An Algorithm for Extracting Schemas from External Memory Graphs

Yoshiki SEKINE

Graduate School of Library, Information and Media Studies
University of Tsukuba

March 2018

Contents

Chapter 1

Introduction

Schema Extraction from Graph Data 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 has been widely used and various kinds of new graph data is actively being created. 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 [1], structure browsing, query formulation [3], 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 thesis we propose an algorithm for extracting a schema from a graph.

To extract an appropriate schema from a graph G, we have to choose, for each node v in G, an appropriate class that v should belong to. The utility function, used in the schema extraction method proposed by Wang et al. [13], is a major function to select such classes. However, the utility function requires a large amount of computation cost as the number of unique edge labels in a graph becomes larger. This means that the utility function is hard to be applied to many of real-world graphs, since real-world graphs tends to have a large number of unique edge labels. Therefore, we propose an algorithm that uses a new utility function that can be computed more efficiently. The function is "looser" but adequate in the sense that the function less depends on the number of unique edge labels but still output appropriate values so that appropriate schemas are extracted.

External Memory Algorithm Meanwhile, 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 I/O efficiently, 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 a class file consisting of the classes of all the nodes. In the edge extraction step, first the algorithm output an edge file consisting of edges between "nodes and classes". Then the algorithm extracts edges between classes by reading the class file and the intermediate edge file 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.

Parallelization of Class Extraction Algorithm Our schema extraction algorithm greedily chooses the class for each node sequentially in the class extraction by using the utility function, which requires the most amount of computation cost in the algorithm. Therefore, in order to reduce the calculation time we also design the class extraction algorithm as a parallel processing algorithm. It means that we sequentially read k nodes and choose the class of each node by calculating the utility function executed in parallel processes. The parallelized class extraction algorithm chooses classes in a similar way to the single processing (non-parallel) algorithm and handles a input file by only sequential read.

Evaluation Experiment We conducted some experiments on our schema extraction algorithm by using RDF benchmark and DBPedia graphs. The results suggest that our algorithm can extract schemas from these graphs more efficiently and appropriately than the previous utility function, and that parallelization of the class extraction makes the execution time faster for DBPedia.

Related Work A number of schema extraction algorithms for graphs have been proposed. DataGuide [3] extracts a schema by grouping nodes reachable from the root via the same label path into the same class. ApproximateDataguite [4] is the approximate version of DataGuide. Nestorov et al. proposes an algorithm for extracting a set of classes by using a clustering approximation method [9]. Wang et al. proposes an algorithm that extracts a schema by an incremental clustering method [13]. 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 [8]. 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 [7]. 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 [11] extracts a relational schema from given DTDs. The XTRACT system [2] extracts a DTD as a schema from given XML documents. XStruct [5] 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 [6], strongly connected components [14], graph reachability [15], and regular path query [12]. To the best of our knowledge, however, no external memory algorithm for schema extraction has been proposed so far.

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

Chapter 2

Preliminaries

In this chapter, we define graph and related notions.

2.1 Labeled Directed Graph

Let L be a set of labels. A *labeled directed graph* (*graph* for short) is denoted $G = (V, E)$, where V is a set of *nodes* and $E \subseteq V \times L \times V$ is a set of *labeled directed edges* (*edges* for short). Let $e \in E$ be an edge labeled by $l \in L$ from a node $v \in V$ to a node $u \in V$. Then e is denoted (v, l, u), v is called *source*, u is called *target* of e, and we say that v has the edge e. The set of outgoing edge labels of v, namely the set of labels which v has, is denoted $L(v)$.

Example 1. *Figure 2.1 represents a fragment of books information and is denoted* $G = (V, E)$*, where*

$$\begin{aligned}
V &= \{\text{author1, book1, book2, “Person1”, “Book 1”, “Book 2”, “1”, “2”}\},\\
E &= \{(\text{author1, name, “Person 1”}),\\
&\quad (\text{author1, is-author-of, book1}),\\
&\quad (\text{author1, is-author-of, book2}),\\
&\quad (\text{book1, author, author1}),\\
&\quad (\text{book2, author, author1}),\\
&\quad (\text{book1, title, “Book 1”}),\\
&\quad (\text{book2, title, “Book 2”}),\\
&\quad (\text{book1, number, “1”}),\\
&\quad (\text{book2, number, “2”})\},\\
L &= \{\text{name, author, is-author-of, title, number}\},\\
L(\text{author1}) &= \{\text{name, is-author-of}\},\\
L(\text{book1}) &= \{\text{author, title, number}\},\\
L(\text{book2}) &= \{\text{author, title, number}\}.
\end{aligned}$$

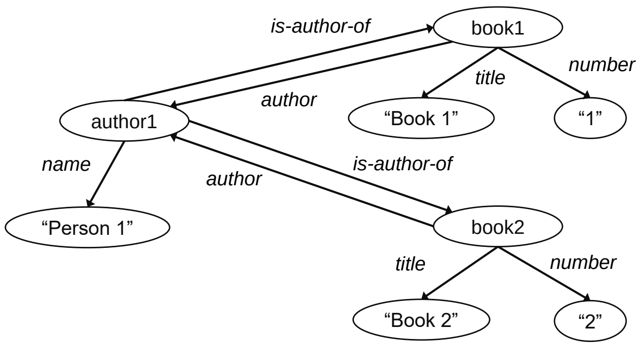

Figure 2.1: Example of a graph

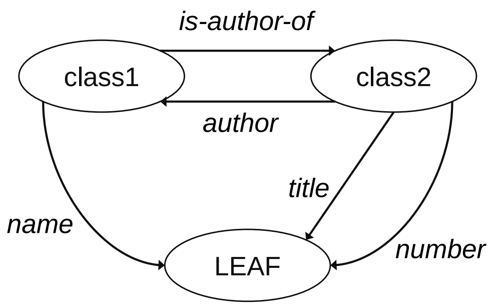

Figure 2.2: Example of a schema

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 graph is mapped to a class in a schema. We assume that every text node belongs to a single class LEAF. For a node v in a (instance) graph, by $class(v)$ we mean the class that v belongs to. A schema is denoted $S = (C, E_s)$, where C is a set of classes and E_s is a set of edges between classes.

Figure 2.2 shows an example of a schema extracted from the graph of Figure 2.1, where book1 and book2 belong to class2, author1 belongs to class1, and the other nodes belong to LEAF. The schema is shown in Example 2.

Example 2. *Consider the graph schema S in Figure 2.2. Then S is defined as a pair*

source	label	target
author1	name	“Person1”
author1	is-author-of	book1
author1	is-author-of	book2
book1	author	author1
book2	author	author1
book1	title	“Book1”
book2	title	“Book2”
book1	number	“1”
book2	number	“2”

Figure 2.3: A graph file of the graph in Figure 2.1

(C, E_s), *where*

$$\begin{aligned}
C &= \{\text{class1, class2, LEAF}\},\\
E_s &= \{(\text{class1, is-author-of, class2}),\\
&\quad (\text{class2, author, class1}),\\
&\quad (\text{class1, name, LEAF}),\\
&\quad (\text{class2, title, LEAF}),\\
&\quad (\text{class2, number, LEAF})\},\\
\mathit{class}(\text{author1}) &= \text{class1}\\
\mathit{class}(\text{book1}) &= \mathit{class}(\text{book2}) = \text{class2}\\
L &= \{\text{name, author, is-author-of, title, number}\}\\
L(\text{class1}) &= \{\text{is-author-of, name}\}\\
L(\text{class2}) &= \{\text{author, title, number}\}
\end{aligned}$$

2.3 Graph and Schema Files

In this thesis, we assume that a graph is stored in a graph file like N-Triples format, which is a container for Resource Description Framework (RDF) data. Each line of a graph file corresponds to an edge, namely, a line consists of $(source, label, target)$. Figure 2.3 shows a graph file that stores the graph of Fig. 2.1.

We assume that a schema is composed of two files denoted schema_classes and schema_edges. The former stores pairs of a node and its class, namely a line consists of $(node, class)$. Note that we do not store leaf nodes and their class LEAF in this file since they are not needed for schema extraction. The latter stores edges between classes, in which each line consists of $(\mathit{source\ class}, \mathit{label}, \mathit{target\ class})$. Figure 2.4 shows an example of the two files representing the schema of Fig. 2.2.

node	class
author1	class1
book1	class2
book2	class2

(a) schema_classes

source class	label	target class
class1	is-author-of	class2
class1	name	LEAF
class2	author	class1
class2	title	LEAF
class2	number	LEAF

(b) schema_edges

Figure 2.4: Example of schema files

Chapter 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 to create edges between classes. In this chapter, we first define our utility function used in the class extraction, then we describe the schema extraction algorithm.

3.1 Utility Function

Wang et al. proposes an algorithm that extracts a graph schema by grouping nodes having a similar set of edge labels into the same class using the utility function [13]. However, the utility function requires a large amount of calculation cost for graphs containing a large number of unique edge labels. To cope with the problem, we define a new utility function, called *light utility function*, so that we can extract schemas from such graphs more efficiently.

Let v be a node, c be a class, and C be a set of classes. In this thesis, we ignore incoming edges of v to make our algorithm simple and to reduce calculation cost. The set of edge labels of v is denoted $L(v)$. By $L(c)$ we mean the set of edge labels of c, that is,

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

Let $|c|$ be the number of nodes in c and $nodes(c,l)$ be the set of nodes in c having an edge labeled by l. By $nodes(C,l)$ we mean the set of nodes in C having an edge labeled by l, that is,

$$nodes(C,l) = \bigcup_{c \in C} nodes(c,l).$$

Then the *light utility function* (*utility function*, for short), denoted $U(C,v,c_i)$, is defined as the product of the Dice coefficient and the mean of the ratio of $|nodes(c,l)|$ to $|nodes(C,l)|$, that is,

$$U(C,v,c_i) = Dice(L(v),L(c_i))^{\alpha} \frac{1}{|L(v)|} \sum_{l \in L(v)} \frac{|nodes(c_i,l)|}{|nodes(C,l)|},$$

where α is a parameter to control which of $Dice$ and the latter ratio is emphasized when extracting classes. U becomes higher if nodes having similar edge labels are grouped into the same class. By this definition we intend to balance how labels of a node v are similar with labels of a class c_i and how much labels of c_i occupy the entire schema. We need to extract classes so that the classes bring a high value of the utility function.

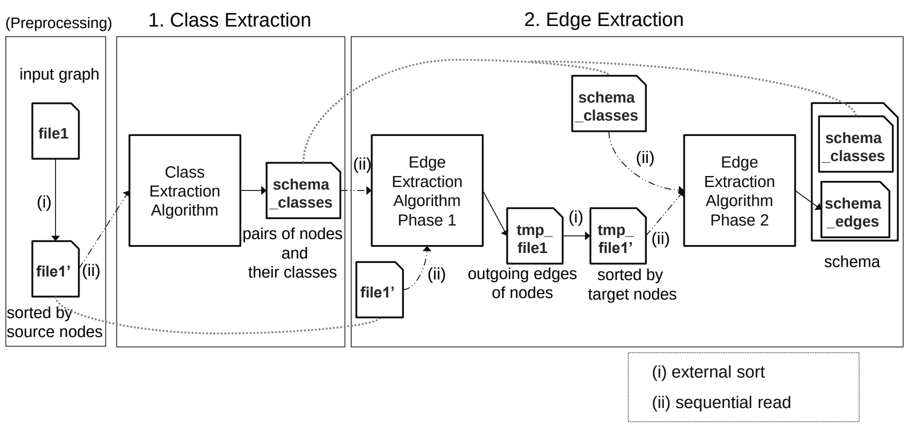

Figure 3.1: Outline of our schema extraction algorithm

3.2 Overview

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 a class file consisting of the classes of all the nodes. Next our algorithm creates an edge file having information required for extracting edges between classes. Note that since the information for edge extraction must include the all edges, 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. 2.3, our algorithm extracts a schema of the graph. An outline of our algorithm is as follows (see Fig. 3.1).

Input: graph file. As shown in Figure 2.3, each line of the file represents an edge, namely, a line consists of the source, the label, and the target of an edge. In the following, we call the input graph file file 1.

Output: schema_classes and schema_edges.

1. **Preprocessing**
   (a) Sort file 1 externally. Let file 1′ be the resulting file (see Figure 3.2).
2. **Class Extraction**
   (a) Read file 1′ sequentially and extract the class of each node based on the utility function.
   (b) Each time the class of a node is extracted, output the node and the class to schema_classes.
3. **Edge Extraction**

source	label	target
author1	is-author-of	book1
author1	is-author-of	book2
author1	name	"Person1"
book1	author	author1
book1	number	"1"
book1	title	"Book1"
book2	author	author1
book2	number	"2"
book2	title	"Book2"

Figure 3.2: file 1'

(a) Read file 1' and schema_classes concurrently and sequentially, and output the outgoing edges of nodes (source nodes are replaced by their classes) to another file tmp_file1.

(b) Sort tmp_file1 externally. Let tmp_file1' be the resulting file. Each line of the file consists of edges between classes and nodes.

(c) Read schema_classes and tmp_file1' concurrently and sequentially, and replace the target node of each edge in tmp_file1' with its class. This results in edges between classes, which are written into schema_edges.

In the following, we give the details of our class extraction and edge extraction algorithms.

3.3 Class Extraction

Let us present the outline of our class extraction algorithm. As shown below, our algorithm greedily chooses, for each node $v \in V$, the class that v belongs to. Actually, the algorithm is designed as an external memory algorithm using parallel processing (details are explained later).

Input: a graph $G = (V, E)$

Output: a set C of classes

```
1. $C \leftarrow \emptyset$.
2. **for each** node $v \in V$ **do**
   (a) **for each** class $c_i \in C$ **do**
       i. Calculate $U(C, v, c_i)$.
   (b) Let $c_v$ be the new class such that $c_v$ has the same set of outgoing edges as $v$
       (edges having the same label are merged into one).
   (c) Calculate $U(C, v, c_v)$.
   (d) Let $class(v)$ be the class such that the value of $U$ is the highest among $C \cup \{c_v\}$.
   (e) $C \leftarrow C \cup \{class(v)\}$.
3. **return** $C$.
```

Algorithm 1 Class Extraction Algorithm (non-parallel version)

```
Input: file 1'
Output: schema_classes
1: $C \leftarrow \emptyset$.
2: while file 1' does not reach EOF do
3:     $L(v) \leftarrow$ the set of outgoing edge labels of $v$ obtained by reading file 1'.
4:     $class(v) \leftarrow \textsc{ClassDetermination}(C, v, L(v))$.
5:     Add a pair $(v, class(v))$ to schema_classes.
6:     $C \leftarrow C \cup \{class(v)\}$.
7: end while
```

We now present the details of our class extraction algorithm. This algorithm is designed as (1) an external memory algorithm to handle large graphs that cannot fit in main memory, and (2) a parallel processing algorithm to reduce the calculation time of utility function. In the class extraction, calculating utility function requires a large amount of computation cost. Therefore we design the class extraction as a parallel process algorithm. We first show a single process algorithm and next show a parallel process algorithm. Both algorithms are also designed as an external memory algorithm. (Note that incoming edges are ignored.)

Data Maintained in Main Memory Values $|nodes(c, l)|$ for each class $c \in C$ and $|nodes(C, l)|$ are kept in main memory to calculate utility function until the classes of all nodes are extracted. That is, we store the number of nodes having an edge labeled by l (1) in $L(c)$ for each class $c \in C$, (2) and in C. On the other hand, for a node v, v's name and v's edge labels $L(v)$ is 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 $L(v)$ are discarded.

Preprocessing The input graph file called file 1 is a list of edges (Figure 2.3). 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 (Figure 3.2). 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.

Class Extraction Algorithm (non-parallel version) Algorithm 1 shows the procedure of the single process (non-parallel) class extraction. An input file, file 1' is a sorted file obtained in the preprocessing. Let v be the node that is currently read. By reading file 1' sequentially, we obtain the edges v has and extract the class of v based on the utility function. This process is repeated until file 1' reaches the end of file.

Specifically, Algorithm 1 works as follows. We read file 1' sequentially and obtain $L(v)$ that is the set of edge labels of v. Note that the edges having the same label are merged into one. ClassDetermination shown in Algorithm 2 is called each time $L(v)$ is obtained in line 4, and it extracts the class of v as follows. We calculate the following utility function and choose the class for which the maximum value is obtained. In particular, if $C = \emptyset$, then the first calculation in lines 2 to 4 is skipped.

- The utility function assuming that v belongs to $c \in C$, for each class c extracted so far.

Algorithm 2 Class Determination

```
1: procedure CLASSDETERMINATION($C, v, L(v)$)
2:     for each class $c_i \in C$ do
3:         Calculate $U(C, v, c_i)$.
4:     end for
5:     Let $c_v$ be the new class having the same set of outgoing edges as $v$ (edges having
           the same label are merged into one).
6:     Calculate $U(C, v, c_v)$.
7:     Let $class(v)$ be the class such that the value of $U$ is the highest among $C \cup \{c_v\}$.
8:     return $class(v)$.
9: end procedure
```

- The utility function assuming that v belongs to a new class c_v having the same edges as v.

Each time $class(v)$ is extracted, we output a pair $(v, class(v))$ to schema_classes in line 5. Finally, we update C, $|nodes(c, l)|$ and $|nodes(C, l)|$ in main memory assuming that the node v belongs to $class(v)$ in line 6. Repeating this process until the input file reaches EOF, we obtain the classes of all nodes.

Class Extraction Algorithm (parallelized version) We parallelize the class extraction algorithm in order to reduce calculation time of the utility function that requires the most amount of computation cost in the class extraction.

Algorithm 3 shows the procedure of the parallelized class extraction. We parallelize the process of calculating the utility function. Like the non-parallel algorithm, an input file, file 1', is a sorted file obtained in the preprocessing. By reading file 1' sequentially, we read k nodes, obtain the edges of them. Then we extract classes of k nodes based on the utility function in k parallel processes. After the parallel processes, some nodes are assigned to existing classes $c \in C$, and the other nodes assigned to the new class c_v. While we confirm that nodes assigned to existing classes belong to the classes, we must determine whether some of nodes assigned to the new class c_v should be merged into or remain. We call this recalculation process *conflict resolution*. By doing that, we finally obtain the classes of k nodes completely. This process is repeated until file 1' reaches the end of file.

Specifically, Algorithm 3 works as follows. We read file 1' sequentially and read k nodes, where k is set to the parallel number. Typically, k is set to the number of cores of a CPU. Let $v_1 \dots v_k$ be the k distinct nodes occurring just after the node currently read. We obtain $L(v_i)$ for every $1 \leq i \leq k$, which is the set of edge labels of v_i in lines 5 to 8 (Note that the edges having the same label are merged into one). Running the codes in lines in 10 to 13 in k parallel processes, we call CLASSDETERMINATION shown in Algorithm 2 to extract the class of v_i based on the utility function. Note that C is not updated during the parallel block. We only refer to C in order to calculate the utility function assuming that nodes belong to each class. When the parallel block is completed, we have a set R of results whose element is a pair $(v_i, class(v_i))$. A set of nodes belonging to c_v is denoted V_{one}. If $|V_{one}| > 1$, then a conflict occurs and CONFLICTRESOLUTION (Algorithm 4) is called. If not, the results in R are confirmed and skip the conflict resolution in lines 15 to 19.

Let us explain the procedure of CONFLICTRESOLUTION. We select the class of each node belonging to existing classes and store them in R' in line 3. (C, $|nodes(c, l)|$ for each

Algorithm 3 Class Extraction Algorithm (parallelized version)

```
Input: file 1'
Output: schema_classes
 1: $C \leftarrow \emptyset$.
 2: $k$ is set to the parallel number.
 3: while file 1' does not reach EOF do
 4:     $V_{tmp} \leftarrow \emptyset$.
 5:     Repeat $k$ times
 6:         $L(v_i) \leftarrow$ the set of outgoing edge labels of $v_i$ obtained by reading file 1'.
 7:         $V_{tmp} \leftarrow V_{tmp} \cup \{v_i\}$.
 8:     End Repeat
 9:     $R \leftarrow \emptyset$.
10:     Parallel for each $v_i \in V_{tmp}$
11:         $class(v_i) \leftarrow$ ClassDetermination$(C, v_i, L(v_i))$.
12:         $R \leftarrow R \cup \{(v_i, class(v_i))\}$.
13:     End Parallel
14:     $V_{one} = \{v \mid (v, class(v)) \in R$ such that $class(v) = c_v\}$.
15:     if $|V_{one}| > 1$ then                                    ▷ conflict occurs
16:         ConflictResolution$(C, R, V_{tmp}, V_{one})$.
17:     else
18:         $C \leftarrow C \cup \{class(v_1), \ldots, class(v_k)\}$.
19:     end if
20:     For each node $v_i \in V_{tmp}$, add a pair $(v_i, class(v_i))$ to schema_classes.
21: end while
```

$c \in C$, and $|nodes(C, l)|$ are updated by these results in line 7.) The first node whose class is c_v is selected in line 4 and add it to R' in line 5. Then in lines 6 to 7 we update C, C_{tmp}, $|nodes(c, l)|$ for each $c \in C$ and each $c \in C_{tmp}$, and $|nodes(C, l)|$ in main memory assuming that the node v selected in line 4 belongs to $class(v)$. For each node $v \in V_{one}$, namely each node in V_{tmp} assigned to c_v except the first node, we recalculate the utility function. We call ClassDetermination2 (Algorithm 5) instead of ClassDetermination so that the existing classes are computed to C_{tmp} only. ClassDetermination2 extracts the class of $v \in V_{one}$ as follows. We calculate the following utility function and choose the class for which the maximum value is obtained.

- The utility function assuming that v belongs to $c \in C_{tmp}$, for each class c extracted so far in a conflict resolution.
- The utility function assuming that v belongs to a new class c_v having the same edges as v.

Note that we recalculate c_v to obtain the utility value from the latest schema because the schema is updated and the resulting utility value may be different. Now the class extraction of v is completed. Each time the class is selected, we update C, C_{tmp}, $|nodes(c, l)|$ for each $c \in C$ and each $c \in C_{tmp}$, and $|nodes(C, l)|$ in main memory assuming that the node v belongs to $class(v)$ in lines 10 to 12. Extracting the class of every node in V_{one} completely, we return R' and back to Algorithm 3.

Finally, we output k pairs $(v_1, class(v_1)), \ldots, (v_k, class(v_k))$ to schema_classes in line 20. Repeating this process until the input file reaches EOF, we obtain the classes of all nodes.

Algorithm 4 Conflict Resolution

```
1: procedure ConflictResolution($C, R, V_{tmp}, V_{one}$)
2:     $C_{tmp} \leftarrow \emptyset$.
3:     $R' = \{(v, class(v)) \in R \mid v \notin V_{one}\}$.        ▷ conflict resolution result
4:     Remove the first node of $V_{one}$. Let $v$ be the first node.
5:     $R' \leftarrow R' \cup \{(v, class(v))\}$.
6:     $C_{tmp} \leftarrow C_{tmp} \cup \{class(v)\}$.
7:     $C \leftarrow C \cup \{class(v) | (v, class(v)) \in R'\}$.
8:     for each $v \in V_{one}$ do
9:         $class(v) \leftarrow$ ClassDetermination2($C, C_{tmp}, v, L(v)$).
10:        $R' \leftarrow R' \cup \{(v, class(v))\}$.
11:        $C_{tmp} \leftarrow C_{tmp} \cup \{class(v)\}$.
12:        $C \leftarrow C \cup \{class(v)\}$.
13:    end for
14:    return $R'$.
15: end procedure
```

Algorithm 5 Class Determination in Conflict Resolution

```
1: procedure ClassDetermination2($C, C', v, L(v)$)
2:     for each class $c_i \in C'$ do
3:         Calculate $U(C, v, c_i)$.
4:     end for
5:     Let $c_v$ be the new class having the same set of outgoing edges as $v$ (edges having
           the same label are merged into one).
6:     Calculate $U(C, v, c_v)$.
7:     Let $class(v)$ be the class such that the value of $U$ is the highest among $C' \cup \{c_v\}$.
8:     return $class(v)$.
9: end procedure
```

3.4 Edge Extraction

In this edge extraction step, we replace nodes in the input graph file by their extracted classes. To do that sequentially, first we use the sorted input graph file and schema_classes, and create an intermediate file, in which source nodes are replaced by their classes (Phase 1). Then by using the intermediate file and schema_classes, we replace target nodes by their classes (Phase 2).

Edge Extraction Phase 1 This phase replaces the source node of each edge by the class of the source node. Algorithm 6 shows the procedure of the edge extraction phase 1. As shown in Figure 3.2, file 1′ is a sequence of triples (*source*, *label*, *target*), and the edge extraction phase 1 is done by replacing *source* of each triple by $class(source)$. Since file 1′ is sorted, the edges having the same source appear consecutively in file 1′, which enables source nodes to be replaced consecutively. Let v be the source node of the "current" edge read from file 1′, and suppose that $class(v)$ is obtained from schema_classes. Then we can replace the source of every edge whose source is v by $class(v)$, which can be done by a sequential read from file 1′.

Specifically, we read a line $(v_s, class(v_s))$ from schema_classes and read a line (v, l, u) from file 1′, where v is the source node, l is the label, and u is the target node in line 3.

target	label	source
book1	is-author-of	class1
book2	is-author-of	class1
“leaf”	name	class1
author1	author	class2
“leaf”	number	class2
“leaf”	title	class2
author1	author	class2
“leaf”	number	class2
“leaf”	title	class2

(a) tmp_file 1

target	label	source
“leaf”	name	class1
“leaf”	number	class2
“leaf”	number	class2
“leaf”	title	class2
“leaf”	title	class2
author1	author	class2
author1	author	class2
book1	is-author-of	class1
book2	is-author-of	class1

(b) tmp_file 1'

Figure 3.3: Intermediate files created in edge extraction

Algorithm 6 Edge Extraction Phase 1

```
Input: file 1' and schema_classes
Output: tmp_file1                                  ▷ sources are replaced by their class
 1: Read a line from schema_classes. Let $v_s$ be the node and $class(v_s)$ be the class of the
       line.
 2: while file 1' does not reach EOF do
 3:     Read a line from file 1'. Let $v, l, u$ be the source, the label, and the target of the
           line, respectively.
 4:     if $v = v_s$ then
 5:         Add a triple $(u, l, class(v_s))$ to tmp_file1 ($u$ is replaced by “leaf” if the target
               $u$ is a leaf node).
 6:     else
 7:         Read schema_classes sequentially and find a line $(v_s, class(v_s))$ such that $v_s = v$.
 8:         Add a triple $(u, l, class(v_s))$ to tmp_file1 ($u$ is replaced by “leaf” if the target
               $u$ is a leaf node).
 9:     end if
10: end while
```

If the class of v is already known, then v is replaced by the class in lines 5. Otherwise, by reading schema_classes sequentially, we obtain the class of v and replace v of each edge in file 1' by the class in line 7. This results in edges between classes and nodes (if the target node u is a leaf, we use “leaf” instead of u since the content is not needed in the next phase). Then we swap the source and the target of the edge, and output the edge to tmp_file 1. Note that we output the triple (target node, label, source class) in this order because in the next phase we sort tmp_file 1 by target nodes and replace target nodes by their classes. Repeating this until file 1' reaches EOF, we obtain the edges between classes and nodes as shown in Figure 3.3a.

Edge Extraction Phase 2 This phase replaces the target node of each edge obtained in Phase 1 by the class of the target node. Algorithm 7 shows the procedure of the edge extraction phase 2. We assume that every text leaf node belongs to a particular class called $LEAF$ and that every non-text leaf node belongs to another particular class called $LEAF2$. Actually, this phase is done in a similar way to the phase 1. As shown in Figure 3.3a, tmp_file 1 is a sequence of triples ($target$, $label$, $class$), and the edge extraction

Algorithm 7 Edge Extraction Phase 2

```
Input: tmp_file 1, schema_classes
Output: schema_edges
 1: Sort tmp_file 1. Let tmp_file 1' be the resulting file.
 2: Read a line from schema_classes. Let $v_t$ be the node and $class(v_t)$ be the class of the
        line.
 3: Read a line from tmp_file 1'. Let $u, l, c_s$ be the target, the label, and the source of the
        line, respectively.
 4: while tmp_file 1' does not reach EOF do
 5:     if $u =$ "leaf" then                                        ▷ $u$ is a text leaf node
 6:         Add a triple $(c_s, l, LEAF)$ to schema_edges.
 7:         Read a line from tmp_file 1'. Let $u, l, c_s$ be the target, the label, and the source
                of the line, respectively.
 8:     else if $u < v_t$ then                                  ▷ $u$ is a non-text leaf node
 9:         Add a triple $(c_s, l, LEAF2)$ to schema_edges.
10:         Read a line from tmp_file 1'. Let $u, l, c_s$ be the target, the label, and the source
                of the line, respectively.
11:     else if $u > v_t$ then
12:         Read a line from schema_classes. Let $v_t$ be the node and $class(v_t)$ be the class
                of the line.
13:     else if $u = v_t$ then
14:         Add a triple $(c_s, l, class(v_t))$ to schema_edges.
15:         Read a line from tmp_file 1'. Let $u, l, c_s$ be the target, the label, and the source
                of the line, respectively.
16:     end if
17: end while
```

is done by replacing *target* of each triple by *class*(*target*). To do this, the algorithm first sorts tmp_file 1 and obtain tmp_file 1' as the result, as shown in Figure 3.3b in line 1. Since tmp_file 1' is sorted, the edges having the same target appear consecutively in tmp_file 1', which enables target nodes to be replaced consecutively. Let v be the target node of the "current" edge read from tmp_file 1', and suppose that the $class(v)$ is obtained from schema_classes. Then we can replace the target of every edge whose target is v by $class(v)$, which can be done by a sequential read from tmp_file 1'.

Specifically, we read a line (u, l, c_s) from tmp_file 1', where u is the target node, l is the label, and c_s is the source class in line 3. If the class of u is already known, then u is replaced by the class in lines 5 to 10. Otherwise, by reading schema_classes sequentially, we obtain the class of u and replace u of each edge in tmp_file 1' by the class in lines 11 to 14. Repeating this until tmp_file 1' reaches EOF, we obtain the edges between classes.

3.5 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 externally: $\mathcal{O}(sort(|E|))$

2. **Class Extraction**
   (a) Reading file 1': $\mathcal{O}(|E|/B)$
   (b) Writing pairs of a node and its class to schema_classes: $\mathcal{O}(|V|/B)$
3. **Edge Extraction Phase 1**
   (a) Reading file 1': $\mathcal{O}(|E|/B)$
   (b) Reading schema_classes: $\mathcal{O}(|V|/B)$
   (c) Writing outgoing edges to tmp_file 1: $\mathcal{O}(|E|/B)$
4. **Edge Extraction Phase 2**
   (a) Sorting tmp_file 1 externally: $\mathcal{O}(sort(|E|))$
   (b) Reading tmp_file 1': $\mathcal{O}(|E|/B)$
   (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.

3.6 Example of Our Algorithm

Let us show an example of our algorithm of parallel implementation. Let file 1 be the file in Figure 2.3, which is obtained from the graph in Figure 2.1. Texts enclosed in double quotes are treated as leafs.

First, in the preprocessing of the class extraction, we sort file 1 and obtain file 1' in Figure 3.2. Next, Algorithm 3 performs the class extraction step as follows. We set $k = 2$ for this simple example. We then read file 1' sequentially and obtain a set $L(v_1)$ that stores all edges of v_1 = author1 in lines 5 to 8. Since $k = 2$, we read the next node v = book1 and obtain $L(v_2)$ that is a set of all labels of edges of book1. Then we have $V_{tmp} = \{\text{author1,book1}\}$.

The parallel block in lines 10 to 13 is executed as follows. (1) Since $|C| = 0$, author1 belongs to a new class $c_{author1} = \text{class1}$ through Algorithm 2. (2) Similarly, since $|C| = 0$, book1 belongs to a new class $c_{book1} = \text{class1}$. The two parallel processes finishing, we have $R = \{(\text{author1}, c_{author1} = \text{class1}), (\text{book1}, c_{book1} = \text{class1})\}$. Note that $c_{author1}$ and c_{book1} are different; however the class number of both being 1, they are numerically identical. That is, a conflict occurs. Let V_{one} be a set of nodes in V_{tmp} whose class are the new class c_v and we have $V_{one} = \{\text{author1,book1}\}$ in line 14.

Since $|V_{one}| = 2 > 1$, we go to ConflictResolution shown in Algorithm 4 as follows. Since we have not updated the schema, $|C| = 0$. Since there is no node belonging to existing classes, $R' = \emptyset$ in line 3. We remove the first element of V_{one} and let it be v = author1 in line 4. Since author1 is the first element of V_{one}, it is confirmed that $class(\text{author1}) = c_{author1} = \text{class1}$ in line 5. Then updating the schema in lines 6 to 7, we have $C_{tmp} = C = \{\text{author1}\}$ and $R' = \{(\text{author1}, c_{author1})\}$. The question is whether book1 should be assigned to $c_{author1}$ or c_{book1}. Through CLASSDETERMINATION2$(C, C_{tmp}, v, L(v))$ shown in Algorithm 5, we calculate following utility function and choose the class for which the maximum value is obtained. (1) book1 belongs to $c_{author1}$, (2) book1 belongs to c_{book1}. Finally we determined that $class(\text{book1}) = c_{book1} = \text{class2}$. The conflict resolution finished, we return to Algorithm 3. We output the two pair (author1, class1) and (book1, class2) to schema_classes in line 20. We continue in a similar manner by the end of file 1′ and we obtain schema_classes in Figure 2.4a.

Algorithm 6 performs the edge extraction phase 1. We read a line from file 1′ and obtain a triple (author1, is-author-of, book1). By reading schema_classes, class1 is identified as the class of author1. We replace the source of the triple with class1, swap the source and the target, and add a triple (book1, is-author-of, class1) to tmp_file 1 in line 5. We continue in a similar manner (and replace the target with "leaf" if the target is a leaf) by the end of file 1′ and we obtain tmp_file 1 in Figure 3.3a.

Algorithm 7 performs the edge extraction phase 2 as follows. We sort tmp_file 1 and obtain tmp_file 1′ in Figure 3.3b as the result in line 1. We read a line from tmp_file 1′ and obtain the triple ("leaf", name, class1). Since the target is a text leaf, add a triple (class1, name, LEAF) to schema_edges in lines 5 to 6. We next read a line from tmp_file 1′ in line 7 and add a triple (class2, number, LEAF) to schema_edges in line 6 similarly. The next three lines of tmp_file 1′ are treated similarly. For the 6th line of tmp_file 1′, we obtain (author1, author, class2). By reading schema_classes, class2 is identified as the class of author1 in line 13. Thus, we add a triple (class2, author, class1) to schema_edges in line 14. We continue similarly by the end of tmp_file 1′ and we obtain schema_edges as shown in Fig. 2.4b.

Chapter 4

Evaluation Experiment

In this chapter, we present experimental results on our algorithm. The algorithm was implemented in Ruby 2.4.2. The parallelized class extraction was implemented by Ruby Gem parallel (version 1.12.0)[1]. All the evaluation experiments were executed on a machine with Intel Xeon E5-2623 v3 3.0GHz CPU, 16GB RAM, 2TB SATA HDD, and Linux CentOS 7 64bit. We used 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".

In our experiments, we use the following two datasets.

SP^{2}Bench SP^{2}Bench [10] (SP2B, for short) is a benchmark tool and generates RDF (N-Triples) files based on DBLP, a computer science bibliography. We generate four graphs of different sizes in Table 4.1. In the following, by V^*, we mean the non-leaf nodes in a set V of nodes. Thus classes of nodes in V^* are extracted.

Figure 4.1 shows an overview of an SP2B graph. The total number of unique RDF types is 12. Nodes whose RDF type is "Article" are the largest number of nodes. Note that we regard every edge label *rdf:_i* as the same regardless of the value of i, since the number i is not important.

The reason why we use this tool is that (1) it has its explicit schema and thus we can compare the schema and the schema extracted by our algorithm and (2) the tool generates graphs of various sizes, which is useful to investigating the performance of our algorithm.

DBPedia DBPedia project extracts structured data from Wikipedia. Among the real-world graphs, it is one of the datasets with the largest number of unique edge labels. We downloaded three benchmark dataset graphs [2] and created another graph with $|E| = 50,000$, which is the first 50,000 lines of the smallest graph of the three. Thus we use the

[1]https://github.com/grosser/parallel
[2]http://benchmark.dbpedia.org/

Table 4.1: Graphs generated by SP^{2}Bench

$\|E\|$	$\|V^*\|$	$\|L\|$	size (GB)
100,073	19,369	24	0.01
1,000,009	187,066	24	0.10
10,000,457	1,730,250	26	1.04
100,000,380	17,823,525	26	10.35

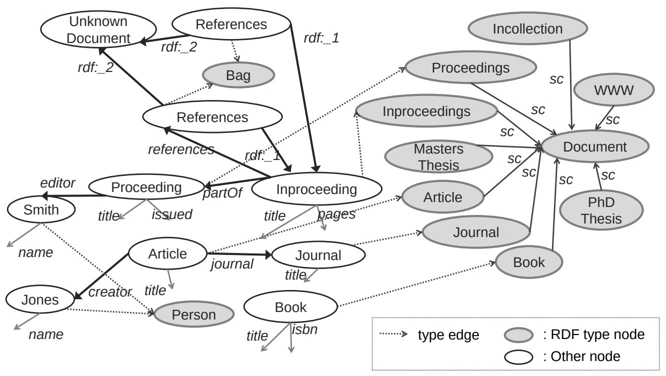

Figure 4.1: Overview of an SP2B graph

Table 4.2: Graphs from DBPedia

$\lvert E\rvert$	$\lvert V^*\rvert$	$\lvert L\rvert$	size (GB)
50,000	1,077	2,772	0.01
15,373,833	313,036	14,130	2.72
76,868,920	1,177,165	22,147	12.80
153,737,783	1,457,983	23,343	25.11

four RDF (N-Triples) graphs in Table 4.2. The total number of unique RDF types in the graph with $|E| = 15,373,833$ is 54,736.

In the following, first we give the evaluation of the class extraction since this is the most complex and time-consuming process. Then, we give the evaluation of the preprocessing and the edge extraction. To evaluate the class extraction quality, we introduce two scores *Score1* and *Score2*, based on RDF types assigned to each node. SP2B is an RDF benchmark tool and each non-leaf node in graphs generated by SP2B has one RDF type. On the other hand, each non-leaf node in DBPedia has one or more RDF types. In the following definition, two particular classes LEAF and LEAF2, which leaf nodes belong to, are omitted.

Score1 becomes larger as extracted classes contain smaller numbers of different types. By *types*(v) we mean the set of types assigned to v. The set of nodes having type t in class c is denoted *nodes*(t, c). Then $Score1$ is defined as follows.

$$Score1 = \frac{1}{|V^*|} \sum_{v \in V^*} \frac{1}{|types(v)|} \sum_{t \in types(v)} \frac{|nodes(t, class(v))|}{|class(v)|}.$$

Score2 becomes larger as a type is distributed to smaller numbers of different classes. Let *total*(t) be the total number of nodes having type t, and let $max(t) = \max_c nodes(t, c)$. Then $Score2$ is the mean of ratio of the two, that is,

$$Score2 = \frac{1}{|T|} \sum_{t \in T} \frac{max(t)}{total(t)}.$$

Thus, the more nodes having type t are grouped into the same class, the higher $Score2$ is.

Firstly, we give the evaluation of the parallelization of the class extraction. We measured the execution time and the memory usage of the class extraction algorithm (non-parallel version and the parallelized version), and calculated the class extraction scores. In this experiment, we set the parameter α to 1 in light utility function and the parallel number k to 4.

Table 4.3: Execution time of the class extraction (non-parallel and parallelized versions) for SP2B graphs

SP2B	$\lvert E\rvert$		
parallel number	1,000,009	10,000,457	100,000,380
1 (non-parallel)	6.70	64.23	651.07
4 (parallelized)	209.31	1,669.85	19,886.80

Table 4.4: Execution time of the class extraction (non-parallel and parallelized versions) for DBPedia graphs

DBPedia	$\lvert E\rvert$		
parallel number	15,373,833	76,868,920	153,737,783
1 (non-parallel)	11,242.80	110,555.53	150,143.69
4 (parallelized)	4,803.85	42,071.79	58,026.58

Tables 4.3 and 4.4 show the results. Each row whose parallel number is 1 represents a result by the non-parallel class extraction algorithm. Each execution time is in seconds. Table 4.3 shows the execution time of the class extraction (non-parallel and parallelized versions) for SP2B graphs of different sizes. This result can be described as follows. In SP2B, (1) the execution time is almost linear to the number of edges $|E|$. The execution time is also almost linear to the number of non-leaf nodes $|V^*|$ since $|V^*|$ is proportional to $|E|$ in SP2B. (2) The parallelization makes the execution time significantly slow. Since the number of classes extracted for SP2B is significantly smaller than DBPedia (details are presented below) and the cost of calculating the light utility function for each node is considerably small, the overhead of parallelization is relatively large. Therefore, the execution time of the parallelized version increased due to the parallelization cost.

Table 4.4 shows the execution time of the class extraction (non-parallel and parallelized versions) for DBPedia graphs of different sizes. This result can be described as follows. (1) The execution time of DBPedia is much longer than that of SP2B since $|L|$ is relatively large in DBPedia and thus the number of extracted classes $|C|$ greatly increases. (2) At first, the growth rate of execution time of DBPedia rapidly grows compared to that of $|V^*|$. As the size of graph grows, the growth rate of execution time is getting closer to that of $|V^*|$. (1) and (2) suggest that $|L|$ and $|V^*|$ mostly affect the execution time of our algorithm. (3) The parallelized version is more than two times faster than the non-parallel version in DBPedia. Since the calculation cost of the utility function is considerably large, the parallelization is effective to make the execution time shortened.

Tables 4.5 and 4.6 show $|C|$ and the scores for the non-parallel version and the parallelized version with $k = 4$. Both of versions were executed with $\alpha = 1$. This result shows that parallelization does not affect $|C|$ and the scores, thus we have almost no difference in the class extraction quality.

Table 4.5: Class extraction scores for the parallelized and non-parallel version ($\alpha = 1$) for the SP2B graph with $|E| = 10,000,457$

parallel number	$\lvert C\rvert$	Score 1	Score 2	Mean
1 (non-parallel)	3	72.53	100.00	86.26
4 (parallelized)	3	72.53	100.00	86.26

Table 4.6: Class extraction scores for the parallelized and non-parallel version ($\alpha = 1$) for the DBPedia graph with $|E| = 15,373,833$

parallel number	$\|C\|$	Score 1	Score 2	Mean
1 (non-parallel)	1,327	70.06	76.97	73.51
4 (parallelized)	1,309	70.60	76.42	73.51

Table 4.7: Memory usage of the class extraction algorithm

Dataset	non-parallel	parallelized
SP2B ($\|E\| = 100,000,380$)	11.1 MB	7.5 MB
DBPedia ($\|E\| = 153,737,783$)	116.6MB	89.6 MB

We also measured the memory usage of the class extraction algorithm. Table 4.7 shows the result. For the SP2B graph with $|E| = 100,000,380$, maximum memory usage of the parallelized version is about 7.5MB. On the other hand, that of the non-parallel version is 11.1MB. We also measured the memory usage for the DBPedia graph with $|E| = 153,737,783$. Maximum memory usage of the parallelized version is about 89.6MB. On the other hand, that of the non-parallel version is about 116.6MB. These results show that the parallelized version was about 10-20 percent less memory usage than non-parallel version. Thus, our class extraction algorithm is completed with sufficiently small memory usage to the input graph file.

Secondly, we give the evaluation of effectiveness of our light utility function. We compare our light utility function and the original utility function. We also examined how parameter α in our light utility function affects the class extraction scores. Let us make a comparison of the following three cases.

- Extracting classes with the original utility function [13]
- Extracting classes with our light utility function with $\alpha = 1$
- Extracting classes with our light utility function with $\alpha = 10$

We ignore incoming edge labels in all the cases. In this experiment, we use parallel number $k = 1$. Tables 4.8, 4.9, and 4.10 show the results. Table 4.8 shows the class extraction scores and the number of classes $|C|$ for the SP2B graph with $|E| = 10,000,457$. The result shows that both of utility functions achieve high scores. The reason why such high scores are obtained is that the graphs were generated the benchmark tool so $|L|$ is small and nodes having the same RDF type have a similar set of labels.

Table 4.9 shows the class extraction scores and the number of classes $|C|$ for the DBPedia graph with $|E| = 50,000$. The maximum mean of score 70.58 is obtained at $\alpha = 1$, which is higher than the value 62.38 obtained by the original.

Table 4.10 shows the class extraction scores and the number of classes $|C|$ for the DBPedia graph with $|E| = 15,373,833$. We could not extract schemas from DBPedia

Table 4.8: Class extraction scores for the SP2B graph with $|E| = 10,000,457$

utility function	$\|C\|$	Score 1	Score 2	Mean
Light($\alpha = 1$)	3	72.53	100.00	86.26
Light($\alpha = 10$)	22	99.45	88.83	94.14
Original	6	97.02	95.32	96.17

Table 4.9: Class extraction scores for DBPedia graph with $|E| = 50,000$

utility function	$\|C\|$	Score 1	Score 2	Mean
Light($\alpha = 1$)	179	67.30	73.87	70.58
Light($\alpha = 10$)	687	89.74	27.77	58.76
Original	81	43.07	81.69	62.38

Table 4.10: Class extraction scores for DBPedia graph with $|E| = 15,373,833$

utility function	$\|C\|$	Score 1	Score 2	Mean
Light($\alpha = 1$)	1,327	70.06	76.97	73.51
Light($\alpha = 10$)	48,631	85.12	3.77	44.44
Original	-	-	-	-

graph with $|E| = 15,373,833$ with the original utility function within reasonable time (24h) because the computation cost of the function is too high.

Overall, the above results suggest that the parameter α introduced in our light utility function works effectively for extracting classes from graphs having fewer unique edge labels such as SP2B. As shown in the tables, our light utility function can extract appropriate schemas efficiently.

Finally, we give the evaluation of the preprocessing and the edge extraction. In this experiment, we used the input files of the edge extraction algorithm obtained by the class extraction algorithm with the parallel number $k = 1$ for SP2B and $k = 4$ for DBPedia. We measured the execution time of the preprocessing and the edge extraction. Tables 4.11 and 4.12 show the details of the execution time. Figures 4.2 and 4.3 plot the execution time of each step. This result means that the execution time of the preprocessing and the edge extraction algorithm are almost linear to $|E|$.

We also measured the memory usage of each step. We observed that each edge extraction step except sorting was executed under 10MB memory usage. External sorting in the preprocessing and the edge extraction step is the most memory consuming step, and its maximum memory usage is about 1.1GB since we limit the maximum memory usage of the sort command to 1GB. Thus, the memory usage of the schema extraction mostly depends on external sorting.

Table 4.11: Execution time of the preprocessing and the edge extraction (SP2B)

SP2B		Edge Extraction (s)		
$\|E\|$	Preprocessing (s)	Phase 1	Phase 2 (sort)	Phase2 (except sort)
100,073	1.55	0.34	0.14	0.34
1,000,009	8.85	2.80	0.52	2.14
10,000,457	84.16	25.17	10.67	21.13
100,000,380	856.52	280.25	83.34	213.92

Table 4.12: Execution time of the preprocessing and the edge extraction (DBPedia)

DBPedia		Edge Extraction (s)		
$\lvert E\rvert$	Preprocessing (s)	Phase 1	Phase 2 (sort)	Phase2 (except sort)
15,373,833	168.66	46.92	22.07	35.36
76,868,920	1,273.80	224.57	171.12	176.57
153,737,783	2,914.30	434.41	418.64	406.48

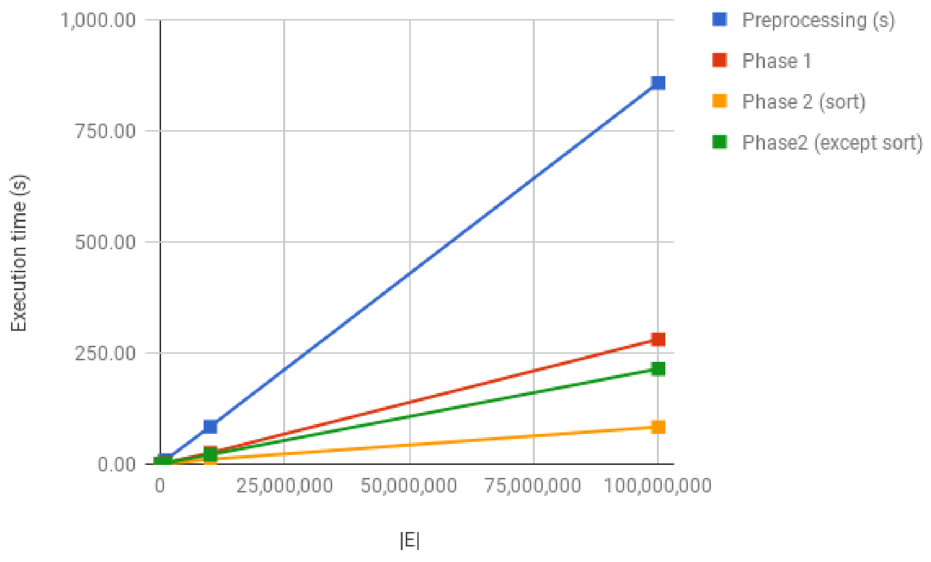

Figure 4.2: Execution time of the preprocessing and the edge extraction (SP2B)

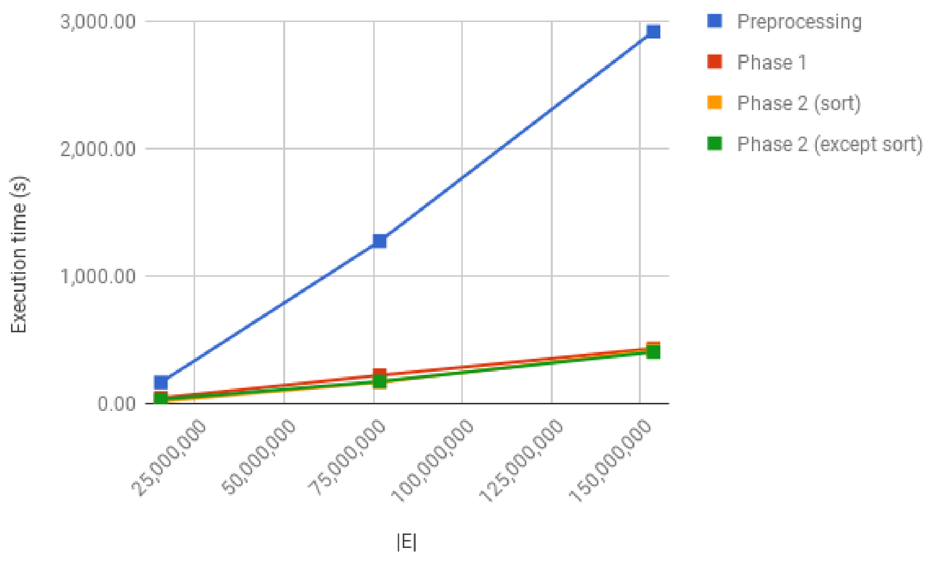

Figure 4.3: Execution time of the preprocessing and the edge extraction (DBPedia)

Chapter 5

Conclusion

In this thesis, we proposed an external memory algorithm for extracting a schema from a 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 the following two steps. (1) The class extraction is to create new classes and to assign each node to a class. (2) The edge extraction is to create edges between classes. The class extraction is also designed as a parallel processing algorithm using our new utility function.

We made some evaluation experiments on our schema extraction algorithm by using SP^{2}Bench, an RDF benchmark tool, and DBPedia graphs. The results suggest that our algorithm can extract schemas from these graphs more efficiently and appropriately than the previous utility function, that the parallelization of the class extraction makes the execution time more than two times faster for DBPedia, and that the memory usage of the schema extraction mostly depends on external sorting.

As a future work, we would like to modify the definition of the light utility function in this thesis so that the class extraction scores become higher and the execution time is further shortened. We also need to examine the extracted classes in more detail.

Acknowledgement

The author would first like to express my sincere gratitude to warm encouragement and support of my adviser, Associate Professor Nobutaka Suzuki in pursuing this study. Completion of this thesis would be impossible without his help. The author would especially like to thank Professor Atsuyuki Morishima for his valuable suggestions and support. The author would also like to thank Associate Professor Tetsuo Sakaguchi for his daily perceptive comments. The author is also grateful to members of Nobutaka Suzuki laboratory for their daily supports, suggestions and encouragement.

Bibliography

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

[2] Garofalakis, M. N., Gionis, A., Rastogi, R., Seshadri, S., and Shim, K. 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)* (2000), pp. 165–176.

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

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

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

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

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

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

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

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

[11] Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D. J., and Naughton, J. F. Relational databases for querying XML documents: Limitations

and opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases* (1999), pp. 302–314.

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

[13] Wang, Q. Y., Yu, J. X., and Wong, K.-F. Approximate graph schema extraction for semi-structured data. In *Proceedings of EDBT 2000*. Springer, 2000, pp. 302–316.

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

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