApproximateDataguite [5] is the approximate version of DataGuide. Nestorov et al. proposes an algorithm for extracting a set of classes by using a clustering approximation method [10]. Wang et al. proposes an algorithm that extracts a schema by an incremental clustering method [14]. These algorithms are in-memory algorithms and cannot handle large graphs that do not fit in main memory. Navlakha et al. proposes a graph summarization algorithm [9]. This is an in-memory algorithm designed for unlabeled undirected graphs, while our algorithm is designed for labeled directed graphs. Luo et al. proposes an external memory algorithm for k-bisimulation [8]. However, the notion of k-bisimulation is too strong to extract classes from usual graphs, since under the condition of k-bisimulation, any two nodes in the same class must have same label paths whose length is k. On the other hand, our algorithm assumes a weaker condition under which nodes having a "similar" set of edges are grouped into the same class.

Schema extraction algorithms have also been proposed for XML documents as well as graphs. The algorithm in [12] extracts a relational schema from given DTDs. The XTRACT system [3] extracts a DTD as a schema from given XML documents. XStruct [6] is a schema extraction system for large XML documents. These algorithms are designed for trees and cannot handle general graph structure.

Several external memory algorithms have been proposed in database research field, e.g., graph triangulation [7], strongly connected components [15], graph reachability [16], and regular path query [13]. To the best of our knowledge, however, no external memory algorithm for schema extraction has been proposed so far.

2. PRELIMINARIES

In this section, we define graph and related notions.

2.1 Labeled Directed Graph

A *labeled directed graph* (*graph* for short) is denoted $G = (V, E)$, where V is a set of *nodes* and E is a set of *edges*. Let e be an edge labeled by l from a node u to a node v. Then e is denoted $u \xrightarrow{l} v$. u is called the *source* and v is called the *target* of e, and u, v are called the *endpoints* of e. If there exists an edge e whose endpoints are u and v, then u (resp. v) is *adjacent* to v (resp. u), and e is *incident* to u and v.

For example, Fig. 1 represents a fragment of article information and is denoted a graph $G = (V, E)$, where

$$V = \{article_1, article_2, journal_1, \text{“}Article1\text{”}, \text{“}Article2\text{”}, \text{“}Journal1\text{”}, \text{“}Person1\text{”}, \text{“}Person2\text{”}, \text{“}Person3\text{”}, \text{“}10\text{”}, \text{“}1\text{”}\},$$

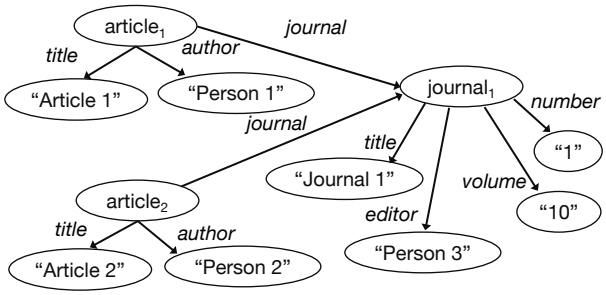


Figure 1: Example of a graph

source	label	target
article1	journal	journal1
article1	title	"Article 1"
article1	author	"Person 1"
article2	title	"Article 2"
article2	author	"Person 2"
article2	journal	journal1
journal1	title	"Journal 1"
journal1	editor	"Person 3"
journal1	volume	"10"
journal1	number	"1"

Figure 2: The graph file of the graph in Fig. 1

$$E = \{article_1 \xrightarrow{journal} journal_1, article_2 \xrightarrow{journal} journal_1, article_1 \xrightarrow{title} \text{“}Article1\text{”}, article_1 \xrightarrow{author} \text{“}Person1\text{”}, article_2 \xrightarrow{title} \text{“}Article2\text{”}, article_2 \xrightarrow{author} \text{“}Person2\text{”}, journal_1 \xrightarrow{title} \text{“}Journal1\text{”}, journal_1 \xrightarrow{editor} \text{“}Person3\text{”}, journal_1 \xrightarrow{volume} \text{“}10\text{”}, journal_1 \xrightarrow{number} \text{“}1\text{”}\}.$$

A graph is stored in a graph file. In this paper, we assume that each line of a graph file corresponds to an edge, namely, a line consists of the *source*, the label, and the *target* of an edge. Figure 2 shows a graph file that stores the graph of Fig. 1.

2.2 Schema of Graph

A *schema* is a summarization of a graph and it is also represented as a graph. A node in a schema is called a *class*. Any node in a (instance) graph is mapped to a class in a schema. We assume that every text node belongs to a single class LEAF. Figure 3 shows an example of a schema extracted from the graph of Fig. 1, where article1 and article2 belong to class1, journal1 belongs to class2, and the other nodes belong to LEAF.

We assume that a schema is composed of two files denoted schema_classes and schema_edges. schema_classes stores pairs of a node and its class. Note that we do not store leaf nodes and their class LEAF in schema_classes since they are not needed for schema extraction. schema_edges stores edges between classes, in which each line has the *source*, the label, and the *target* of an edge respectively. Figure 4 shows an

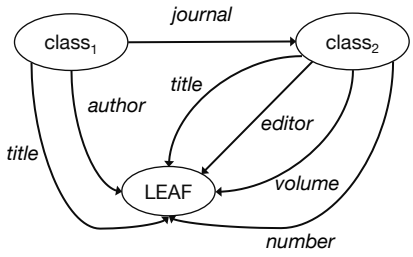


Figure 3: Example of a schema

node	class
article1	class1
article2	class1
journal1	class2

(a) schema_classes

source class	label	target class
class1	journal	class2
class1	title	LEAF
class1	author	LEAF
class2	title	LEAF
class2	editor	LEAF
class2	volume	LEAF
class2	number	LEAF

(b) schema_edges

Figure 4: Example of schema files

example of the two files representing the schema of Fig. 3.

3. THE ALGORITHM

Our schema extraction algorithm is designed as an external memory algorithm. To achieve this, our algorithm consists of two steps: class extraction and edge extraction. The class extraction is to create new classes and to assign each node to a class, and the edge extraction is creating edges between classes. In this section, we first define utility value used in the class extraction, then we describe the schema extraction algorithm.

3.1 Extended Utility Value

Given a graph, our algorithm groups nodes having similar set of labels into the same class. Our class extraction is based on the utility value proposed in [14]. However, the utility value in [14] does not distinguish incoming and outgoing edges at class extraction. Due to this, nodes which should be in distinct classes are sometimes grouped into the same class. To cope with the problem, we extend the definition of utility value so that incoming edges and outgoing edges are treated separately at the calculation of utility value. We also refine the relationship between class and its labels in the definition of utility value.

Let v be a node, c be a class, and C be a set of classes. The set of labels of incoming edges of v is denoted $L_{in}(v)$. By $L_{in}(c)$ we mean the set of labels of incoming edges of c, that is,

$$L_{in}(c) = \bigcup_{v \in c} L_{in}(v).$$

Let $|c|$ be the number of nodes in c and $c_{in}(l)$ be the set of nodes in c having an incoming edge labeled by l. Then $P_{in}(l|c)$ is defined as the ratio of $|c_{in}(l)|$ to $|c|$, that is,

$$P_{in}(l|c) = \frac{|c_{in}(l)|}{|c|}.$$

By $C_{in}(l)$ we mean the set of nodes in C having an incoming edge labeled by l, that is,

$$C_{in}(l) = \bigcup_{c \in C} c_{in}(l).$$

$P_{in}(c|l)$ is defined as the ratio of $|c_{in}(l)|$ to $|C_{in}(l)|$, that is,

$$P_{in}(c|l) = \frac{|c_{in}(l)|}{|C_{in}(l)|}.$$

$T_{in}(l, c)$, representing the strength of the relationship between incoming label l and c, is defined as follows.

$$T_{in}(l, c) = P_{in}(l|c)^k \cdot P_{in}(c|l)^{\frac{1}{k}},$$

where $k > 0$ is a parameter to control which of $P_{in}(l|c)$ and $P_{in}(c|l)$ is emphasized when extracting classes. $E_{in}(c)$ is defined as the mean of $T_{in}(l, c)$, that is,

$$E_{in}(c) = \frac{1}{|L_{in}(c)|} \sum_{l \in L_{in}(c)} T_{in}(l, c).$$

The *utility value* w.r.t. incoming edge, denoted U_{in}, is defined as the mean of $E_{in}(c)$, that is,

$$U_{in} = \frac{1}{|C|} \sum_{c \in C} E_{in}(c),$$

where $|C|$ is the number of classes in C.

Similarly, the utility value w.r.t. outgoing edge is defined as follows. The set of labels of outgoing edges of v is denoted $L_{out}(v)$. By $L_{out}(c)$ we mean the set of labels of outgoing edges of c, that is,

$$L_{out}(c) = \bigcup_{v \in c} L_{out}(v).$$

Let $c_{out}(l)$ be the set of nodes in c having an outgoing edge labeled by l, and $C_{out}(l)$ be the set of nodes in C having an outgoing edge labeled by l. Moreover, $P_{out}(l|c)$ is defined as the ratio of $|c_{out}(l)|$ to $|c|$, and $P_{out}(c|l)$ is defined as the ratio of $|c_{out}(l)|$ to $|C_{out}(l)|$. Then $T_{out}(l, c)$ represents the strength of the relationship between outgoing label l and c, that is,

$$T_{out}(l, c) = P_{out}(l|c)^k \cdot P_{out}(c|l)^{\frac{1}{k}}.$$

And $E_{out}(c)$ is the mean of $T_{out}(l, c)$, that is,

$$E_{out}(c) = \frac{1}{|L_{out}(c)|} \sum_{l \in L_{out}(c)} T_{out}(l, c).$$

The *utility value* w.r.t. outgoing edge, denoted U_{out}, is defined as the mean of $E_{out}(c)$, that is,

$$U_{out} = \frac{1}{|C|} \sum_{c \in C} E_{out}(c).$$

Finally, the *extended utility value* (*utility value*, for short), denoted U, is defined as the linear sum of the U_{in} and U_{out}, that is,

$$U = \alpha U_{in} + \beta U_{out}, \quad \alpha + \beta = 1.$$

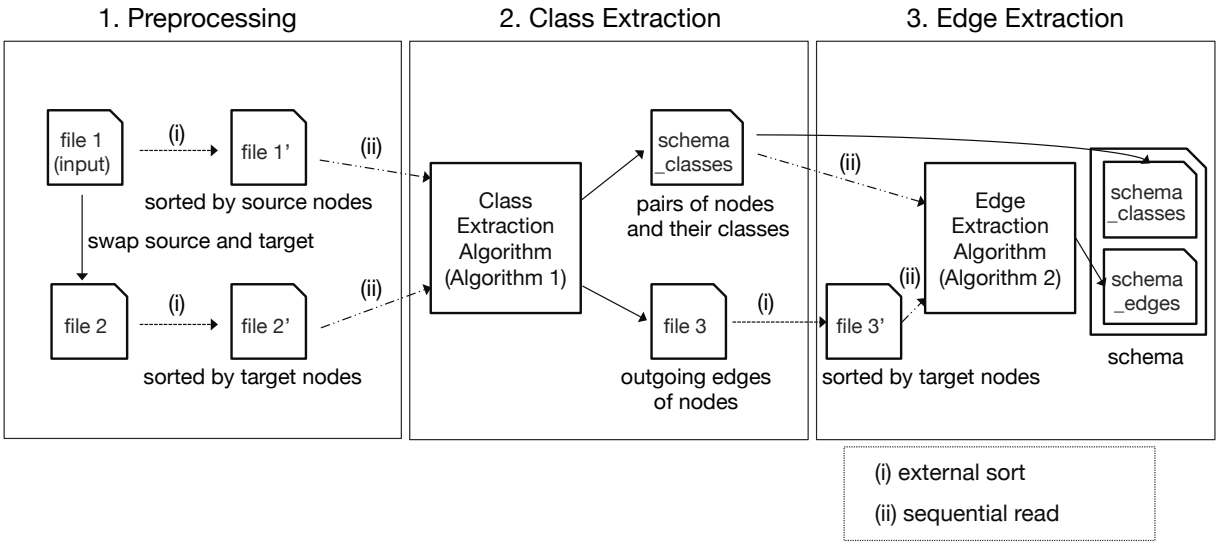


Figure 5: Outline of our algorithm

Thus, U becomes higher if nodes having similar labels of incoming/outgoing edges are grouped into the same class. Our aim is to extract classes so that the classes bring a high utility value.

3.2 Schema Extraction Algorithm

In-memory schema extraction algorithms assume that the entire graph is stored in main memory. However, since recent large graphs are too large to fit in main memory, we need another approach to handling such large graphs.

In order to deal with such large graphs, we take the following approach. First, our algorithm sequentially reads a graph, extracts classes with maintaining minimum information to extract classes in main memory, and outputs files having information required for extracting edges between classes. Note that since the information for edge extraction must include the classes of all nodes, the files cannot be fit in main memory. Then the algorithm extracts edges by reading these files sequentially.

Our algorithm consists of preprocessing, class extraction, and edge extraction. Given a graph file like Fig. 2, our algorithm extracts a schema of the graph. An outline of our algorithm is as follows (see Fig. 5).

Input: graph file. In the following, we call the input graph file file 1. As shown in Fig. 2, each line of file 1 represents an edge, namely, a line consists of the source, the label, and the target of an edge.

Output: schema_classes and schema_edges

1. **Preprocessing**
   (a) Read file 1 sequentially, swap the source and the target of each line, and output the result as file 2. Thus, each line of file 2 consists of the target, the label, and the source of an edge.
   (b) Sort file 1 and file 2 externally. Let file 1′ and file 2′ be the resulting files, respectively.
2. **Class Extraction**
   (a) Read file 1′ and file 2′ concurrently and sequentially, and extract the class of each node based on utility value.
   (b) Each time the class of a node is extracted, output the node and the class to schema_classes, and output the outgoing edges of the node to file 3. These files are used in the edge extraction step.
3. **Edge Extraction**
   (a) Sort file 3 externally. Let file 3′ be the resulting file.
   (b) Read schema_classes and file 3′ concurrently and sequentially. By using schema_classes, replace the node of each edge in file 3′ with a class. This results in edges between classes, which are written into schema_edges.

In the following, we describe the details of our algorithm.

3.2.1 Preprocessing

The input graph file called file 1 is a list of edges (Fig. 2). Each line represents an edge, namely, a line consists of the source, the label, and the target of an edge. We sort file 1 and let file 1′ be the resulting file (Fig. 7a). Since file 1′ is sorted, edges having the same source appear consecutively in file 1′. Therefore, we can obtain the outgoing edges of each node by one sequential read only. We also need to obtain the incoming edges of nodes. However, if only file 1 and file 1′ are given, we need an exhaustive search on the files.

To cope with this problem, we also make another sorted file. First, we create file 2 from file 1 by swapping the source and the target of each line (Fig. 6). Thus, each line of file 2 consists of the target, the label, and the source of an edge. Then we sort file 2 and let file 2′ be the resulting file (Fig. 7b). Now we can obtain the incoming edges of each node by a

sequential read only because edges having the same target appear consecutively in file 2'.

3.2.2 Class Extraction

Values $|c|$, $|c_{in}(l)|$, and $|c_{out}(l)|$ for each class c are kept in main memory to calculate utility value until the classes of all nodes are extracted. On the other hand, for a node v, labels of incident edges and adjacent nodes of v are kept in main memory until the algorithm outputs the class of v to schema_classes. When the class extraction of v is completed, v and its incident edges are discarded.

Algorithm 1 shows the procedure of the class extraction. Input files, file 1' and file 2', are sorted files obtained in the preprocessing. We assume that every leaf node belongs to the same class called $LEAF$. Throughout this algorithm, by $c(v)$ we mean the class of a node v. Let *current* be the node that is currently read. By reading file 1' and file 2' sequentially, we obtain the incident edges of *current* and extract the class of *current* based on utility value. This process is repeated until file 1' and file 2' reach the end of file.

Specifically, our algorithm works as follows. We read file 1' and file 2' sequentially so that the source of the current line of file 1' coincides with the target of the current line of file 2', and *current* is set to the source of the current line of file 1' (and the target of the current line of file 2') in lines 2 to 5. The class of *current* is extracted as follows. If *current* is a leaf node, then $c(current)$ is set to LEAF (lines 7 to 8). Then we calculate the following utility values and choose the class c_{max} for which the maximum utility value is obtained (lines 9 to 16). In particular, if $C = \emptyset$, then the first calculation (lines 10 to 12) is skipped.

- The utility value assuming that *current* belongs to c, for each class c extracted so far.
- The utility value assuming that *current* belongs to a new class having the same edges as *current*.

Each time $c(current)$ is extracted, we output a pair $(current, c(current))$ to schema_classes (line 17). Repeating this process until the input files reach EOF, we obtain the classes of all nodes.

Besides the classes extracted above, we prepare the following for the edge extraction step presented below.

- For each node *current*, replace the sources of outgoing edges of *current* with $c(current)$, swap the source and the target of each of the edges, and output these edges to file 3 (lines 18 to 19).

Thus, each line of file 3 consists of a node v (target), a label l, a class c (source), respectively (Fig. 8a). If the target v is a leaf, we use "LEAF" instead of v since the content is not needed in the next step.

3.2.3 Edge Extraction

Algorithm 2 shows the procedure of the edge extraction. As shown in Fig. 8a, file 3 is a sequence of triples (*target*, *label*, *class*), and the edge extraction is done by replacing *target* of each triple by $c(target)$. To do this, the algorithm first sorts file 3 and obtain file 3' as the result as shown in Fig. 8b (line 1). Since file 3' is sorted, the edges having the same target appear consecutively in file 3', which enables target nodes to be replaced consecutively. Let v be the target node of the "current" edge read from file 3', and suppose

Algorithm 1 Class Extraction

Input: file 1', file 2'
Output: schema_classes, file 3

1: Let C be the set of extracted classes. Initially, $C = \emptyset$.
2: Read a line from file 1'. Let v be the source of the line.
3: Read a line from file 2'. Let u be the target of the line.
4: **while** file 1' does not reach EOF **or** file 2' does not reach EOF **do**
5: $current \leftarrow \min\{u, v\}$
6: Read file 1' and file 2' sequentially, and obtain the incoming and the outgoing edges of *current*.
7: **if** *current* is a leaf node **then**
8: $c(current) \leftarrow LEAF$
9: **else**
10: **for each** $c \in C$ **do**
11: Calculate the utility value assuming that *current* belongs to c.
12: **end for**
13: Calculate the utility value assuming that *current* belongs to a new class having the same edges as *current*.
14: Let c_{max} be the class for which the maximum utility value is obtained in lines 10 to 13.
15: $c(current) \leftarrow c_{max}$
16: Add $c(current)$ to C.
17: Output a pair $(current, c(current))$ to schema_classes.
18: Replace the sources of outgoing edges of *current* with $c(current)$.
19: For each edge $c(current) \overset{label}{\rightarrow} target$ obtained in line 18, output a triple $(target, label, c(current))$ to file 3 (*target* is replaced by "LEAF" if the target is a leaf node).
20: **end if**
21: Let v be the source of current line of file 1', and let u be the target of current line of file 2'.
22: **end while**

that the class c_t of v is obtained from schema_classes. Then we can replace the target of every edge whose target is v by c_t, which can be done by a sequential read from file 3'.

Specifically, we read a line (v, l, c_s) from file 3', where v is the target node, l is the label, and c_s is the source class (line 4). If the class of v is already known, then v is replaced by the class (lines 5 to 8). Otherwise, by reading schema_classes sequentially, we obtain the class of v and replace v of each edge in file 3' with the class (line 10). Repeating this until file 3' reaches EOF, we obtain the edges between classes.

3.3 I/O Cost

We consider the I/O cost of our algorithm. Let $G = (V, E)$ be a graph, $|V|$ be the number of nodes and $|E|$ be the number of edges. We assume that data is transferred between external memory and main memory in blocks of size B. $\mathcal{O}(sort(|E|))$ represents the I/O complexity of external merge sort. The I/O cost of each step is as follows.

1. **Preprocessing**

   Sorting file 1' and file 2' externally: $\mathcal{O}(sort(|E|))$

2. **Class Extraction**

Algorithm 2 Edge Extraction

Input: file 3, schema_classes
Output: schema_edges

```
1: Sort file 3. Let file 3' be the resulting file.
2: Read a line from schema_classes. Let v_t be the node and
       c_t be the class of the line.
3: while file 3' does not reach EOF do
4:     Read a line from file 3'. Let v, l, c_s be the target, the
           label, and the source of the line, respectively.
5:     if v is a leaf node then
6:         Add a triple (c_s, l, LEAF) to schema_edges.
7:     else if v = v_t then
8:         Add a triple (c_s, l, c_t) to schema_edges.
9:     else
10:        Read schema_classes sequentially and find a line
               (v_t, c_t) such that v_t = v.
11:        Add a triple (c_s, l, c_t) to schema_edges.
12:    end if
13: end while
```

target	label	source
journal1	journal	article1
"Article 1"	title	article1
"Person 1"	author	article1
"Article 2"	title	article2
"Person 2"	author	article2
journal1	journal	article2
"Journal 1"	title	journal1
"Person 3"	editor	journal1
"10"	volume	journal1
"1"	number	journal1

Figure 6: file 2

(a) Reading file 1' and file 2': $\mathcal{O}(|E|/B)$

(b) Writing pairs of a node and its class to schema_classes: $\mathcal{O}(|V|/B)$

(c) Writing outgoing edges to file 3: $\mathcal{O}(|E|/B)$

3. **Edge Extraction**

(a) Sorting file 3 externally: $\mathcal{O}(sort(|E|))$

(b) Reading file 3': $\mathcal{O}(|E|/B)$

(c) Reading schema_classes: $\mathcal{O}(|V|/B)$

(d) Writing edges to schema_edges: $\mathcal{O}(|E|/B)$

Thus, the I/O cost of our algorithm is as follows.

$$\mathcal{O}\left(\frac{|E|}{B} + \frac{|V|}{B} + sort(|E|)\right) = \mathcal{O}\left(\frac{|V|}{B} + sort(|E|)\right)$$

The external R-way merge sort algorithm is an efficient algorithm for sorting large files externally, and we have a number of implementations of the algorithm, e.g., UNIX sort. Therefore, the above estimation suggests that our algorithm extracts a schema from a large graph efficiently, if only such commands are available.

3.4 Example of Our Algorithm

Let us show an example of our algorithm. Let file 1 be the file in Fig. 2, which is obtained from the graph in Fig. 1. Texts enclosed in double quotes are treated as leafs.

source	label	target
article1	author	"Person 1".
article1	journal	journal1.
article1	title	"Article 1".
article2	author	"Person 2".
article2	journal	journal1.
article2	title	"Article 2".
journal1	editor	"Person 3".
journal1	number	"1".
journal1	title	"Journal 1".
journal1	volume	"10".

(a) file 1'

target	label	source
"1"	number	journal1
"10"	volume	journal1
"Article 1"	title	article1
"Article 2"	title	article2
"Journal 1"	title	journal1
"Person 1"	author	article1
"Person 2"	author	article2
"Person 3"	editor	journal1
journal1	journal	article1
journal1	journal	article2

(b) file 2'

Figure 7: Input files of Algorithm 1

First, the preprocessing step is as follows. We read file 1 sequentially, swap the source and the target of each line, and output the result as file 2 (Fig. 6). Then we sort file 1 and file 2, and obtain file 1' and file 2', respectively (Fig. 7).

Next, Algorithm 1 performs the class extraction step as follows. We read file 1' and file 2' and we have $v = article1$ and $u =$ "1". Since "1" $< article1$, we have $current =$ "1" in line 5. Since *current* is a leaf node, its class is set to LEAF in lines 7 to 8. We read the next line from file 2' and obtain $u =$ "10". The class of u is LEAF again. We proceed similarly, and let us consider the case where $v = article1$ and $u = journal1$. We have $current = article1$. We then read file 1' sequentially and obtain all incident edges of *current* in line 6. Since $|C| = 0$, *current* belongs to a new class c_1 (lines 13 to 16). We output a pair (article1, c_1) to schema_classes in line 17. We also replace the sources of outgoing edges of *current* with c_1 (and replace the target with "LEAF" if the target is a leaf node) and output the edges to file 3 in lines 18 to 19. Then we read file 1' and file 2', and we have $v = article2$ and $u = journal1$ (line 21). We have $current = article2$ in line 5. Since $|C| = 1$, we calculate utility values in the cases where *current* belongs to c_1 and belongs to a new class c_2 in lines 10 to 13. Since the former is larger than the latter, c_1 is chosen as the class of *current* in line 14. We continue in a similar manner by the end of file 1' and file 2' and we obtain file 3 and schema_classes (Figs. 8a and 9a).

Finally, Algorithm 2 performs the edge extraction step as follows. We sort file 3 and obtain file 3' (Fig. 8b) as the result in line 1. We read a line from file 3' and obtain $v =$ "$LEAF$", $l = author$ and $c_s = c_1$ in line 4. Since the target is a leaf, add a triple $(c_1, author, LEAF)$ to schema_edges in lines 5 to 6. We next read a line from file 3' and obtain

target	label	source
"LEAF"	author	c_1
journal1	journal	c_1
"LEAF"	title	c_1
"LEAF"	author	c_1
journal1	journal	c_1
"LEAF"	title	c_1
"LEAF"	editor	c_2
"LEAF"	number	c_2
"LEAF"	title	c_2
"LEAF"	volume	c_2

(a) file 3

target	label	source
"LEAF"	author	c_1
"LEAF"	author	c_1
"LEAF"	editor	c_2
"LEAF"	number	c_2
"LEAF"	title	c_1
"LEAF"	title	c_1
"LEAF"	title	c_2
"LEAF"	volume	c_2
journal1	journal	c_1
journal1	journal	c_1

(b) file 3'

Figure 8: file 3 and file 3'

node	class
article1	c_1
article2	c_1
journal1	c_2

(a) schema_classes

source	label	target
c_1	author	LEAF
c_1	title	LEAF
c_1	journal	c_2
c_2	editor	LEAF
c_2	number	LEAF
c_2	title	LEAF
c_2	volume	LEAF

(b) schema_edges

Figure 9: schema files

a triple $(c_1, author, LEAF)$ in lines 5 to 6 again. The next six lines of file 3' are treated similarly. For the 9th line of file 3', we obtain $v = journal1, l = journal$ and $c_s = c_1$. By reading schema_classes, c_2 is identified as the class of $journal1$ (line 10). Thus, we add a triple $(c_1, journal, c_2)$ to schema_edges in line 16. We continue similarly by the end of file 3' and we obtain schema_edges as shown in Fig. 9b.

4. EXPERIMENTAL RESULTS

We implemented the algorithm in Ruby and made evaluation experiments on effectiveness of our utility value, execution time and memory usage. We use two graph datasets generated by SP^{2}Bench [11] and by Berlin SPARQL Benchmark (BSBM) [1]. SP^{2}Bench generates RDF (Notation 3) files based on DBLP, while BSBM is based on an e-commerce use case. The reason why we use these RDF tools is that (1) these tools have their explicit schemas and thus we can compare these schemas and schemas extracted by our algorithm and (2) the tools generate graphs of various sizes, which is useful to investigating the performance of our algorithm.

Table 1: 6 Graphs generated by SP^{2}Bench

	size (GB)	$\|V\|$	$\|E\|$
1	0.01	61,107	100,073
2	0.11	593,379	1,000,009
3	1.1	5,582,764	10,000,457
4	5.4	27,452,918	50,000,869
5	10.7	55,182,878	100,000,380
6	20.7	111,027,855	200,000,007

All the evaluation were executed on a machine with Intel Xeon E5-2623 v3 3.0GHz CPU, 16GB RAM, 2TB SATA HDD, and Linux CentOS 7 64bit. In the following, we use $\alpha = 0.2$, $\beta = 0.8$, and $k = 1$ unless otherwise stated. We use UNIX sort command in order to sort files externally in the preprocessing and the edge extraction, and we limited the maximum memory usage of the sort command to 1GB by using option "-S".

4.1 Experiment for SP^{2}Bench Dataset

We use 6 graphs of different sizes generated by SP^{2}Bench, ranging from 0.01 to 20.7 GB (Table 1). Each line of the files consists of triple (source, label, target). For the 6 graphs, we measured the execution time of preprocessing, class extraction and edge extraction. Table 2 shows the details of the execution time. Figure 10 plots the total execution time of preprocessing, class extraction, and edge extraction. This result means that the execution time of our algorithm is almost linear to input size.

We measured the memory usage of the algorithm for the 6th graph in Table 1. The class extraction step is the most memory consuming step, and its maximum memory usage is 3.1GB. The reason is as follows. Some of the labels occur very frequently. Specifically, every node has an edge labeled by rdf:type and the target of the edge is an RDF-class node. Thus, when we extract a class of an RDF-class node, large memory is needed since the node has a huge number of incoming edges. On the other hand, in the preprocessing and edge extraction steps, their maximum memory usages are both 1.1GB. As a result, the memory usage of the schema extraction is approximately 15% of the input graph file.

As for the effectiveness of our extended utility value, let us make a comparison of the following two cases.

- Extracting classes with the original utility value [14]
- Extracting classes with our extended utility value

The main difference between the two cases is that the former does not distinguish incoming and outgoing edges while the latter does distinguish them. Figure 11 summarizes the extracted classes of the two cases. As shown in the figure, Inproceeding, Proceeding, Article, Journal, Book, Person, and Bag nodes are grouped into the same class in the former case. On the other hand, the 8 kinds of nodes are grouped into 5 different classes in the latter case. This refinement is achieved by the distinction between incoming and outgoing edges.

Table 2: Execution time of our algorithm (SP^{2}Bench)

Graph no.	Preprocessing (s)	Class Extraction (s)	Edge Extraction (s)	Total (s)
1	1.5	5.8	1.2	8.6
2	8.9	47.4	4.0	60.2
3	84.2	464.6	36.7	585.4
4	408.3	2516.3	170.3	3094.9
5	856.5	5028.1	350.9	6235.5
6	1701.3	10298.7	823.4	12823.4

Table 3: Correct class rates (SP^{2}Bench)

(a) Classes extracted by using the original utility value

Graph no.	$\|C\|$	$\|E\|$	Correct class rate
1	4	142	0.811
2	4	188	0.747
3	4	220	0.782
4	4	244	0.872
5	4	254	0.899
6	4	144	0.915

(b) Classes extracted by using the extended utility value

Graph no.	$\|C\|$	$\|E\|$	Correct class rate
1	8	134	0.982
2	7	151	0.974
3	7	169	0.985
4	7	171	0.992
5	7	173	0.994
6	7	172	0.990

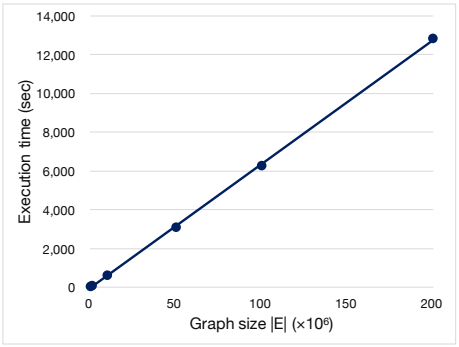
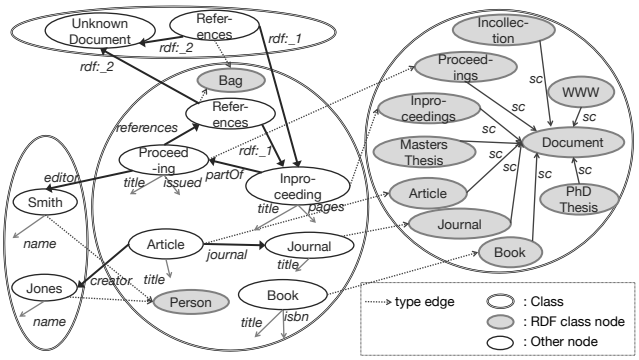
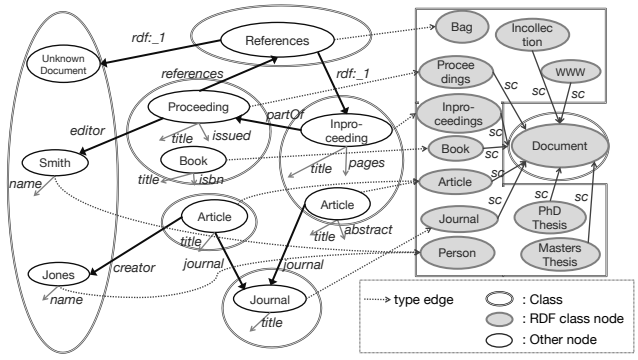


Figure 10: Total execution time of SP^{2}Bench



(a) Classes extracted by using the original utility value



(b) Classes extracted by using the extended utility value

Figure 11: Summary of results of SP^{2}Bench

To evaluate the quality of the class extraction further, we introduce the notion of *correct class rate* as follows. Since SP^{2}Bench is an RDF benchmark tool, each node in an extracted class c has an rdf-type. Let t^c_{max} be the most "major" rdf-type in c, that is, $|c(t^c_{max})| \geq |c(t)|$ for any rdf-type t, where $c(t)$ is the set of nodes in c whose rdf-type is t. Then the *correct class rate* of a set C of classes is defined as follows (class LEAF is ignored).

$$\frac{\sum_{c \in C} |c(t^c_{max})|}{\sum_{c \in C} |c|},$$

where $|c|$ denotes the number of nodes in class c. Table 3 shows the correct class rates of C in the above two cases. As shown in the table, our extended utility value brings more refined classes that gives better correct class rates.

We also examined how parameter k affects correct class rate. Figure 12 plots correct class rate values for seven different k values under the 3rd graph of Table 1. The maximum correct class rate 0.995 is obtained at $k = 4.0$, which is higher than the value 0.985 shown in Table 3b. In the case of $k = 4.0$ we have $|C| = 21$, and 16 out of 21 classes have exactly one rdf-type, that is, any nodes in the same class have the same rdf-type.

Overall, the above results suggest that the parameters α, β, and k introduced in our extended utility value works effectively for extracting classes. However, the effectiveness of parameter k should be investigated further, and this is left as a future work.

4.2 Experiment for BSBM Dataset

We used 3 graphs of different sizes generated by BSBM, ranging from 0.09 to 8.6 GB (Table 4). For the 3 graphs, we

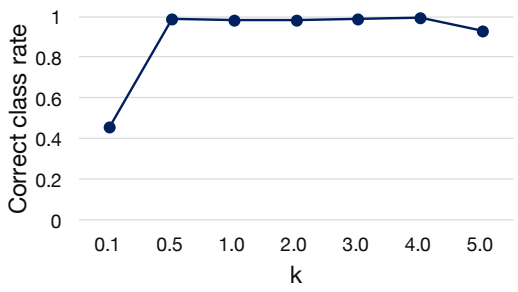


Figure 12: Correct class rate of SP^{2}Bench

Table 4: 3 graphs generated by BSBM

	size (GB)	$\|V\|$	$\|E\|$
1	0.09	116,726	374,911
2	0.86	1,007,560	3,564,773
3	8.6	8,886,078	35,272,182

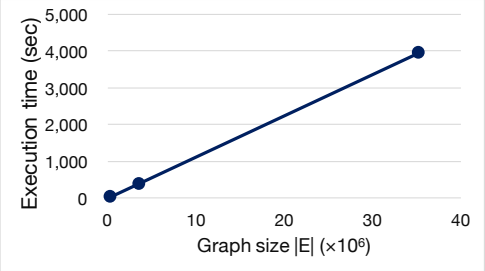


Figure 13: Total execution time of BSBM

measured the execution time of preprocessing, class extraction and edge extraction. Table 5 shows the details of the execution time. Figure 13 plots the total execution time of preprocessing, class extraction, and edge extraction. This result means that the execution time of our algorithm is almost linear to input size again.

We measured the memory usage of the algorithm for the 3rd graph in Table 4. In the class extraction step, maximum memory usage is 0.63GB. On the other hand, in the preprocessing and edge extraction steps, their maximum memory usages are both 1.1GB. As a result, the memory usage of the schema extraction is approximately 12% of the input graph file.

Table 6 shows the correct class rates for the two cases. Correct class rate of our extended utility value is 99.8% on average. As shown in the table, almost all the nodes are grouped into "correct" classes.

5. CONCLUSION

In this paper, we proposed an external memory algorithm for extracting a schema from a large graph. Our algorithm is designed so that each file is read sequentially in most cases and very few random accesses are required for schema extraction. Our algorithm consists of two steps: class extraction and edge extraction. The class extraction is to create new classes and to assign each node to a class, and the edge extraction is creating edges between classes. The experiments suggest our approach is I/O efficient and can group nodes more correctly and execution time is almost linear.

As a future work, we need to investigate the performance of our algorithm by using other datasets, to reduce used memory size, and to develop a method to update schema in responce to the graph modified. Moreover, we also need to compare our algorithm and other methods including in-memory schema extraction algorithm.

6. REFERENCES

[1] C. Bizer and A. Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems*, 5:1–24, 2009.

[2] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the Fourteenth International Conference on Data Engineering (ICDE 1998)*, pages 14–23, 1998.

[3] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A system for extracting document type descriptors from XML documents. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2000)*, pages 165–176, 2000.

[4] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. Technical Report 1997-50, Stanford InfoLab, 1997.

[5] R. Goldman and J. Widom. Approximate Dataguides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, volume 97, pages 436–445, 1999.

[6] J. Hegewald, F. Naumann, and M. Weis. Xstruct: Efficient schema extraction from multiple and large XML documents. In *Proceedings of the 22nd International Conference on Data Engineering Workshops, ICDE 2006*, page 81, 2006.

[7] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*, pages 325–336, 2013.

[8] Y. Luo, G. H. Fletcher, J. Hidders, Y. Wu, and P. De Bra. External memory k-bisimulation reduction of big graphs. In *Proc. CIKM 2013*, pages 919–928, 2013.

[9] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *Proceedings of the ACM SIGMOD international conference on Management of data (SIGMOD 2008)*, pages 419–432, 2008.

[10] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1998)*, pages 295–306, 1998.

[11] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^{2}Bench: a SPARQL Performance Benchmark. In *Proceedings of the 25th International Conference on Data Engineering (ICDE 2009)*, pages 222–233, 2009.

[12] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases*,

Table 5: Execution time of our algorithm (BSBM)

Graph no.	Preprocessing (s)	Class Extraction (s)	Edge Extraction (s)	Total (s)
1	5.9	30.6	3.7	40.1
2	58.6	294.6	23.4	376.6
3	505.4	3217.7	244.1	3967.1

Table 6: Correct class rates (BSBM)

(a) Classes extracted by using the original utility value

Graph no.	$\|C\|$	$\|E\|$	Correct class rate
1	7	52	0.963
2	7	58	0.976
3	7	58	0.969

(b) Classes extracted by using the extended utility value

Graph no.	$\|C\|$	$\|E\|$	Correct class rate
1	10	65	0.998
2	10	65	0.998
3	10	65	0.999

pages 302–314, 1999.

[13] N. Suzuki, K. Ikeda, and Y. Kwon. An algorithm for all-pairs regular path problem on external memory graphs. *IEICE Transactions*, 99-D(4):944–958, 2016.

[14] Q. Y. Wang, J. X. Yu, and K.-F. Wong. Approximate graph schema extraction for semi-structured data. In *Proceedings of the International Conference on Extending Database Technology (EDBT 2000)*, pages 302–316. Springer, 2000.

[15] Z. Zhang, J. X. Yu, L. Qin, L. Chang, and X. Lin. I/O efficient: Computing sccs in massive graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*, pages 181–192, 2013.

[16] Z. Zhang, J. X. Yu, L. Qin, Q. Zhu, and X. Zhou. I/O cost minimization: Reachability queries processing over massive graphs. In *Proceedings of the International Conference on Extending Database Technology (EDBT 2012)*, pages 468–479, 2012.