Designing algorithms for big graph datasets: A study of computing bisimulation and joins

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op maandag 16 maart 2014 om 16:00 uur

door

Yongming Luo

geboren te Fushun, China

Designing algorithms for big graph datasets: A study of computing bisimulation and joins

Yongming Luo

Designing algorithms for big graph datasets: A study of computing bisimulation and joins / by Yongming Luo

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-3801-0

The work in this thesis has been funded by The Netherlands Organisation for Scientific Research (NWO) SeeQR project (612.001.005).

SIKS Dissertation Series No. 2015-11
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The thesis cover is designed by Jiazuo Hou.
The thesis is printed and published by Proefschriftmaken.nl || Uitgeverij BOXPress, ′s-Hertogenbosch.

Summary

Recently, graph data from domains such as social networks, biological systems, sensor networks, and linked data have become major sources for analysis and decision making. Many new platforms have been developed to store and digest such graphs, which are typically massive in size. Traditional algorithms often fail to perform well on these systems (and data), due to their lack of features such as batch processing, low main memory consumption, scalability, robustness against data skew, and even ease of implementation.

In this thesis, over the course of the SeeQR project, we study the design of graph algorithms that fulfill these practical requirements. We first summarize connections between computation models for massive data, and propose an algorithm transformation framework, that can automatically transform an algorithm from one model to another. By using this framework, in the first part, we design algorithms for the problem of localized bisimulation partitioning of graphs, which is an essential step for many graph-based applications. Bisimulation partitioning can significantly reduce the graph size, while still preserving useful structural information. In RDF data management for example, bisimulation partitioning is used to create structural indexes and accelerate query processing. In bisimulation computation, and many other graph algorithms, one common operation is set comparison, which is to compare entities (e.g., nodes) that are associated with certain sets. Therefore in the second part, we study a fundamental set comparison operator, set-containment join, which computes the containment relation between massive collections of sets. We propose efficient algorithms for both of these problems under main-memory, external-memory, and distributed settings. We demonstrate that these algorithms are efficient and practical to use for big graphs. For example, computing localized bisimulation for big graphs with millions of nodes and billions of edges can be efficiently achieved with even a single machine. Set containment join between millions of sets can be computed within minutes instead of hours. These results also prove the effectiveness of the algorithm transformation framework.

We conclude with indications for how our design process and the insights we obtained in our investigations provide value for the design and study of other algorithms over big graphs.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Graphs have long been a fundamental data model in mathematics and computer science. Graph structures are used to model pairwise relations (called *edges*) between entities (called *nodes*). Figure 1.1 shows an example of a small social network graph. Recently, massive graph-structured datasets are becoming increasingly common in a wide range of applications. Graphs of interest, such as social networks [Sco12], internet graphs [FFF99] and linked open data [HB11], are on the order of millions or billions of nodes and edges. As technologies for generating and capturing data continue to improve and proliferate, the size of graphs will only continue to grow rapidly in the near future.

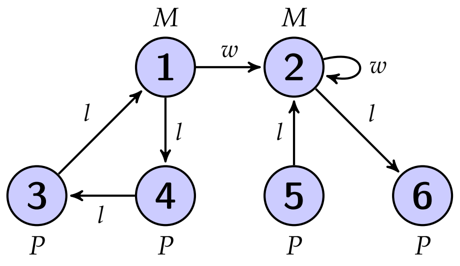


Figure 1.1.: Example graph of a social network, where nodes 1 and 2 have label M (short for "manager"), and the other nodes have label P (short for "people"). The edge label l is short for "likes", while w is short for "works for".

Graph data is being used among varied disciplines for different tasks. Social networks, for example, can be used to discover potential business customers, social friends, or even criminals. The analysis of protein networks, on the other hand, can help scientists to develop new drugs. At the basis of these applications, however, are some fundamental graph problems, which require the help from graph algorithms. Subgraph matching and graph traversal problems, for instance, are the central problems of many social network analysis and bioinformatics applications (e.g., [ZLY09, CWY09, OR02]).

Classic graph algorithms are designed for the Random Access Machine (RAM) model, i.e., a single-processor abstract machine with unified memory-access cost and infinite main memory. These algorithms work pretty well when data can fit into the main memory of a single machine. If that's not the case, graph data (or at least a part of the data) will either (i) be stored on slower secondary memory devices with more space (e.g., disks), or (ii) be distributed and in transit among a cluster of machines while the program is running. In such cases the RAM model cannot represent the behavior of the system correctly, and algorithms designed for the RAM model will not run as expected. In other words, the RAM model is not sufficient for big graph algorithms. To cope with this need, many other models were investigated and implemented. The External Memory (EM) model was created for scenario (i), while the Bulk Synchronous Parallel (BSP) and MapReduce models were designed for scenario (ii) (more details in Section 2.1). These models among others have been widely used in academia and industries. So the ultimate question is:

> Q0: Is there a paradigm for designing algorithms for massive graph data under various computation models?

In this thesis, we do not intend to give a complete answer to this huge and extensive question. Instead we would like to show, using a case study, the course through the design of some concrete algorithms for big graphs. While the problems are specific, the rationale behind the approaches is general. We hope that one can get some inspiration while reading the thesis. Specifically, in this thesis we would like to answer the following questions:

Algorithm transformation Based on different computation models and running environments, algorithms can roughly be categorized into in-memory algorithms, external-memory algorithms, and parallel/distributed algorithms. In practice, many graphs of interest are too large to be processed in main memory for a single machine, therefore, we must necessarily turn to either external memory or distributed/parallel solutions. While there is a lot of research on building these models and designing algorithms for specific models, it is unclear how these algorithms are connected, or how the experience of designing one algorithm can be transferred to another model, or when a new problem comes out, what model should we use first? So here comes our first research question:

> Q1: What is the workflow to design algorithm for different computation models?

In the thesis we present a universal way, namely the algorithm transformation framework, that can automatically transform one algorithm among different computation models. By using this framework, we start our algorithm design with one model, then we could have algorithms transformed to all platforms. In Chapter 2 and Part I, we present this framework and demonstrate its usage via several concrete examples.

Graph reduction When working with big graphs, the algorithm is only part of the story, the other part is the data. If we can make the graph smaller, naturally algorithms will run faster on them. Such technique is called *graph reduction*. It is a way to shrink the size of the graphs, while still maintaining certain characteristics (e.g., topological structure) of them. Graph bisimulation partitioning (and its many variants) is such a reduction operation. Intuitively, bisimulation partitioning groups nodes together as disjoint sets based on the local topology of each node. These partition "blocks" and the relationships between them form an *abstracted* graph where the graph size is reduced but the structural information (e.g., path information) is preserved. With the help of such abstracted graph, many graph queries can be answered or filtered out without probing the real graph, therefore the performance of the whole process is greatly enhanced.

Bisimulation is a ubiquitous notion across many fields [San11]. In the context of graph reduction, graph bisimulation finds its applications in various data management problems, such as constructing structural indexes for XML and RDF databases [FVW$^+$09,MS99,PLF$^+$12a], graph compression [BGK03,FLWW12], and subgraph matching [Fan12]. Despite these many applications, we find little work on computing bisimulation reductions for big graphs. Our second question is:

> Q2: Can we design a practical bisimulation reduction algorithm for big graphs?

In Part I of the thesis, we study how to design such algorithms in detail. We make use of the algorithm transformation framework and develop bisimulation reduction algorithms under different models. The disk-based construction and update algorithms, for example, can handle graphs with millions of nodes and billions of edges on a single machine.

Set comparison In graph computation and data analysis, a commonly found task is to compare sets. In social network analysis, for example, by comparing a set of features (e.g., friends, hobbies) of people, we can derive a lot of other information (e.g., clustering) from the graph. One key task of set comparisons is to discover the *containment* relations between sets, which is called set-containment join in literature.

Set-containment join has been a well-studied problem for over a decade. Many in-memory and disk-based solutions have been proposed. These solutions, however, share some common in-memory processing strategies, whose performance is critical and will become a bottleneck if we want the algorithms to cope with the massive volume of sets from big graphs. Therefore, our third question is:

> Q3: How can we accelerate state-of-the-art set-containment join algorithms?

In Part II, we revisit the in-memory processing strategies and carefully analyze the existing solutions, using advanced data structures to design more efficient (in many cases 10$\times$ faster) set-containment join algorithms.

By answering Q1 to Q3, we also partly answer Q0. Using the algorithm transformation framework, a large collection of existing graph algorithms can be transformed to different models. It also suggests that to design new algorithms for big graphs, the BSP model is a good starting point. However this does not mean that in-memory processing strategies in distributed algorithms are not important. On the contrary, especially for computationally intensive tasks, in-memory processing strategies can make a huge difference on algorithm performance.

1.1. Thesis outline

The thesis is organized as follows:

Chapter 2 We introduce several computational models and our algorithm transformation framework. Here we use the PageRank problem and triangle counting problem as examples to illustrate the transformation mechanism.

Part I studies the problem of localized bisimulation (referred as k-bisimulation) partitioning for big graphs. This part is an extension of papers [LFH$^+$13b] and [LFH$^+$13a]:

> Yongming Luo, George H. L. Fletcher, Jan Hidders, Yuqing Wu, and Paul De Bra. External memory k-bisimulation reduction of big graphs. In *CIKM*, pages 919–928, San Francisco, CA, USA, 2013.

> Yongming Luo, George H. L. Fletcher, Jan Hidders, Paul De Bra, and Yuqing Wu. Regularities and dynamics in bisimulation reductions of big graphs. In *GRADES*, pages 13:1–13:6, New York, NY, USA, 2013.

Chapter 3 We introduce the k-bisimulation problem and discuss its properties. Based on the algorithm transformation framework, we propose several efficient algorithms for various computational models. We prove the I/O complexity and space complexity of the I/O-efficient k-bisimulation construction algorithm and edge update algorithm.

Chapter 4 We conduct extensive empirical analysis on a variety of massive real-world and synthetic graph datasets for the I/O-efficient algorithms. Results show that our algorithms perform efficiently in practice, scaling gracefully as graphs grow in size.

Chapter 5 We take a closer look into various aspects of bisimulation results on big graphs, from both real-world scenarios and synthetic graph generators. We draw the following observations: (1) A certain degree of regularity exists in real-world graphs' bisimulation results. Specifically, power-law distributions appear in many of the results' properties. (2) Synthetic graphs fail to fulfill one or more of these regularities that are revealed in the real-world graphs.

Chapter 6 We discuss more properties of k-bisimulation, and take a look at a related problem, k-simulation. We discuss the connections between k-simulation and k-bisimulation, and propose ideas to efficiently compute k-simulation.

Inspired by k-simulation computation and many other real-world applications, in **Part II** we study the set-containment join problem. This part is an extension of paper [LFHDar]:

> Yongming Luo, George H. L. Fletcher, Jan Hidders, and Paul De Bra. Efficient and scalable trie-based algorithms for computing set containment relations. In *ICDE*, Seoul, Korea, 2015, to appear.

Chapter 7 We formally introduce the problem of set-containment join. We survey the state-of-the-art approaches and discuss related research problems.

Chapter 8 We introduce the problem of subset enumeration within limited sets. We propose several new algorithms for this problem. The result of this chapter is used in a later chapter to develop new set-containment join algorithms. Readers can jump to Chapter 9 first and then go back to Chapter 8 if necessary.

Chapter 9 We present two novel trie-based set-containment join algorithms: the Patricia Trie-based Signature Join (PTSJ) and PRETTI+, a Patricia trie enhanced extension of the state-of-the-art PRETTI join. The compact trie structure not only enables efficient use of main-memory, but also significantly boosts the performance of both approaches. By carefully analyzing the algorithms and conducting extensive experiments with various synthetic and real-world datasets, we show that, in many practical cases, our algorithms are an order of magnitude faster than the previous state-of-the-art.

Chapter 10 We conclude the thesis with a discussion of future work based on the proposed techniques and results.

2. Connections between computation models and programming frameworks

Classic in-memory algorithms are designed and analyzed under the Von Neumann (RAM) model, while other models are created for algorithms running in various environments. Lots of efforts have been devoted to design algorithms for specific models, yet the connections between algorithms of different models have not been well-studied. In this chapter, we are curious about the possibility of designing a single algorithm for different models. In Section 2.1 we introduce several widely-used computation models and programming frameworks. Then we study ways to automatically translate an algorithm from one model (BSP) to other models. In the end, we use the PageRank problem and triangle counting problem as examples to show how the translation strategy works in practice.

2.1. Practical computation models and frameworks

Computation models and programming frameworks not only serve as tools for analyzing algorithm performance, but also are guidance or restrictions for designing real programs. In this section we introduce several well-known computation models and frameworks that we will use later, namely the RAM model, EM model, BSP model, Pregel-like framework, and MapReduce framework (a more detailed discussion of computation models can be found in [AM10]). We further discuss the connections between these models and show how they influence the algorithm design.

Random-access machine (RAM) model The most commonly-used model for analyzing algorithm performance is the random-access machine (RAM) model. It assumes a one-CPU machine with infinite main memory. Read and write accesses to any memory cell take unit cost and instructions roughly take the same amount of time to finish. While the RAM model is simple to reason about and gives a good estimation of algorithm performance when data can fit into main memory, it becomes less accurate when data volume exceeds the main memory. This leads to the External Memory (EM) model.

External Memory (EM) model When some data is too big to be held in the main memory, algorithms need to transfer data (called I/O cost) between main memory and secondary storage devices (e.g., magnetic disk). Due to the nature of external memory, this communication process is usually more time consuming than the in-memory processing part. A special kind of algorithms (called external memory algorithms) are designed to minimize the I/O cost, while preserving the performance of algorithms themselves. The External Memory (EM) model [AV88, Vit08] is designed to model the behavior of such algorithms. EM considers a single processor, single I/O computing device where data is organized by blocks (for ease of discussion, we do not consider machines with parallel disks, i.e., Parallel Disk Model [Vit08]). The algorithm performance is measured by the number of I/O operations. Secondary storage space is considered to be infinite, while in-memory computation is considered free.

Suppose we have table X (file sequentially filled with records) with size $|X|$, a machine with the main memory capacity of M and data transfer block size B (page size). Therefore X occupies $\frac{|X|}{B}$ pages on disk. In what follows, we will use the following notation [Vit08] to estimate I/O cost:

- $sort(|X|)$ denotes the number of I/Os when sorting table X on some given column(s). This will take $\Theta(\frac{|X|}{B} log_{\frac{M}{B}}(\frac{|X|}{B}))$ I/Os for a standard external memory based merge sort.

- $scan(|X|)$ denotes the number of I/Os when scanning over table X. This will take $\frac{|X|}{B}$ I/Os.

The Bulk Synchronous Parallel (BSP) model The Bulk Synchronous Parallel (BSP) model is proposed by Leslie Valiant [Val90], intended to be used as a bridging model between parallel computing infrastructure (hardware) and programming paradigm (software). BSP consists of three components: (1) a set of computation nodes; (2) a global end-to-end network that connects all nodes; and (3) some synchronization facility. One BSP program consists of *supersteps*. Inside each superstep, computation nodes do their own work in parallel. At the end of each superstep, there is a synchronization barrier at which nodes communicate with each other and synchronize their states. This decomposition of computation and communication makes the BSP program easy to understand and implement. Some difficulties of programming in distributed systems, such as deadlock handling, are alleviated by the BSP model.

BSP can serve as a programming model. Various tools have been developed to help programmers build distributed applications upon such model. Platforms like MulticoreBSP [YBRM14] and Apache Hama [Ham14] are examples of such efforts.

The performance of a BSP computer is determined by three parameters: (p, g, l), where p is the number of processors, g is a parameter relating to the network throughput, and l is the cost of barrier synchronization. These flexible parameters can be adapted according to different computer architectures. Then the performance of a superstep of a BSP algorithm can be estimated as

$$T_{superstep} = w + h \cdot g + l, \tag{2.1}$$

where w is the maximum local computation in a superstep, $h \cdot g$ is the maximum communication time in a superstep (h is the maximum number of messages sent or received by any processing unit) [McC96].

Vertex-centric programming frameworks, Pregel-like systems Recently, with the increasing interests of graph database and graph computation, various companies and research groups propose the idea of vertex-centric graph computing, and some release programming frameworks with them. These platforms are essentially the realization of BSP in the context of graph computation. Here we briefly introduce the Pregel framework [MAB+10], which is one of the first among these. Pregel-like systems are those such as GraphLab [Low13], GPS [SW13], Apache Giraph [Gir14], and Mizan [KAA+13].

One the model level, the main difference between BSP and Pregel is that Pregel treats each node in a graph as a computing machine, meaning that conceptually each node works on its own computation problem in parallel, and nodes communicate with each other at the synchronization barrier. The key point, when designing algorithms for Pregel, is the ability to "think like a vertex", to decompose the problem to small pieces at vertex level.

During our discussion, we use BSP and Pregel-like model interchangeably. In graph algorithms, indeed in many cases, nodes (vertexes) are the central of attention, however we sometimes also treat other entities as computing units when necessary, which makes our model more similar to the original BSP model. Such modification can be easily handled by Pregel-like systems.

MapReduce framework The MapReduce programming model [DG08, LD10] is designed to process large datasets in parallel. A MapReduce *job* takes a set of key/value pairs as input and outputs another set of key/value pairs. A MapReduce *program* consists of a series of MapReduce jobs, where each MapReduce job implements a *map* and a *reduce* function ("[ ]" means a list of elements)[1]:

$$\begin{array}{lll} \textit{map} & (k_1, v_1) & \rightarrow [(k_2, v_2)] \\ \textit{reduce} & (k_2, [v_2]) & \rightarrow [(k_3, v_3)]. \end{array}$$

The *map* function takes key/value pair (k_1, v_1) as the input, and emits a list of key/value pairs (k_2, v_2). In the *reduce* function, all values with the same key are grouped together as a list of values v_2 (this is achieved via the *shuffling* stage) and are processed to emit another list of key/value pairs (k_3, v_3). Users define the *map* and *reduce* functions, letting the framework take care of all other aspects of the computation (synchronization, I/O, fault tolerance, etc.).

The open source Hadoop implementation of the MapReduce framework is a mature system and is widely used in industry and research [Had14]. Hadoop is often used together with the Hadoop Distributed File System (HDFS), which is designed to provide high-throughput access to application data. Besides *map* and *reduce* functions, in Hadoop a user can also write a custom *partition* function, which is applied after the *map* function to specify to which reducer each key/value pair should go.

2.2. Connections

After introducing the models, one natural question to ask is, what is the connection between these models? For instance, if we design some algorithm for one model, can we easily adapt the algorithm to another model? We try to answer these questions in this section and summarize some research efforts in this area. The short answer is that the BSP model connects all models together.

We write model$_1$ $\rightarrow$ model$_2$ if the algorithm of model$_1$ can be transformed to the algorithm of model$_2$. For a BSP program, we use compute(i) to denote the amount

[1]Here we use the model description from [LD10] instead of the original paper, which is easier to understand and closer to implementations such as Hadoop.

of local computation of node *i* in one superstep. We assume that the synchronization happens at the end of each superstep.

BSP → EM [SK97] BSP programs can be translated into external memory programs. The main idea of this approach is to mimic the BSP platform behavior using one machine by creating several virtual machines. At each superstep, we create two tables: virtual machine (*vm*, Table 2.2a) and message (*msg*, Table 2.2b). We store information of computation nodes in *vm* and the communication between nodes in *msg*.

Table 2.1.: Tables for each superstep, BSP → EM

(a) table *vm*

node_id	node_state

(b) table *msg*

from_node	to_node	message_content
`//node id of the source`	`//node id of the target`	

Since each virtual machine state is stored in one row of *vm*, every local computation on some virtual machine is about reading and changing its state. Then at the communication stage, all we need to do is to fill in the *msg* table, and perform a join of it with *vm* table of the next superstep on *msg.to_node* and *vm.node_id*. The join process can be implemented via various join algorithms (such as sort-merge join) from database research (e.g., [Gra93]). Pseudo code can be found in Algorithm 1.

Algorithm 1: EM simulates BSP, one superstep

```
  Input: table vm
  Output: table vm
1 create table msg
2 for each (node_id, node_state) ∈ vm do
3     node_state ← compute(node_id, node_state)
4     update (node_id, node_state) to vm
5     append (node_id,...) to msg
6 Join msg with vm on msg.to_node and vm.node_id
7 return vm
```

The transformed EM algorithm involves I/O of table *vm* and *msg*. Therefore to estimate the cost of the algorithm, the most interesting values are the sizes of these two tables (referred as $|vm|$ and $|msg|$). If we estimate the performance of a BSP algorithm by Equation 2.1, we get that $|vm|$ is bounded by $p \times w$, $|msg|$ by $p \times h$. The I/O cost of Algorithm 1 is bounded by $3 \times Scan(|vm|) + 2 \times Scan(|msg|) + Sort(|msg|)$, which consists of (1) load and write back to vm ($2 \times Scan(|vm|)$), (2) sort msg ($Sort|msg|$), and (3) merge join on vm and msg ($Scan(|vm|) + Scan(|msg|)$). This of course assumes that we use a simple sort-merge join for message passing.

Tighter bounds are obtained in paper [SK97], e.g., what is the EM algorithm's I/O complexity compared to the original BSP algorithm. But such result relies on the assumption that each message size is at most $\frac{m}{p}$, where m is the virtual machine memory size, and p is the number of nodes (virtual machines). In many graph problems, each message size can potentially be p, which makes $m \geq p^2$, so the assumption does not hold anymore.

BSP $\rightarrow$ MapReduce [GSZ11, Pac12] Due to the similarity of BSP and MapReduce, there is a straightforward way to run BSP algorithm on MapReduce platforms. It works as follows [GSZ11]:

- Each superstep corresponds to one MapReduce task;
- Mappers are of no use, only the reducers are used as processors in a BSP computer;
- The shuffling stage (grouping the output of map function by keys) is used for message passing;
- Data are stored on the filesystem (global memory) after every MapReduce task finishes.

The above approach only makes use of the reduce function of the framework, while we can do more on the map side. Alternatively, same as BSP $\rightarrow$ EM, we can use the virtual machines to mimic BSP framework using MapReduce. Again considering tables from Table 2.1, the most tricky part of this translation would be the joining of two tables, which can be easily handled by the shuffling step within MapReduce. Then one superstep corresponds to one MapReduce task. The sketch of the simulation is described in Algorithm 2. We note that the local computation (line 2) can also be done in the reduce procedure, this translation is then the same as in paper [GSZ11]. Also note that the table concept in the algorithm is used for the ease of discussion, a MapReduce system does not hold the tables in its global memory but stores them in the distributed file system. Essentially, tables are just a collection of key-value pairs distributed among machines.

Algorithm 2: MapReduce simulates BSP, one superstep

```
  Input: table $vm$
  Output: output $vm$ for the next iteration
1 Procedure Map((node_id, node_state))
      // compute() can be moved to reduce procedure
2     node_state ← compute(node_id, node_state)
3     emit(node_id, node_state)
4     for each receiver of node_id do
          // msg contains the sender information
5         emit(receiver, msg)
6 Procedure Reduce((node_id, [node_id infos]))
      // compute() can be done here as well
7     save the input to table $vm$
```

EM (Streaming) → MapReduce [FMS+10] It is also worth mentioning the connection from EM to MapReduce. For a general EM model, a machine can essentially perform random access at the disk-block level. This indicates that EM algorithms, making use of this property, are not suitable for distributed models. If we restrict the EM model's I/O access pattern to sequential scans on one disk (allowing multiple passes), the model essentially becomes a streaming model [HRR98]. It has been shown that any algorithm under the streaming model that computes symmetric (order-invariant) functions is guaranteed to have a corresponding MapReduce Algorithm [FMS+10]. However, if we allow the EM model to have more than one disk, even with only sequential scan enabled, the model becomes strictly more powerful than the streaming model and other models [ADRR04], e.g., there exists some problem that can be solved with two passes on the latter model, but needs at least polynomial number of passes on the streaming model.

Other types of algorithm transformations are possible as well, such as MapReduce → BSP [Pac12] and Relational Algebra → BSP [Suj96].

As we have seen so far, connections between models are discussed in various contexts. But there is not much work that puts these efforts together and shows the big picture. In this section, we find that BSP is a model that can connect all models together. BSP algorithms, using the transformation techniques in this section, can be adapted to algorithms for other models. We call this strategy an *algorithm transformation framework* and will illustrate it with two concrete examples in Section 2.3 and 2.4.

2.3. Example: PageRank

In this section we take the well-known PageRank problem as an example to show how algorithm design is transformed between different computation platforms.

2.3.1. Problem definition and algorithm under the RAM model

PageRank, proposed by Brin et al. [BP98], is a measurement of importance of nodes in a graph. It has been extensively studied in various contexts. For a graph $G = \langle N, E\rangle$ (N for node set and E for edge set), a simple version of PageRank of some node x is defined recursively in Equation 2.2[2].

$$pageRank(x) = \frac{0.15}{|N|} + 0.85 \times \sum_{p \in pred(x)} \frac{pageRank(p)}{|succ(p)|}. \tag{2.2}$$

Here $pred(x)$ is the set of direct predecessors of x ($\{y|(y,x) \in E\}$), $succ(x)$ is the set of direct successors of x ($\{y|(x,y) \in E\}$), and $|M|$ is the size of set M. The initial value of $pageRank(x)$ is set to $\frac{1}{|N|}$ for every node. Naturally, one algorithm to compute PageRank is to iteratively compute the numbers of each node until they are stable enough.

$$pageRank_{i+1}(x) = \frac{0.15}{|N|} + 0.85 \times \sum_{p \in pred(x)} \frac{pageRank_i(p)}{|succ(p)|}. \tag{2.3}$$

After all values of $pageRank_i$ are computed, the $pageRank_{i+1}$ computation can start.

For example, we want to apply Equation 2.3 to a simple graph in Figure 2.1. First we set the PageRank value of each node to be $\frac{1}{3}$. Then in the first iteration, $pageRank_1(1) = 0.15 \times \frac{1}{3} + 0.85 \times (\frac{1}{3} + \frac{1}{3}) = 0.617$, and $pageRank_1(2) = pageRank_1(3) = 0.15 \times \frac{1}{3} + 0.85 \times (\frac{1}{3} \times \frac{1}{2}) = 0.1917$. Because node 1 has two direct predecessors, its PageRank value is higher than the others, even in the very first iteration.

[2]The weight 0.15 and 0.85 are decided based on good practices and can be arbitrary positive numbers as long as they add up to 1. Such assignment has no effect on our algorithm discussion.

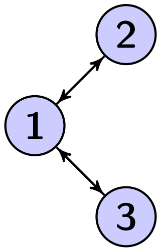


Figure 2.1.: Example graph for PageRank

2.3.2. BSP algorithm (Pregel)

The Pregel version of the PageRank algorithm was introduced in the original Pregel paper [MAB+10]. We describe the algorithm in Algorithm 3. Here for simplicity, we omit the strategy for checking stop conditions. We can see that the algorithm is quite simple, only includes the computation on each node and sends its rank value to its children (direct successor).

Algorithm 3: PageRank in BSP, i_{th} iteration, [MAB+10]

```
Input: graph structure, $pageRank_i$ for all nodes
Output: $pageRank_{i+1}$ for all nodes
1 for each $n \in N$ do
2     compute $pageRank_{i+1}(n)$ by Equation 2.3
3     send $pageRank_{i+1}(n)$ to children of $n$
4 synchronize
```

2.3.3. EM algorithm

A series of EM algorithms for PageRank can be found in Chen et al. [CGS02]. We present one of them in Algorithm 4. The algorithm has the same structure as in Algorithm 1. All we need to do is to replace the *compute(node_id)* function with Equation 2.3. For the message passing part, both sort merge join and hash join are discussed and experimented in [CGS02].

2.3.4. MapReduce algorithm

It is well-known that the MapReduce framework is not the ideal platform for iterative algorithms such as PageRank. Nonetheless, PageRank is a common task

Algorithm 4: PageRank in EM, i_{th} iteration, [CGS02]

```
Input: table vm
Output: table vm
1 create table msg
2 for each (nid, nstate) ∈ vm do
      // information to compute $pageRank_{i+1}(nid)$ is in nstate
3     nstate ← compute $pageRank_{i+1}(nid)$ by Equation 2.3
4     update (nid, nstate) to vm
5     for each $s \in succ(nid)$ do
          // to send $pageRank_{i+1}(nid)$ to nid's direct successors
6         append (nid, s, $pageRank_{i+1}(nid)$) to msg
7 join msg with vm on msg.to_node and vm.node_id
8 return vm
```

to be performed in MapReduce. Algorithm 5 from Lin et al. [LD10, Chapter 5] describes the PageRank algorithm in MapReduce framework, which essentially uses the same framework of Algorithm 2.

Algorithm 5: PageRank in MapReduce, i_{th} iteration, [LD10]

```
Input: table vm
Output: output vm for the next iteration
1 Procedure Map((nid, node info))
      // $pageRank_i(nid)$ is in node info
2     emit (nid, node info)
3     for each successor of nid do
4         emit(successor, $pageRank_i(nid)$)
5 Procedure Reduce((nid, [nid info]))
      // [node info] also holds $pageRank_i$ values of predecessor nodes
6     compute $pageRank_{i+1}(nid)$ by Equation 2.3
7     update $pageRank_{i+1}(nid)$ in table vm
```

2.4. Example: triangle counting

Triangle counting is another important graph problem. It has been extensively studied due to its applications in network analysis and graph mining (e.g., [TKMF09]). In this section we take this problem as another example to illustrate the algorithm transformation framework.

2.4.1. Problem definition and algorithm under the RAM model

As the name suggested, in the triangle counting problem, we want to get the number of triangles in a graph. Assume an undirected graph $G = \langle N, E \rangle$ without self-loop, we would like to get $|T|$ where $T = \{\{u, v, w\} | \{u, v\}, \{v, w\}, \{w, u\} \in E\}$.

One classic and straightforward algorithm for triangle counting is called the *node-iterator* algorithm [Sch07]. The algorithm iterates over the neighbor set of each node, and tries to find out if there exists an edge between any two nodes in the neighbor set. Pseudo code of node-iterator is in Algorithm 6.

Algorithm 6: Algorithm node-iterator for triangle counting in the RAM model, [Sch07]

```
Input: $G = \langle N, E \rangle$
Output: Number of triangles in $G$
1 $count \leftarrow 0$
2 for each $n \in N$ do
3     for each $u, v \in n.neighbors()$ do        // count both (u, v) and (v, u)
4         if $(u, v) \in E$ then
5             $count \leftarrow count + 1$
6 return $count/6$
```

Note that here we consider all pairs of nodes in a neighbor set (line 3 of Algorithm 6) where further optimizations may apply (e.g., node-iterator++ in [Sch07]). For instance, only consider (u, v) such that $u < v$. For illustration purpose we omit such techniques and show a simpler version of the algorithm.

2.4.2. BSP algorithm

To design a distributed version of the node-iterator, one difficult part is the edge existence check (line 4 in Algorithm 6). The BSP algorithm (Algorithm 7) achieves this by enumerating all pairs (u, v) from n's neighboring set, and sending this pair and the possible edge (u, v) to the same machine for checking. Essentially, it creates all paths of length two in the graph and tries to determine if the path can be closed by a third edge.

2.4.3. EM algorithm

We can easily derive the EM algorithm from Algorithm 7. We notice that there is basically no computation in each superstep in the BSP algorithm, but merely

Algorithm 7: Algorithm node-iterator in BSP

```
1  for each $n \in N$ do
2      for each $u, v \in n.neighbors()$ do
3          send $[(u, v), n]$ to $(u, v)$
4  for each $(u, v) \in E$ do
5      send $[(u, v), true]$ to $(u, v)$
6  synchronize
7  for each $(u, v)$ do
8      if $[(u, v), true]$ exist then
9          count $u, v, n$ as a triangle
10 synchronize
   // omit the superstep for gathering counts
```

message passing. And we already know that the algorithm is about materializing paths of length two and three in a graph. So the algorithm mainly consists of joins on the edge table. In fact, node-iterator can be implemented using a relational database and SQL [Wal].

Algorithm 8: Algorithm node-iterator in EM

```
   Input: Edge table $E_t$ of $G$, with schema $(source, target)$
   Output: Number of triangles in $G$
1  $F \leftarrow E_t \bowtie_\phi E'_t$        // $E'_t$ is another reference to $E_t$
   // $\phi : E_t.target = E'_t.source$
   // $F$ is with schema $(source, source', target)$
2  $H \leftarrow F \bowtie_\theta E_t$
   // $\theta : F.target = E_t.source \wedge F.source = E_t.target$
3  return $|H|/6$
```

Note that Dementiev's approach [Dem], which makes use of many EM-specific techniques, is more similar to the original node-iterator algorithm.

2.4.4. MapReduce algorithm

The MapReduce version of node-iterator [SV11] is a direct translation of Algorithm 7. We describe it in Algorithm 9. Note that the edge existence check (line 4 to line 5 of Algorithm 7) can be a separated task, or can be merged together with other tasks like we do here, since it only uses the map function.

Algorithm 9: Algorithm node-iterator in MapReduce, [SV11]

```
Input: Edge file with schema $(source, target)$
Output: Number of triangles in $G$
Procedure Map($(u, v)$ in edge file)
    emit($u, v$)
// neighbor set of nodes are grouped together
Procedure Reduce($n, n.neighbors()$)
    for each $u, v \in n.neighbors()$ do
        emit($(u, v), n$)
Procedure Map()
    if input is from last job then
        emit($(u, v), n$)
    if input is from edge file then
        emit($(u, v), true$)
Procedure Reduce($(u, v), [value]$)
    if $true \in [value]$ then
        for $n \in [value]$ do
            count $(u, v, n)$ as a triangle
// omit the task for gathering counts
```

2.5. Conclusion

To sum up, one effective way to design algorithms for massive graphs is to start with the BSP model or Pregel-like platform, then we can translate the algorithm to corresponding external memory and MapReduce algorithms or at least get inspiration. The mechanism of this algorithm transformation framework is described in Section 2.2. Section 2.3 and 2.4 discuss two real-world graph problems. Using the algorithm transformation framework, we start with BSP algorithms and find efficient algorithms for each model, many of which appear in scientific publications as research results. This proves the effectiveness of the framework.

Of course things can get tricky during the design process. Then we take a step back and apply more specific techniques (e.g., model-specific techniques). In Part I we will explain in detail how to apply the framework and the techniques to another fundamental graph problem, namely k-bisimulation partitioning.

Part I.

Localized bisimulation

3. Scalable k-bisimulation reduction of big graphs

3.1. Introduction

In reasoning over graphs, a fundamental and ubiquitous notion is that of bisimulation, which is a characterization of when two nodes in a graph share basic structural properties such as neighborhood connectivity. Bisimulation arises and is widely adopted in a surprisingly large range of research fields [SR11]. In data management, bisimulation partitioning (i.e., grouping together bisimilar nodes in order to reduce graph size) is often a basic step in indexing semi-structured datasets [MS99], and also finds fundamental applications in RDF [PLF$^+$12a] and general graph data (e.g., compression [BGK03, FLWW12], query processing [KSBG02a], and data analytics [Fan12, THP08]).

It is often the case that bisimulation reductions of real graphs result in partitions which are too refined for effective use. Hence, a notion of localized bisimulation, or *k-bisimulation* has proven to be quite successful in data management applications (e.g., [FVW$^+$09, KSBG02a, QLO03, YHSY04]). k-bisimulation is the variant of bisimulation where topological features of nodes are only considered within a local neighborhood of radius $k \geqslant 0$. With a pay-as-you-go nature, k-bisimulation is cheaper to compute and maintain, cost adjustable, and faithfully representative of the bisimulation partition within the local neighborhood.

State of the art

Algorithms for bisimulation partitioning have been studied for decades, with well-known algorithms such as those of Paige and Tarjan [PT87] and more recent work (e.g., [DPP04]), having theoretically effective behavior.

In practice, however, state-of-the-art solutions face a critical challenge: all known approaches for computing bisimulation on a compute node are internal-memory based solutions[1]. In these solutions, operations such as directly changing other node's state and the lack of efficient ordering inherently lead to random memory access patterns. Therefore, these algorithms do not translate to efficient I/O-bound solutions in the EM model. Consequently, when processing graphs which do not fit entirely in main memory the performance of these algorithms decreases drastically.

The reality is that, in practice, many graphs of interest are too large to be processed in main memory. Indeed, massive graphs are now ubiquitous [Fan12, HB11]. To process real graphs, therefore we must necessarily turn to either external memory, distributed, or parallel solutions. There has been some work on parallel (e.g., [RL98,SSZ95]) and distributed (e.g., [BO05]) approaches to bisimulation computation, and recently, external memory solutions on restricted acyclic and tree-structured graphs [HFH12]. However, to our knowledge there is no known effective solution for computing bisimulation and k-bisimulation partitions on arbitrary graph structures in external memory. Such an algorithm would not only enable us to process big graphs on single machines, but also provide an essential step for parallel and distributed solutions (e.g., MapReduce [LdLF+13]) to further scale their performance on real graphs. As noted in paper [LdLF+13] and many other researches (e.g., [KBG12]), in many cases, single machine external memory algorithms are more competitive than distributed algorithms due to their lack of communication overhead and their effective use of available infrastructure. Therefore, the study of external memory solutions is clearly warranted.

Our contributions

Given these motivations, we have studied external memory solutions for reasoning about k-bisimulation on arbitrary graphs. In this chapter, we present the results of our study, which makes the following high-level contributions.

- We present k-bisimulation partitioning algorithms for BSP and MapReduce.
- Following the algorithm transformation framework from Chapter 2, we design the first known I/O efficient external memory based algorithm for constructing the k-bisimulation partition of a disk-resident graph. The I/O cost of this algorithm is bounded by $O(k \cdot sort(|E_t|) + k \cdot scan(|N_t|) + sort(|N_t|))$, with space complexity $O(|N_t| + |E_t|)$, where E_t and N_t are the the input graph's edge set and node set.

[1]With the single exception of Hellings et al. [HFH12] which we discuss below in Section 3.5.2.

- We present the first known I/O efficient external memory based algorithms for performing maintenance on a disk-resident k-bisimulation graph partition, with I/O cost bounded by $O(k \cdot sort(|E_t|) + k \cdot sort(|N_t|))$, and space complexity $O(k \cdot |N_t| + k \cdot |E_t|)$.

The rest of the chapter is organized as follows. In the next section we give our basic definitions and foundations for our solution. In Section 3.3 we describe the k-bisimulation partitioning algorithms under BSP and MapReduce. We introduce the data structures used and the cost model in Section 3.4. We then describe in Section 3.5 our solution for constructing k-bisimulation partitioning. Next, Section 3.6 presents algorithms for keeping an existing partition up to date, in the face of updates to the underlying graph.

3.2. Data model and definitions

Our data model is that of finite directed node- and edge-labeled graphs $\langle N, E, \lambda_N, \lambda_E \rangle$, where N is a finite set of nodes, $E \subseteq N \times N$ is a set of edges, λ_N is a function from N to a set of node labels $\mathcal{L}_N$, and λ_E is a function from E to a set of edge labels $\mathcal{L}_E$.

Definition 3.1 (*k-bisimilar*). *Let k be a non-negative integer and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph*[1]*. Nodes $u, v \in N$ are called k-*bisimilar *[KSBG02b] (denoted as $u \approx^k v$) iff the following holds:*

1. *$\lambda_N(u) = \lambda_N(v)$,*
2. *if $k > 0$, then $\forall u' \in N[(u,u') \in E \Rightarrow \exists v' \in N[(v,v') \in E,\ u' \approx^{k-1} v'$ and $\lambda_E(u,u') = \lambda_E(v,v')$*[2]*]], and*
3. *if $k > 0$, then $\forall v' \in N[(v,v') \in E \Rightarrow \exists u' \in N[(u,u') \in E,\ v' \approx^{k-1} u'$ and $\lambda_E(v,v') = \lambda_E(u,u')]]$.*

It can be easily shown that the *k-bisimilar* relation is an equivalence relation.

We illustrate Definition 3.1 with an example. Consider the graph given in Figure 1.1. In this graph, nodes 1 and 2 are 0- and 1- bisimilar but not 2-bisimilar.

It is attempting to reason about the bisimilarity of the nodes by their pathes, e.g., to think that nodes u and v are k-bisimilar iff for any path of length at most k starting

[1]We assign a default label for all nodes and edges if they are not labeled.
[2]Note that we use $\lambda_E(u,u')$, instead of $\lambda_E((u,u'))$, for ease of readability.

at u there is an equivalent path starting at v. Such intuition however, is not correct. One counter example is to consider a graph as follows:

$$n_1 \xrightarrow{a} n_2 \xrightarrow{b} n_3, n_4 \xrightarrow{a} n_5 \xrightarrow{b} n_6, n_4 \xrightarrow{a} n_7.$$

Here n_1 and n_4 have the same path of length 2, but are not 2-bisimilar. Another counter example (Figure 6.3) and more discussions can be found in Chapter 6.

Our interest in this part is in computing the k-bisimulation partition of a massive graph, and performing maintenance on the result under updates to the original graph. By *massive*, we mean that both the set of nodes and the set of edges of the graph are too big to fit into main memory. By a *partition* of the graph, we mean an assignment of each node u of the graph to a *partition block*, which is the unique subset of nodes in the graph of which the members are k-bisimilar to u.

In particular, we are interested in constructing partition "identifiers."

Definition 3.2 (*k-partition identifier*)**.** *A* k-partition identifier *for graph* $G = \langle N, E, \lambda_N, \lambda_E \rangle$ *and* $k \geq 0$ *is a set of* $k+1$ *functions* $\mathcal{P} = \{pId_0, \ldots, pId_k\}$ *such that, for each* $0 \leq i \leq k$, pId_i *is a function from* N *to the integers, and, for all nodes* $u, v \in N$, *it holds that* $pId_i(u) = pId_i(v)$ *iff* $u \approx^i v$.

A fundamental tool in our reasoning about k-bisimulation is the notion of node signatures. Intuitively, a node's signature is an encoding of its neighborhood information up to a certain radius, by which we can determine the node's partitioning information.

Definition 3.3 (*k-bisimulation signature*)**.** *Let* $G = \langle N, E, \lambda_N, \lambda_E \rangle$ *be a graph,* $k \geq 0$, *and* $\mathcal{P} = \{pId_0, \ldots, pId_k\}$ *be a k-partition identifier for G. The* k-bisimulation signature *of node* $u \in N$ *is the pair* $sig_k(u) = (pId_0(u), L)$ *where:*

$$L = \begin{cases} \emptyset & \text{if } k = 0, \\ \{(\lambda_E(u, u'), pId_{k-1}(u')) \mid (u, u') \in E\} & \text{if } k > 0. \end{cases}$$

We then have the following fact.

Proposition 3.1. $pId_k(u) = pId_k(v)$ *iff* $sig_k(u) = sig_k(v)$ $(k \geq 0)$ *[KS90].*

To prove proposition 3.1 we first need to prove the following proposition.

Proposition 3.2. $u \approx^k v \Rightarrow u \approx^{k-1} v$ $(k > 0)$.

Proof. By induction on k.

(1) $k = 1$. This is obvious, as 0-bisimilarity just enforces equality of node labels.

(2) $k > 1$. Assume that this holds for $j - 1$ ($\approx^{j-1} \Rightarrow \approx^{j-2}, 0 < j - 1 < k$), we want to show that this also holds for j ($u \approx^{j} v \Rightarrow u \approx^{j-1} v$). Let $u \approx^{j} v$. According to the definition, for all outgoing edges $(u, u') \in E$, there exists some edge $(v, v') \in E$, such that $u' \approx^{j-1} v'$ and $\lambda_E(u, u') = \lambda_E(v, v')$, and vice versa. Since $\approx^{j-1} \Rightarrow \approx^{j-2}$, we have $u' \approx^{j-2} v'$, then we have $u \approx^{j-1} v$. So $u \approx^{j} v \Rightarrow u \approx^{j-1} v$. □

Now we prove Proposition 3.1:

Proof for Proposition 3.1. $\Rightarrow$:

(1) For $k = 0$, this is trivial, since $pId_0(u) = pId_0(v)$.

(2) For $k > 0$, (which also means $u \approx^{k} v$), we want to show that $sig_k(u) = sig_k(v)$. According to Proposition 3.2, $u \approx^{k} v \Rightarrow u \approx^{0} v$, so that $pId_0(u) = pId_0(v)$. And for each outgoing edge (u, u') of u, there exists some outgoing edge (v, v') of v, such that $u' \approx^{k-1} v'$, then $pId_{k-1}(u') = pId_{k-1}(v')$, and $\lambda_E(u, u') = \lambda_E(v, v')$. Therefore each pair in $sig_k(u)$ equals to some pair in $sig_k(v)$, and vice versa. Then we have $sig_k(u) = sig_k(v)$.

$\Leftarrow$:

(1) For $k = 0$, this is obvious.

(2) For $k > 0$. Let $sig_k(u) = sig_k(v)$, we want to show that $pId_k(u) = pId_k(v)$ (or $u \approx^{k} v$). Since $sig_k(u) = sig_k(v)$, we know that for every outgoing edge (u, u') of u, we have a pair $(\lambda_E(u, u'), pId_{k-1}(u'))$ in $sig_k(u)$, we can find an equal pair $(\lambda_E(v, v'), pId_{k-1}(v'))$ in $sig_k(v)$, such that $pId_{k-1}(u') = pId_{k-1}(v')$ and $\lambda_E(u, u') = \lambda_E(v, v')$. By definition, this means $u \approx^{k} v$. Then we have $pId_k(u) = pId_k(v)$. □

Proposition 3.1 is the basis of all algorithms in this chapter. The core idea is that a node's k-bisimulation partition block can be determined by its k-bisimulation signature, which in turn is determined by the $(k - 1)$-bisimulation partition of the graph. Intuitively, in order to compute the k-bisimulation partition, we compute the graph's *j-bisimulation* ($0 \leq j \leq k$) partitions bottom-up, starting from $j = 0$. We call each such intermediate computation the *iteration j computation*.

It is straightforward to show that the k-bisimulation partition of a graph is unique. Hence, in the sequel, we can safely talk about k-partition identifiers as unique objects. Also, note that we will use integer node identifier values to designate nodes in N. Therefore, in the following discussions the functions sig_k and pId_k both could take node identifiers (i.e., integers) as input.

Table 3.1.: *k-bisimulation* for the example graph in Figure 1.1 ($k = 0, 1, 2$)

nId	$pId_0(nId)$	$sig_1(nId)$	$pId_1(nId)$	$sig_2(nId)$	$pId_2(nId)$
1	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, E)\}$	G
2	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, F)\}$	H
3	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
4	B	$B, \{(l, B)\}$	E	$B, \{(l, D)\}$	J
5	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
6	B	$B, \{\}$	F	$B, \{\}$	K

Table 3.1 shows one way of assigning k-bisimulation ($k = 0, 1, 2$) partition identifiers and signatures for the example graph in Figure 1.1, where the *nId* denotes the unique identifier for each node, and $pId_i(nId)$ and $sig_j(nId)$ ($0 \leq i \leq 2$ and $0 < j \leq 2$) are presented accordingly. For $k = 0$, nodes are grouped into two partitions by node labels (given identifiers A and B). Then for $k = 1, 2$, signatures are constructed according to Definition 3.3 and distinct partition identifiers are assigned to distinct signatures, following Proposition 3.1.

3.3. k-bisimulation under RAM, BSP and MapReduce

Proposition 3.1 defines an iterative algorithm for computing k-bisimulation. The algorithm is not so different from the PageRank algorithm we've seen in Section 2.3. Essentially, for computing k-bisimulation of graph $G = \langle N, E, \lambda_N, \lambda_E \rangle$, we need to get $pId_k(v)$ for all $v \in N$. $pId_k(v)$ can be created by looking into $sig_k(v)$, which in turn depends on $pId_{k-1}(v)$. Pseudo code of this algorithm is in Algorithm 10.

Algorithm 10: Signature-based k-bisimulation algorithm for iteration k

```
Input: $(k-1)$-bisimulation partitioning of $G = \langle N, E, \lambda_N, \lambda_E \rangle$
Output: $k$-bisimulation of $G$
1 for each $v \in N$ do
2     construct $sig_k(v)$
3 create one-to-one mapping between $sig_k$ and $pId_k$    // e.g., use a dictionary
4 for each $v \in N$ do
5     save $pId_k(v)$ according to $sig_k(v)$
```

Following the algorithm transformation framework, we should first design the algorithm under the BSP model. From a vertex-centric point of view, for some node v, we need to send $pId_{k-1}(v)$ to v's direct predecessors in order for them to construct $sig_k(v)$. Furthermore, we need to assign for each different signature a distinct pId_k value. We achieve this by creating a new graph $G_s = \langle N_s, E_s \rangle$, where $N_s = N \cup \{sig_k(v) | v \in N\}$, and $E_s = \{(v, sig_k(v)) | v \in N\}$. In G_s, if we follow the edge directions, we can send nodes in N to their corresponding signatures. Then we can assign pId_k distributively on each processor. Then we follow the reverse edge direction of G_s, to send back the pId_k assignment to nodes. Pseudo code is described in Algorithm 11, with three supersteps.

Algorithm 11: *k*-bisimulation algorithm for iteration *k* under BSP model

```
Input: $(k-1)$-bisimulation partitioning of $G = \langle N, E, \lambda_N, \lambda_E \rangle$
Output: k-bisimulation of $G$
1 for each $v \in N$ do
2     send $pId_{k-1}(v)$ to its parent
3 synchronize
4 for each $v \in N$ do
5     construct $sig_k(v)$, send $v$ to $sig_k(v)$        // following edges in graph $G_s$
6 synchronize
7 for each $sig_k$ do
      // following reverse edges in graph $G_s$
8     assign $pId_k$, and send $pId_k$ back to all its related nodes
9 synchronize
```

Now we can design the corresponding MapReduce algorithm. Recall from Section 2.2 that one superstep in BSP corresponds to one MapReduce task. Such transformation is similar to that of PageRank and the pseudo code can be found in paper [LdLF$^+$13]. It is worth mentioning that line 5 to line 9 of Algorithm 11 can be heavily skewed, meaning that many nodes could have the same signature, therefore should be sent to the same processor. The same problem happens in the MapReduce algorithm as well. In [LdLF$^+$13], several skew elimination techniques are proposed to tackle this problem.

3.4. Data structures for EM model

We assume that graphs are saved on disk in the form of fixed column tables (node set as table N_t and edge set as table E_t). We also assume that these tables can have several copies sorted on different columns. In later discussions, we will use the notation $X.y$ to refer to column y of table X.

We have the following possible attributes for N_t (some attributes are iteration-specific):

nId	node identifier (note that this is the same as row identifier in the table; we leave this attribute here for clarity of the discussion).
nLabel	node label
pId_{old_nId}	*bisimulation* partition identifier for the given *nId* from last computation iteration
pId_{new_nId}	*bisimulation* partition identifier for the given *nId* from the current computation iteration
pId_{j_nId}	*j bisimulation* partition identifier for the given *nId* ($j = 0, 1, \ldots, k$)

and for E_t:

sId	source node identifier
tId	target node identifier
eLabel	edge label
pId_{old_tId}	*bisimulation* partition identifier for the given *tId* from last computation iteration

We further assume that we have a *signature storage facility* S, which stores the mapping between signatures and their corresponding partition identifiers. S is a data structure having only one idempotent function called *S.insert()*. For node $u \in N$, *S.insert()* takes $sig_j(u)$ $(0 \leq j \leq k)$ as input, and provides $pId_j(u)$ as output. Essentially *S.insert()* implements the one to one mapping function from sig_j to pId_j. The implementation details of S will be discussed in Section 3.5.2.

For ease of discussion and investigation, we assume in what follows that the N_t and E_t are each just one file sequentially filled with fixed length records. Moreover, in this chapter we make use of sort merge join to the extent possible, since it is a very basic way to achieve I/O efficient results. However, many possibilities could be explored for implementing these data structures (e.g., indexing techniques) and join algorithms to further optimize our presented results. We leave such investigations open for future research.

Finally, we also assume that we have a (possibly external memory based) priority queue available. In our empirical study below, we use the off-the-shelf I/O efficient priority queue implementation provided by the open source STXXL library [DKS08]. We use standard I/O complexity notions to analyze our algorithms [AV88] (EM model in Section 2.1).

3.5. Constructing k-bisimulation partitions in EM model

Algorithm 12: BUILD_BISIM(), compute the *k-bisimulation* equivalence classes of a graph

```
1  Input: $N_t$, $E_t$, $k$
   Output: $N_t$, $E_t$
2  if $k = 0$ then
3      fill in the $pId_{0\_nId}$ and $pId_{new\_nId}$ columns of $N_t$   // $O(sort(|N_t|)) + O(scan(|N_t|))$
4      return $N_t$, $E_t$
5  else
6      ($N_t$,$E_t$)← BUILD_BISIM($N_t$, $E_t$, $k-1$)   // $k > 0$, recursive call
7      if $k = 1$ then
8          $N_t \leftarrow$ sort($N_t$) by $nId$   // $O(sort(|N_t|))$
9          $E_t \leftarrow$ sort($E_t$) by $tId$   // $O(sort(|E_t|))$
10     scan $N_t$, move content of column $pId_{new\_nId}$ to $pId_{old\_nId}$   // $O(scan(|N_t|))$
11     fill in the $pId_{old\_tId}$ column of $E_t$   // $O(scan(|E_t|)) + O(scan(|N_t|))$
12     initialize $S$
13     $F \leftarrow \pi_\alpha(E_t)$, where $\alpha = (sId, eLabel, pId_{old\_tId})$
14     $F \leftarrow$ sort($F$) by $sId, eLabel, pId_{old\_tId}$, removing duplicates   // $O(sort(|E_t|))$
15     for each $uId \in \pi_{nId}(N_t)$ do
           // overall $O(scan(|E_t|)) + O(scan(|N_t|)) + cost\ of\ S$
16         construct $sig_k(uId)$ from $F$   // merge join with $F$
17         $pId_k(uId) \leftarrow S.insert(sig_k(uId))$
18         record $pId_k(uId)$ in $N_t.pId_{new\_nId}$ where $nId = uId$
19 fill in the $pId_k[nId]$ column of $N_t$
20 return $N_t$, $E_t$
```

We present our algorithm for *k*-bisimulation partition computation in Algorithm 12. The algorithm is inspired by Proposition 3.1, meaning for each node in the input graph, to construct its signature and find a one-to-one mapping number (partition identifier) for that signature.

In iteration $j = 0$, we assign distinct partition identifiers to nodes based on their *nLabel*s. For other iterations $j > 0$, our algorithm mainly performs two things for each node ID $uId \in \pi_{nId}(N_t)$ (line 15 to 18): (1) construct $sig_j(uId)$; and (2) insert $sig_j(uId)$ to S, record the returning $pId_j(uId)$ in the corresponding row in N_t. To prepare the necessary information for constructing $sig_j(uId)$, we need to fill in the missing columns of E_t (line 6 to 11). Several scans and sorts on tables are involved for each iteration. Note that some operations in the algorithm can be merged as one in practice. We present them separately just to make the presentation clearer. A detailed description is given in Section 3.5.1.

3.5.1. Details of Algorithm 12 (Build_Bisim())

Input and output The input variables of Algorithm 12 are node table N_t, edge table E_t and k, which is the degree of local bisimilarity from Definition 3.1. The output variables are N_t and E_t. The schema of N_t is (nId, $nLabel$, pId_{0_nId}, pId_{old_nId}, pId_{new_nId}); the schema of E_t is (sId, $eLabel$, tId, pId_{old_tId}).

$k = 0$, line 2 to 4 According to Definition 3.1, $k = 0$ means nodes having the same labels should be assigned the same partition identifier. We achieve this by sorting N_t on *nLabel* column. When scanning N_t, for each new *nLabel* we encounter, we assign a new integer (e.g., a predefined counter) to the corresponding *nId*, filling it in the pId_{0_nId} and pId_{new_nId} columns. This will take $O(sort(|N_t|)) + O(scan(|N_t|))$ I/Os. Using a hash map could achieve the same goal as well, with the same I/O upper bound.

Details of line 3 of Algorithm 12

```
1 sort $N_t$ by $nLabel$                                   // $O(sort(|N_t|))$
2 create variable $current\_pId$
3 for each $(nId, nLabel, pId_{0\_nId}, pId_{old\_nId}, pId_{new\_nId}) \in N_t$ do      // $O(scan(|N_t|))$
4     if $nLabel$ is new then
5         $current\_pId \leftarrow$ request a new $pId$
6     save $current\_pId$ to $pId_{0\_nId}$ and $pId_{new\_nId}$
```

$k > 0$, line 6 to 18 For $k > 0$, we first perform a recursive call to the algorithm, ensuring we work in a bottom-up manner. For iteration 1 ($k = 1$), we sort N_t and E_t on *nId* and *tId*, preparing them for later merge join operations. The algorithm's idea is to construct the signature of each node in order to distinguish it from other nodes according to the k-bisimilar relation. If we can properly fill in the pId_{old_tId} column of E_t, and join it with N_t on *nId=sId*, the information combined from columns $\{pId_{0\_nId}, eLabel, pId_{old\_tId}\}$ is enough for constructing the signature. The column *eLabel* is already filled in before algorithm starts. The column pId_{0_nId} is filled in during iteration 0 (line 2 to 4). The column pId_{old_tId} is filled in during each iteration $j > 0$ (line 11). Then for each node ID $uId \in N_t$, we get its $sig_k(uId)$, insert it to S in an I/O efficient way, getting $pId_k(uId)$ in return, and then placing this value in the pId_{new_nId} column of N_t.

At line 11 of Algorithm 12, to fill in the pId_{old_tId} column of E_t, we conduct a sort merge join of E_t and N_t (since both tables are sorted properly in iteration 1), replacing the content of pId_{old_tId} in E_t with pId_{old_nId} in N_t.

```
Details of line 11 of Algorithm 12
1 $E_t \leftarrow \pi_\alpha(E_t \bowtie_\phi N_t)$                    // merge join of $E_t$ and $N_t$
  // $\alpha : (E_t.sId, E_t.eLabel, E_t.tId, N_t.pId_{old\_nId})$, $\phi : E_t.tId = N_t.nId$
```

At line 16 of Algorithm 12, we sequentially construct the signature $sig_k(uId)$ for each $uId \in \pi_{nId}(N_t)$ according to Definition 3.3, and get the corresponding $pId_k(uId)$ (using *S.insert()*). All $pId_k(uId)$ will be written back to the pId_{new_nId} column of N_t (where *nId=uId*) right after, so that there is no random access to N_t. Note that although by definition sig_k is a set, we construct $sig_k(uId)$ as a string, maintaining elements of the set in sorted order. It is both an easy way for storing a set and handy for implementing S later on (e.g., using a trie).

```
Details of line 16 of Algorithm 12
1 create string $sig_k(uId) \leftarrow pId_0(uId)$                    // overall scan $N_t$
2 if $uId \in \pi_{sId}(F)$ then
3     for each $(uId, eLabel, pId_{old\_tId}) \in F$ do                // sequentially scan $F$
4         $sig_k(uId) \leftarrow sig_k(uId)+(eLabel, pId_{old\_tId})$
5 Get $pId_k(sId)$ from $S.insert(s)$, and save it to $nodeTable.pId_k(nId)$ column
  // $sId = nId$
```

3.5.2. Further discussion of Algorithm 12

Example run If we assume the numbering scheme for S is a self-increased counter across iterations, Table 3.1 (p. 28) would have the intermediate results for running Algorithm 12 on the example graph in Figure 1.1 (p. 1, $k = 2$), and Table 3.2 gives the final output of the algorithm.

Table 3.2.: Output of Algorithm 12 on example graph in Figure 1.1 ($k = 2$)

(a) N_t

nId	$nLabel$	pId_{0_nId}	pId_{old_nId}	pId_{new_nId}
1	M	A	C	G
2	M	A	C	H
3	P	B	D	I
4	P	B	E	J
5	P	B	D	I
6	P	B	F	K

(b) E_t

sId	$eLabel$	tId	pId_{old_tId}
3	l	1	C
1	w	2	C
2	w	2	C
5	l	2	C
4	l	3	D
1	l	4	E
2	l	6	F

Early stopping condition It is not always necessary to let the algorithm run k iterations. Indeed, it can be shown (Proposition 6.2) that after a bounded number of computation iterations, the partitioning result of Algorithm 12 will stop changing (i.e., achieve classical full bisimulation partitioning, Section 6.1). By simply checking if two consecutive iterations produce the same number of partition blocks (Proposition 6.1), we could decide whether the computation can stop.

Numbering schemes of partition identifier and S In the algorithm, the correctness of the partition identifiers' assignment is guaranteed level by level, meaning that the partition block numbering scheme from iteration j has nothing to do with that of iteration $j + 1$, for example. This means that we could use one counter for the whole computation, or could use different counters for each computation iteration.

The same idea also applies for implementing S. As long as S returns distinct pIds for different signatures for each computation iteration, it is immaterial to the work performed by Algorithm 12 if S is a new one for each iteration or not. So, we could use one S for all iterations (when we have a global counter), to reuse some signature pId across iterations. Furthermore, in practice there could potentially be benefits from warm caching (get a better hit ratio) for this approach. Moreover, for the maintenance algorithms presented in Section 3.6, we would only need to store one S instead of k of them. Essentially if the same signature appears many times in different iterations, we only save it once in S. The drawback of this method is that the size of S will keep increasing as the algorithm runs. This issue will become acute when the number of partitions becomes large and the signatures are long.

Data structures for S The signature storage facility S clearly plays an important role in Algorithm 12. In principle, any data structure that permits an efficient set-equality check will be sufficient. Trie and dictionary are such data structures, for instance. During our experiments, we see that in many of the cases, partition sizes are small and the signatures are short, for which a main memory based data structure is enough. In other cases, signature length could reach several million and partition size into tens of millions, then we need some external memory based solution for S. We could, for example, sort all signatures from F in an I/O efficient way [AFGV97], then when scanning these signatures, partition identifiers are assigned. In this case, the overall cost of the *S.insert()* operation could still be bounded by $O(sort(|E_t|))$. Other disk based solutions, such as disk-based tries (e.g., String B-Tree [FG99] or [GO12]) or inverted files (e.g., [Mam03]) could also be considered.

In our experiments we use BerkeleyDB (B-Tree or Hash index) to mimic a trie, which, as we show in the experimental results, has acceptable empirical behavior.

Complexity and correctness We have the following characterization of Algorithm 12.

Theorem 3.1. *Let $k \geq 0$ and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. Algorithm 12 computes the k-bisimulation partition of G with I/O complexity of $O(k \cdot sort(|E_t|) + k \cdot scan(|N_t|) + sort(|N_t|))$, and space complexity of $O(|N_t| + |E_t|)$.*

Proof. After all the I/O cost of one iteration of *k-bisimulation* computation is bounded by $O(sort(|E_t|)) + O(scan(|N_t|))$, k is a given input, and there is one extra sort on N_t in iteration 1. Hence Algorithm 12 has the I/O complexity of $O(k \cdot sort(|E_t|) + k \cdot scan(|N_t|) + sort(|N_t|))$.

During computation, only one N_t and E_t are used, and S is used. The space upper bound for S is the same as the space upper bound for all signatures. Since in the algorithm, we construct all signatures by joining the information from N_t and F (which is a projection of E_t), the space upper bound of S is $O(|N_t| + |E_t|)$. Therefore, the overall space complexity upper bound of Algorithm 12 is $O(|N_t| + |E_t|)$.

We prove correctness inductively.

(1) $k = 0$. Since we are following the definition, this is obvious.

(2) $k > 0$. Assume we get the correct $(k-1)$ *bisimulation* partitioning results. In iteration k, for each node u in N_t, we construct $sig_k(u)$ and insert it in S to get $pId_k(u)$. According to Proposition 3.1 and the definition of S, we are sure that $pId_k(u)$ is correct. □

Differences and connections with Hellings et al. As indicated in Section 3.1, the only known solutions for computing bisimulation on graphs in external memory are those of Hellings et al. [HFH12], with I/O complexity of $O(sort(|N_t| + |E_t|))$. There are two critical differences between their work and ours. (1) *Targeting different problems.* The solutions of Hellings et al. are designed specifically for computing the *full* bisimulation of acyclic graphs. Our approach does not rely on such structure, computing k-bisimulation regardless of the presence or absence of cycles in the graph. (2) *Using different techniques.* Hellings et al. compute partition blocks level by level, starting from the leaf nodes of the graph. Our approach constructs all partition blocks at each iteration, using data structures and processing strategies which are not tied to any (a)cyclic structure in the graph.

It is also worth mentioning the connections between these two methods. As pointed out by prof. dr. M.T. de Berg, there is a way to transform an arbitrary graph to an acyclic graph, and applying the method of Hellings et al. to it will produce the same result as ours. The graph transformation works as follows.

For some graph $G = \langle N, E\rangle$, build graph $G^T = \langle N^T, E^T\rangle$, such that for each $n \in N$, there are k copies of n in $N^T : n_1, n_2, \ldots, n_k$. For each edge $(u, v) \in E$, we put (u_{i+1}, v_i) in E^T $(1 \leq i < k)$. Then G^T is guaranteed to be acyclic. This approach essentially materializes the message passing routes between iterations of Algorithm 12. And it is easy to show that we can in the end get the k-bisimulation partitioning results of G from G^T. The construction of G^T however, needs k times of sort merge join of N_t and E_t, which is exactly the same as in Algorithm 12. Furthermore, the size of G^T is $k \cdot |N_t| + k \cdot |E_t|$, which makes the overall I/O cost of this approach $O(sort(k \cdot |N_t| + k \cdot |E_t|))$, a bit more than Algorithm 12.

3.6. Maintenance of k-bisimulation partitions in EM model

It is easy to show that any edge and node updates on a graph can potentially change the complete k-bisimulation partition of the graph. Therefore, in the worst case, the upper bound of such maintenance cost is the cost of recomputing the k-bisimulation partition from scratch. However, when dealing with real graphs, as we shall see in Chapter 4, in many cases there is still hope to use data structures such as dictionary (S) and priority queue to maintain the correct partition result instead of recomputing everything. In this section we propose several algorithms for this purpose.

For maintenance algorithms we assume that we have constructed the k-bisimulation partition of graph $G = \langle N, E, \lambda_N, \lambda_E \rangle$, where, as before, G's N_t and E_t are stored on disk, containing the historical information kept in N_t (Table 3.4); E_t is the same as in Algorithm 12, but has two copies with sort orders (*sId,tId*) and (*tId,sId*) to boost performance. We use E_{tst} and E_{tts} to refer to each of these copies.

Table 3.4.: N_t for maintenance algorithms

nId	*nLabel*	pId_{0_nId}	pId_{1_nId}	$\dots$	pId_{k_nId}

We further assume that we save the signature storage facility S on disk, which we use and update throughout the maintenance process.

The maintenance problem includes the following subproblems.

Change k If k increases, we carry out another iteration of computation. If k decreases, the result can be returned directly since we keep the history information in N_t.

Add a set of new nodes (Add_Nodes()) When adding a set of new nodes, we assume the new nodes are isolated, stored in the *newNodes* table, which has the same schema as N_t, and that $|newNodes| = O(|N_t|)$. We first sort N_t and *newNodes* by *nLabel*, then perform a merge join on the *nLabel* column to fill in the pId_{0_nId} column of *newNodes* for all the existing *nLabel*. For the missing ones, we request a new *pId* for each of the new *nLabel*. Then we get the $pId_1, \dots, pId_k$ of the *newNodes* by inserting its pId_0 to S. At the end we append the whole *newNodes* to N_t. The I/O complexity of ADD_NODES() is bounded by $O(sort(|N_t|))$. Pseudo code is in Algorithm 16.

Add a set of new edges (Add_Edges()) For adding a set of edges, we assume that the edges are added between existing nodes. If this is not the case, we first call procedure ADD_NODES(). The new edges are stored in the *newEdges* table, having the same schema as E_t. For inserting one edge (s, l, t) to G, the potential changes are to $sig_j(s)$ $(1 \leq j \leq k)$, as well as those signatures of all ancestors of s within k steps. So the main work is to detect whether there is some change in $sig_j(s)$ and propagate those change(s) to its parent nodes' signatures in later iterations. We use a priority queue ***pQueue*** to record and process such changes in a systematic, level-wise manner. For some node ID *uId* and iteration *j*, *pQueue* stores the pair *(j,uId)* as

Algorithm 16: Add_Nodes(), add a set of new nodes to existing *k-bisimulation* partition

```
  Input: $N_t$, $S$, table of new nodes $newNodes$, $k$
  Output: $N_t$, $S$
1 $N_t \leftarrow$ sort($N_t$) by $nLabel$                                   // $O(sort(|N_t|))$
2 $newNodes \leftarrow$ sort($newNodes$) by $nLabel$                          // $O(sort(|N_t|))$
3 $newNodes \leftarrow \pi_\alpha(newNodes \rtimes_\phi (N_t))$, remove duplicates    // $O(scan(|N_t|))$
   // $\alpha : (newNodes.nId, newNodes.nLabel, N_t.pId_{0\_nId}, \dots)$
   // $\phi : newNodes.nLabel = N_t.nLabel$,  $\beta : (nLabel, pId_{0\_nId})$
4 request a new $pId$ for each new $nLabel$ in $newNodes$, fill in all the NULL fields in
  $newNodes.pId_{0\_nId}$
5 for each $uId \in \pi_{nId}(newNodes)$ do                  // overall $O(scan(|N_t|))$ + cost of $S$
6     get value of $S.insert(pId_0(uId))$, use it for $pId_{1\_nId}, \dots, pId_{k\_nId}$ of $uId$
7 append $newNodes$ to $N_t$
8 return $N_t$, $S$
```

priority reference. Then whenever we dequeue one element from *pQueue*, we get the smallest node ID from the lowest iteration (lowest priority reference). Therefore *pQueue* indicates those nodes whose signatures could change in each iteration level (from 1 up to k).

At the beginning of the algorithm, we enqueue (j, s) to *pQueue* $(\forall(s, l, t) \in E_t, 0 < j \leq k)$. Then, while *pQueue* is not empty, we dequeue the list of (j, uId) pairs with the same j out of the queue, construct the new signature of each such *uId*, insert it to S, and compare the returning $pId_j(uId)$ with the old pId_{j_nId} value of *uId*. If the *pId* remains the same as the old one, we continue; if it changes, we record $pId_j(uId)$ in N_t, and enqueue all $(j+1, vId)$ pairs to *pQueue* where $vId \in \pi_{sId}(\sigma_{tId=uId}(E_t))$. Pseudo code is given in Algorithm 17, and a detailed discussion is in Section 3.6.1.

Deletions Deletions follow a similar idea to insertions. For example, when removing an edge (s, l, t), it is the same idea as adding one. We also (potentially) modify the signature of s, propagating changes to its ancestors via *pQueue*, then the reasoning is the same. When removing a node, we first remove each incoming edge and each outgoing edge for that node. Then we remove the node from N_t.

3.6.1. Details of Algorithm 17 (Add_Edges())

Input and output The input variables of Algorithm 17 are node table N_t, edge tables E_{tst} and E_{tts}, the signature storage facility S, the new edge set *newEdges* and k.

Algorithm 17: ADD_EDGES(), add a set of new edges to existing *k-bisimulation* partition

```
Input: $N_t$, $E_{tst}$, $E_{tts}$, $S$, a table of new edges $newEdges, k$
Output: $N_t$, $E_{tst}$, $E_{tts}$, $S$
1  if $k = 0$ then
2      merge $newEdges$ into $E_{tst}$ and $E_{tts}$                // $O(sort(|E_t|))$
3  else                                                              // $k > 0$
4      $N_t \leftarrow$ sort($N_t$) by $nId$                         // $O(sort(|N_t|))$
5      create empty priority queue $pQueue$                          // overall $O(sort(|N_t|))$
6      for $j \in \{1, \ldots, k\}$ and $(s, l, t) \in newEdges$ do
7          enqueue $(j, s)$ to $pQueue$
8      merge $newEdges$ into $E_{tst}$ and $E_{tts}$, fill in the $pId_{old\_tId}$ column  // $O(sort(|E_t|))$
9      while $pQueue$ is not empty do
10         dequeue all pairs $(j, uId)$ from $pQueue$ with the same (i.e., smallest) $j$
           value, save all distinct $uId$ to $M$                     // remove duplicates
11         $F \leftarrow \sigma_{sId \in M}(E_{tst})$                // merge join, $O(scan(|N_t|) + scan(|E_t|))$
12         fill in the $pId_{old\_tId}$ column of $F$  // $O(scan(|N_t|) + O(sort(|E_t|)) + O(scan(|E_t|)))$
13         $H \leftarrow \pi_\alpha(F)$, where $\alpha$=($sId$, $eLabel$, $pId_{old\_tId}$)
14         $H \leftarrow$ sort $H$ on $sId$, $eLabel$, $pId_{old\_tId}$, and remove duplicates
           // scan $M$, $N_t$ and $H$, overall $O(scan(|N_t|)) + O(scan(|E_t|))$ + cost of $S$
15         for each $uId \in M$ do
16             construct $sig_j(uId)$ from $H$
17             $pId_j(uId) \leftarrow S.insert(sig_j(uId))$
18             if $pId_j(uId)$ is not the same as the corresponding value in $N_t.pId_{j\_nId}$
               then
19                 propagate changes to $N_t$ and $pQueue$  // $O(scan(|N_t|)) + O(scan(|E_t|))$
20 return $N_t$, $E_{tst}$, $E_{tts}$, $S$
```

The output variables of Algorithm 17 are N_t, E_{tst}, E_{tts} and S. N_t's schema is given in Table 3.4, while E_{tst}, E_{tts} and *newEdges*'s schema is the same as E_t in Algorithm 12.

$k = 0$, line 1 to 2 of Algorithm 17 For $k = 0$, since all nodes' information is properly filled (including the pId_{0_nId} column in N_t), we only need to add new rows to E_{tst} and E_{tts} according to *newEdges*.

$k > 0$, line 3 to 19 of Algorithm 17 For $k > 0$, for each iteration, which is indicated by j in the algorithm, we need to (1) find out the potential nodes whose signatures could have changed; (2) check whether these signatures have been changed or not;

and, (3) propagate any such changes to the parents of these nodes. To record the potential nodes and to perform the propagation, we use a priority queue *pQueue*. To check signature changes, we reuse the signature storage facility S.

When adding a new edge $(s, l, t) \in newEdges$ to the graph, all $sig_j(s)$ $(j > 0)$ have the potential to change, and hence we add all pairs (j, s), for $j \in \{1, \ldots, k\}$, to *pQueue*, indicating that we need to check the signature of s in every iteration (line 6 to 7). For each iteration $j > 0$, we dequeue from *pQueue* all node IDs in the smallest iteration j, remove duplicates, and save them to a temporary table M, so that M contains in sorted order all node IDs whose signatures would change in iteration j. Then we create an extra table F, preparing for signature constructions. This is achieved by performing a merge join of E_{tst} and M (where $E_{tst}.sId \in M$). Then we fill in $F.pId_{old\_tId}$ column, as in Algorithm 12.

After projection on the (sId, $eLabel$, pId_{old_tId}) of F and removing duplicates, we get H (line 14), and are ready to construct the signatures. For each $uId \in M$, we construct $sig_j(uId)$ according to the signature definition. The idea of constructing the nodes' signatures is the same as line 16 of Algorithm 12, only in this case we are not considering every node but only those appearing in *pQueue* (and later in M).

We then call $S.insert(sig_j(uId))$ for all such uId. If S returns the same $pId_j(uId)$ as recorded in $N_t.pId_{j\_nId}$, nothing will happen; otherwise we change the $N_t.pId_{j\_nId}$ entry of uId accordingly, and propagate the changes to *pQueue*. If $j < k$, we add all parents of uId to *pQueue* to indicate that we will check these nodes' signatures in the $j + 1$ iteration.

3.6.2. Further discussion of Algorithm 17

Example run We present different behaviors of Algorithm 17 using two examples. Here we will extend the graph from Figure 1.1 as in Figure 3.1. The dashed lines in this figure indicate the two edges which we will add in our examples.

First suppose we add edge $(2, l, 7)$ to the original graph of Figure 1.1, where node 7 is a new node with label P. Table 3.5 shows the resulting partition after this insertion. The new/changed part of the table is indicated in gray. When the algorithm starts, (1, 2) and (2, 2) are added to *pQueue*. Then after checking each of these, the algorithm finds no change in node 2's signature, therefore no change propagates, and the algorithm stops. We see that comparing with Table 3.1, the only thing that changes is to add one more row (node 7) to the table. Since node 7 does not have outgoing edges, adding one edge that points into node 7 will not change any existing nodes's

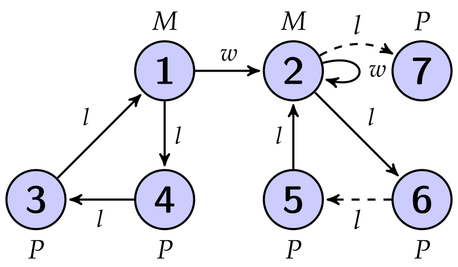


Figure 3.1.: Updates on the example graph

signature. Node 7 belongs to the group of node 6, and no other node changes group membership.

Table 3.5.: 2-*bisimulation* for the example graph after edge insertion $(2, l, 7)$

nId	$pId_0(nId)$	$sig_1(nId)$	$pId_1(nId)$	$sig_2(nId)$	$pId_2(nId)$
1	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, E)\}$	G
2	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, F)\}$	H
3	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
4	B	$B, \{(l, B)\}$	E	$B, \{(l, D)\}$	J
5	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
6	B	$B, \{\}$	F	$B, \{\}$	K
7	B	$B, \{\}$	F	$B, \{\}$	K

In the second case, suppose we add edge $(6, l, 5)$ to the original graph of Figure 1.1. The algorithm first add (1, 6) and (2, 6) to *pQueue*. Then in iteration 1, the algorithm detects that the signature of node 6 does change, and therefore adds one new pair (2, 2) to *pQueue*. In iteration 2, both node 2 and node 6's signatures are checked, and they are both changed. We see that in Table 3.6 $pId_2(1)$ and $pId_2(2)$ become the same, while $pId_2(6)$ changes from K to I.

Complexity and correctness We have the following characterization of Algorithm 17.

Theorem 3.2. *Let $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph and $k \geq 0$. After adding a set of new edges to G, Algorithm 17 correctly updates the k-bisimulation partition of G with I/O complexity of $O(k \cdot sort(|E_t|) + k \cdot sort(|N_t|))$, and space complexity of $O(k \cdot |N_t| + k \cdot |E_t|)$.*

Table 3.6.: 2-*bisimulation* for the example graph after edge insertion $(6, l, 5)$

nId	$pId_0(nId)$	$sig_1(nId)$	$pId_1(nId)$	$sig_2(nId)$	$pId_2(nId)$
1	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, E)\}$	G
2	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, E)\}$	G
3	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
4	B	$B, \{(l, B)\}$	E	$B, \{(l, D)\}$	J
5	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
6	B	$B, \{(l, B)\}$	E	$B, \{(l, D)\}$	J

Proof. After all the I/O cost of one iteration of Algorithm 17 is bounded by $O(sort(|E_t|)) + O(sort(|N_t|))$, and the upper bound of the number of iterations is k. In the worst case, *pQueue* will sort all edges for all iterations, which gives us $O(sort(k \cdot |E_t|))$. Hence Algorithm 17 has the given I/O complexity.

During computation, only one N_t and E_t are used, and S is used. Here the node table contains historical information from iteration 0 to k, so comparing with the original N_t, the space upper bound is $O(k \cdot |N_t|)$. Also according to the algorithm, every iteration would have to save its signature mapping to S, so the space upper bound of S is $O(k \cdot |E_t|)$. Therefore, the overall space complexity upper bound of Algorithm 17 is $O(k \cdot |N_t| + k \cdot |E_t|)$.

Let (s, l, t) be the new edge. After we insert s, t to N, $pId_0(u)$ will not change for any $u \in N$. So, according to Definition 3.3, there are only two ways that $sig_j(u)$ $(0 < j \leq k)$ could be affected:

(1) a new pair $(\lambda_E(v), pId_{j-1}(v))$ appears, or

(2) changes of $pId_{j-1}(v)$ in some existing pair $(\lambda_E(v), pId_{j-1}(v))$, where v is some child of u.

Case (1) can only be caused by adding a new edge to u, so that in our case this can only happen to $sig_j(s)$ $(0 < j \leq k)$, and we capture these changes in line 7 of Algorithm 17. The second case can only happen when the pId_{j-1} for the children of u changes. We capture (and propagate) these changes in line 19 of Algorithm 17. Therefore, we capture all changes in the signatures of $u \in N$, and recompute the signatures accordingly. Hence Algorithm 17 produces the correct *k-bisimulation* partitioning result. □

When to switch back to Algorithm 12 As we will see in our empirical study (Section 4.4.4), it is not always beneficial to use Algorithm 17, since it performs extra work in each iteration. Heuristics could be adopted to decide when to switch back to Algorithm 12. For example, if at a certain iteration, most of the nodes are placed into *pQueue*, it is more beneficial to switch back to Algorithm 12. This could be done by simply checking the size of *pQueue* at the beginning of each iteration.

3.7. Conclusion

In this chapter, with the help of the algorithm transformation framework from Chapter 2, we have presented signature-based *k*-bisimulation partitioning algorithms under the RAM model and BSP model, and to our knowledge, the first I/O-efficient general-purpose algorithms for constructing and maintaining *k*-bisimulation partitions on massive disk-resident graphs. The I/O cost of the construction algorithm is bounded by $O(k \cdot sort(|E_t|) + k \cdot scan(|N_t|) + sort(|N_t|))$, and the maintenance algorithms are bounded by $O(k \cdot sort(|E_t|) + k \cdot sort(|N_t|))$. It is clear that the worst case I/O-bound of the maintenance algorithms is slightly more than that of the construction algorithm, also with a bigger storage requirement. Comparisons of both methods are presented in the next chapter.

4. Empirical analysis of k-bisimulation algorithms

4.1. Introduction

In Chapter 3, we learned the rich history of bisimulation and studied efficient algorithms for computing k-bisimulation in various settings. In this chapter, we present the results of an in-depth experimental study of the algorithms on both synthetic and real datasets. After introducing the experiment setup, we show the performance of the construction algorithm (Build_Bisim()) and edge update algorithm (Add_Edges()) under the EM model in Section 4.3 and 4.4 respectively.

4.2. Experiment setting

Environment The following experiments are run on a machine with 2.27 GHz Intel Xeon (L5520, 8192KB cache) processor, 12GB main memory, running Fedora 14 (64-bit) Linux. We use C++ to implement all the algorithms, using GCC 4.4.4 as the compiler. We use the open-source STXXL library [DKS08] to construct the tables and perform the external memory sorting, and use Berkeley DB to implement S. One S is used for all computation iterations (as discussed in Section 3.5.2). In the experiments we do not exploit any parallelism and restrain ourselves with predefined buffer sizes. We record the running time as well as the I/O volume between the buffer and the disk system. Therefore, the performance (time) of the experiments is comparable to a commodity PC, and the I/O volume can be repeated on other systems. In the following experiments, we set both the STXXL buffer and Berkeley DB buffer to be 128MB, if not otherwise indicated. Please note that we run experiments for the Twitter dataset on a different machine (Intel Xeon E5520, 2.27 GHz, 8192KB cache, 70G main memory, same OS) for limited disk space reason, using a 512MB/512MB buffer setting.

Datasets To prove the practicability of the algorithms, we experiment with various graph datasets. The datasets are collected from public repositories, ranging from synthetic data to real-world data, from several million of edges to more than 1.4 billion edges. In Table 4.1 we give a description of the datasets, as well as some simple statistics of them. All datasets are accessed on 15 May 2012. Note that in the following we show the experiment results on a subset of the datasets when the result is representative enough, which is consistent as in paper [LFH+13b].

Table 4.1.: Description and statistics of the real and synthetic graph datasets

Data Name	Description	$\|N\|$	$\|E\|$	$\frac{\|E\|}{\|N\|}$
Jamendo (**RE**)	A repository of music metadata in RDF format [RSM08]	0.49M	1.05M	2.16
LinkedMDB (**RE**)	A repository of movie metadata in RDF format [HC09]	2.33M	6.15M	2.64
DBLP (**RE**)	An RDF format DBLP dump[1]	23M	50.2M	2.18
WikiLinks (**R**)	A page-to-page linking graph of Wikipedia[2]	5.71M	130.16M	22.79
DBPedia (**RE**)	An early RDF dump of DBPedia[3]	38.62M	115.3M	2.99
Twitter (**R**)	A following relationship graph of Twitter [KLPM10]	41.65M	1468.4M	35.25
Flickr-Grow (**R**)	A following relationship graph of Flickr [MKG+08]	1.5M to 2.3M	17.7M to 33.1M	11.68 to 14.39
SP2B (**SE**)	A RDF data generator for arbitrarily large DBLP-like data [SHLP09]	280.91M	500M	1.78
BSBM (**SE**)	A RDF data generator for e-commerce use case [BS09]	8.89M	34.87M	3.92
Random (**SE**)	Uniform distribution graph generated by GTgraph [BM]	10M	200M	20
Power (**SE**)	Power-law distribution graph generated by GTgraph [BM]	8.39M	200M	23.85

* **R** and **S** indicate whether the graph is from real data or synthetic data. **E** and **N** indicate if the graph is labeled on edge and/or node.

4.3. Experiments on the construction algorithm (Build_Bisim())

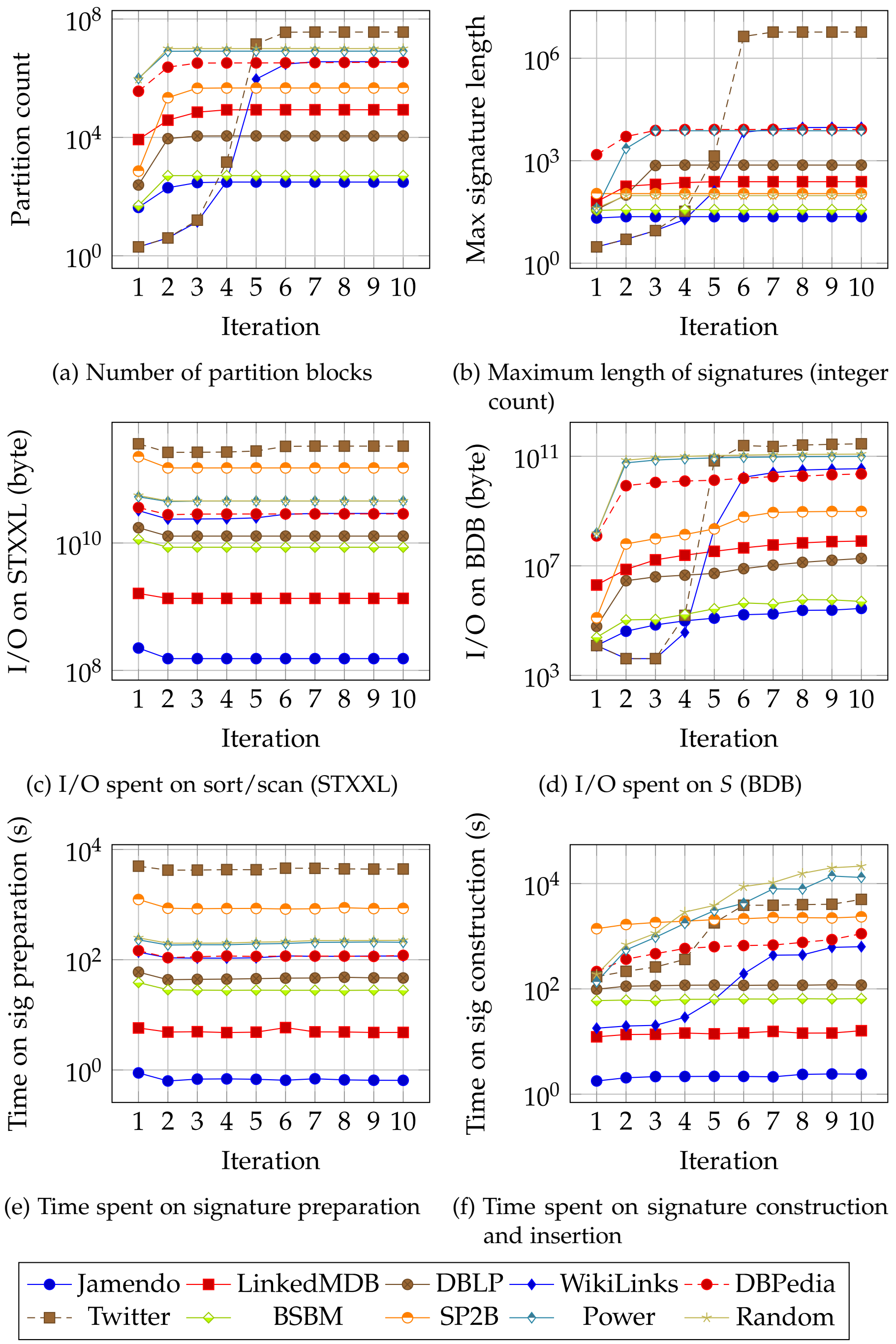
In Figure 4.1 we show the experiment results for Algorithm 12 on all datasets. We compute the 10-bisimulation (i.e., $k = 10$) of these datasets and measure many aspects of the running behavior for each iteration. Concerning time measurement, we run every experiment 5 times and take the average number. We notice that the standard deviation over the average is less than 10% for all datasets. S uses BerkeleyDB's B-Tree index in this experiment.

Partition blocks count In Figure 4.1a, we show the number of partition blocks every iteration produces for all datasets. We see that the numbers vary from one dataset to another, where the difference is sometimes more than an order of magnitude, and interestingly, does not directly relate to the size of the dataset. In certain cases (e.g., Twitter) the partition size is quite large. Moreover, many of the datasets (e.g., Jamendo, LinkedMDB, DBLP, etc.) reach full bisimulation after 5 iterations. In fact, all datasets (including Twitter) get sufficient partitioning resultd after 5 iterations of computation. Here we can reasonably argue that even for the Twitter dataset, the partition results after 5 iterations are too refined (the reduced graph is almost as big as the original graph, e.g., (partition count)/(node count) > 0.8).

Maximum signature length Figure 4.1b shows the maximum length of signatures for each iteration. We observe that the signature length is usually quite short, especially comparing with the size of the graph. But there are still cases (e.g., Twitter) for which the signature becomes very long (more than 1 million integers), which stresses the need for an I/O efficient solution for S. Note that the synthetic datasets, such as BSBM and SP2B, reach their full bisimulation partition after 3 iterations of computations, and have rather short signatures, indicating that they are highly structured.

I/O measure Figures 4.1c and 4.1d show the I/O volume spent on sorting/scanning (STXXL) and on interacting with S (Berkeley DB). We see for most of the datasets, there is no dramatic change across different iterations. But for Wikilinks and Twitter,

[1] `http://thedatahub.org/dataset/l3s-dblp`
[2] `http://haselgrove.id.au/wikipedia.htm`
[3] `http://www.cs.vu.nl/~pmika/swc/btc.html`



(a) Number of partition blocks

(b) Maximum length of signatures (integer count)

(c) I/O spent on sort/scan (STXXL)

(d) I/O spent on S (BDB)

(e) Time spent on signature preparation

(f) Time spent on signature construction and insertion

Figure 4.1.: Experiment results for Algorithm 12 for real and synthetic datasets for each iteration ($k = 10$)

the two datasets which have very few partition blocks at the beginning and many at the end, there is a noticeable difference on S for different iterations. In this case I/O on S becomes a comparable factor with sort and scan (I/O on STXXL).

Time measure Figure 4.1e shows the time spent on preparing the signature (line 6 to 14 in Algorithm 12) for each iteration, which is quite stable for all datasets. Figure 4.1f shows the time on constructing the signature and insert into S (line 15 to 18 in Algorithm 12). In this case datasets with higher degrees tend to cost more time in later iterations, which correlate with their longer signatures and larger number of partition blocks. For all datasets, however, the operations on constructing and looking for signatures are the dominant factor for each iteration. This brings us to think about further optimization tasks on the construction of signatures and the implementation of S.

We can conclude that the algorithm is practical to use. For graphs with 100 million edges (e.g., WikiLinks and DBPedia), the algorithm can process them in under 700 seconds for one iteration, or 1 hour for them to achieve full bisimulation. For use cases such as building structural indexes for graph database, time-consuming index construction is expected. 1 hour of index creation on a hundred-million-edge graph is quite efficient. Furthermore, the operation is bounded by machine's I/O performance, and scales (almost) linearly with the number of nodes and edges. If we switch to higher throughput devices (e.g., SSD/SCSI), the result will be even better.

4.3.1. Different implementations of S

As we mentioned in Section 3.5.2, S could be implemented in several ways. In Figure 4.2 we compare the overall I/O performance of Build_Bisim() using B-Tree and Hash indexes for S on several datasets. We notice that the B-Tree implementation slightly outperforms Hash Index for all datasets. This is most likely due to small caching effects and locality of references during construction of the signatures.

4.3.2. The effect of different buffer sizes

We allocate two buffers, one for scan and sort (STXXL buffer in our case), one for S (BerkeleyDB buffer in our case), in order to analyze the impact of buffer size on our algorithms. To illustrate, we take the DBPedia dataset since it is large enough to show buffer effects. For the sort/scan setting, we set the buffer size ranging from 16MB to 512MB, while keeping the S buffer to 128MB, recording the I/O between

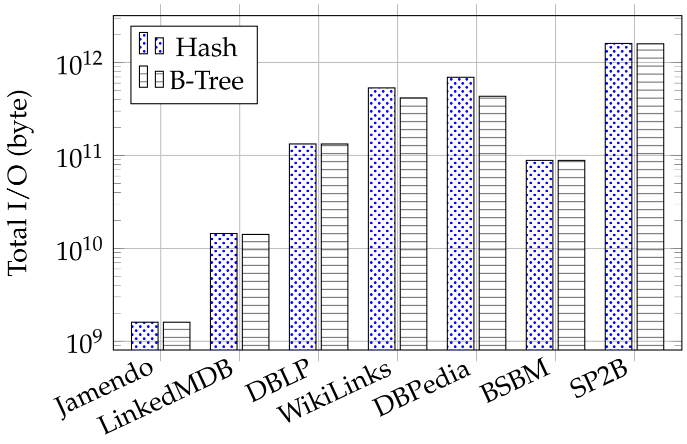
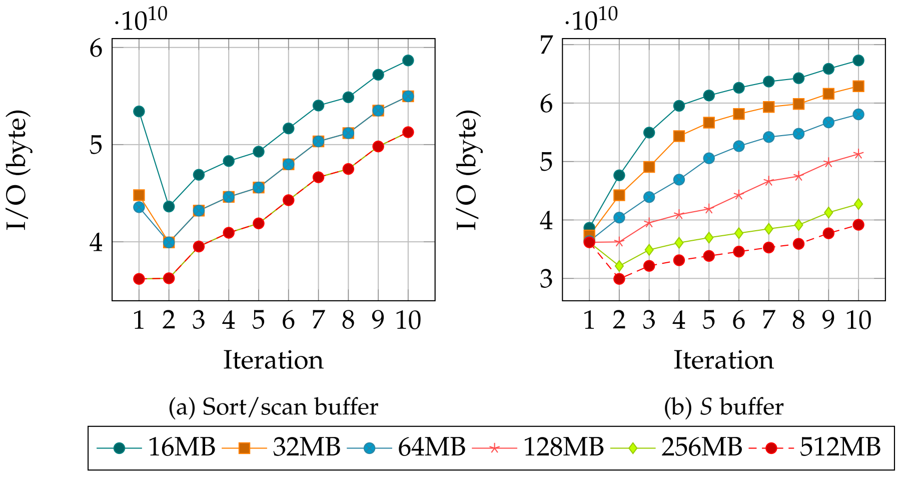


Figure 4.2.: I/O comparison for B-Tree and Hash index of S ($k = 10$)

the buffer and the disk system. From Figure 4.3a we see that a bigger buffer does improve the performance. But since we only gain in the external memory sorting part, a certain amount of I/Os is inevitable for each iteration. Note that the reason why iteration 1 has higher I/O cost is that in iteration 1 extra sorts on N_t and E_t are performed.



(a) Sort/scan buffer

(b) S buffer

Figure 4.3.: I/O for different buffer size setting for sort/scan and S ($k = 10$)

For the setting on S, we set the buffer size ranging from 16MB to 512MB, while keeping the sort/scan buffer to be 128MB, recording the I/O of the buffer to the disk system. From Figure 4.3b we also see that more buffer brings less I/O, as expected. However, in this case the buffer size change has a bigger impact on the I/O performance. This indicates that if we have a certain amount of memory space, it is more beneficial to allocate more memory to the S buffer than to the sort/scan buffer. Note that the S buffer also shows a quite high hit ratio during execution (more than 0.98 for DBPedia in all settings).

4.3.3. Scalability

In order to measure how well the algorithm scales, we generate different size of SP2B datasets (edge count 1M, 5M, 10M, 50M, 100M, 500M), and measure the I/O and elapsed time for each dataset. In Figure 4.4 we see that the time spent on each edge is on the order of 10^{-5} seconds, and the I/O spent on each edge is under 4000 bytes (which is one typical disk page size). The algorithm's performance scales (almost) linearly with the data size.

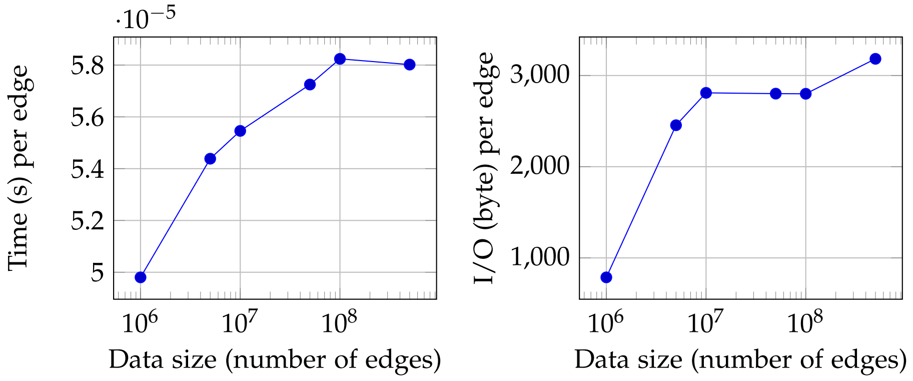


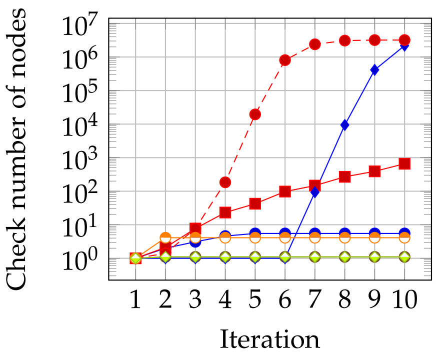
Figure 4.4.: Time and I/O spent on each edge on average ($k = 10$)

4.4. Experiments on the edge update algorithm (Add_Edges())

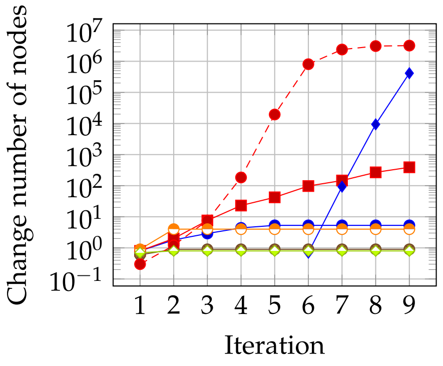
Edge updates are common operations for graph data. For our datasets, adding one edge means to add a link between two wiki pages (WikiLinks), to add more information to one publication or author (DBLP), to follow one more person (Twitter) and so on. Sometimes we would like to also add several edges together at once. So in this subsection we test the performance of Algorithm 17 (Add_Edges()), first adding a single edge and then adding a set of edges.

4.4.1. Observations on single edge update

To create the dataset for testing, we randomly take one edge from the edge set, perform BUILD_BISIM() on the rest of the dataset, and apply ADD_EDGES() on this edge. We believe the edge selection is more natural this way, since it takes into account the distribution of edges among nodes. We repeat the experiment 10 times and take the average of the measured numbers. In Figure 4.5a we show how many nodes are checked for adding one edge to the graph in each iteration. In Figure 4.5b we show how many nodes actually change their partition IDs in each iteration. From the figures we see that the behavior varies for different datasets; graphs that have larger degrees tend to propagate more changes to later iterations, which complies with our intuition.



(a) Number of nodes being checked



(b) Number of nodes that change partitions

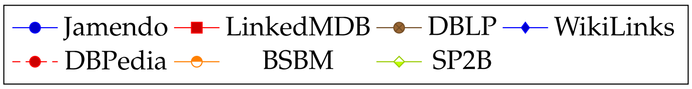


Figure 4.5.: Experiment on ADD_EDGES() when $k = 10$

Since there is a chance that many nodes are changed, but may all belong to a certain set of partitions, we also examine how many partitions change their members in each iteration. We see that the behavior is closely related to that of Figure 4.5b.

4.4.2. Single edge update (Build_Bisim() vs. Add_Edges())

After edge insertion, if there is no update algorithm available, the only choice to get the *k*-bisimulation partition is to execute the Build_Bisim() from scratch on the new dataset. So this would be the baseline for the Add_Edges() algorithm to compare. In the following we compare the overall I/O and time (Figure 4.6) of the two algorithms. We see that indeed the Add_Edges() algorithm always achieves a better performance than using Build_Bisim() to recompute the *k*-bisimulation partition result from scratch, with up to an order of magnitude improvement.

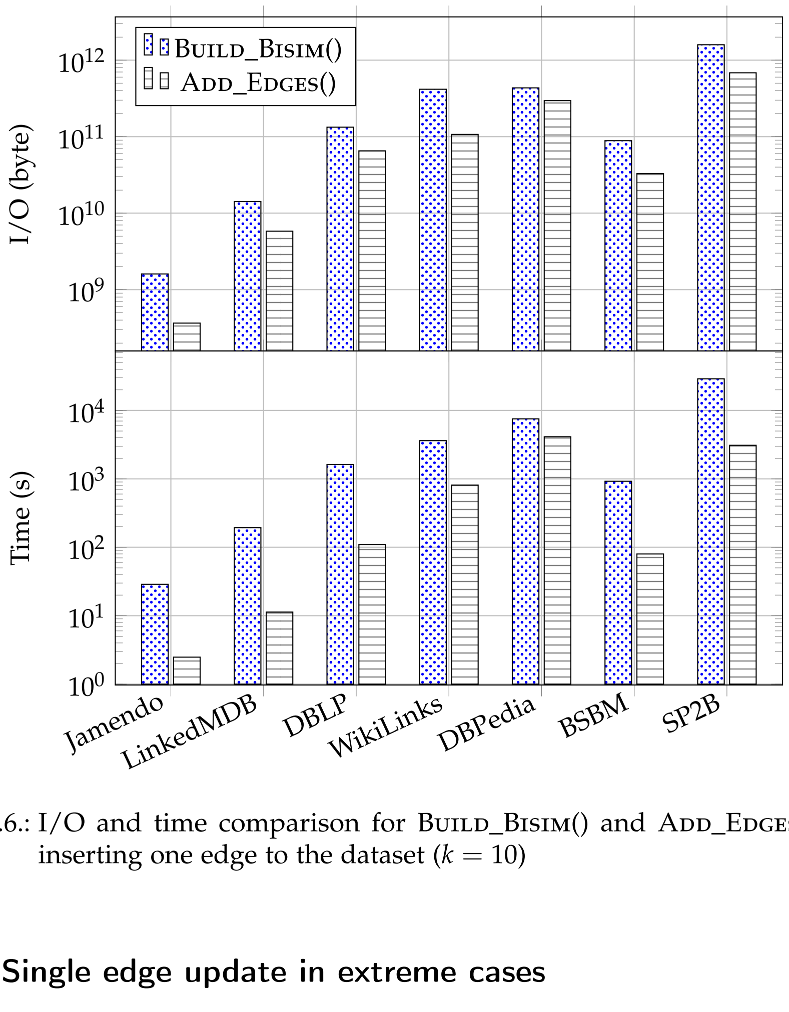


Figure 4.6.: I/O and time comparison for Build_Bisim() and Add_Edges() after inserting one edge to the dataset ($k = 10$)

4.4.3. Single edge update in extreme cases

From the previous experiments, we see that the performance of the algorithms is highly related to the datasets they process. For some datasets the edge update algorithm is very much favorable compared to the construction algorithm, while in other cases not so much. In the following we would like to gain a better understanding of this phenomenon.

We achieve this with two synthetic datasets, triggering both the extreme cases: one where the construction algorithm benefits the most and one where the update algorithm benefits the most. The first dataset, Dbest, shows a best-case scenario that the update algorithm can achieve relative to the construction algorithm. In this case we create a full k-ary tree, with edges pointing from parents to their children. When adding one edge to the tree, we add one edge to the leaf node, so that no node's signature would change after the insertion. In this case the update algorithm does the least amount of work, without propagating any change to further iterations during execution. Figure 4.7a shows an example of Dbest, which is a binary tree with height 3. The dashed edge is the newly added edge.

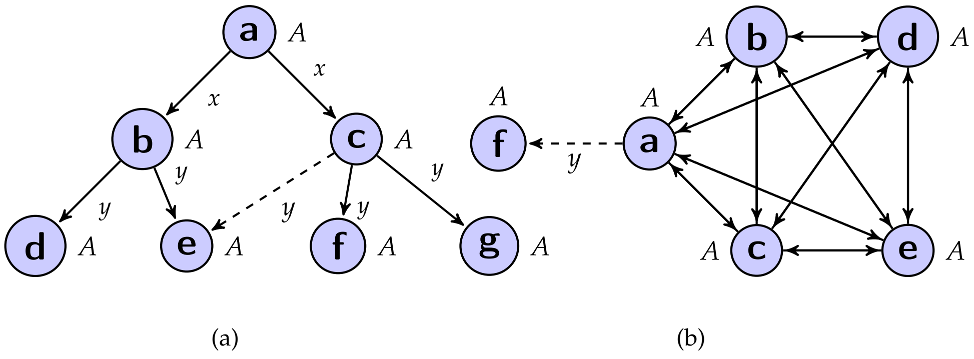


Figure 4.7.: Examples for Dbest (4.7a) and Dworst (4.7b) datasets

The second dataset, Dworst, exhibits a worst-case scenario for the update algorithm, relative to construction. In this case we create a complete graph, with edges all labeled with x. Then when adding one more edge (labeled y) to one of the nodes, every other node in each iteration is affected and therefore all the nodes' signatures are changed. The update algorithm has to check all nodes in every iteration. Figure 4.7b shows an example of Dworst, a complete graph with 5 nodes. The dashed edge is the newly added edge.

We generate Dbest and Dworst on the scale of 100 million edges, and measure the elapsed time and I/O costs (Figure 4.8) for both the construction (Build_Bisim()) and edge update (Add_Edges()) algorithms in each iteration. We see that indeed for Dbest, the update algorithm shows a 4 times speed-up in time compared with the

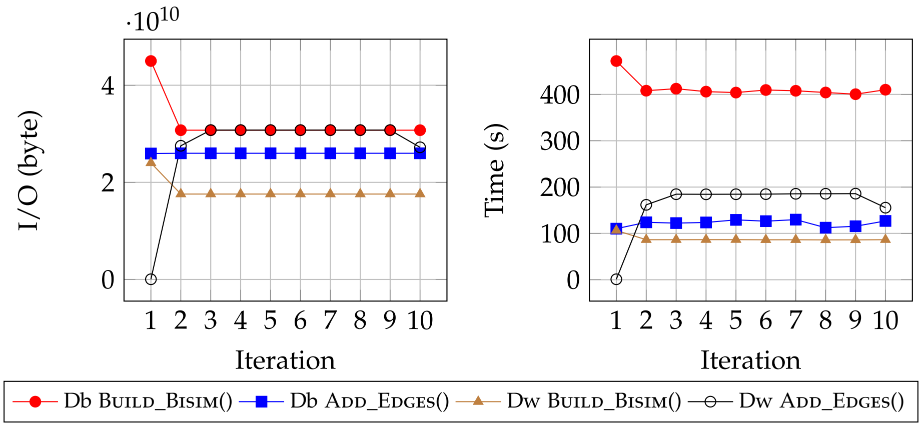


Figure 4.8.: Time and I/O comparison for Db(est) and Dw(orst) by applying BUILD_-BISIM() and ADD_EDGES() algorithms on both ($k = 10$)

construction algorithm. For Dworst, the update algorithm is 2 times slower in time than the construction algorithm.

4.4.4. Experiment on multiple edges update

To test the performance of multiple edges update, we randomly split a set of edges from the datasets (edge count 1, 10, 100, . . . , 1M), construct k-bisimulation partitioning on the rest of the graphs, and apply the algorithm ADD_EDGES() upon the set of edges, recording the I/O and elapsed time of the experiments. In Figure 4.9 we show the I/O improvement ratio and time speed up ratio (both construct/update) for all cases (taking the average). A gray line is drawn at $y = 1$ for both figures to split the space, to indicate whether ADD_EDGES() performs better than BUILD_BISIM() or not. From the figure we see that, for many of the datasets, it is beneficial to do batch update (ADD_EDGES()) up until 10^4 edges. An order of magnitude time speed up is observed for Jamendo, LinkedMDB and DBLP. In fact, if we consider the time cost for Jamendo and DBLP, it is always favorable to use ADD_EDGES() in all cases. For dataset DBPedia, however, changes propagate rapidly in the first few iterations, therefore the construction algorithm (BUILD_BISIM()) becomes a better choice when there are more than ten edges to be updated.

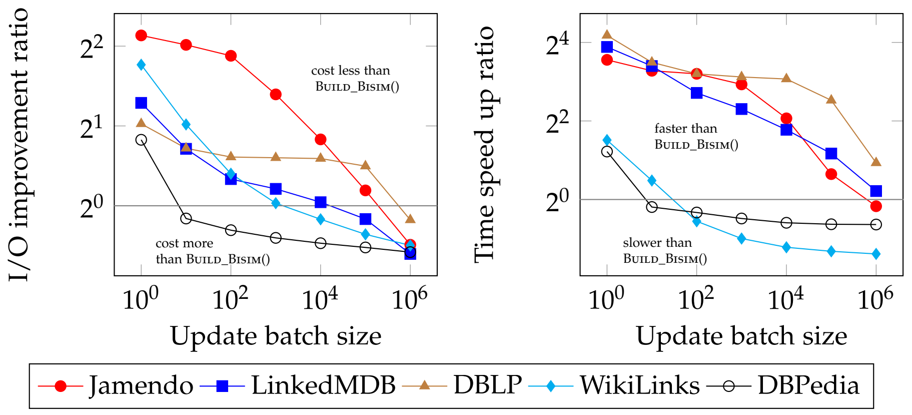


Figure 4.9.: I/O (left) and time (right) improvement ratio $\frac{cost(\textsc{Build\_Bisim}())}{cost(\textsc{Add\_Edges}())}$ for batch edge updates ($k = 10$)

4.5. Conclusion

In this chapter we conducted an extensive empirical study on the k-bisimulation partitioning algorithms for disk-resident graphs. In both Chapter 3 and Chapter 4, we showed that our algorithms are not only efficient and practical to use, but also scale well with the size of the data.

5. Regularities and dynamics in k-bisimulation results

5.1. Introduction

While a lot of effort has been focused on computing (k-)bisimulation, little work has been carried out to take a deep look into the bisimulation result itself, which is essential for applications (e.g., indexing, query optimization, compression, load balancing) to take into consideration. Indeed, it is well known that graph properties, or data properties in general, such as skewness (e.g., power-law distribution [CSN09]) can hugely influence the performance of data-intensive processing. This applies to both single-machine algorithms (e.g., caching effects [Dem02]) and distributed algorithms (e.g., [ALPH01,HL91]). Therefore characteristics of the input data must be examined and reflected at the stage of algorithm design.

Motivated by these observations, in this chapter we analyze the k-bisimulation partitioning results of many real and synthetic big graphs. We compare the graph properties of the *abstracted bisimulation graph* (defined as k-BPR graph in Def. 5.1) both with each other and with the original underlying graph. We also analyze a dynamic social network graph (Flickr-Grow), and examines the behavior of the k-BPR graph as the original graph grows.

We make the followings observations:

- Regularities exist in the bisimulation results of real-world graphs. Power-law distributions hold for partition block size distribution, signature length distribution, degree distributions for the k-BPR graph. The k-BPR graphs are usually denser than their original graphs.

- In the context of bisimulation results, the synthetic graph generators that we examined fail to fulfill one or more of the regularities that are observed in real-world graphs.

- For the dynamic social network that we examined, its k-BPR graph also grows, but the growth is stable (related by a constant factor) with respect to the original graph.

To the best of our knowledge, we are the first to make these observations.

5.1.1. Definition and experiment setup

We define the *k-bisimulation partition relation graph* from the $\approx^k$ relation (Definition 3.1).

Definition 5.1. *Let $G = \langle N, E, \lambda_N, \lambda_E \rangle$be a graph and $k \geq 0$. The k-*bisimulation partition relation graph *for G (denoted as k-*BPR *graph [PLF+12b]) is the directed graph $G_k = \langle N_k, E_k \rangle$, such that*

- *N_k consists of the equivalence classes of $\approx^k$, i.e., if for node $v \in N$, we let $[v]_{\approx^k} = \{u \in N \mid v \approx^k u\}$, then $N_k = \{[v]_{\approx^k} \mid v \in N\}$.*
- *$E_k \subseteq N_k \times N_k$, and $(X, Y) \in E_k$ iff $\exists x \in X, y \in Y$ s.t. $(x, y) \in E$.*

For example, consider the social network graph in Figure 1.1 (p. 1) and its k-bisimulation results in Table 3.1 (p. 28). For $k = 1$, nodes in G are grouped into partitions C, D, E, F, which form N_k in G_k. Edges are merged together as well, and the k-BPR graph is drawn in Figure 5.1.

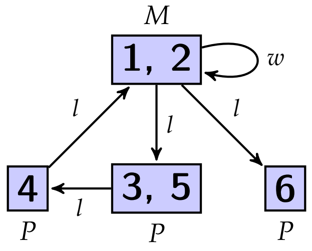
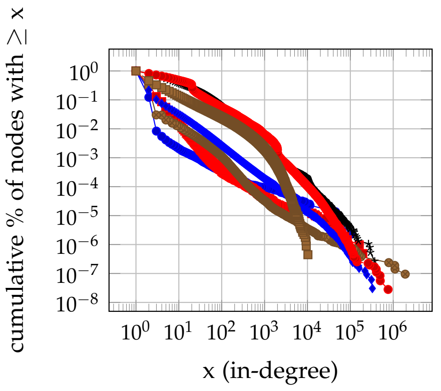


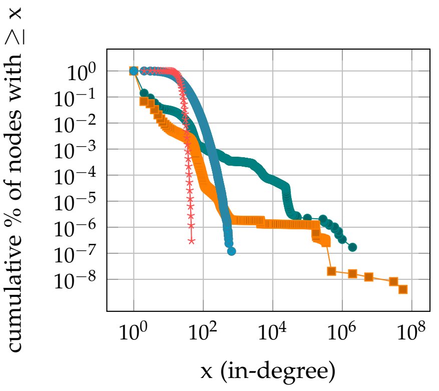
Figure 5.1.: k-BPR graph of Figure 1.1 $(k = 1)$

Since both G and G_k are directed graphs, we define for each node in G and G_k the in-degree (out-degree) as the number of incoming (outgoing) edges of that node.

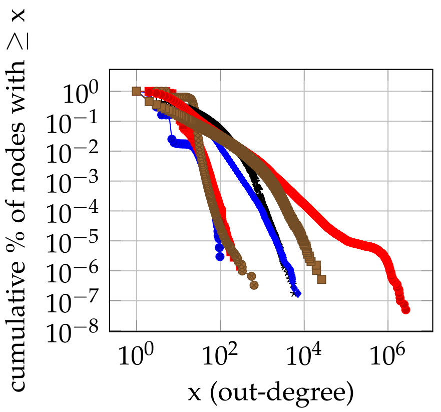
Experiment setup for this investigation is the same as in Section 4.2. Figure 5.2 presents the in-degree and out-degree distributions for the real graphs and synthetic graphs respectively. We see that all the real graphs and some synthetic graphs (i.e. BSBM, SP2B, Power) show a certain power-law distribution. For Flickr-Grow we plot the grown graph.



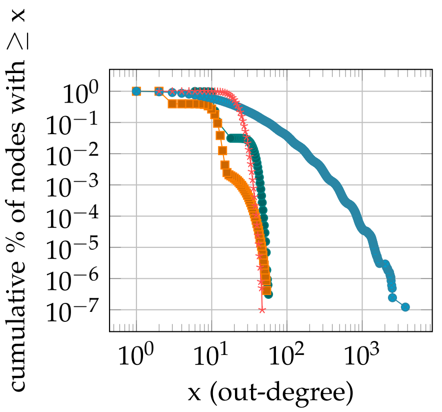
(a) in-degree distribution for real graphs



(b) in-degree distribution for synthetic graphs



(c) out-degree distribution for real graphs



(d) in-degree distribution for synthetic graphs

Figure 5.2.: In-degree and out-degree distributions for graphs

5.2. Static properties of k-BPR graphs

In this section we examine the properties of the static graphs (we treat the grown Flickr-Grow as a static graph in this section). Specifically, we are interested in the comparison of basic structural properties of the k-BPR graph G_k and its original graph G.

5.2.1. Comparison of G_k and G

In Figure 5.3a and 5.3b we show $\frac{|N_k|}{|N|}$ and $\frac{|E_k|}{|E|}$ for $k \in \{1, \ldots, 10\}$ for all graphs, where $|X|$ denotes the size of set X. The figures indicate the reduction (compression) rate we can get. In general, we see that localized bisimulation reduction provides good compression on the original graphs, with a reduction rate between 10^{-4} and 10^{-1}, and the rate becomes stable around $k = 5$. We also see that, compared with the real graphs, the partition results from synthetic datasets {BSBM, Power, Random} are either too coarse or too refined. However, this also happens for the real graphs without labels (i.e., WikiLinks, Twitter, Flickr-Grow).

In Figure 5.3c, we plot the average degree of the partition graph for each dataset for $k \in \{1, \ldots, 10\}$. Comparing with the original graph degree in Table 4.1, we see that the partition block graphs usually have higher degrees and at the beginning of the computation, the average degree tends to drop. In the case of graphs without labels, the degrees first rise until k is 4 or 5 and then drop.

Overall, for the purpose of compression or structural indexing, we observe that choosing $k = 5$ is usually sufficient. A larger k value would lead to a too refined partitioning. k-BPR graphs are usually denser than their original graphs.

5.2.2. Power-law distribution in G_k

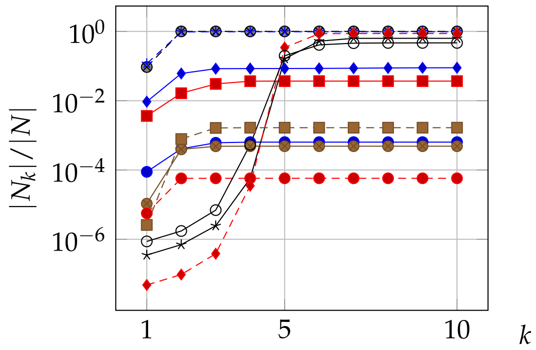
In Figure 5.2 we see that many of the original graphs follow a power-law distribution in their structure. We are curious about whether this is also true for their k-BPR graphs.

We first study some graph properties of the k-BPR graphs. In Figure 5.4 we plot the in-degree and out-degree of the k-BPR graphs for real graphs and synthetic graphs, respectively.

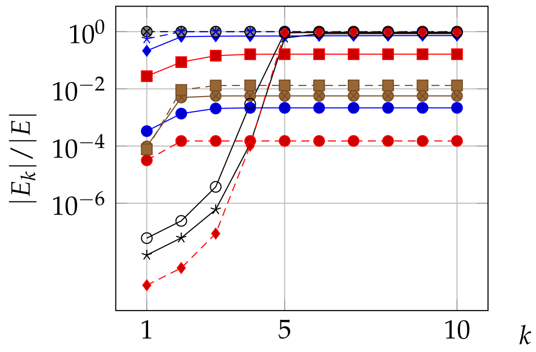
Figure 5.5a and 5.5b show the distribution of the partition block size for each graph. Note here that for the Random dataset, each node belongs to its own partition.

In Chapter 3 we defined a notion of *signature* for each node, which is essentially an encoding of the bisimulation equivalence class of the node. The length of a node's signature gives us insight into the complexity of the local topology of the node. Figure 5.5c and 5.5d show the distribution of signature lengths.

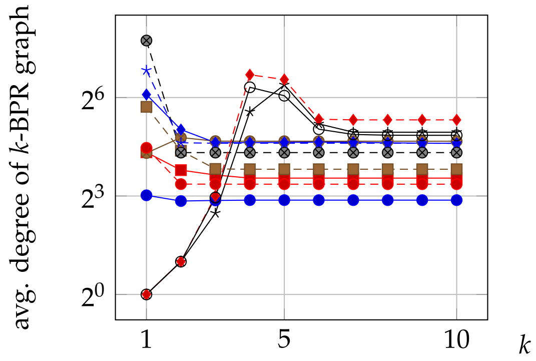
In general, we observe that all examined properties show certain power-law distribution nature for real graphs. This gives us some insights when we want to build applications of k-BPR graphs. Furthermore, we note that not a single synthetic dataset fulfils all power-law distribution graph properties as shown in real data.



(a) Node reduction ratio of G_k to G



(b) Edge reduction ratio of G_k to G



(c) Average degree of k-BPR graphs

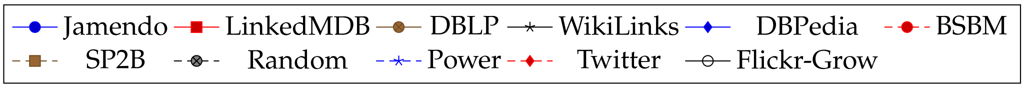
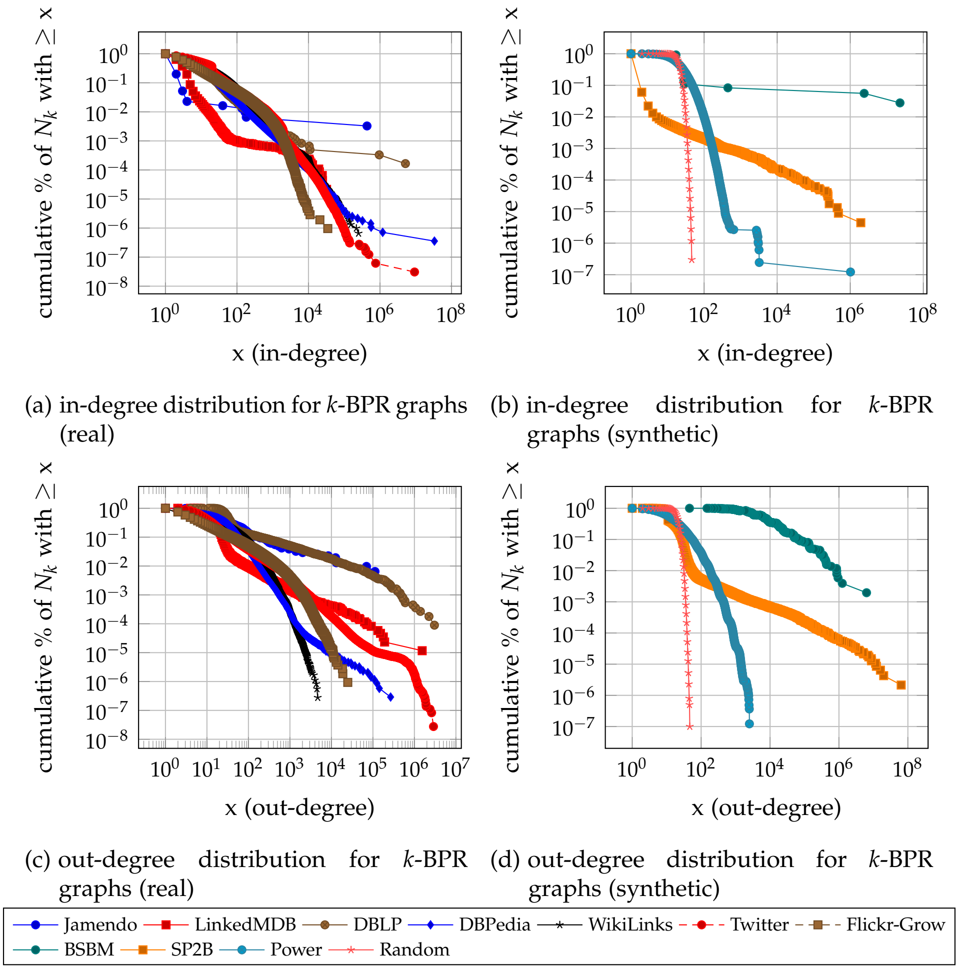


Figure 5.3.: Comparison of k-BPR graph to its original graph



(a) in-degree distribution for k-BPR graphs (real)

(b) in-degree distribution for k-BPR graphs (synthetic)

(c) out-degree distribution for k-BPR graphs (real)

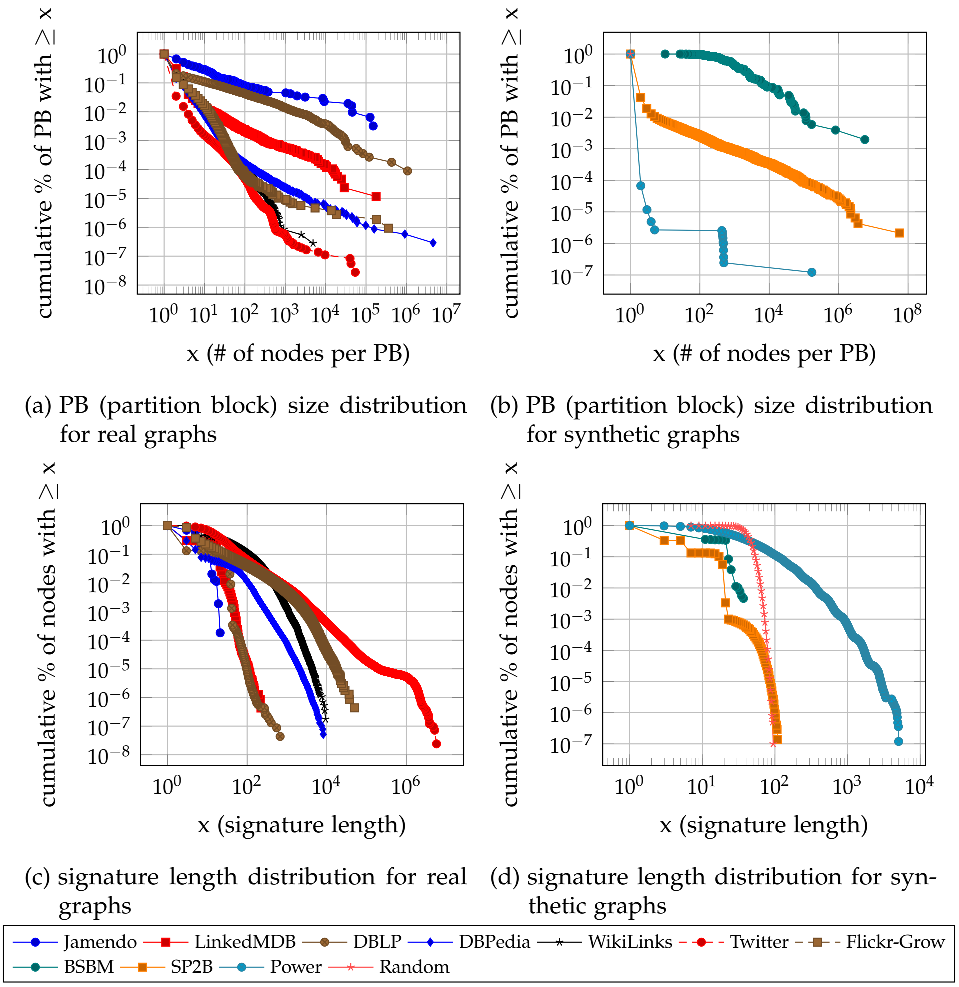
(d) out-degree distribution for k-BPR graphs (synthetic)

Figure 5.4.: In-degree and out-degree distributions for k-BPR graphs

From the bisimulation partition perspective, the most *real* synthetic graph is SP2B, which still lacks of the power-law distribution on signature length. This indicates that benchmark graph generators still need to be improved in this direction to reflect the structure of real graphs.

5.3. Dynamic properties of k-BPR graphs

While Section 5.2 studies the properties for static graphs and their k-BPR graphs, in this section we want to look into growing graphs. Note that for our growing graph



(a) PB (partition block) size distribution for real graphs

(b) PB (partition block) size distribution for synthetic graphs

(c) signature length distribution for real graphs

(d) signature length distribution for synthetic graphs

Figure 5.5.: Partition block size and signature distributions in *k*-bisimulation results

(Flickr-Grow), the findings in Section 5.2 still hold.

It is easy to design synthetic graphs such that their corresponding *k*-BPR graph either shrinks or grows, as the original graph grows. For real-world social graphs however, we are interested to know: (**P1**) is the *k*-BPR graph growing when the original graph grows? and (**P2**) is the *k*-BPR graph growing faster than the original graph? We use the Flickr-Grow graph for this investigation. The original Flickr-Grow graph includes a time stamp for each edge. We separate the edge set into 14 subsets based on the time stamp, grouping edges together for every 10 days. In this way we can examine graph growth in a coarse granularity.

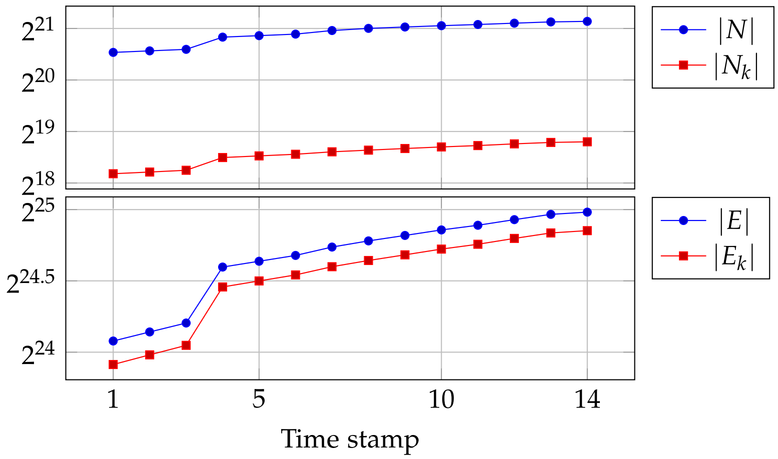


Figure 5.6.: k-BPR graph growth trend in $|N|$, $|E|$, $|N_k|$, and $|E_k|$

To answer P1, we plot in Figure 5.6 the trend of $|N|$ and $|E|$ of G, $|N_k|$ and $|E_k|$ of G_k with time, where $k = 5$. Other k values show the same behavior as well. Essentially, we examine the k-BPR graph growth in terms of nodes and edges. We see that during the whole period, $|N_k|$ increased by 1.5× and $|E_k|$ by 2×, while the original graph grows with the same ratios.

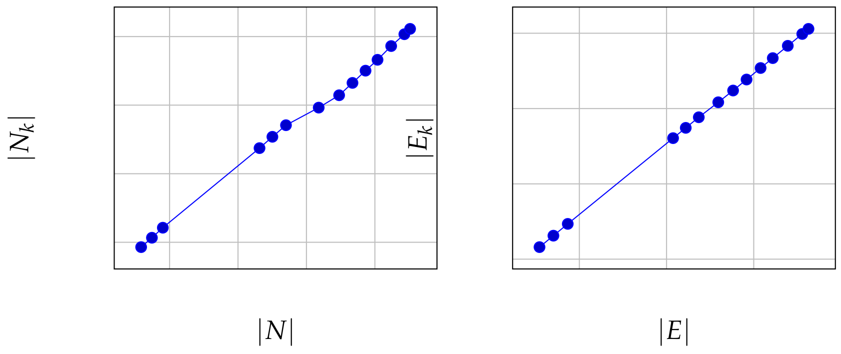


Figure 5.7.: k-BPR graph growth trend in $|N_k|$ w.r.t. $|N|$ and $|E_k|$ to $|E|$, all axes are in linear scale

To answer P2, we plot Figure 5.7, showing the growth of $|N_k|$ (y-axis) w.r.t. $|N|$ (x-axis) and $|E_k|$ to $|E|$. We see that there is clearly a constant factor between $|N_k|$ and $|N|$ ($|E_k|$ and $|E|$). So we conclude that (1) the k-BPR graph grows with the original graph, but (2) the growth is stable with respect to the original graph.

5.4. Conclusion

In this chapter we have examined many aspects of the k-bisimulation partitioning results for massive real-world and synthetic graphs. Extensive experiments have shown basic regularities in the k-BPR graphs for both static and dynamic real graphs, while the synthetic graphs fail to mimic real graphs in this respect. This indicates that synthetic graph generators aiming to generate “real graphs” should take the k-bisimulation properties into account during development. To our knowledge, we are the first to make these observations.

6. Further discussion of k-(bi)simulation

From Chapter 3 to 5, we study many aspects of *k*-bisimulation, from properties to efficient algorithms. In this chapter, we would like to discuss two related notions: full bisimulation, which considers *global* structural information, and *k*-simulation, which is a more *relaxed* partitioning method (i.e., leads to bigger partitions). Not only because these two concepts are also fundamentally important to study, but also by studying the connections between these problems, we will gain a better understanding of all problems, and may develop better solutions for all. We first introduce the notions of full bisimulation and *k*-simulation, and examine the relations of them with *k*-bisimulation. Then we propose an algorithm for computing *k*-simulation, within which we identify a core operation, that we further study in the second part of the thesis.

6.1. Connection between k-bisimulation and full bisimulation

First we give the definition of full bisimulation.

Definition 6.1 (full bisimulation). *Let $k \geq 0$ and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. Nodes $u, v \in N$ are called* bisimilar *(denoted as $u \approx v$), iff the following holds:*

1. $\lambda_N(u) = \lambda_N(v)$,
2. $\forall u' \in N[(u,u') \in E \Rightarrow \exists v' \in N[(v,v') \in E,\ u' \approx v'$ *and* $\lambda_E(u,u') = \lambda_E(v,v')]]$, *and*
3. $\forall v' \in N[(v,v') \in E \Rightarrow \exists u' \in N[(u,u') \in E,\ v' \approx u'$ *and* $\lambda_E(v,v') = \lambda_E(u,u')]]$.

As we mentioned earlier, there is an upper bound for the number of computation rounds for *k*-bisimulation. And this upper bound is no larger than the number of nodes in the graph.

Proposition 6.1. *If $\approx^j = \approx^{j+1}$, then $\approx^j = \approx^{j'}$ $(\forall j' \geq j)$.*

Proof. Since $\approx^j = \approx^{j+1}$, $\forall u \in N$, we could assign $pId_j(u) = pId_{j+1}(u)$. Then, according to Definition 3.3 (signature) and Proposition 3.1, it holds that $pId_{j+2}(u) = pId_{j+1}(u)$, and the same applies for any further $j' \geq j$. □

Proposition 6.2. *The j in Proposition 6.1 always exists, and its upper bound is $|N|$ (number of nodes).*

Proof. From Proposition 3.2 we know that $\forall u, v \in N$, if $u \approx^{j+1} v$, then $u \approx^j v$, which is equivalent of saying partitions will either split or stay the same. If they stay for one time, they will stay forever (Proposition 6.1). Otherwise, G has to at least split one of its partition blocks for each $\approx^i$ where $i \leq j$, in which case j reach the upper bound $|N|$. □

Then we observe the following useful connection between k-bisimulation and full bisimulation.

Proposition 6.3. *Let $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. There exists a $k \geq 0$ such that for any $u, v \in N$ it holds that $u \approx_k v$ iff $u \approx v$.*

Proof. First we want to show $u \approx_k v \Rightarrow u \approx v$. From Proposition 6.2 we know that k has an upper bound $|N|$. Here we set k to $|N|$, which means that $\approx_k = \approx_{k+1}$. Then according to the definition, in iteration $k+1$, for $u \approx_{k+1} v$, we have:

1. $\lambda_N(u) = \lambda_N(v)$,

2. $\forall u' \in N[(u, u') \in E \Rightarrow \exists v' \in N[(v, v') \in E,\ u' \approx^k v' \text{ and } \lambda_E(u, u') = \lambda_E(v, v')]]$, and

3. $\forall v' \in N[(v, v') \in E \Rightarrow \exists u' \in N[(u, u') \in E,\ v' \approx^k u' \text{ and } \lambda_E(v, v') = \lambda_E(u, u')]]$.

Since $\approx_k = \approx_{k+1}$, we can replace $\approx_k$ with $\approx_{k+1}$, then the relationship $\approx_{k+1}$ has the same definition as $\approx$. So that $\approx_{k+1} = \approx$.

Then we want to show that $u \approx v \Rightarrow u \approx_k v$. We will do it inductively.

1. $k = 0$. This is obvious.

2. $k > 0$. Assume that this holds for $j - 1$, we want to show that this also holds for j. Let $u \approx v$, we want to show that $u \approx_j v$. According to the definition, we want to have for all outgoing edges $(u, u') \in E$, there exists some edge $(v, v') \in E$,

such that $u' \approx^{j-1} v'$ and $\lambda_E(u,u') = \lambda_E(v,v')$, and vice versa. Because of $u \approx v$, we already have $u' \approx v'$; and because of $u \approx v \Rightarrow u \approx_{j-1} v$, we have $u' \approx_{j-1} v'$. Then all the requirements for $u \approx_j v$ are fulfilled. So $\approx \Rightarrow \approx_k$.

□

6.2. Connection between k-bisimulation and k-simulation

Similar to k-bisimulation, the notion of (k-)*simulation* also plays an important role in a wide range of applications (e.g., [PLF+12a, PFHV14, FGL+14]). In this section we briefly discuss some properties of k-simulation and the relation of it with k-bisimulation.

We first introduce the *k-simulate* relation between nodes.

Definition 6.2 (*k-simulate*). *Let k be a non-negative integer and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. For nodes $u, v \in N$, we say v k-*simulates *u (denoted as $u \preceq_k v$), iff the following holds:*

- *$\lambda_N(u) = \lambda_N(v)$, and*
- *if $k > 0$, then $\forall u' \in N[(u,u') \in E \Rightarrow \exists v' \in N[(v,v') \in E,\ u' \preceq_{k-1} v'$ and $\lambda_E(u,u') = \lambda_E(v,v')]]$.*

Using $\preceq_k$, we can build the equivalence relation *k-similar*.

Definition 6.3 (*k-similar*). *Let k be a non-negative integer and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. Nodes $u, v \in N$ are called k-*similar *(denoted as $u \sim^k v$) iff $u \preceq_k v$ and $v \preceq_k u$.*

We illustrate the concept of $\preceq_k$ and $\sim^k$ using the same example graph in Figure 1.1. It is easy to show that, for $k = 0, 1$, $\sim^k = \approx^k$, so we start with the situation when $k = 2$. For $k = 2$, to construct $\sim^2$, we need to first construct $\preceq_2$. We show $\preceq_2$ using a matrix in Figure 6.1. Then by checking with the matrix, we can easily get the $\sim^2$ relation: among six nodes, only *node 3* $\sim^2$ *node 5*.

Both $\preceq_k$ and $\sim^k$ relations can be built in a bottom-up manner.

Proposition 6.4. *Let k be a non-negative integer and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. For all $u, v \in N$ it holds that $u \preceq_k v \Rightarrow u \preceq_{k-1} v$.*

	1	2	3	4	5	6
1						
2	$\preceq_2$					
3					$\preceq_2$	
4						
5			$\preceq_2$			
6			$\preceq_2$	$\preceq_2$	$\preceq_2$	

Figure 6.1.: 2-simulate matrix example for Figure 1.1, omit self-simulate

Proof. By induction on k.

(1) $k = 1$. This is obvious, since $\lambda_N(u) = \lambda_N(v)$.

(2) $k > 1$. Assume that this holds for j ($\preceq_j \Rightarrow \preceq_{j-1}, 0 < j < k$), we also want to show that this holds for $j+1$. Let $u \preceq_{j+1} v$. According to Definition 6.2, for all outgoing edges $(u, u') \in E$, there exists some edge $(v, v') \in E$, such that $u' \preceq_j v'$ and $\lambda_E(u, u') = \lambda_E(v, v')$. Since $\preceq_j \Rightarrow \preceq_{j-1}$, we have $u' \preceq_{j-1} v'$, then we have $u \preceq_j v$. So $u \preceq_{j+1} v \Rightarrow u \preceq_j v$. □

Corollary 6.1. *Let k be a non-negative integer and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. For all $u, v \in N$ it holds that $u \sim^k v \Rightarrow u \sim^{k-1} v$.*

Definition 6.4. *Let k be a non-negative integer and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. The k-simulation partition relation graph for G (denoted as k-SPR graph [FHV+11]) is the directed graph $G_k = \langle N_k, E_k \rangle$, such that*

- *N_k consists of the equivalence classes of $\sim^k$, i.e., if for node $v \in N$, we let $[v]_{\sim^k} = \{u \in N \mid v \sim^k u\}$, then $N_k = \{[v]_{\sim^k} \mid v \in N\}$.*
- *$E_k \subseteq N_k \times N_k$, and $(X, Y) \in E_k$ iff $x \preceq_k y$ holds for all $x \in X$ and $y \in Y$.*

Similar to $\approx^k$, here we also define *k-partition identifier* for $\sim^k$, i.e., $pId_i(u) = pId_i(v)$ iff $u \sim^i v$. $pId_k(u)$ could be used to denote the node set N_k in the k-SPR graph G_k. We write $pId_i(u) \preceq_i pId_i(v)$ iff $(pId_k(u), pId_k(v)) \in E_k$, and write $pId_i(u) \prec_i pId_i(v)$ iff $pId_i(u) \preceq_i pId_i(v)$ and $pId_i(u) \neq pId_i(v)$. Then k-SPR graph could be expressed by $pId_k(u)$ and the $\prec_k$ relations.

Consider again the example graph in Figure 1.1. Though its $\sim^1 = \approx^1$, i.e., nodes are partitioned to $\{1, 2\}, \{3, 5\}, \{4\}, \{6\}$, its 1-SPR graph (Figure 6.2) is totally different from its 1-BPR graph (Figure 5.1).

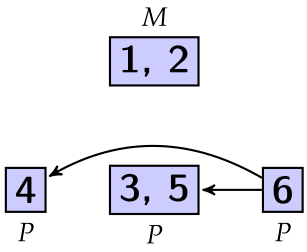


Figure 6.2.: k-SPR graph of Figure 1.1 ($k = 1$)

6.3. Signature-based k-simulation algorithm

It is easy to show that k-simulation is a more relaxed relation than k-bisimulation, i.e., $\approx^k \Rightarrow \sim^k$, but $\sim^k \not\Rightarrow \approx^k$. In graph of Figure 6.3 for example, $n_1 \sim^2 n_5$ but $n_1 \not\approx^2 n_5$. Computing simulation equivalence however is more difficult. Even the state-of-the-art in-memory algorithms have at least the time complexity of $O(|N|^3)$ (e.g., [Ran14]). Inspired by studies we have done to k-bisimulation, we are curious if it is possible to reuse the concept of signature for k-simulation.

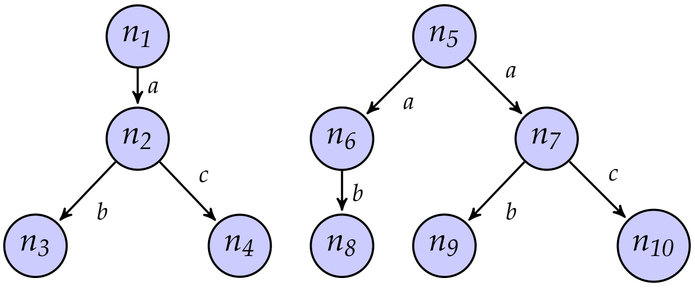


Figure 6.3.: Example graph where $n_1 \sim^2 n_5$ but $n_1 \not\approx^2 n_5$

We define the *k-simulation signature* as follows:

Definition 6.5. *Let k be a non-negative integer, $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph, and $\mathcal{P} = \{pId_0, \ldots, pId_k\}$ be a k-partition identifier for G. The k* simulation signature *of node $u \in N$ is the pair $sig_k(u) = (pId_0(u), L)$ where:*

$$L = \begin{cases} \emptyset & \text{if } k = 0, \\ \{(\lambda_E(u,u'), pId_{k-1}(u')) \mid (u,u') \in E\} & \text{if } k > 0. \end{cases}$$

Definition 6.5 is essentially the same as Definition 3.3 (p. 26). Only in this case, we need to do some processing to the signatures.

Definition 6.6 (*reduced signature*)**.** *Let $k > 0$, $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph, and for some node $u \in N$, $sig_k(u) = (pId_0(u), L)$ be its k-simulation signature. For any two pairs $(\ell_1, p_1), (\ell_2, p_2)$ in L, if $\ell_1 = \ell_2$, and $p_1 \preceq_{k-1} p_2$, we delete the pair (ℓ_1, p_1) from L. After we delete all such pairs, we call the signature a* reduced signature, *denoted as $rsig_k(u)$.*

Proposition 6.5. *Let k be a non-negative integer, $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. For any node $u \in N$, if there exists two nodes u', u'' such that $(u, u'), (u, u'') \in E$, $\lambda_E(u, u') = \lambda_E(u, u'')$ and $u'' \preceq_{k-1} u'$, then removing edge (u, u'') from E will not change the $\sim^k$ partition for G.*

Proof. At the very least, the removal of edge (u, u'') will potentially affect the partition result of node u. If u's partition block is not affected, then clearly $\sim^k$ will not change. Assume there is a node v such that $u \sim^k v$. From Definition 6.3 we have $u \preceq_k v$ and $v \preceq_k u$. It is obvious that $u \preceq_k v$ will not be affected, since v has fewer edges to simulate with. For $v \preceq_k u$, every edge (v, v') that has to be simulated by (u, u'') can be replaced with (u, u') (transitive relation). Then the proposition holds. □

Proposition 6.6. *$pId_k(u) = pId_k(v)$ iff $rsig_k(u) = rsig_k(v)$ $(k \geq 0)$.*

Proof. $\Rightarrow$.

Assume that $pId_k(u) = pId_k(v)$. According to Proposition 6.5, we remove all edges (u, u'') and $(v, v'') \in E$ if there exists u' and v', where $u'' \preceq_{k-1} u'$, $v'' \preceq_{k-1} v'$, and $(u, u'), (v, v') \in E$. After the edge removal, $pId_k(u) = pId_k(v)$ still holds. We refer to this reduced graph in later discussion. Then we have $u \preceq_k v$ and $v \preceq_k u$. Then the set $U = \{(\lambda_E(u, u'), pId_{k-1}(u')) \mid (u, u') \in E\}$ and $V = \{(\lambda_E(v, v'), pId_{k-1}(v')) \mid (v, v') \in E\}$ must be equal. In the following we prove that.

(1) $U \subseteq V$. According to Definition 6.3, we have for any pair $(\lambda_E(u, u'), pId_{k-1}(u')) \in U$, there exists a pair $(\lambda_E(v, v'), pId_{k-1}(v')) \in V$, such that $pId_{k-1}(u') \preceq_{k-1} pId_{k-1}(v')$; and for any pair in V, there exists a pair in U, such that any pair in V can be simulated by a pair in U. If there exists one $pId_{k-1}(u') \prec_{k-1} pId_{k-1}(v')$, since we refer to a reduced graph, there is no pair in U that can simulate the pair $(\lambda_E(v, v'), pId_{k-1}(v'))$, otherwise $(\lambda_E(u, u')$ will be removed. Then $pId_{k-1}(u') = pId_{k-1}(v')$. So $U \subseteq V$.

(2) $V \subseteq U$. Ditto.

$\Leftarrow$. It is obvious that in the reduced graph, $rsig_k(u) = rsig_k(v) \Rightarrow pId_k(u) = pId_k(v)$, then according to Proposition 6.5, in the original graph $pId_k(u) = pId_k(v)$ still holds. $\square$

Based on Proposition 6.6 we could directly design an algorithm to compute localized simulation partitioning for graphs. The sketch of the algorithm can be found in Algorithm 18. The idea is to create reduced signatures for all nodes and assign partition identifiers according to the signatures. In every iteration i, the i-SPR graph is created and used for maintaining the $\prec_{i-1}$ relation between nodes. Here we reuse the same signature storage facility S as described in Section 3.4 (p. 30).

Algorithm 18: k-simulation partition construction algorithm based on k-SPR

```
1 build 1-(bi)simulation partition as we did before
2 for iteration $i \in \{2, \ldots, k\}$ do
3     create $i$-SPR graph $G_{i-1}$ based on $\sim^{i-1}$
4     create signatures for all nodes
5     create reduced signatures based on graph $G_{i-1}$
6     use $S$ to assign new partition identifiers for iteration $i$
```

An example may illustrate the algorithm better. Considering the graph in Figure 6.3, Table 6.1 shows the intermediate result comparison of localized simulation ($rsig_2(nId)$) and bisimulation ($sig_2(nId)$) computation for the example graph. Note that the 1-similar and 1-bisimilar relations are the same. Here for sig_2 of n_5, because $p_5 \preceq_1 p_3$, we delete pair (a, p_5), therefore $rsig_2(n_5)$=$rsig_2(n_1)$, so $n_1 \sim^2 n_5$.

Table 6.1.: Intermediate results for example graph in Figure 6.3 ($k = 1, 2$)

nId	$sig_1(nId)$	$pId_1(nId)$	$sig_2(nId)$	$rsig_2(nId)$
n_1	$p_1, \{(a, p_1)\}$	p_2	$p_1, \{(a, p_3)\}$	$p_1, \{(a, p_3)\}$
n_2	$p_1, \{(b, p_1), (c, p_1)\}$	p_3	$p_1, \{(b, p_4), (c, p_4)\}$	$p_1, \{(b, p_4), (c, p_4)\}$
n_3	$p_1, \{\}$	p_4	$p_1, \{\}$	$p_1, \{\}$
n_4	$p_1, \{\}$	p_4	$p_1, \{\}$	$p_1, \{\}$
n_5	$p_1, \{(a, p_1)\}$	p_2	$p_1, \{(a, p_3), (a, p_5)\}$	$p_1, \{(a, p_3)\}$
n_6	$p_1, \{(b, p_1)\}$	p_5	$p_1, \{(b, p_4)\}$	$p_1, \{(b, p_4)\}$
n_7	$p_1, \{(b, p_1), (c, p_1)\}$	p_3	$p_1, \{(b, p_4), (c, p_4)\}$	$p_1, \{(b, p_4), (c, p_4)\}$
n_8	$p_1, \{\}$	p_4	$p_1, \{\}$	$p_1, \{\}$
n_9	$p_1, \{\}$	p_4	$p_1, \{\}$	$p_1, \{\}$
n_{10}	$p_1, \{\}$	p_4	$p_1, \{\}$	$p_1, \{\}$

Many optimization techniques (e.g., sort merge join, limited search space, partition ID encoding) can be applied to Algorithm 18, just as we did for k-bisimulation (Algorithm 12 in Chapter 3). The new part where we need to create the i-SPR graph (line 5 of Algorithm 18) is the most difficult part of the algorithm. To determine whether two partitions have the simulate relation, essentially we need to compute containment relation between sets. Since we need to perform the computation for all partitions (at most $|N|$ partitions), the number of set containment checks is massive. This leads to the problem of *set-containment join*, which we will study in more detail in Part II.

6.4. Conclusion

In this chapter we discussed two important concepts that relate to k-bisimulation: full bisimulation and k-simulation. We proposed an algorithm sketch for computing k-simulation partitions. Inside the computation we identified that a certain set computation is the core of the problem, which will be carefully studied in the second part of the thesis.

Part II.

Set-containment join

7. Algorithms for computing set-containment relations

7.1. Introduction

Sets are ubiquitous in data processing and analytics. A fundamental operation on massive collections of sets is computing containment relations. Indeed, bulk comparison of sets finds many practical applications in domains ranging from graph analytical tasks (e.g., [SZZ13, PLF+12a, ZÖC+14]) and query optimization [CB07] to OLAP (e.g., [LHYS14, BW14]) and data mining systems [Ran03]. In Chapter 6 for example, in the problem of computing k-simulation, the most complex operation is to check the containment relations between sets of partition IDs, where the number of sets is the number of nodes in graph. In order to design k-simulation algorithms for big graphs, it is therefore essential to first develop efficient algorithms that can work with such massive number of sets.

Table 7.1.: Example of set-containment join. If we perform a set-containment join ($\bowtie_{\supseteq}$) between user profiles and user preferences, we retrieve matching pairs $\{(u_1, p_1), (u_1, p_2), (u_2, p_3)\}$.

(a) user profiles

id	set	signature
u_1	{b, d, f, g}	0111
u_2	{a, c, h}	1011
u_3	{a, c, d}	1011

(b) user preferences

id	set	signature
p_1	{b, d}	0101
p_2	{b, f, g}	0110
p_3	{a, c, h}	1011

As a simple example, consider an online dating website where each user has an associated profile set listing their characteristics such as hobbies, interests, and so forth. User dating preferences are also indicated by a set of such characteristics. By executing a *set-containment join* of the set of user preferences with the set of user profiles, the dating website can determine all potential dating matches for users, pairing each preference set with all users whose profiles *contain* all desired characteristics. A concrete illustration can be found in Table 7.1.

In this part we study efficient and scalable solutions to the following formalization of this common problem. Consider two relations R and S, each having a set-valued attribute *set*. The set containment join of R and S ($R \bowtie_{\supseteq} S$) is defined as

$$R \bowtie_{\supseteq} S = \{(r, s) \mid r \in R \wedge s \in S \wedge r.set \supseteq s.set\}.$$

It is known that set-containment joins are expensive to compute [LV07,CCKN01]. Yet, due to its fundamental nature, the theory and engineering of set-containment join have been intensively studied (e.g., [HM97,HM03,HANM07,RPNK00,MGM01, MGM03,TBM02,Mam03,JP05,TPVS06,TBV$^+$11,LV07,CCKN01]). In this chapter we briefly survey many of the approaches. Existing solutions fall into two general categories: *signature-based* and *information-retrieval-based* (IR) methods, which we'll cover in Section 7.2 and Section 7.3. We further summarize techniques to make disk-based extensions for all the algorithms in Section 7.4. At the end of the chapter, we discuss some research topics that are related to set-containment join.

7.2. Signature-based methods

Signature-based algorithms (e.g., [HM97,HM03,HANM07,RPNK00,TBM02,MGM01, MGM03]) encode set information into fixed-length bit strings (called *signatures*), and perform a containment check on the signatures, as an initial filter followed by a validation of the resulting pairs using actual set comparisons. In this way, many set comparisons can be avoided, and the whole process gets accelerated.

We first introduce the definition of *signature* [HM97]. A signature of tuple t ($t.sig$) can be seen as the output of some hash function h (i.e., $t.sig = h(t.set)$) such that

$$t_1.set \subseteq t_2.set \Rightarrow h(t_1.set) \sqsubseteq h(t_2.set).$$

Here we define the containment relation $\sqsubseteq$ between two binary strings as $str_1 \sqsubseteq str_2 \Leftrightarrow str_1 \& \neg str_2 = 0$, where & and $\neg$ are bitwise AND and NOT operations. We will also refer to the $\sqsubseteq$ relation as "subset" containment when there is no possibility of confusion.

A straightforward implementation of a signature hash function is as follows: assume the signature length $|t.sig|$ is b bits, all initially set to zero. If integer x is in *t.set* we set the $(x \bmod b)$th higher-order bit of *t.sig* to 1. The resulting signature is essentially a *compressed* bitmap representation of *t.set*. In the signature column of Table 7.1 we show the 4-bit signature for each set in our example relations. Alphabets are mapped to integers starting from 1, in alphabetical order (i.e., 'a' is mapped to 1, 'b' to 2, and so forth). Note that tuples u_2 and u_3 have the same signature, but different set values. These are called false drops in literature. More advanced hash function implementations are discussed in papers such as [HM97, MGM03].

Signatures can be applied to nested-loop join algorithms. Algorithms loop over signatures of both the inner relation and the outer relation, and perform pairwise containment check on the signatures. Obviously, $|R| \times |S|$ comparisons are needed to finish the signature check. This approach is usually considered as a baseline solution in papers e.g., [HM97, HM03, RPNK00, MGM03].

Index structures can be used to reduce such comparisons. In Signature Tree [HM03, TBM02], for example, a tree-shaped filter index is used to guide comparisons. Nodes near the root hold signatures that are union of the children, so that filtering done at the parents will prevent signature comparisons at the children.

To further avoid pairwise signature comparison, in the spirit of hash join, a series of algorithms are proposed. In these algorithms, tuples are grouped together by some hash values (may or may not relate to signature value), and the signature comparisons only take place between certain groups instead of all. The Signature Hash Join (SHJ) algorithm is the first approach of such idea, followed by Lattice Set Join [MGM01] and Extendible Signature Hashing [HM03], which are disk-based extensions of SHJ. We describe SHJ in more detail in Section 9.1.1.

7.3. IR-based methods

IR-based methods (e.g., [Mam03, JP05, TPVS06, TBV$^+$11]) build inverted indexes upon sets storing tuple IDs in the inverted lists. A merge join between inverted lists will produce tuples that contain all such set elements. Typically auxiliary indexes are created to accelerate inverted index entry look-ups and joins. Standard optimization such as compression can be applied to inverted files to further boost the performances.

When inverted files are stored on disk, one drawback of IR-based methods is that they trigger random access on disk blocks. Many efforts are spent to tackle this problem. In Mamoulis [Mam03], the author proposes a block-based nested-loop join (BNL) to avoid such access pattern. Terrovitis et al. in [TPVS06, TBV$^+$11] study ways to combine inverted files with other data structures such as B-Tree to further reduce I/O cost.

In this category, to the best of our knowledge, PRETTI (PREfix Tree based seT joIn) [JP05] is the most recent and efficient in-memory set-containment join algorithm. It uses a prefix tree (trie) to organize set elements of one relation, and an inverted file for the other relation. By only traversing the prefix tree once while joining on the inverted lists, PRETTI can produce all join results. PRETTI will be further discussed in Section 9.1.2.

7.4. Disk-based extensions

For a single machine, when data is too big for main memory, disk-based extensions are needed for set-containment join algorithms to run. In this section we introduce several commonly used strategies for this purpose. These works are inspired by the study of (parallel) join algorithms in database research (e.g., [SKS10, Chapter 12 and 18]).

Nested-loop join on partitions is a widely used strategy. Given relations S and R, we partition S into $S_1, \ldots, S_m$ and R to $R_1, \ldots, R_n$, and process each pair (S_i, R_j) ($i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$) in memory. BNL [Mam03] and PRETTI [JP05] use this strategy to scale.

Another strategy follows the grace hash join style, where two relations are partitioned by the same hash function, so that not all pairs of partitions need to be examined to produce the join results. Only in our case, since it's not an equijoin, tuples of one relation need to be replicated to several partitions. In one of the first approaches PSJ [RPNK00], tuple $r \in R$ is partitioned by some random set element of r and hash function h, then tuple $s \in S$ is replicated to all partitions with hash value $\{h(e) | \forall e \in s\}$. APSJ and (A)DCJ from Melnik et al. [MGM03] develop this idea further by using boolean hash functions to lower the relation replication factor, and yield better results.

Last but not least, disk-based data structures also play an important role to build disk-based extensions. Linear Hashing [TBM02], Extendible Hashing [HM03], tree structures [HM03, TBM02, TPVS06], and sorted inverted files [TBV$^+$11] are examples of this kind.

7.5. Related problems and research

Problems involving discovering relations between sets exist in many topics in database research. Research problems such as relational division, set similarity join, and nested relations (sets) are such examples. In the following we give a brief summary of some of the efforts that have been carried out.

Relational division As an implementation for universal quantification, the division operator (÷) from relational algebra is used to describe *for-all* semantics. Formally, the operator is defined as:

$$R(A, B) \div S(D) = \{a|\{b|R(a, b)\} \supseteq \{d|S(d)\}\}$$

Continuing our example from 7.1, user profiles ÷ {b,d} returns all users that contain elements b and d, which is u_1. Extensive research has been done to develop efficient algorithms for implementing relational division operator (see [GC95, RSMW03] for detailed surveys), and integrate the operator into query languages [BC13, BW14] and real systems [LHYS14]. Relational division can be seen as a special case of set-containment join [RM06], with the time complexity of $O(n \log n)$, where n is the size of the bigger relation [LV07].

Set similarity join The Set Similarity Join (SSJ) returns pairs of sets that are *similar* by some measurement. It can be viewed as a generalization of set-containment join, where conditions on sets are similarity measures (Hamming distance, edit distance, etc.) instead of containment check. Due to its applications in data integration, bioinformatics and data mining, SSJ receives plenty of attention in recent years. Similar to set-containment join, SSJ algorithms also use techniques such as signatures for filtering [AGK06], partitioning methods [AGK06, LDWF11], trie structure [FWL12, QZWX13] and inverted index [SK04] for speeding up the process. A detailed performance comparison of the state-of-the-art SSJ algorithms is conducted in [WDG$^+$14], with more complete references in it. In Section 9.2.3 we'll see that, certain set-containment join algorithm infrastructure can be easily adapted to perform SSJ as well.

Nested relations, trees and nested sets Till now we only consider sets that are *flat* (unnested). It is worth mentioning works that have been done to compute relations on *nested* sets. Garani and Johnson [GJ00] surveyed varied join operations on nested relations, which are essentially nested sets with schema. Another line of research on this matter is tree pattern matching on XML, where both the tree pattern query and XML document are considered as labeled nested sets. A comprehensive survey of this research can be found in [HD13]. Last but not least, there is some recent work on set containment query processing for nested sets [IF13]. In this work the authors assume no obligatory labeling on sets (schema free). IR-related techniques are heavily used in many of these works.

7.6. Conclusion

In this chapter we briefly surveyed the state-of-the-art set-containment join algorithms. We categorized these algorithms into signature-based algorithms and IR-based algorithms. Disk-based extensions were discussed for both categories. Some research topics that also involve set comparisons were mentioned at the end.

8. Subset enumeration within limited sets

8.1. Introduction

Suppose we represent sets over a finite domain as fixed-length binary strings (i.e., bitmaps). Generating all subsets of a given set r is a fundamental and well-studied combinatorial problem (e.g., [NW78, Chapter 1]), i.e., to return $K = \{k | k \sqsubseteq r\}$[1]. Efficient algorithms and bitwise solutions are developed to make sure the procedure is as fast as possible.

In many cases, it is not necessary to loop over all subsets of r. Limitations can be introduced while generating subsets. The problem of *k-subsets*, for example, returns subsets that have exactly k elements. In this chapter we focus on the limitation that we can only return subsets that already exist in the system, i.e., given a binary string r, and a set of binary strings S, we want to generate all subsets of r that also belong to S, i.e., to return $K = \{k | k \sqsubseteq r \wedge k \in S\}$. This problem is closely related to the set containment query answering problem in database research [LHYS14, HM97] and association rule discovery in data mining [TL02, LAN06, Sav13], finding its applications in various data warehousing and OLAP systems.

One obvious solution of this problem is to reuse the subset generation algorithms to first generate all possible subsets of r and then filter out the ones that are not in S, but this approach could be computationally expensive (2^b checks where b is the number of 1s in r). In this chapter we study ways to avoid generating all such subsets, but still output the correct results. We first introduce the baseline solution that needs to generate all subsets. Then we present Algorithm JumpEnum, that considers the constraint set *while* generating subsets. Last, we propose two trie-based algorithms, TrieEnum and PTrieEnum, that can finish the task in linear time (w.r.t. the output size), which is in general more efficient than the baseline solution. We note that the trie-based algorithms are output-sensitive, i.e., in addition to the input,

[1]Here we reuse the notation $\sqsubseteq$ from Section 7.2 to represent containment relation between binary strings

their performance depends on the output size. We conduct a series of experiments to show the performance differences of these algorithms. In Chapter 9 we will integrate algorithms developed in this chapter to set-containment join solutions, therefore some discussions of the algorithms are presented there as well.

In the pseudo code, we often treat bit strings as integers in two's complement form. Hence we can apply standard "C" style operators on such binary strings, including bitwise operators (e.g., AND &, OR $|$, XOR ˆ, NOT $\sim$, left shift $<<$, right shift $>>$), arithmetic operators (e.g., $-$) and so on.

8.2. Generate-and-filter enumeration

In this subsection we introduce the baseline solution FilterEnum (Algorithm 19). The algorithm uses existing algorithms to generate all subsets of r, then for each subset, it checks with the constraint set S. It only outputs a subset if the subset exists in S. A hashmap can be used for S to accelerate the existence checking.

Algorithm 19: FilterEnum() generate and filter, the baseline algorithm

```
  Input: binary string $r$, candidates $S$
  Output: list of subsets of $r$, that belong to $S$
1 create list $m$
2 $l \leftarrow$ generate subsets of $r$, e.g., algorithm 20 or 21
3 for each $t \in l$ do
4     if $t \in S$ then
5         add $t$ to $m$
6 return $m$
```

For example, suppose we have $S = \{1011, 0110, 0101, 0001\}$ and $r = 0111$. According to Algorithm 19, we first generate all subsets of r, e.g., $0111, 0110, 0101, 0100, \ldots, 0000$. Then for each subset, we check whether it belongs to S and we output 0110,0101,0001 as the result.

There are quite a few algorithms available for generating subsets of a given string (e.g., [LvHS00]), all of which can be plugged into Algorithm 19. Here we present Algorithm 20 and 21. The former generates subsets in ascending order, while the latter in descending order, both using efficient bitwise operations.

Proposition 8.1. *Algorithm 19 has the time complexity of $O(2^b \times L)$, Algorithms 20 and 21 have the time complexity of $O(2^b)$, where b is the number of 1s in the input binary string r, and L is the length of r.*

Algorithm 20: Generate subsets of a given binary string, counting up, from [HM97]

```
  Input: binary string $r$
  Output: list of subsets of $r$
1 create list $l$
2 $t \leftarrow r\& - r$
3 add $t$ to $l$
4 while $t \neq 0$ do
5     $t \leftarrow r\&(t - r)$
6     add $t$ to $l$
7 return $l$
```

Algorithm 21: Generate subsets of a given binary string, counting down, adapted from [Cod]

```
  Input: binary string $r$
  Output: list of subsets of $r$
1 create list $l$
2 $t \leftarrow r$
3 while $t \neq 0$ do
4     add $t$ to $l$
5     $t \leftarrow (t - 1)\&r$
6 add $t$ to $l$
7 return $l$
```

Proof. Algorithm 20 and 21 are bounded by their output size ($O(2^b)$). Algorithm 19 is bounded by its subset generation routine times the cost of each containment check ($O(L)$). □

Algorithm 19 is sufficient when the candidate space $|l|$ (line 2 of Algorithm 19) is small, but will become slow when the strings are long, due to the exponential behavior of the generation algorithms. One quick fix is to solve the problem from the other side, meaning to go through each element in S, and check whether it is a subset of r. This will give us a solution with the complexity of $O(|S| \times L)$. In Section 8.3 we will combine both of the ideas above to design a new algorithm. After that we propose a totally different trie-based algorithm in Section 8.4.

8.3. Check-and-jump enumeration

Our new algorithm JumpEnum (Algorithm 22) is based on the idea to generate subsets of r *while* checking with the given set S. Assume we have the candidate set S sorted. While generating subsets in descending order (e.g., using Algorithm 21), we also scan the candidate set. Whenever we find a match, we output such element; if not, we then go to the next value in the candidate set, using that as a guidance to *jump* in the subset space. To use this method, we need an auxiliary function that can make the jump, which is described in Section 8.3.1.

Algorithm 22: JumpEnum() check-and-jump enumeration

```
   Input: binary string $r$, candidates $S$
   Output: list of subsets of $r$, that also belong to $S$
1  create list $m$
2  $t \leftarrow r$
3  while $t$ do
4      $index \leftarrow$ locate $t$ in $S$
       // If $t \in S$, $index$ holds the index of $t$, otherwise it holds the index of the
       most significant value that is smaller than $t$ in $S$
5      $u \leftarrow S[index]$
6      if $t = u$ then                                       // $t \in S$
7          add $t$ to $m$
8          $t \leftarrow (t-1)\&r$                           // simply get the next subset
9          continue
10     else
11         $t \leftarrow$ subsetSmallerThan($r, u$)          // e.g., Algorithm 23
12 if $t \in S$ then                                         // now $t$ is 0
13     add $t$ to $m$
14 return $m$
```

Table 8.1 shows a running example of Algorithm 22. The algorithm first checks whether S contains r and moves the cursor on S accordingly. Then, after the check of 1011, the subset generation of r jumps with the cursor on S. At the last row, the subset generation does not continue with the sequence $0101, 0100, 0011, \ldots$, but skips 0100 to 0010 and directly jumps to 0001 based on the observation on S.

Proposition 8.2. *Algorithm 22 has the time complexity of $O(|S| \times L)$, where S is the constraint set, and L is the length of the input string.*

Proof. In the worst case, the jump facility does not work with the input string.

Table 8.1.: Running example for Algorithm 22, where $S = \{1011, 0110, 0101, 0001\}$ and $r = 0111$

generate subset of r	check point on S
0111	$\underset{\uparrow}{1011}$, 0110, 0101, 0001
0110	1011, $\underset{\uparrow}{0110}$, 0101, 0001
0101	1011, 0110, $\underset{\uparrow}{0101}$, 0001
0001	1011, 0110, 0101, $\underset{\uparrow}{0001}$

Therefore all elements in S will be checked, where each comparison will cost $O(L)$. □

8.3.1. Get the biggest subset that is smaller than some value

Given bit strings *mask* and *value*, we want the function to return some subset of *mask* (called *sub*) such that (1) $sub \leq value$ and (2) *sub* is the biggest of all valid subsets. For example, given mask 10110 and value 10001, we want the function to return 10000, since (1) 10000 $\sqsubseteq$ 10110 and (2)10000 < 10001. Algorithm 23 implements such functionality.

Algorithm 23: SUBSETSMALLERTHAN()

```
Input: binary string mask, boundary value
Output: greatest subset of mask that is smaller than value
len ← length of mask
for i = 0, ..., len-1 do
    if (mask[i] = value[i]) or (mask[i] = 1 and value[i] = 0) then
        output value[i]
    else
        output mask[i:] from this point
        break
```

Proposition 8.3. *Algorithm 23 is correct, i.e., the output of Algorithm 23 is (1) less than or equal to value; (2) subset of mask; (3) biggest among all possible subsets.*

Proof. (1) The output's prefix is the same as *value*; then for the rest part of the output, the most significant bit is zero instead of one, so the output is less than or equal to *value*.

(2) The output's prefix is either $mask[i]$ or 0, a subset of $mask[i]$; the rest part of the output is from $mask[i]$, so overall output is a subset of $mask$.

(3) Assume there is some other subset c that satisfies (1) and (2), and $c >$ output. Then in order for c to happen, c needs to flip some bits from 0 to 1 in the prefix of the output. That can only happen when mask[i] = 1 and value[i] = 0. Then we get a c that is strictly bigger than $value$, contradicting with (1). □

Using the same reasoning, we know that the next subset right after the output of Algorithm 23 is the smallest subset that is greater than $value$.

Algorithm 23 is correct, but may be considered not efficient enough, since we have to call it many times in Algorithm 22. A better choice would be to carefully examine the loop in the procedure, to see if that can be replaced with bitwise operations. The answer is yes, and the procedure can be found in Algorithm 24.

Algorithm 24: SUBSETSMALLERTHANBITOP(), bitwise version

```
Input: binary string mask, boundary value
Output: greatest subset of mask that is smaller than value
temp ← (value ^ mask) & value
i ← number of leading zeros in temp
if i = 0 then
    return mask
prefix ← (1 << 31) >> (i − 1)
return (prefix & value) | ((~prefix) & mask)
```

Proposition 8.4. *Algorithm 24 has the same output as Algorithm 23.*

Proof. The core operation of Algorithm 23 is to find the splitting point, where we use $value$ for the prefix, and $mask$ for the rest. Line 1 to 5 are designed for this purpose: $value$ XOR $mask$ sets only the bits that are not the same to 1. Then an AND operation with $value$ eliminates the 1s that are caused by $mask$ = 1. The leading 1 position points out the splitting point. Therefore everything before that 1 we use $value$, afterwards we use $mask$. Two bitmasks and an OR operation are created to output the correct parts. □

8.4. Trie-based enumeration

Let's reconsider the problem: for string r and set S, return $K = \{k | k \sqsubseteq r \wedge k \in S\}$. The performance of Algorithm 19 relates to 2^b, where b is the number of 1s in r.

Algorithm 22 on the other hand is bounded by $|S|$. The ideal case is to create an algorithm, that is bounded by the output $|K|$. In this section we propose algorithms fulfilling this idea. Here we totally ignore the exponential subset enumeration, but build a trie for S and operate on it directly. Recall that a trie is a basic tree data structure for storing strings. One property of tries is that strings within a subtree share the same path (prefix) from the root to the subtree. Here we use a binary trie to store binary strings of the same length. Strings themselves are stored at the leaves of the trie. After we insert all strings into the trie, since strings have the same length, we get a trie with the height of string length. From the root, each level of trie nodes represents one position bit in binary strings. An example of a binary trie can be found in Figure 8.1.

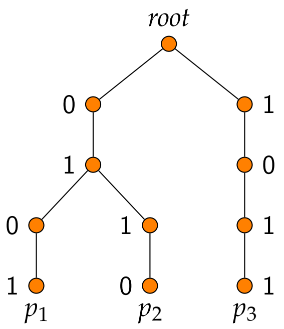


Figure 8.1.: Trie example, after inserting strings 0101:p_1, 0110:p_2, 1011:p_3 into an initially empty trie. Here we let left branches store strings with prefix bit 0 and right branches store strings with prefix bit 1.

When performing a breadth-first search on a trie, in the end we enumerate all existing strings by visiting the leaves. If we restrict our search at each level of the trie using some given string as guidance, we get the subset enumeration algorithm TrieEnum(), given in Algorithm 25. The basic idea is that, while traversing the trie level by level, we are examining all strings bit by bit. Then if we take the input string into consideration, the search space shrinks every time when a bit “0” is encountered. We use a queue to hold nodes whose prefixes are subsets of the input string. When Algorithm 25 finishes, all bits of the input string are examined, and all strings that are a subset of the input string are in the queue.

For example, if we want to find subsets of a string 0111 for all strings in Figure 8.1, we run Algorithm 25, then all nodes in the left branch of Figure 8.1 are visited and placed on the queue. In the end, 0101 and 0110 at leaf nodes are returned.

Algorithm 25: TrieEnum() subset enumeration using trie

```
Input: binary string $r$, candidates $S$
Output: list of subsets of $r$, that belong to $S$
1  create binary trie for $S$
2  create queue $q$
3  $i \leftarrow 0$
4  $current\_bit \leftarrow r[i{+}{+}]$
5  enqueue $trie.root$ on $q$
6  while $q.top$ has children do
7      $node \leftarrow$ dequeue from $q$
8      if $current\_bit = 0$ then
9          enqueue $node.left$ on $q$
10     else
11         enqueue $node.left$ and $node.right$ on $q$
12     $current\_bit \leftarrow r[i{+}{+}]$
13 return $q$
```

Similar ideas to construct trie structure for constrained set exist in literature [LAN06, Sav13]. In [LAN06] a trie is constructed for binary strings with delta-encoding (i.e., string that records bit shifting positions). In [Sav13], for a collection of sets, a trie is constructed in the set element space (i.e., on set values instead of binary space). Querying subsets on these data structures are also similar to that of Algorithm 25. A limitation of such approaches is that there are many unnecessary nodes that only have one child in the trie (which we later refer to as single-branch nodes). We also see this in Figure 8.1. For k strings (with b bits each), if there are no single-branch nodes, ideally the trie should have around $2k$ nodes. But instead, it will in the worst case need $k(b - lg_2k) + 2k$ nodes. The longer the string is, the more single-branch nodes it has. Moreover, these nodes all need to be enqueued and visited. In the empirical study, we witnessed that Algorithm 25 usually performs slower than Algorithm 19.

8.4.1. Introduce Patricia Trie

Knowing what is the weakness, we can improve the design accordingly. To avoid single-branch nodes, we adopt a data structure called Patricia trie [Mor68, Sed03], which is specifically designed for this purpose. Essentially, a Patricia trie merges single-branch nodes into one node in a trie, so it can guarantee that all nodes have full branches (in our case two-way branches). Of course in the worst case a Patricia trie is not better than a regular trie, but as we'll see in the experiments, that rarely happens for randomly-generated and real-world datasets. Figure 8.2 shows what a Patricia trie would look like if we insert the same strings as in Figure 8.1. First, because there is bit difference on position 0, one node is created on this position. Here, the right branch has no more splitting points, so it directly points to 1011. For the left branch, there is another splitting point on position 2, so another node is created accordingly, and each string belongs to one of the branches. Overall, 2 extra nodes are created and there is no single-branch node in the trie.

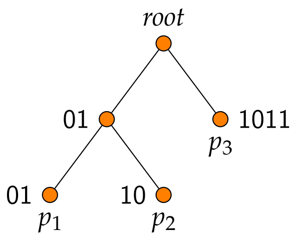


Figure 8.2.: Patricia trie example, inserting the same strings as in Figure 8.1 into a Patricia trie

In this chapter we apply a slight modification to the original Patricia trie. In our version of a Patricia trie node, we store (1) pointers to the left and right nodes, (2) the indexes at which point the prefix starts and splits, and (3) the common prefix from the last split point to the current split point.

We define a subset generation procedure on Patricia tries in Algorithm 26. It is similar to Algorithm 25 with the only difference being that, instead of comparing one bit at a time, segments of bits (which come from merged single-branch nodes) are compared at each node. In the end, strings that are subsets of s are stored in the *result* list instead of queue q.

To continue our example, if we run the same string 0111 on Figure 8.2 using Algorithm 26, we still need to visit the left branch of the trie. Only at this time, three instead of six nodes need to be traversed. In practice, bit strings (in our case signatures in Chapter 9) can be much longer and sparse, and therefore more node visits are saved compared to Algorithm 25.

Proposition 8.5. *If we have the trie built for S beforehand, the time complexity of Algorithm 25 and 26 are bounded by $O(|K| \times L)$, where $|K|$ is the output size, and L is input string length.*

Algorithm 26: PTrieEnum() subset enumeration using Patricia trie

```
Input: binary string $r$, candidates $S$
Output: list of subsets of $r$, that belong to $S$
1  create Patricia trie ptrie for $S$
2  create queue $q$
3  create list result
4  enqueue ptrie.root on $q$
5  while $q \neq \emptyset$ do
6      node ← dequeue from $q$
       // r.prefix moves forward when traverse deeper to the trie
7      if node.prefix $\sqsubseteq$ r.prefix then
8          if node.split = $|r|$ then
9              add node to result
10         else
11             split_bit ← r[node.split]
12             if split_bit = 0 then
13                 enqueue node.left on $q$
14             else
15                 enqueue node.left and node.right on $q$
16 return result
```

Proof. In the worst case, a Patricia trie will not change the trie structure. For generating each subset element of the output, the algorithm needs to traverse at most L nodes, therefore to output $|K|$ elements cost $|K| \times L$ node visiting. □

The above analysis gives a very rough upper bound for trie-based enumeration algorithms, which does not consider the shared traverse path between outputs for tries. Moreover, in practice a Patricia trie is much more compact than a regular trie, therefore practically it is more efficient than the worst case scenario.

8.5. Experimental study

In this section we empirically compare the performance of the four enumeration algorithms, namely FilterEnum (Algorithm 19), JumpEnum (Algorithm 22), TrieEnum (Algorithm 25) and PTrieEnum (Algorithm 26). We evaluate the effect of (1) the number of 1s in input bit string r (referred as b), (2) the size of the constraint set (referred as $|S|$), and (3) the size of the result set (referred as $|K|$) one the performance of the algorithms. We achieve this by fixing two parameters while varying on the third one. Data configurations of our investigation are in Table 8.2. All data are generated under uniform distribution. While changing the parameters of b and $|S|$, the possibility to get a certain result (to K) is very low, that is why $|K| = 0$ in both cases, and that is why we test $|K|$ for other values to complete the picture.

We implement all experiments in Java, and the source code can be found online[2].

Experiments are executed on a single machine (Intel Xeon 2.27 GHz processor, 12GB main memory, Fedora 14 64-bit Linux). Every experiment is loaded and run from cold cache for ten times. We take the average running time from them, and we observe that the standard deviation is not significant comparing with the average (less than 15% for JumpEnum and FilterEnum, less than 5% for TrieEnum and PTrieEnum). Note that we will use FilterEnum and PTrieEnum in applications in Chapter 9, so further comparison of these two algorithms is discussed there as well.

Table 8.2.: Dataset configurations

fixed parameters	changing parameter
$\|S\| = 2^{18}$, $\|K\| = 0$	$b \in \{10, 15, 20, 25, 30\}$
$b = 25$, $\|K\| = 0$	$\|S\| \in \{2^{15}, 2^{16}, 2^{17}, 2^{18}, 2^{19}, 2^{20}\}$
$\|S\| = 2^{18}$, $b = 25$	$\|K\| \in \{2^{14}, 2^{15}, 2^{16}, 2^{17}\}$

Impact of number of 1s in r (b) In Figure 8.3 we show the impact of the number of 1s in the input string on different algorithms. Algorithm names with suffix "B" indicate the algorithms' running time is taking into account the index building time (e.g., build hash map, sorting, trie structure, etc.). If we take a look at the running time without index building (left figure), we see that FilterEnum is indeed sensitive to the number of ones in the input, while other algorithms are not. JumpEnum and PTrieEnum have the best performance among all, since the result set size is zero. If we consider the index building time (right figure), we see that when number of 1s (b) is small, FilterEnum is actually a better choice over others, but that advantage disappears when b goes larger. While TrieEnum is the slowest of all, PTrieEnum, which is equipped with the Patricia trie structure, performs efficiently and stable among all algorithms, even when we consider the index building time.

Impact of constraint set size $|S|$ In Figure 8.4 we show the algorithms' behavior under different constraint result set sizes $|S|$. Again because the result set is empty,

[2] `https://github.com/lgylym/subset_enum`

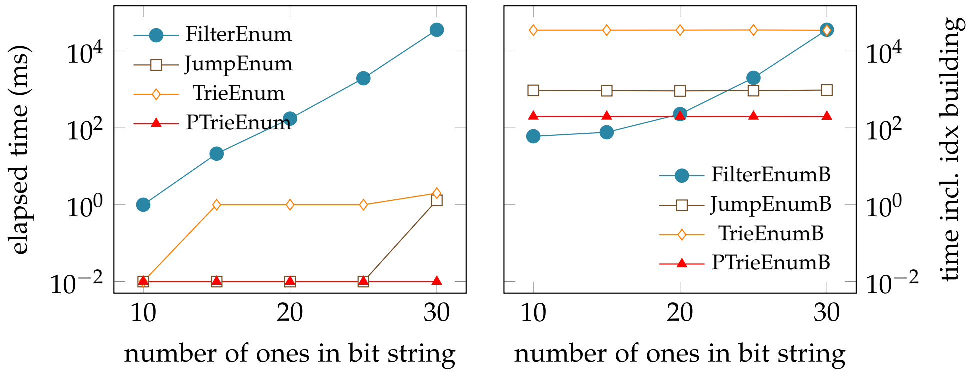


Figure 8.3.: Impact of number of ones in bit string r

JumpEnum and PTrieEnum perform very well when we do not consider index building time. FilterEnum however needs to enumerate subsets in all cases, therefore it is the slowest of all. But its performance is not sensitive to $|S|$. If we take into account index building time, PTrieEnum is still the most efficient of all, but the cost to build the index increases with the constraint set size. This suggests that PTrieEnum may benefit more when it can reuse the trie structure over time.

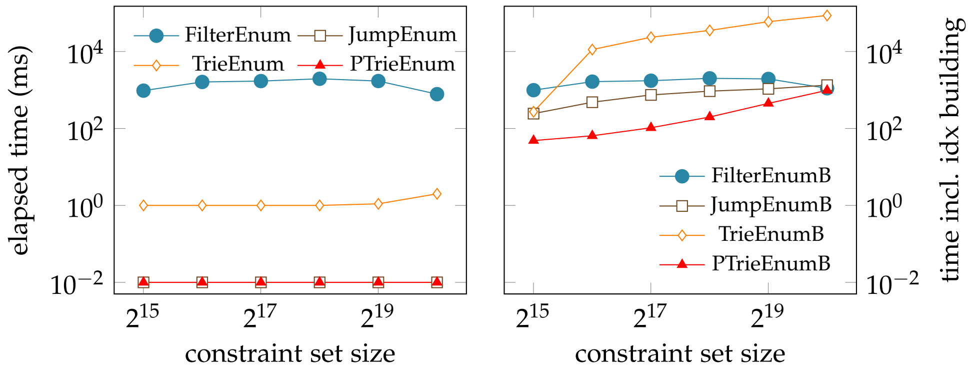


Figure 8.4.: Impact of constraint set size $|S|$

Effect of result set size $|K|$ It is interesting to see how algorithms react to different result set sizes. We show the result in Figure 8.5. By taking a look at the algorithms, we know that FilterEnum is not sensitive to this parameter, but the other three algorithms are. This is validated in the left figure of Figure 8.5. Index building time however is not affected by this parameter. For TrieEnum and PTrieEnum, due to caching effect of CPU, the index building time actually is slightly shorter when more subsets of r are inserted to the tries.

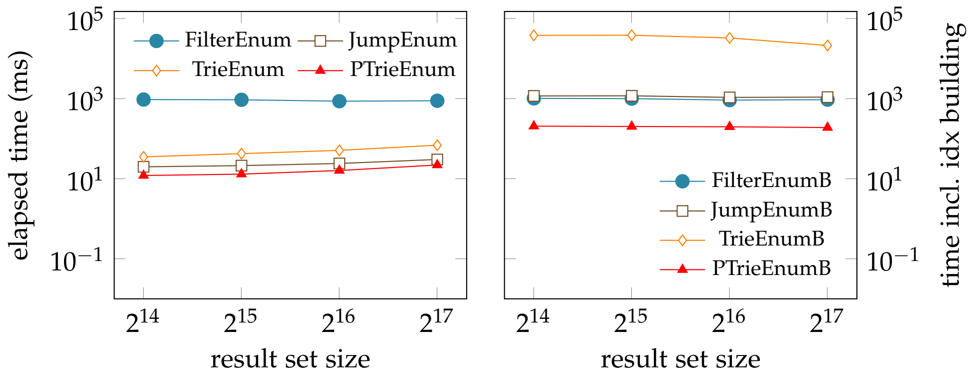


Figure 8.5.: Impact of result set size $|K|$

Summary In this experiment we evaluated four different subset enumeration algorithms in various settings. We found that FilterEnum is efficient only when the number of 1s in r is small. In other cases, whether considering the indexing building time or not, the two new algorithms, JumpEnum and PTrieEnum perform more efficiently, in many cases orders of magnitude better than the baseline solution. PTrieEnum may benefit more if its index structure is reused.

8.6. Conclusion

In this chapter we presented several algorithms for enumerating subsets of a given binary string within limited sets of strings. We first introduced the baseline solution FilterEnum, which first generates subsets of a given string, and then filters with the constraint set. We then proposed a novel algorithm JumpEnum, that can generate the subsets while checking with the constraint set. At last, we proposed trie-based algorithms (TrieEnum and PTrieEnum) for the enumeration, where a compact Patricia trie plays an important role to make things fast. An empirical study showed that JumpEnum and PTrieEnum are indeed much faster than the baseline solution in most of the cases, whether we consider the index building time or not. One application of the above-mentioned algorithms is to integrate them into set-containment join algorithms, which will be discussed in the next chapter.

9. Trie-based Set-containment Join

As we discussed in Chapter 7, most of the focus of the state-of-the-art set-containment join algorithms has been on disk-based algorithms. Though these algorithms have proven quite effective for joining massive set collections, the performance of these solutions is bounded by their underlying in-memory processing strategies, where less work has been done (see Section 9.1). To keep up with ever-increasing data volumes and modern hardware trends we need to push the performance of set-containment join to the next level. Therefore, it is essential to revisit (and develop new) in-memory set-containment join algorithms. Such algorithms will serve both as an essential component for main memory databases [LL13] as well as building blocks and inspiration for external memory and other computation models and platforms.

In this chapter we study in-memory algorithms for computing set-containment joins between massive collections (relations) of sets, where we scale the relations along three basic dimensions: set cardinality, domain cardinality, and relation size. Here, *set cardinality* is the size of *set* values in the relations; *domain cardinality* is the size of the underlying domain from which set elements are chosen; and *relation size* is the number of tuples in each relation.

In particular, our contributions are as follows:

- We propose two novel algorithms for set-containment join. One is for the low set cardinality, high domain cardinality setting (PRETTI+); the other is for the remaining scenarios (PTSJ). Both algorithms make use of the compact Patricia trie data structure.

- Our PTSJ proposal is a signature-based method. Hence, the length of the signature is a critical parameter for the algorithm's performance. Therefore, we perform a detailed analysis on PTSJ for determining the proper signature length. We also detail how PTSJ can (1) be easily extended to answer other set-oriented queries, such as set-similarity joins, and (2) efficiently be adapted to a disk-based environment.

- We present the results of an extensive empirical study of our solutions on a variety of massive real-world and synthetic datasets which demonstrate that our algorithms in many cases perform an order of magnitude faster than the previous state-of-the-art and scale well with relation size, set cardinality, and domain cardinality.
- We propose several ideas to make distributed extensions for PTSJ.

The rest of the chapter is organized as follows. In the next section, we introduce the state-of-the-art solutions for set-containment join. In Sections 9.2 and 9.3 we propose PTSJ and PRETTI+, our two new algorithms. Section 9.4 presents the results of our empirical study of all algorithms. We describe the ideas of making distributed extensions of PTSJ in Section 9.5 and then conclude this chapter in Section 9.6.

9.1. State-of-the-art Algorithms

In this section we describe two efficient in-memory set-containment join algorithms, SHJ and PRETTI. These solutions are representatives of the state-of-the-art, and serve as baseline solutions in our later development and experiments. For simplicity we assume in the following that domain values and tuple IDs are represented as integers.

9.1.1. Signature Hash Join

The Signature Hash Join (SHJ) was proposed by Helmer and Moerkotte [HM97]. As we mentioned in Section 7.2, SHJ uses the signature structure as a concise representation for sets, and uses signature comparisons as filtering operations before performing real set comparisons. In the spirit of hash join, SHJ works as follows: (1) for each tuple s in S, compute $s.sig$, and insert ($s.sig$, s) into a hash map (idx); (2) for each tuple r in R, compute $r.sig$, enumerate all subsets of $r.sig$, examine all tuples with such signatures in the hash map (hence in S), comparing them with r. Pseudo code of this approach can be found in Algorithm 27 and Algorithm 19. Here we split SHJ into two parts: a generalized signature join framework (Algorithm 27) that can be reused for other algorithms, and an enumeration algorithm used in SHJ (Algorithm 19) that can be replaced with more efficient algorithms as we discussed in Chapter 8.

SHJ inspired other algorithms (e.g., PSJ [RPNK00] and APSJ [MGM03]). It is one of the most efficient in-memory solutions for computing set-containment join.

Algorithm 27: SIGNATURE_JOIN() signature join framework

```
Input: relations $S$ and $R$
Output: pairs of tuple IDs that have the set containment relation
1 create index $idx$                                   // e.g., in SHJ is a hashmap
2 for each $s \in S$ do
3     insert $(s.sig, s)$ into $idx$
4 for each $r \in R$ do
5     $subset \leftarrow$ Call subset enumeration algorithm // e.g., Algorithm 19
6     for each $s \in subset$ do
7         if $r.set \supseteq s.set$ then
8             output $(s, r)$
```

One drawback of SHJ comes from line 2 of Algorithm 19, where all subsets of a given signature are enumerated and validated in the hash map. Though the authors provide a very efficient procedure (with bitwise operations) to perform this enumeration, such a mechanism cannot scale with respect to signature length, and therefore cannot scale with relation size and set cardinality. Consequently, all algorithms using this mechanism suffer also from the same problem. In Section 9.2, we provide a solution to this problem, with the introduction of an alternative data structure.

9.1.2. PRETTI Join

Recall from Section 7.3, PRETTI is an IR-based approach. In contrast with SHJ, PRETTI operates on the space of set elements instead of on the space of signatures. In particular, PRETTI works as follows: given relations S and R, first build a prefix tree (trie) based on the ordered set elements of tuples in S; then build an inverted file based on set elements of tuples in R. In the same root-to-leaf path of the trie, tuples of the descendants contain tuples of the ancestors. Then, when traversing the trie from root to leaf, at each node a list of containment tuples can be generated by joining the tuples in the node and in the inverted list. The list is passed down the trie for further refinement. A sketch of the PRETTI join can be found in Algorithm 28. The recursive call operates on each child of the root node and goes down the tree in a depth-first-search manner. Figure 9.1 illustrates the trie structure after inserting sets in user preferences from Table 7.1.

PRETTI is a very efficient algorithm. It only traverses the trie once to generate

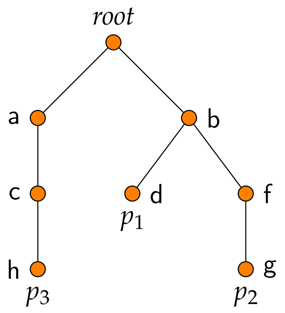


Figure 9.1.: Trie example for PRETTI, after inserting sets from user preferences (Table 7.1)

Algorithm 28: PRETTI_JOIN() recursively join and output

```
Input: subtree root node, current_list, inverted index idx
Output: pairs of tuple IDs that have signature containment relation
// Initially, current_list ← idx[node.label]
1 for each s in node.tuples do
2     for each r in current_list do
3         output (s,r)
4 for each child c of node do
5     child_list = current_list ∩ idx[c.label]
6     PRETTI_JOIN(c, child_list, idx)
```

all results. Set comparisons are naturally performed while traversing, and most interestingly, early containment results are reused for further comparisons.

PRETTI has two main weak points. First, many auxiliary data structures such as trie and inverted index are built for the algorithm, which can consume too much space if set cardinality is high. Second, varied-length set comparisons can be time consuming in comparison with fixed-length signature comparisons, especially when set cardinality is high. In our later empirical evaluation we will see that PRETTI can perform quite well for low set cardinality datasets. However, due to excessive main memory consumption and element comparisons, it cannot scale with either larger relations or higher set cardinalities. Later in this chapter, we develop extensions to PRETTI to overcome this main-memory consumption problem.

9.2. Patricia Trie-based Signature Join (PTSJ)

Let's reconsider SHJ from Section 9.1.1. After all signatures are computed, given one signature *r.sig*, SHJ needs two steps to get its subset results: (1) enumerate all subsets of *r.sig*; (2) check whether some subset exists in the hash map entry and perform set comparison afterwards. It is difficult for this mechanism to scale to longer signatures, because the number of possible subsets of a given signature is exponential (2^b) to the signature length b. Therefore in real cases, only part of the signature is used for enumeration purposes (and for creating hash map entries). Based on our experience, this partial signature length cannot even reach 20 bits. This mechanism essentially limits the possible performance gain of SHJ. However, it is not necessary to enumerate all possible subsets, but rather only those that actually exist in a relation. Hence, the space for enumeration is $O(|S|)$. This is the core idea of our first algorithm. We simply reuse the join framework of Algorithm 27, and replace the enumeration algorithm with Algorithm 25 and more advanced Algorithm 26. We call the latter approach **P**atricia **T**rie-based **S**ignature **J**oin (PTSJ).

9.2.1. Cost analysis of PTSJ

In this section we give some cost estimation of PTSJ under simple conditions. Some notation we use are given in Table 9.1. The cost of PTSJ (C_{PTSJ}) can be broken down to

$$C_{\text{PTSJ}} = C_{\text{create\_PT}} + C_{\text{query\_PT}} + C_{\text{compare\_set}},$$

where $C_{\text{create\_PT}}$ is the cost to build the Patricia trie on relation S, $C_{\text{query\_PT}}$ is the cost to compare signatures on the trie, and $C_{\text{compare\_set}}$ is the cost to actually perform set comparisons. We first identify that $C_{\text{create\_PT}}$ and $C_{\text{compare\_set}}$ are not the major costs of PTSJ. Then we dig deeper into $C_{\text{query\_PT}}$, giving an estimation of how many integer comparisons will it cost. We find that under simple natural assumptions, $C_{\text{query\_PT}}$ is mostly influenced by set cardinality c and signature length b. In the end, based on these analyses, we propose a strategy to choose a good signature length for PTSJ.

C_{create_PT} and $C_{compare\_set}$

$C_{\text{create\_PT}}$ During Patricia trie creation, only $2|S|$ nodes are created in total. Even in the worst case H nodes are visited during each signature insertion. Obviously, $C_{\text{create\_PT}}$ does not take the major part of PTSJ's running time.

Table 9.1.: Notation for cost analysis

Notation	Explanation
b	Signature length in bits
c	Average set cardinality
d	Domain cardinality, set element domain
Int	Integer size in bits
$\|X\|$	Size of relation X
H	Average height of Patricia trie
N	Number of tuples in S that have the signature containment relation with some tuple in R
V	Number of trie nodes one query has to visit

$C_{\text{compare\_set}}$ Assume that on average N tuples remain for set comparison for each tuple in R. Then $C_{\text{compare\_set}} = N \times c \times |R|$. It is easy to see that N decreases when signature length grows, and increases when $|R|$ increases. In general this is a small value (from 10's to 100's), proportional to the result output size (see below). Therefore $C_{\text{compare\_set}}$ is also not the major cost of PTSJ.

Estimation of N To estimate N, we start with a rather simple situation. Consider two signatures d and q, with set cardinalities (and hence number of bits set to 1 in signature) c_d and c_q, resp., and with signature length b. We want to know what is the probability that $d \sqsubseteq q$. For each element in a set, the probability that it appears on each bit is $\frac{1}{b}$. For $d \sqsubseteq q$ to happen, d should have 1's on only the positions that q has 1's. For each element in d, they have c_q positions to choose from, so each element has the probability $\frac{c_q}{b}$ to be a subset. In total, the probability is $(\frac{c_q}{b})^{c_d}$, and $N = |S| \times (\frac{c_q}{b})^{c_d}$.

We next consider a more complicated scenario. For example, if d's set cardinality is uniformly distributed between 1 and c_d, then the estimated probability of $d \sqsubseteq q$ would be $\frac{p^1+p^2+\ldots+p^{c_d}}{c_d} \approx \frac{p}{c_d \times (1-p)}$, where $p = \frac{c_q}{b}$.

In general, N gets smaller when signature length (b) grows. High set cardinality query (c_q) tends to have more results, while low set cardinality data (c_d) tend to produce more results. All these intuitions are confirmed by our formula. The main take-home message here is that N is a small value, so that set comparisons do not take the significant part of the overall running time.

C_{query_PT}

Let's assume that the number of trie nodes each tuple in R has to visit is V. Then the number of comparisons to be done on the trie is

$$C_{\text{query\_PT}} = |R| \times V \times \left\lceil \frac{b}{H \times \text{Int}} \right\rceil .$$

Here each node on average compares $\frac{b}{|H|}$ bits, which costs $\left\lceil \frac{b}{|H| \times |\text{Int}|} \right\rceil$ actual integer comparisons. We know that $\left\lceil \frac{x}{y} \right\rceil \leq \frac{x}{y} + 1$, so we get the upper bound

$$C_{\text{query\_PT}} \leq |R| \times V \times \left(\frac{b}{H \times \text{Int}} + 1 \right) . \tag{9.1}$$

For low cardinality settings, H can be as high as $\frac{b}{2}$, so it is rare for a single node to take more than two integer comparisons. For higher cardinalities, H is a smaller value closer to $log_2(2|S|)$, but still grows with respect to b. Then we can expect a small but slowly increasing value for comparisons per node. The more important factor however is V.

Estimation of V There are $\binom{b}{c}$ possible signatures with c bits set to 1. When set cardinality is small (i.e., when $\binom{b}{c} \leq |S|$), it is highly probable that all possible signatures exist in the trie. For example, in the extreme case that set cardinality is 1, there are only $2b$ possible nodes in the trie. Since $2b << 2|S|$, the trie is likely to be full. In such cases, V tends to reach the maximum possible, i.e., $2^c \times H$. Here H is approximately $\frac{b}{2}$.

This becomes less obvious when c and b grow to larger values. In such cases, the trie will not contain all possible cases, and the average height usually does not reach $\frac{b}{2}$. If we have an all-one signature as the query, all nodes ($2|S|$) will be visited. Therefore $2^H = |S|$. If on the lowest level, only one branch is included, the number of nodes to visit becomes $2^{H-1} + 1 \times 2^{H-1}$. Similarly, if single-branches happen for the lowest x levels (which yields the most number of nodes), we get $2^{H-x} + x \times 2^{H-x} = (1 + x) \times 2^{H-x}$. Furthermore, if we assume that the number of single-branch nodes in a result is proportional to the number of zeros in a signature $(1 - \frac{c}{b})$, so $x = (1 - \frac{c}{b}) \times H$, then the number of visited nodes is estimated to be

$$V = \left(1 + H \times \left(1 - \frac{c}{b}\right)\right) \times 2^{H \times \frac{c}{b}} \leq (1 + H) \times |S|^{\frac{c}{b}} \tag{9.2}$$

Here, we see that with the increase of $|S|$, the number of visited nodes increases. Bigger set cardinality also indicates more visited nodes, while longer signatures reduce the number of visited nodes. As we'll see later, we usually select b between $\frac{c}{2} \times Int$ and $c \times Int$, so $(|S|)^{\frac{c}{b}}$ is around 2 even for a million tuples. In such case we say the V is bounded by $O(H)$. And if we bring formula 9.2 into formula 9.1, we get $C_{\text{query\_PT}}$ is bounded by $O(c \times |R|)$.

Space complexity of PTSJ

Since to build a Patricia trie for some relation S, only $2|S|$ nodes are created, and for each tuple the signature size is usually no more than its set values, the space complexity of PTSJ is $O(|S|)$.

9.2.2. Choosing the signature length for PTSJ

Because there is no need for exhaustive subset generation, in practice, signature length can be set to thousands of bits in PTSJ without any problem. Generally, longer signatures provide more effective filtering, but bring more signature comparisons and higher main memory consumption. So there is a need for finding the balance point for signature length.

First of all, there is an absolute upper bound for signature length, which is domain cardinality d. Letting $b = d$ essentially makes the signature a bitmap representation of the sets. In many cases this number can be achieved. For example, for a domain that has 1024 elements, the maximum signature length is $\frac{1024}{Int}$ integers.

It is obvious that there is a lower bound for b as well, which is c. If $b < c$, there is a high chance that all bits in a signature are set to 1, which is not useful anymore.

Apart from these two bounds, we find that the "optimum" signature length depends on many properties of input relations, such as set cardinality, domain cardinality, relation size, and data distributions. Among these, we notice both from formula (9.2) and empirical study that the set cardinality has a bigger impact on signature length selection, and usually $\frac{c}{2} \times Int \leq b \leq c \times Int$ can yield a good result. This also prevents the algorithm from using more signature comparisons than set comparisons. If not specified otherwise, we use the lower bound of the range ($\frac{c}{2} \times Int$).

Finally, we can set a maximum length in the algorithm, to prevent it from being extremely long. In our experiments, this limit is set to 256 integers.

Overall, our signature length is set to

$$\text{minimum of } \left\{ d, \frac{c}{2} \times Int, 256 \times Int \right\}.$$

9.2.3. Extensions to PTSJ

Merge identical sets

With the help of the trie, tuples of the same signature are naturally grouped together. If we go one step further, maintaining a mapping list of tuples that have the same set elements, taking them into consideration while outputing, we save the cost of comparing duplicates over time. This strategy is applied in our PTSJ implementation. It works well without introducing noticeable overhead while creating the trie, and saves quite some comparisons while performing joins, especially for real-world datasets.

Superset and set-equality joins

While our algorithms are designed for $R \bowtie_{\supseteq} S$, it can be easily modified to perform $R \bowtie_{\subseteq} S$. Here we take Algorithm 25 as an example to illustrate; Algorithm 26 can be changed in a similar manner. The only place that needs to be touched is the if-else statements (lines 8 to 11). Two case handling statements should be switched, as given in Algorithm 29. Furthermore, in Algorithm 27 the set value containment check (line 7) will change accordingly, to "if *r.set* $\subseteq$ *s.set*."

Algorithm 29: Replace Algorithm 25 line 8 to 11 for superset join

```
if $current\_bit = 0$ then
    enqueue $node.left$ and $node.right$ on $q$
else
    enqueue $node.left$ on $q$
```

Set-equality joins ($R \bowtie_{=} S$) can be answered efficiently as well. In this case, a simple search on the trie will return a list of tuples with the same signature. Further set comparisons are needed to validate the search results. Since we already merge tuples with the same set values, as discussed above in Section 9.2.3, many set comparisons are saved.

Set similarity joins

Apart from being used for set containment computations, a Patricia trie can be (re)used to answer set similarity join [AGK06] queries as well. Set similarity join has been well-studied in the literature [WDG+14]. Solutions that make use of a trie have been proposed as well (e.g., [FWL12, QZWX13]), but these do not operate on (and cannot be easily adapted to) the signature space as PTSJ does. For instance, given query signature q, we want to find signatures within hamming distance k. We can use Algorithm 30 to achieve this goal, where we extend Algorithm 25 for illustration purposes.

Algorithm 30: TRIE_SSJ() hamming distance set similarity join using trie

```
Input: signature, trie, threshold k
Output: tuple IDs that have similar signature within hamming distance k
create queue q
i ← 0
current_bit ← signature[i++]
enqueue (trie.root, 0) on q
while q.top has children do
    (node, i) ← dequeue from q
    if i ≤ k then
        if current_bit = 0 then
            enqueue (node.left, i) on q
            enqueue (node.right, i+1) on q
        else
            enqueue (node.left, i+1) on q
            enqueue (node.right, i) on q
        current_bit ← signature[i++]
return q
```

We use a counter to remember the hamming distance between some prefix and our query. In the end, all signatures (therefore tuples) that are within the distance are in the queue, waiting for other operations (validation, output) to take action. Systems such as OLAP can benefit a lot from reusing one index for different purposes.

Disk-based algorithm

PTSJ can be easily extended to an external memory setting. A straightforward implementation is to perform a nested-loop join over partitions of the data (same as PRETTI), as we discussed in Section 7.4. Only in our case, PTSJ has a much smaller memory footprint than PRETTI, which makes it more suitable for this strategy. Smarter partitioning techniques (e.g., [RPNK00, MGM03]) can be integrated into PTSJ as well. Moreover, some advanced data structures such as a disk-based (Patricia) trie (e.g., [AZ09]) will further boost the performance.

9.2.4. Discussion

SHJ can be viewed as a one-level, multi-way trie, where each branch starts with a different prefix. PTSJ, on the other hand, is a multi-level, binary trie. The main benefits of PTSJ over SHJ come from longer signatures, which can filter out more unnecessary set comparisons. Furthermore, the trie structure guarantees that only interesting subset prefixes are visited, instead of the whole exponential space.

PRETTI, on the other hand, does make use of a trie structure, but it operates on the set element space instead of signature space. The benefit is that it does not need to be validated twice. The downside, however, is that trie height is as high as the set cardinality, making it only suitable for low set cardinality settings. This brings us to an advanced version of PRETTI, using a Patricia trie.

9.3. PRETTI+

Since the Patricia trie is so useful for PTSJ, it is natural to ask if this data structure can be used to enhance PRETTI. We have integrated a Patricia trie with PRETTI, calling this new join algorithm PRETTI+. Modifications have to be done both on trie construction and on the join procedures.

Inserting sets to the trie can be a bit trickier than with PTSJ, since sets are not necessarily of the same size. In Algorithm 31, we show the trie construction function for PRETTI+. Here we assume that each node maintains a prefix, a set of related tuples, and a set of children nodes. The main idea is that, depending on the common prefixes, the newly arrived tuple may be inserted to different positions with respect to the given node.

The join operation is almost the same as for PRETTI, except that lists of tuples from the inverted index have to be joined several times in each node, since each node holds several set elements. By replacing a standard trie with a Patricia trie, PRETTI+ consumes much less main memory than PRETTI. However, set comparisons and tuple list joins still take place, same as in PRETTI. As we'll see in our empirical study, PRETTI+ is always a better choice than PRETTI.

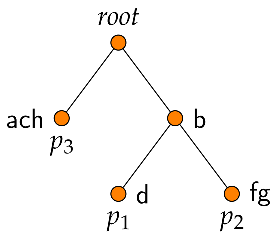


Figure 9.2.: Trie example for PRETTI+, after inserting sets from user preferences (Table 7.1)

Algorithm 31: PRETTI+_INSERT() trie construction for PRETTI+

```
Input: subtree root node, tuple s, cursor on s.set: from
Output: root for the subtree
// insert s.set[from:] to subtree node, here we treat s.set as a string
1  clen ← |common prefix of node.prefix and s.set[from:]|
2  nlen ← |node.prefix|
3  tlen ← |s.set[from:]|
4  if clen = nlen then
5      if clen < tlen then
6          c ← some child of node that matches s.set[(from+clen):]
7          call PRETTI+_INSERT(c, s, from + clen)
8      else                                                    // clen = tlen
9          put s into node
10     return node
11 else                                                        // clen < nlen
12     if clen = tlen then
13         create new_node for s, insert new_node between node and its parent
14     else
15         create new_node as parent for node and tuple
16     return new_node
```

9.4. Empirical Study

In this section we empirically compare the performance of SHJ, PRETTI, PTSJ, and PRETTI+. We first introduce the experiment settings. Then we validate the signature length selection strategy discussed above in Section 9.2.2. After that we conduct the main comparison of the four algorithms on a variety of synthetic and real-world datasets.

9.4.1. Experiment setting

Synthetic datasets

We create a data generator to generate synthetic relations. The generator can generate relations with varying sizes, set cardinalities, domain cardinalities and so on. The distribution of data can vary on both set cardinality and elements. The distributions are generated using Apache Commons Math[1], a robust mathematics and statistics package. We start with a simple setting, with uniform distribution on different set cardinalities and set elements. Later we test the algorithms' performance on relations with Zipf and Poisson distributions, which are commonly found in real-world scenarios.

Real-world datasets

We experiment with four representative real-world datasets, covering the scenarios of low, medium and high set cardinalities. Some statistics of the datasets[2] are shown in Table 9.2.

Table 9.2.: Statistics for real-world datasets

data	$\|R\|$	c avg.	c median	d
flickr	3.55×10^6	5.36	4	6.19×10^5
orkut	1.85×10^6	57.16	22	1.53×10^7
twitter	3.7×10^5	65.96	61	1318
webbase	1.69×10^5	462.64	334	1.51×10^7

Flickr-3.5M (flickr) The flickr dataset[3] associates photos with tags [LSW10]. Naturally, here we treat tags as sets, to perform a set-containment join on photo IDs. In this way, we create the containment relation between photos. Further operations such as recommendation can be investigated upon such relations. This is a low set-cardinality scenario.

Orkut community (orkut) The Orkut dataset[4] contains relations of people from an online social network and the communities they belong to [YL12]. Here we treat

[1]`http://commons.apache.org/proper/commons-math/`
[2]Can be downloaded at `http://goo.gl/EBaHbA`
[3]`http://staff.science.uva.nl/~xirong/index.php?n=DataSet.Flickr3m`
[4]`http://snap.stanford.edu/data/com-Orkut.html`

each person as a tuple and the communities they belong to as a set. Set-containment join in this case, can help people discover new communities and new friends with similar hobbies. Set cardinality for this dataset is higher than Flickr, and we further keep tuples with $c \geq 10$ to exhibit a low-to-medium set cardinality scenario.

Twitter k-bisimulation (twitter) We derive this dataset from paper [LFH$^+$13b]. Bisimulation is a method to partition the nodes in a graph, based on the neighborhood information of nodes. In this dataset, tuples are the partitions of the graph, and sets are the encoded neighborhood information each partition represents. Here we define the neighborhood of each node to be within 5 steps from the node. On such dataset, set-containment join could be used for graph similarity detection and graph query answering. For this dataset, we select tuples with $c \geq 30$, to exhibit a medium set-cardinality scenario.

WebBase Outlinks-200 (webbase) This dataset is a web graph from the Stanford WebBase project [HRGMP00]. We extract the data[5] using tools from the WebGraph project [BV04]. We only keep pages that have more than 200 outlinks, following Melnik et al. [MGM03], to exhibit a high set-cardinality scenario.

Implementation details

We implement all algorithms in Java. The signature length of SHJ is set to optimal according to paper [HM97]. The signature length of PTSJ is set as suggested in section 9.2.2. For PRETTI and PRETTI+ we maintain a hash map in each trie node to enable fast access to children while traversing. This is costly but necessary for the algorithm to reach its best performance. Note that here we tried various efficient implementations of hash map (e.g., Fastutil[6], CompactCollections[7], Trove[8]), and we found the HashMap implementation from JDK 7 itself has both the best performance and lowest main memory consumption. The open-source code of all implemented algorithms is available online[9].

[5]http://law.di.unimi.it/datasets.php
[6]http://fastutil.di.unimi.it/
[7]https://github.com/gratianlup/CompactCollections
[8]http://trove.starlight-systems.com/
[9]https://github.com/lgylym/scj

Test environment

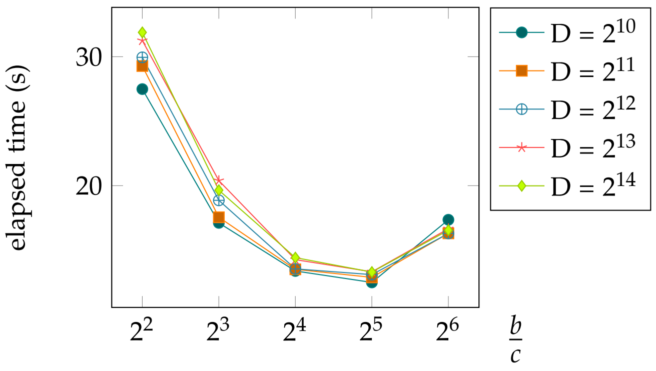
All experiments are executed on a single machine (Intel Xeon 2.27 GHz processor, 12GB main memory, Fedora 14 64-bit Linux, JDK 7). The JVM maximum heap size is set to 5GB, which we think is a decent setting even for today's computers. In the experiments we run each algorithm ten times, and record the average, standard deviation and median of running times. We observe in our measurements that the average gives a good estimate of the running time, and the standard deviation is not significant when compared with the overall time ($< 10\%$). Hence in the following we only show the average running time. We tend to test with bigger relations if possible, since larger relations and longer running times eliminates the random behavior introduced by OS scheduling. We run programs with `taskset` command, to restrict the execution on one CPU core. The running time we later present include the time to build indexes (e.g., hash map for SHJ and trie structures for the rest algorithms). We notice a trend here, that with the increase of set cardinality, the percentage of index build time over running time decreases. This is due to the fact that bigger set cardinality leads to more set element comparisons, which takes a larger portion of running time accordingly. But in general, the index build time of SHJ and PTSJ are less than 1% and 5% of the overall running time; PRETTI and PRETTI+ on the other hand take more than 70% and 20% of the running time to build indexes.

For PRETTI and PRETTI+ certain datasets are too big to run in the given memory. In such cases we switch the algorithms to the nested-loop on-disk versions. We notice that PRETTI and PRETTI+ may gain some efficiency by this approach, since the in-memory trie of a partition can be shallower than the global trie. This is more noticeable for high set cardinality scenarios.

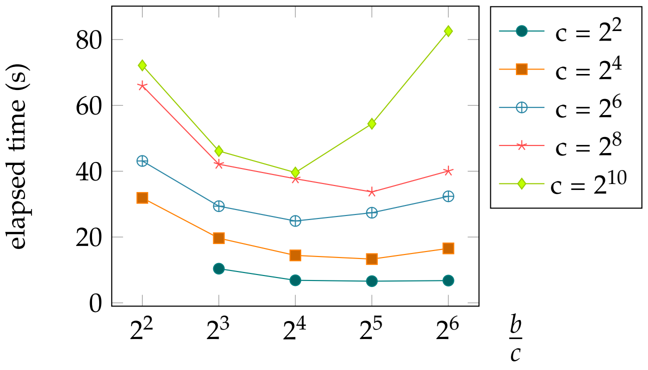
9.4.2. The optimal signature length of PTSJ

As we discussed, the signature length has a huge impact on PTSJ's performance, sometimes an order of magnitude difference. In Section 9.2.2 we gave some suggestions on how to choose signature length. In this section, we want to empirically validate these suggestions.

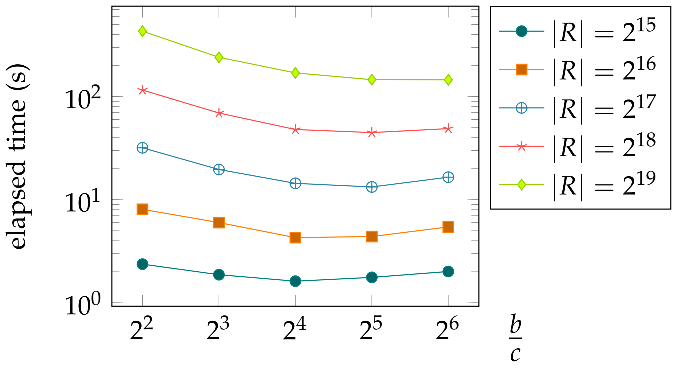
Given a dataset, there are three main properties: the relation size, the set cardinality, and the domain cardinality, all of which are independent from others. We want to know how these properties affect the behavior of PTSJ. The strategy of this investigation is to change one property while keeping the other two fixed. By examining the performance under different signature lengths, we can then clearly



(a) Impact of domain cardinality setting



(b) Impact of set cardinality setting



(c) Impact of relation size

Figure 9.3.: Performance of PTSJ with different signature length settings

see whether there is a correlation between a certain property and signature length. Table 9.3 summarizes the settings for this investigation.

Table 9.3.: Dataset configurations

fixed parameters	changing parameter
$\lvert R\rvert = 2^{17}$, $c = 2^4$	$d \in \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}\}$
$\lvert R\rvert = 2^{17}$, $d = 2^{14}$	$c \in \{2^2, 2^4, 2^6, 2^8, 2^{10}\}$
$c = 2^4$, $d = 2^{14}$	$\lvert R\rvert \in \{2^{15}, 2^{16}, 2^{17}, 2^{18}, 2^{19}\}$

Figure 9.3 shows the performance results of PTSJ, where the x-axis is the ratio between signature length b and set cardinality c. The strategy given in Section 9.2.2 suggests that a ratio between 16 and 32 is sufficient. In Figure 9.3a, we see that indeed, a ratio between 16 and 32 gives the best performance. Domain cardinality does not have a big impact on the signature selection. In Figure 9.3b we show how the algorithm performs under different set cardinality settings. Again PTSJ finds its best performance point between 16 and 32. We notice that for some high cardinality settings ($c = 2^8, 2^{10}$), comparing signatures themselves becomes an expensive operation. In these cases shorter signatures are preferred in general. Figure 9.3c shows the impact of relation size over signature length selection. We see a slow trend that when relations grow in size, the optimal signature length tends to move to larger values. This is indicated by formula 9.2, where $|R|$ is part of the factor. But as we observe, a ratio between 16 and 32 can already give a good result.

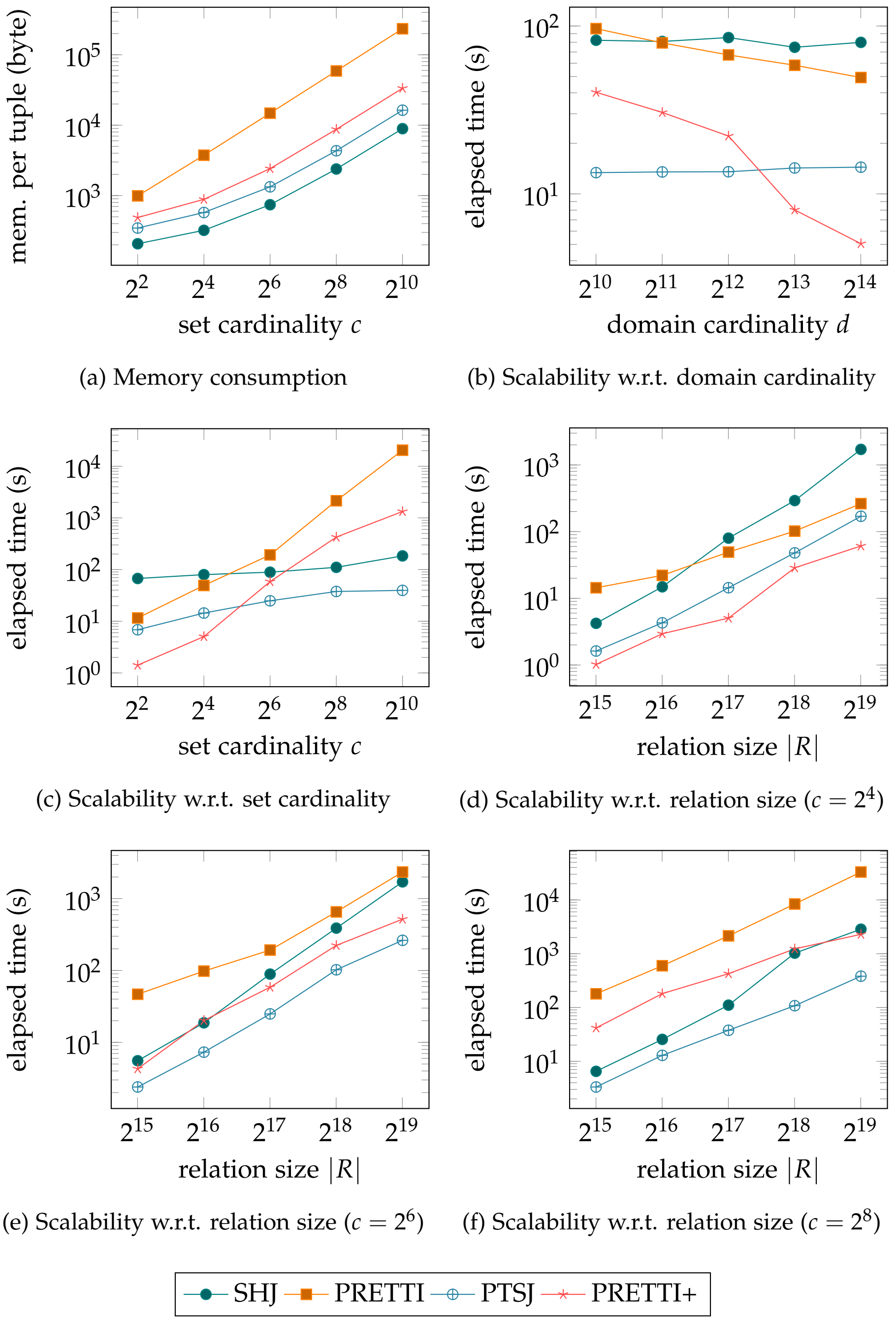
Overall, these experiments support our signature selection strategy of Section 9.2.2. A signature of length between $16c$ and $32c$ is usually a good selection.

9.4.3. Comparison of algorithms

In this section we discuss the experimental results of the four algorithms on various synthetic datasets. We test on different settings to show the scalability of all algorithms. Figure 9.4 shows experiments on uniformly distributed datasets. Figure 9.5 further shows performance on Poisson and Zipf distributions. Dataset configuration is the same as in Table 9.3.

Space efficiency for different algorithms

Main-memory consumption is an essential factor for evaluating main memory algorithms. Low main-memory consumption indicates better scalability of the



(a) Memory consumption

(b) Scalability w.r.t. domain cardinality

(c) Scalability w.r.t. set cardinality

(d) Scalability w.r.t. relation size ($c = 2^4$)

(e) Scalability w.r.t. relation size ($c = 2^6$)

(f) Scalability w.r.t. relation size ($c = 2^8$)

Figure 9.4.: Comparison of different algorithms for uniformly distributed data

algorithm with respect to larger datasets. It is not difficult to get a rough estimation of memory consumption for the algorithms mentioned in this paper. The main differences come from the different data structures (indexes) each algorithm uses. For instance, for SHJ, a hash table has to be built; for PRETTI and PRETTI+, a prefix tree and an inverted index; for PTSJ, a Patricia trie.

In general, two factors influence memory consumption: (1) relation size $|R|$ and (2) set cardinality c. The influence of relation size is obvious: the number of hash table entries grows linearly with relation size, and so does the size of the prefix tree and inverted index, and the Patricia trie. Set cardinality, on the other hand, has a larger impact on PRETTI and PRETTI+, while SHJ and PTSJ are not so sensitive to it.

We can clearly see this via our experiments. In Figure 9.4a, we plot, for each join algorithm with different set cardinality settings, the main memory consumption per tuple. Here we note that, though the experiment runs with 2^{17} tuples, the result stays the same for much larger relations. This means that we can estimate how much memory we need, given information about relation size and set cardinality.

We see that the memory consumption basically has a linear relationship with set cardinality. SHJ, PTSJ and PRETTI+ vary by a constant factor, which is basically the cost of longer signatures (PTSJ), Patricia trie (PTSJ and PRETTI+) and inverted index (PRETTI+). PRETTI on the other hand, needs around ten times more main-memory than others. For a relation with set cardinality 2^6, it needs more than 10KB per tuple, which means 10GB for just one million tuples. This empirically substantiates our remarks on PRETTI.

Scalability with different domain cardinality settings

Figure 9.4b depicts performance with different domain cardinality settings. We see that the signature-based solutions (SHJ and PTSJ) are not sensitive to changes in domain cardinality, since they operate on the signature space instead of on the set element space. PRETTI and PRETTI+, on the other hand, operate directly on the set element space. Larger domain cardinality indicates more entries in the inverted index, and shorter inverted lists (therefore faster merge joins on the lists). So PRETTI and PRETTI+ perform better when domain cardinality is high.

Scalability with different set cardinality settings

In order to determine the scalability of the algorithms with respect to set cardinality, we set the relation size to 2^{17}, with average set cardinality varying from 2^2 to 2^{10}.

The very high set cardinality scenarios (2^{10}) are not uncommon, especially in the context of graph analytics. We'll see more data of this kind from experiments with real data. In Figure 9.4c, we see that PRETTI and PRETTI+ are both more sensitive to set cardinalities, compared to the signature-based solutions. When set cardinality is lower (below 2^5), PRETTI+ is a better choice over the other alternatives; but beyond that point, PTSJ is a better choice. In each case, one of our new algorithms will achieve nearly an order of magnitude performance gain over the best of SHJ and PRETTI.

Scalability with different relation sizes

Algorithm scalability with respect to relation size may be the most important factor in practice. From Figure 9.4d to 9.4f, we show performance with difference set cardinality scenarios ($c = 2^4, 2^6, 2^8$). Just as we saw earlier, for low cardinality settings (Figure 9.4d), PRETTI+ is a clear winner, followed by PTSJ, PRETTI and SHJ. When set cardinality grows, the advantages of signature-based solutions start to show. PTSJ becomes a better choice over the others. The difference becomes more significant with larger relation sizes. In Figure 9.4f we see that in many cases in-memory PRETTI (and PRETTI+) cannot finish the experiments, so we switch the algorithm to a disk-based nested-loop version.

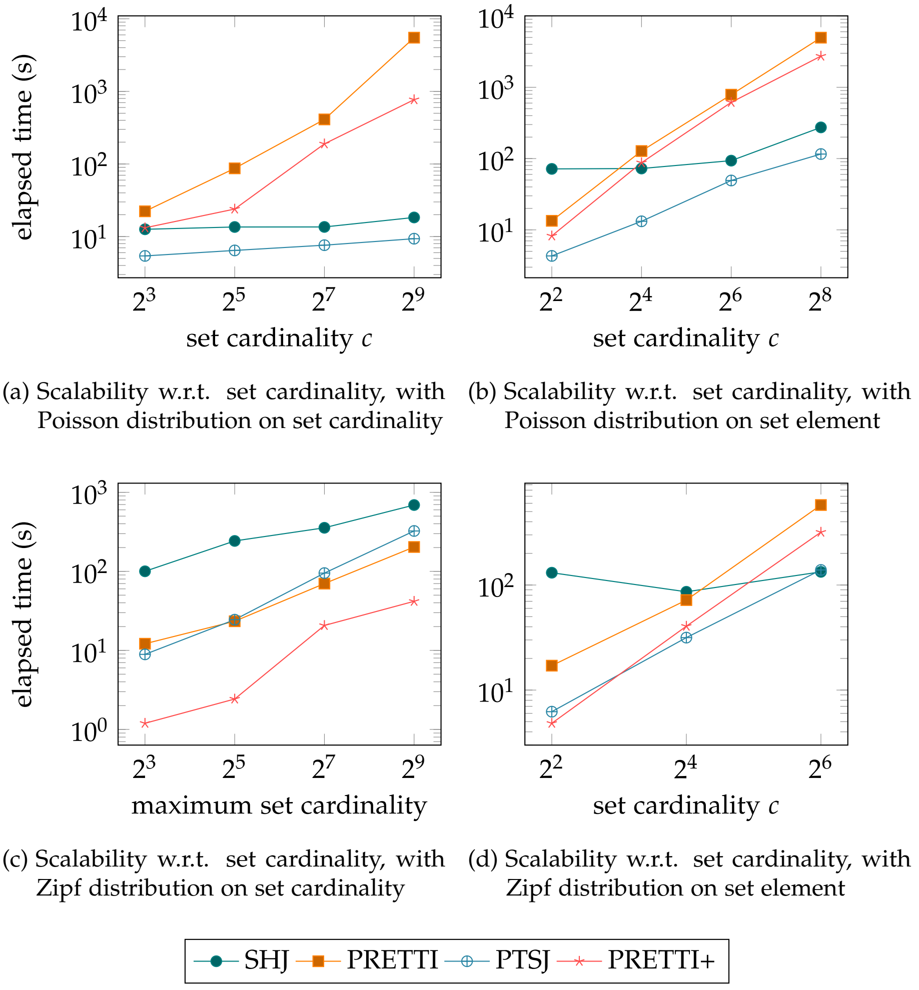
Poisson distribution and Zipf distribution

Here we want to determine if different distributions on the set cardinality and set elements have an impact on performance. We test datasets ($|R| = 2^{17}$) with two distributions: Poisson distribution and Zipf distribution, which are widely found in real-world datasets. Distributions are applied to either set cardinality or set elements. We expect that the distribution on set cardinality will have a greater impact, as shown previously. Unless specified otherwise, the x-axis shows the average set cardinalities.

In Figure 9.5a we show datasets with Poisson distribution on set cardinalities. This setting is bad news for PRETTI and PRETTI+, because then the set cardinality can be potentially large. We see that indeed, even when $c = 2^3$, PRETTI and PRETTI+ are not competitive with PTSJ. Indeed, PTSJ performs the best in all cases.

Figure 9.5b shows Poisson distribution on set elements. This distribution does not make a significant difference for all algorithms, which behave as in Figure 9.4c.

Zipf distribution on set cardinality favors PRETTI and PRETTI+. As in Figure 9.5c, we see that PRETTI+ becomes the best solution on all settings. Note that in this case



(a) Scalability w.r.t. set cardinality, with Poisson distribution on set cardinality

(b) Scalability w.r.t. set cardinality, with Poisson distribution on set element

(c) Scalability w.r.t. set cardinality, with Zipf distribution on set cardinality

(d) Scalability w.r.t. set cardinality, with Zipf distribution on set element

Figure 9.5.: Comparison of different algorithms for skewed distributions

the x-axis is the maximum set cardinality instead of average. Since c follows a Zipf distribution, many sets have small c and only a few have larger ones. In fact, the median set cardinality for the dataset with $max\ c = 2^9$ is only 17. This explains why PRETTI+ performs so well.

Zipf distribution on set elements, as in Figure 9.5b, does not have a huge impact on performance differences. PRETTI and PRETTI+ perform slightly better than in uniform distribution, since they could produce results earlier due to the nature of Zipf distribution (frequent elements are placed near the trie root).

Overall, our observation is that distributions on set cardinality have a big impact on performance. In such cases, we need to not only examine the average set cardinality, but also the median of set cardinality of data, for choosing the right algorithm. Nonetheless, either PTSJ or PRETTI+ will be the best choice, with sometimes a 10-fold speedup compared with the previous state-of-the-art.

9.4.4. Experiments on real-world datasets

Figure 9.6 summarizes performance on various real-world datasets, where we plot the ratio of a certain algorithm's running time over the best algorithm for that dataset. We see that the performance can vary in an order of magnitude for many algorithms. In low-to-medium set cardinality settings (flickr, orkut), PRETTI+ is the clear winner, where signature based methods, even PTSJ, are at least three times slower. SHJ in these two cases runs longer than a day. When it comes to medium-to-high set cardinality settings (twitter) however, the benefit of signatures starts to appear, PTSJ can make the computation 3.6 times faster than the second best (SHJ). For webbase, PTSJ again is at least 8 times faster than the state-of-the-art, and 2.6 times faster than PRETTI+.

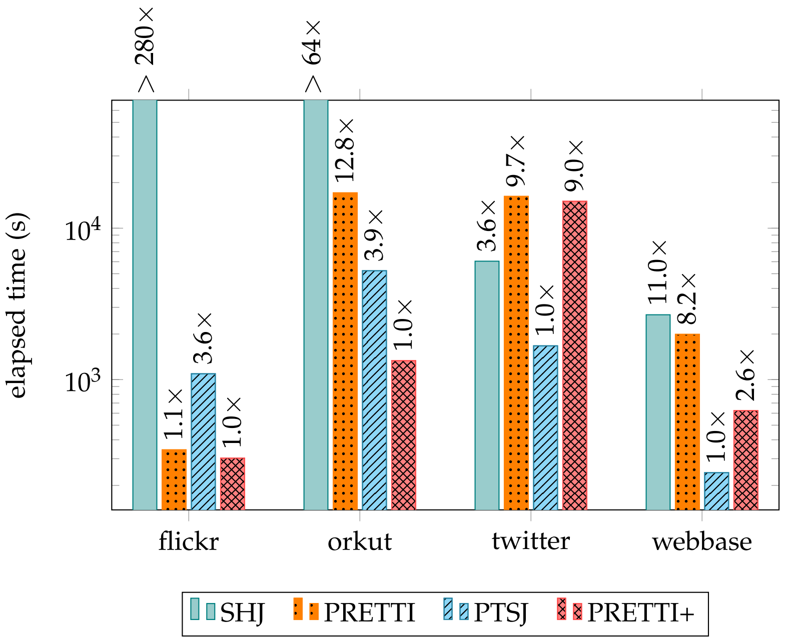


Figure 9.6.: Algorithm performance comparison for different real-world datasets

9.5. Set-containment join algorithms, the BSP version

Since we now have efficient in-memory set-containment join algorithms, one natural next step is to investigate on distributed and external memory solutions. As we've seen in Chapter 7, many algorithms have been developed for external memory setting. However, not much work has been done for distributed algorithms for set-containment join. The closest works we are aware of are papers about join algorithms on MapReduce framework (e.g., [MF12, LOÖW14]), which cannot be easily adapted to the set-containment join problem. In this section, we propose some ideas for developing set-containment join algorithms for the BSP model, that also make use of a trie structure. Such algorithms are also suitable for adapting to the algorithm transformation framework.

Can we simply create BSP versions of PTSJ/PRETTI+? As we observed, both algorithms depend on their in-memory data structures, and are computation-bounded instead of I/O-bounded. Learning from classic parallel join algorithms [SKS10, Chapter 18], we know there are several ways to make an in-memory algorithm distributed, all of which contain some data distribution (partitioning) strategy.

A straight forward strategy is to perform a nested-loop join on partitions of both relations, and perform set-containment join on each pair of partitions on some machine. Then we can plug in any set-containment join algorithm from this chapter for the in-memory processing part.

A Fragment-and-Replicate Join can also be applied. We can send partitions of one relation to machines, and replicate the other on all machines. Then on each machine, we build a trie on the partitioned relation (in memory) and query over the other relation on this partial trie (with a sequential scan over the relation).

It is not necessary to replicate the whole relation to all machines. The partitioning strategy from PSJ, APSJ and ADCJ (see Section 7.4) can be applied in a distributed environment as well. Only in this case, tuples are not saved to disk blocks, but sent over to different machines.

Based on the available trie structure, we can do even more than the above approaches. The idea is to use the enumeration facility from Chapter 8 to distribute tuples. The algorithm runs as follows:

1. Assume we have relations S and R, both with signatures created. We distribute tuples in S to machines basing on the first several bits of their signature

(referred as partial signature).

2. We collect partial signatures from all machines and build a *trie*.
3. We distribute the *trie* to all machines.
4. We distribute tuples in R randomly among machines. For each tuple of R on some machine, by using the *trie*, we determine which machines will contain potential subsets of the tuple, and send the tuple to these machines.
5. On each machine, we now have tuples that have the containment relation on the partial signature (some prefix of the signature). We just need to compare the rest part of the signatures and validate them on set elements (using for example, PTSJ).

9.6. Conclusion

Motivated by recent hardware trends and practical applications from graph analytics, query processing, OLAP systems, and data mining tasks, in this chapter we proposed and studied two efficient and scalable set-containment join algorithms: PTSJ and PRETTI+. The latter is suitable for low set cardinality, high domain cardinality settings, while the former is a more common algorithm suitable for the other scenarios. As shown in the experiments, these two new algorithms can be remarkably faster in many cases than the existing state-of-the-art, and scale gracefully with set cardinality, domain cardinality, and relation size. Detailed analysis has been carried out for PTSJ, especially for finding the optimal value for the critical parameter (signature length). Various extensions of PTSJ make it possible to reuse the index structure to answer other types of join queries, such as set-similarity joins. Finally we proposed several ideas to design trie-based set-containment join algorithm under the BSP model.

10. Conclusions

10.1. Research summary

In this thesis, we have demonstrated, through a series of concrete examples, ways to design algorithms for big graphs. Along the way we also gave answers to the three concrete research questions proposed in Chapter 1.

> Q1: What is the workflow to design algorithm for different computation models?

After investigating on various computational models, we answered Q1 by proposing the algorithm transformation framework. Using this framework, it is possible to first design BSP algorithms and then transform the algorithms to other models. We used several real-world graph problems, namely PageRank, triangle counting and k-bisimulation to demonstrate the practicality of the framework.

> Q2: Can we design a practical bisimulation reduction algorithm for big graphs?

We answered Q2 by developing a series of efficient k-bisimulation partitioning algorithms. The design process followed the algorithm transformation framework. The I/O-efficient algorithms, among all algorithms, are the first known I/O-efficient solutions, that can easily handle big graphs on a single commodity PC.

> Q3: How can we accelerate state-of-the-art set-containment join algorithms?

Q3 is answered in Part II. By carefully analyzing the existing algorithms and applying novel data structures and design choices, we showed that it is possible to improve the performance of the previous state-of-the-art set-containment join algorithms to an order of magnitude faster.

10.2. Future work

In Chapter 1, we raised the overall research question:

> Q0: Is there a paradigm for designing algorithms for massive graph data under various computation models?

Lots of research problems will need to be studied to answer this huge question. But based on the focus of this thesis, we discuss some interesting topics for further investigation.

Algorithm transformation framework In Chapter 2, we used the algorithm transformation framework as a conceptual tool for designing algorithms. We are curious if this idea can turn into a real system, which serves as a middle layer, doing the transformations automatically. This middle layer can then be integrated with many big-data platforms as back-ends. In this case users can achieve the dream of "write code once, run everywhere".

Algorithm extensions First, it would be interesting to explore adaptations and extensions of our algorithms for alternative hardware platforms (e.g., multicore, SSD). Second, as we indicated at various points, many alternative data structures and join algorithms can be investigated for optimizing various aspects of the proposed algorithms (e.g., multi-way trie and trie-tire join for set-containment join). Third, since we now have efficient set-containment join algorithms, integrating them into the k-simulation computation is a natural next step. Last but not least, for set-containment join, it would be interesting to study how our efficient solutions can be adapted to other related data models such as uncertain sets [ZCS$^+$12] and complex sets [IF13].

K-(Bi)simulation result analysis First, other interesting measurements on the k-BPR graphs can be performed; features such as diameter and clustering coefficient may show different properties when compared with the original graphs. Second, it would be interesting to analyze the different behaviors of labeled and unlabeled graphs (as in Sec. 5.2), and determining the causes. Third, as we have seen throughout the chapter, synthetic graph generators fail to deliver power-law distribution bisimulation results as observed in real graphs. Studying ways to solve this problem on existing graph generation models or with new models is an important research direction. Last but not least, similar research could be carried out on other related reductions, such as simulation partition graphs [HHK95].

The bigger picture It is becoming clearer that in the near future, the line between analytical systems and transaction systems will blur [Pla09]. Moreover, the line between database systems, batch processing systems, and stream processing systems will also blur (e.g., [Zah13, SGL13, MSZ12, Rus13]). Essentially, we will have one system that stores all data, that will not only accept well-defined queries, but also can execute code/script based on certain programming paradigm (e.g., MapReduce, Pregel-like), and the system's response is expected to be real-time or near real-time. Our algorithm transformation framework can certainly be the middle layer between the user-interaction functionality (query engine and algorithm interpreter) and back-end storage systems (distributed or single machine, in memory or on disk). The k-BPR graphs, as a result of the k-bisimulation preprocessing step, can serve as a structural index for many graph related query answering and computation tasks.

Bibliography

[ADRR04] Gagan Aggarwal, Mayur Datar, Sridhar Rajagopalan, and Matthias Ruhl. On the streaming model augmented with a sorting primitive. In *FOCS*, pages 540–549, Rome, Italy, 2004.

[AFGV97] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory. In *STOC*, pages 540–548, El Paso, TX, USA, 1997.

[AGK06] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, Seoul, Korea, 2006.

[ALPH01] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernardo A. Huberman. Search in power-law networks. *Phys. Rev. E*, 64:046135, Sep 2001.

[AM10] Deepak Ajwani and Henning Meyerhenke. Realistic computer models. In *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice*, pages 194–236, 2010.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[AZ09] Nikolas Askitis and Justin Zobel. B-tries for disk-based string management. *The VLDB Journal*, 18(1):157–179, 2009.

[BC13] Antonio Badia and Bin Cao. Efficient implementation of generalized quantification in relational query languages. In *VLDB*, pages 241–252, 2013.

[BGK03] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *VLDB*, pages 141–152, Berlin, Germany, 2003.

[BM] David A. Bader and Kamesh Madduri. GTgraph: A suite of synthetic graph generators. `http://www.cse.psu.edu/~madduri/software/GTgraph/index.html`.

[BO05] Stefan Blom and Simona Orzan. Distributed state space minimization. *Int. J. STTT*, 7(3):280–291, 2005.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998.

[BS09] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[BV04] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *WWW*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[BW14] Antonio Badia and Anna Wagner. Complex SQL predicates as quantifiers. *IEEE Trans. Knowl. Data Eng.*, 26(7):1617–1630, 2014.

[CB07] Bin Cao and Antonio Badia. SQL query optimization through nested relational algebra. *ACM Trans. Database Syst.*, 32(3):1–46, August 2007.

[CCKN01] Jin-Yi Cai, Venkatesan T. Chakaravarthy, Raghav Kaushik, and Jeffrey F. Naughton. On the complexity of join predicates. In *PODS*, Santa Barbara, California, USA, 2001.

[CGS02] Yen-Yu Chen, Qingqing Gan, and Torsten Suel. I/O-efficient techniques for computing PageRank. In *CIKM*, pages 549–557, McLean, VA, USA, 2002.

[Cod] CodeChef. Tutorial for bitwise operations. `http://www.codechef.com/wiki/tutorial-bitwise-operations`.

[CSN09] A. Clauset, C. Shalizi, and M. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

[CWY09] Wei Chen, Yajun Wang, and Siyu Yang. Efficient influence maximization in social networks. In *KDD*, pages 199–208, Paris, France, 2009.

[Dem] Roman Dementiev. Pipelined external memory triangle counting/listing algorithm. `http://algo2.iti.kit.edu/dementiev/tria/algorithm.shtml`.

[Dem02] Erik D. Demaine. Cache-oblivious algorithms and data structures. Lecture Notes from the EEF Summer School on Massive Data Sets. University of Aarhus, Denmark, June 27–July 1, 2002.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DKS08] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Softw. Pract. Exper.*, 38(6):589–637, May 2008.

[DPP04] Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient algorithm for computing bisimulation equivalence. *Theor. Comp. Sci.*, 311(1-3):221–256, 2004.

[Fan12] Wenfei Fan. Graph pattern matching revised for social network analysis. In *ICDT*, pages 8–21, Berlin, Germany, 2012.

[FFF99] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, Cambridge, Massachusetts, USA, 1999.

[FG99] Paolo Ferragina and Roberto Grossi. The String B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[FGL$^+$14] George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Similarity and bisimilarity notions appropriate for characterizing indistinguishability in fragments of the calculus of relations. *J. Log. Comput.*, page exu018, 2014.

[FHV$^+$11] George H. L. Fletcher, Jan Hidders, Stijn Vansummeren, Yongming Luo, Francois Picalausa, and Paul De Bra. On guarded simulations and acyclic first-order languages. In *DBPL*, Seattle, US, 2011.

[FLWW12] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, Scottsdale, AZ, USA, 2012.

[FMS$^+$10] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Transactions on Algorithms*, 6(4), 2010.

[FVW$^+$09] George H. L. Fletcher, Dirk Van Gucht, Yuqing Wu, Marc Gyssens, Sofia Brenes, and Jan Paredaens. A methodology for coupling fragments of XPath with structural indexes for XML documents. *Inf. Syst.*, 34(7):657–670, 2009.

[FWL12] Jianhua Feng, Jiannan Wang, and Guoliang Li. Trie-join: A trie-based method for efficient string similarity joins. *The VLDB Journal*, 21(4):437–461, August 2012.

[GC95] Goetz Graefe and Richard L. Cole. Fast algorithms for universal quantification in large databases. *ACM Trans. Database Syst.*, 20(2):187–236, 1995.

[Gir14] Apache giraph. `http://giraph.apache.org/`, 2014.

[GJ00] Georgia Garani and Roger Johnson. Joining nested relations and subrelations. *Inf. Syst.*, 25(3):287–307, 2000.

[GO12] Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. In *ALENEX*, pages 65–74, Kyoto, Japan, 2012.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[GSZ11] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *ISAAC*, pages 374–383, Yokohama, Japan, 2011.

[Had14] Hadoop. `http://hadoop.apache.org/`, 2014.

[Ham14] Hama. `http://hama.apache.org/`, 2014.

[HANM07] Sven Helmer, Robin Aly, Thomas Neumann, and Guido Moerkotte. Indexing set-valued attributes with a multi-level extendible hashing scheme. In *DEXA*, pages 98–108, Regensburg, Germany, 2007.

[HB11] Tom Heath and Christian Bizer. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.

[HC09] Oktie Hassanzadeh and Mariano P. Consens. Linked Movie Data Base. In *LDOW*, Madrid, Spain, 2009.

[HD13] Marouane Hachicha and Jérôme Darmont. A survey of XML tree patterns. *IEEE Trans. Knowl. Data Eng.*, 25(1):29–46, 2013.

[HFH12] Jelle Hellings, George H. L. Fletcher, and Herman Haverkort. Efficient external-memory bisimulation on DAGs. In *SIGMOD*, pages 553–564, Scottsdale, AZ, USA, 2012.

[HHK95] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pages 453–462, Washington, DC, USA, 1995.

[HL91] Kien A. Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, pages 525–535, Barcelona, Catalonia, Spain, 1991.

[HM97] Sven Helmer and Guido Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *VLDB*, pages 386–395, Athens, Greece, 1997.

[HM03] Sven Helmer and Guido Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal*, 12(3):244–261, 2003.

[HRGMP00] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. Webbase: A repository of web pages. *Computer Networks*, 33(1):277–293, 2000.

[HRR98] Monika Rauch Henzinger, Prabhakar Raghavan, and Sridar Rajagopalan. Computing on data streams. In *External Memory Algorithms: DIMACS Workshop External Memory and Visualization*, volume 50, page 107. American Mathematical Soc., 1998.

[IF13] Ahmed Ibrahim and George H. L. Fletcher. Efficient processing of containment queries on nested sets. In *EDBT*, pages 227–238, Genoa, Italy, 2013.

[JP05] Ravindranath Jampani and Vikram Pudi. Using prefix-trees for efficiently computing set joins. In *DASFAA*, pages 761–772, Beijing, China, 2005.

[KAA$^+$13] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *EuroSys*, pages 169–182, Prague, Czech Republic, 2013.

[KBG12] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: large-scale graph computation on just a pc. In *OSDI*, pages 31–46, Hollywood, CA, USA, 2012.

[KLPM10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, Raleigh, North Carolina, USA, 2010.

[KS90] Paris C. Kanellakis and Scott A. Smolka. {CCS} expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43 – 68, 1990.

[KSBG02a] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, San Jose, CA, USA, 2002.

[KSBG02b] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proc. ICDE*, pages 129–140, San Jose, CA, USA, 2002.

[LAN06] Shaimaa Y. Lazem, Noha Adly, and Magdy Nagi. A new index structure for querying association rules. In *AINA*, pages 876–880, Vienna, Austria, 2006.

[LD10] Jimmy Lin and Chris Dyer. Data-intensive text processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[LdLF$^+$13] Yongming Luo, Yannick de Lange, George H. L. Fletcher, Paul De Bra, Jan Hidders, and Yuqing Wu. Bisimulation Reduction of Big Graphs on MapReduce. In *BNCOD*, Oxford, UK, 2013.

[LDWF11] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. PASS-JOIN: A partition-based method for similarity joins. In *VLDB*, pages 253–264, Istanbul, Turkey, 2011.

[LFH$^+$13a] Yongming Luo, George H. L. Fletcher, Jan Hidders, Paul De Bra, and Yuqing Wu. Regularities and dynamics in bisimulation reductions of big graphs. In *GRADES*, pages 13:1–13:6, New York, NY, USA, 2013.

[LFH$^+$13b] Yongming Luo, George H. L. Fletcher, Jan Hidders, Yuqing Wu, and Paul De Bra. External memory k-bisimulation reduction of big graphs. In *CIKM*, pages 919–928, San Francisco, CA, USA, 2013.

[LFHDar] Yongming Luo, George H. L. Fletcher, Jan Hidders, and Paul De Bra. Efficient and scalable trie-based algorithms for computing set containment relations. In *ICDE*, Seoul, Korea, 2015, to appear.

[LHYS14] Chengkai Li, Bin He, Ning Yan, and Muhammad Safiullah. Set predicates in SQL: Enabling set-level comparisons for dynamically formed groups. *IEEE Trans. Knowl. Data Eng.*, 26(2):438–452, 2014.

[LL13] David Lomet and Paul Larson, editors. *IEEE Data Eng. Bull., Special Issue on Main-Memory Database Systems*, volume 36, 2013.

[LOÖW14] Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 46(3):31, 2014.

[Low13] Yucheng Low. *GraphLab: A Distributed Abstraction for Large Scale Machine Learning*. PhD thesis, University of California, Berkeley, 2013.

[LSW10] Xirong Li, Cees G. M. Snoek, and Marcel Worring. Unsupervised multi-feature tag relevance learning for social image retrieval. In *CIVR*, pages 10–17, Xi'an, China, July 2010.

[LV07] Dirk Leinders and Jan Van den Bussche. On the complexity of division and set joins in the relational algebra. *J. Comput. Syst. Sci.*, 73(4):538–549, 2007.

[LvHS00] J Loughry, J van Hemert, and L Schoofs. Efficiently enumerating the subsets of a set. `http://www.applied-math.org/subset.pdf`, 2000.

[MAB+10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, Indianapolis, Indiana, USA, 2010.

[Mam03] Nikos Mamoulis. Efficient processing of joins on set-valued attributes. In *SIGMOD*, pages 157–168, San Diego, California, USA, 2003.

[McC96] William F. McColl. Universal computing. In *Euro-Par*, pages 25–36, Lyon, France, 1996.

[MF12] Ahmed Metwally and Christos Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. In *VLDB*, pages 704–715, Istanbul, Turkey, 2012.

[MGM01] Sergey Melnik and Hector Garcia-Molina. Divide-and-conquer algorithm for computing set containment joins (extended technical report). Technical Report 2001-32, Stanford InfoLab, September 2001.

[MGM03] Sergey Melnik and Hector Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28(1):56–99, March 2003.

[MKG+08] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the Flickr Social Network. In *WOSN*, pages 25–30, Seattle, WA, USA, August 2008.

[Mor68] Donald R. Morrison. Patricia–practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.

[MS99] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *ICDT*, pages 277–295, Jerusalem, Israel, 1999.

[MSZ12] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. An extension of datalog for graph queries. In *SEBD*, pages 177–184, 2012.

[NW78] Albert Nijenhuis and Herbert S Wilf. Combinatorial algorithms for computers and calculators. *Computer Science and Applied Mathematics, New York: Academic Press, 1978, 2nd ed.*, 1, 1978.

[OR02] Evelien Otte and Ronald Rousseau. Social network analysis: a powerful strategy, also for the information sciences. *J. Information Science*, 28(6):441–453, 2002.

[Pac12] Matthew Felice Pace. BSP vs MapReduce. In *ICCS*, pages 246–255, Omaha, Nebraska, USA, 2012.

[PFHV14] François Picalausa, George H. L. Fletcher, Jan Hidders, and Stijn Vansummeren. Principles of guarded structural indexing. In *ICDT*, pages 245–256, Athens, Greece, 2014.

[Pla09] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *SIGMOD*, pages 1–2, Providence, Rhode Island, USA, 2009.

[PLF$^+$12a] François Picalausa, Yongming Luo, George H. L. Fletcher, Jan Hidders, and Stijn Vansummeren. A Structural Approach to Indexing Triples. In *ESWC*, pages 406–421, Heraklion, Greece, 2012.

[PLF$^+$12b] François Picalausa, Yongming Luo, George H. L. Fletcher, Jan Hidders, and Stijn Vansummeren. A structural approach to indexing triples. In *ESWC*, pages 406–421, Crete, Greece, 2012.

[PT87] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16:973, 1987.

[QLO03] Chen Qun, Andrew Lim, and Kian Win Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, pages 134–144, San Diego, CA, USA, 2003.

[QZWX13] Jianbin Qin, Xiaoling Zhou, Wei Wang, and Chuan Xiao. Trie-based similarity search and join. In *Proc. of the Joint EDBT/ICDT 2013 Workshops*, pages 392–396, Genoa, Italy, 2013.

[Ran03] Ralf Rantzau. Processing frequent itemset discovery queries by division and set containment join operators. In *DMKD*, pages 20–27, San Diego, California, 2003.

[Ran14] Francesco Ranzato. An efficient simulation algorithm on Kripke structures. *Acta Inf.*, 51(2):107–125, 2014.

[RL98] S. Rajasekaran and I. Lee. Parallel algorithms for relational coarsest partition problems. *IEEE Trans. Parallel and Distributed Syst.*, 9(7):687–699, 1998.

[RM06] Ralf Rantzau and Christoph Mangold. Laws for rewriting queries containing division operators. In *ICDE*, page 21, Atlanta, GA, USA, 2006.

[RPNK00] Karthikeyan Ramasamy, Jignesh M Patel, Jeffrey F Naughton, and Raghav Kaushik. Set containment joins: The good, the bad and the ugly. In *VLDB*, pages 351–362, Cairo, Egypt, 2000.

[RSM08] Yves Raimond, Mark B Sandler, and Queen Mary. A Web of Musical Information. In *ISMIR*, pages 263–268, Philadelphia, PA, USA, 2008.

[RSMW03] Ralf Rantzau, Leonard D. Shapiro, Bernhard Mitschang, and Quan Wang. Algorithms and applications for universal quantification in relational databases. *Inf. Syst.*, 28(1-2):3–32, 2003.

[Rus13] John Russell. *Cloudera Impala*. O'Reilly, 2013.

[San11] Davide Sangiorgi. Origins of bisimulation and coinduction. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, pages 1–37. Cambridge University Press, 2011.

[Sav13] Iztok Savnik. Index data structure for fast subset and superset queries. In *CD-ARES*, pages 134–148, Regensburg, Germany, 2013.

[Sch07] Thomas Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, University Karlsruhe, 2007.

[Sco12] John Scott. *Social network analysis*. SAGE Publications Limited, 2012.

[Sed03] Robert Sedgewick. Radix Search. In *Algorithms in Java*. Addison-Wesley Professional, 2003.

[SGL13] Jiwon Seo, Stephen Guo, and Monica S Lam. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289, 2013.

[SHLP09] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. In *ICDE*, pages 222–233, Washington, DC, USA, 2009.

[SK97] Jop F. Sibeyn and Michael Kaufmann. BSP-like external-memory computation. In *CIAC*, pages 229–240, Rome, Italy, 1997.

[SK04] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, pages 743–754, Paris, France, 2004.

[SKS10] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw-Hill Book Company, 2010.

[SR11] Davide Sangiorgi and Jan Rutten. *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 1st edition, 2011.

[SSZ95] Scott A. Smolka, Oleg Sokolsky, and Shipei Zhang. On the parallel complexity of bisimulation and model checking. *Modal Logic and Process Algebra: A Bisimulation Perspective in CSLI Lecture Notes*, 53:257–288, 1995.

[Suj96] K Ronald Sujithan. Towards a scalable parallel object database-the bulk synchronous parallel approach. Technical Report PRG-TR-17-96, Oxford University, 1996.

[SV11] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, Hyderabad, India, 2011.

[SW13] Semih Salihoglu and Jennifer Widom. GPS: A graph processing system. In *SSDBM*, pages 22:1–22:12, Baltimore, Maryland, 2013.

[SZZ13] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. Graph queries in a next-generation datalog system. In *VLDB*, pages 1258–1261, Trento, Italy, 2013.

[TBM02] Eleni Tousidou, Panayiotis Bozanis, and Yannis Manolopoulos. Signature-based structures for objects with set-valued attributes. *Inf. Syst.*, pages 93–121, 2002.

[TBV$^+$11] Manolis Terrovitis, Panagiotis Bouros, Panos Vassiliadis, Timos Sellis, and Nikos Mamoulis. Efficient answering of set containment queries for skewed item distributions. In *EDBT*, pages 225–236, Uppsala, Sweden, 2011.

[THP08] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *SIGMOD*, pages 567–580, Vancouver, Canada, 2008.

[TKMF09] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. DOULION: Counting triangles in massive graphs with a coin. In *KDD*, pages 837–846, Paris, France, 2009.

[TL02] Alexander Tuzhilin and Bing Liu. Querying multiple sets of discovered rules. In *KDD*, pages 52–60, Edmonton, Alberta, Canada, 2002.

[TPVS06] Manolis Terrovitis, Spyros Passas, Panos Vassiliadis, and Timos Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *CIKM*, pages 728–737, Arlington, Virginia, USA, 2006.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[Vit08] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2008.

[Wal] Stephen Walkauskas. Counting triangles. `http://www.vertica.com/2011/09/21/counting-triangles/`.

[WDG$^+$14] Sebastian Wandelt, Dong Deng, Stefan Gerdjikov, Shashwat Mishra, Petar Mitankin, Manish Patil, Enrico Siragusa, Alexander Tiskin, Wei Wang, Jiaying Wang, and Ulf Leser. State-of-the-art in string similarity search and join. *SIGMOD Rec.*, 43(1):64–76, May 2014.

[YBRM14] A. N. Yzelman, Rob H. Bisseling, Dirk Roose, and Karl Meerbergen. MulticoreBSP for C: A high-performance library for shared-memory parallel programming. *International Journal of Parallel Programming*, 42(4):619–642, 2014.

[YHSY04] Ke Yi, Hao He, Ioana Stanoi, and Jun Yang. Incremental maintenance of XML structural indexes. In *SIGMOD*, pages 491–502, Paris, France, 2004.

[YL12] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *ICDM*, pages 745–754, Brussels, Belgium, 2012.

[Zah13] Matei Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters*. PhD thesis, UC Berkeley, 2013.

[ZCS$^+$12] Xiaolong Zhang, Ke Chen, Lidan Shou, Gang Chen, Yuan Gao, and Kian-Lee Tan. Efficient processing of probabilistic set-containment queries on uncertain set-valued data. *Information Sciences*, 196:97–117, 2012.

[ZLY09] Shijie Zhang, Shirong Li, and Jiong Yang. GADDI: Distance index based subgraph matching in biological networks. In *EDBT*, pages 192–203, Saint Petersburg, Russia, 2009.

[ZÖC$^+$14] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. gStore: a graph-based SPARQL query engine. *The VLDB Journal*, 23(4):565–590, 2014.

Acknowledgements

Writing down the acknowledgement chapter is a difficult moment. I put off this moment as much as possible, because that is a sign that my 4-year PhD journey at TU/e is about to end.

I would like to thank my supervisors, dr. George Fletcher, prof. Paul De Bra, and dr. Jan Hidders. I could not ask for more support from all of you. It is indeed a luxury and privilege for me to have three people spending so much time with me on my project. Especially George, my daily supervisor, gives me all the support and freedom that I could imagine. Whenever I have questions and doubts, I know that I can always knock on your door.

I would like to thank the thesis committee members and the reviewers of my paper (though anonymous). Your insightful and detailed comments make my text more complete, and help me to understand the problems better.

I would like to thank The Netherlands Organisation for Scientific Research (NWO) for founding my research, and SIKS school for organizing many useful and interesting events for PhD students in the field. This thesis could not be done without these supports.

I would like to thank my colleagues, dr. Francois Picalausa and dr. Stijn Vansummeren from ULB, and prof. Yuqing Wu from Indiana. You have helped me so much since I started my project. I thank Yannick de Lange for your good work while we were working together. Overall, it has been a pleasure to work with all of you.

During the years, I share the office and enjoy vlaai/lunches together with many of the former and current information system group members. Thank you all for the interesting discussions, and for providing an energetic and peaceful atmosphere. My special thanks goes to our secretary Riet, for the very detailed proof reading of the whole thesis.

I thank prof. Jianzhong Li, prof. Hong Gao, and dr. Hongzhi Wang from Harbin Institute of Technology, database group. Thank you for bringing me into the field of database research and letting me try a lot of new things while I was in the group. The interest I grew there inspired me to continue my PhD study in Eindhoven.

A special thank you goes to my family. Words cannot express how grateful I am to my parents, for their love and support over the years. Thank you my beloved fiancee Jiazuo, for your encouragement and caring. My life is so different and colorful since we met.

By the time of my defense, I will just reach 30. According to Confucius, 30 is the year to start "standing firm", to be independent and take responsibility, and more importantly, to have the goal for life. Thanks again to all of you, I can see that I'm gradually reaching this point.

Curriculum Vitae

Yongming Luo was born on 25-02-1985 in Fushun, China. From 2003 to 2009, he took the honors Bachelor-Master program at Harbin Institute of Technology in Harbin, China, with degrees in Computer Science and Technology. In 2009 and 2010, he studied within the Master of Science in Information Networking (MSIN) Athens program of Carnegie Mellon University, USA, provided with full scholarship. From March 2011 he started working on the SEEQR PhD project at Eindhoven University of Technology at Eindhoven, the Netherlands, of which the results are presented in this dissertation.

Publications

- *Efficient and scalable trie-based algorithms for computing set containment relations.* Yongming Luo, George H. L. Fletcher, Jan Hidders, and Paul De Bra. ICDE 2015, to appear, Seoul, Korea.

- *Regularities and dynamics in bisimulation reductions of big graphs.* Yongming Luo, George H. L. Fletcher, Jan Hidders, Paul De Bra, Yuqing Wu. GRADES@SIGMOD 2013, New York, US.

- *Bisimulation Reduction of Big Graphs on MapReduce.* Yongming Luo, Yannick de Lange, George H. L. Fletcher, Paul De Bra, Jan Hidders and Yuqing Wu. BNCOD 2013, Oxford, UK.

- *External Memory k-Bisimulation Reduction of Big Graphs.* Yongming Luo, George H.L. Fletcher, Jan Hidders, Yuqing Wu, Paul De Bra. CIKM 2013, San Francisco, US.

- *A Structural Approach to Indexing Triples.* Francois Picalausa, Yongming Luo, George H. L. Fletcher, Jan Hidders and Stijn Vansummeren. ESWC 2012, Heraklion, GR.

- *Storing and Indexing Massive RDF Datasets.* Yongming Luo, Francois Picalausa, George H. L. Fletcher, Jan Hidders and Stijn Vansummeren. In: Semantic Search over the Web, Data-Centric Systems and Applications, pp. 31âĂŞ60. Springer, Heidelberg (2012).
- *On guarded simulations and acyclic first-order languages.* George Fletcher, Jan Hidders, Stijn Vansummeren, Yongming Luo, Francois Picalausa, and Paul De Bra. DBPL 2011, Seattle, US.

SIKS Dissertations

2009

2009-01 Rasa Jurgelenaite (RUN), *Symmetric Causal Independence Models.*
2009-02 Willem Robert van Hage (VU), *Evaluating Ontology-Alignment Techniques.*
2009-03 Hans Stol (UvT), *A Framework for Evidence-based Policy Making Using IT.*
2009-04 Josephine Nabukenya (RUN), *Improving the Quality of Organisational Policy Making using Collaboration Engineering.*
2009-05 Sietse Overbeek (RUN), *Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality.*
2009-06 Muhammad Subianto (UU), *Understanding Classification.*
2009-07 Ronald Poppe (UT), *Discriminative Vision-Based Recovery and Recognition of Human Motion.*
2009-08 Volker Nannen (VU), *Evolutionary Agent-Based Policy Analysis in Dynamic Environments.*
2009-09 Benjamin Kanagwa (RUN), *Design, Discovery and Construction of Service-oriented Systems.*
2009-10 Jan Wielemaker (UVA), *Logic programming for knowledge-intensive interactive applications.*
2009-11 Alexander Boer (UVA), *Legal Theory, Sources of Law & the Semantic Web.*
2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin), *Operating Guidelines for Services.*
2009-13 Steven de Jong (UM), *Fairness in Multi-Agent Systems.*
2009-14 Maksym Korotkiy (VU), *From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA).*
2009-15 Rinke Hoekstra (UVA), *Ontology Representation - Design Patterns and Ontologies that Make Sense.*
2009-16 Fritz Reul (UvT), *New Architectures in Computer Chess.*
2009-17 Laurens van der Maaten (UvT), *Feature Extraction from Visual Data.*
2009-18 Fabian Groffen (CWI), *Armada, An Evolving Database System.*

2009-19 Valentin Robu (CWI), *Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets.*
2009-20 Bob van der Vecht (UU), *Adjustable Autonomy: Controling Influences on Decision Making.*
2009-21 Stijn Vanderlooy (UM), *Ranking and Reliable Classification.*
2009-22 Pavel Serdyukov (UT), *Search For Expertise: Going beyond direct evidence.*
2009-23 Peter Hofgesang (VU), *Modelling Web Usage in a Changing Environment.*
2009-24 Annerieke Heuvelink (VUA), *Cognitive Models for Training Simulations.*
2009-25 Alex van Ballegooij (CWI), *"RAM: Array Database Management through Relational Mapping".*
2009-26 Fernando Koch (UU), *An Agent-Based Model for the Development of Intelligent Mobile Services.*
2009-27 Christian Glahn (OU), *Contextual Support of social Engagement and Reflection on the Web.*
2009-28 Sander Evers (UT), *Sensor Data Management with Probabilistic Models.*
2009-29 Stanislav Pokraev (UT), *Model-Driven Semantic Integration of Service-Oriented Applications.*
2009-30 Marcin Zukowski (CWI), *Balancing vectorized query execution with bandwidth-optimized storage.*
2009-31 Sofiya Katrenko (UVA), *A Closer Look at Learning Relations from Text.*
2009-32 Rik Farenhorst (VU) and Remco de Boer (VU), *Architectural Knowledge Management: Supporting Architects and Auditors.*
2009-33 Khiet Truong (UT), *How Does Real Affect Affect Affect Recognition In Speech?.*
2009-34 Inge van de Weerd (UU), *Advancing in Software Product Management: An Incremental Method Engineering Approach.*
2009-35 Wouter Koelewijn (UL), *Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling.*
2009-36 Marco Kalz (OUN), *Placement Support for Learners in Learning Networks.*
2009-37 Hendrik Drachsler (OUN), *Navigation Support for Learners in Informal Learning Networks.*
2009-38 Riina Vuorikari (OU), *Tags and self-organisation: a metadata ecology for learning resources in a multilingual context.*
2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin), *Service Substitution – A Behavioral Approach Based on Petri Nets.*
2009-40 Stephan Raaijmakers (UvT), *Multinomial Language Learning: Investigations into the Geometry of Language.*

2009-41 Igor Berezhnyy (UvT), *Digital Analysis of Paintings.*

2009-42 Toine Bogers (UvT), *Recommender Systems for Social Bookmarking.*

2009-43 Virginia Nunes Leal Franqueira (UT), *Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients.*

2009-44 Roberto Santana Tapia (UT), *Assessing Business-IT Alignment in Networked Organizations.*

2009-45 Jilles Vreeken (UU), *Making Pattern Mining Useful.*

2009-46 Loredana Afanasiev (UvA), *Querying XML: Benchmarks and Recursion.*

2010

2010-01 Matthijs van Leeuwen (UU), *Patterns that Matter.*

2010-02 Ingo Wassink (UT), *Work flows in Life Science.*

2010-03 Joost Geurts (CWI), *A Document Engineering Model and Processing Framework for Multimedia documents.*

2010-04 Olga Kulyk (UT), *Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments.*

2010-05 Claudia Hauff (UT), *Predicting the Effectiveness of Queries and Retrieval Systems.*

2010-06 Sander Bakkes (UvT), *Rapid Adaptation of Video Game AI.*

2010-07 Wim Fikkert (UT), *Gesture interaction at a Distance.*

2010-08 Krzysztof Siewicz (UL), *Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments.*

2010-09 Hugo Kielman (UL), *A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging.*

2010-10 Rebecca Ong (UL), *Mobile Communication and Protection of Children.*

2010-11 Adriaan Ter Mors (TUD), *The world according to MARP: Multi-Agent Route Planning.*

2010-12 Susan van den Braak (UU), *Sensemaking software for crime analysis.*

2010-13 Gianluigi Folino (RUN), *High Performance Data Mining using Bio-inspired techniques.*

2010-14 Sander van Splunter (VU), *Automated Web Service Reconfiguration.*

2010-15 Lianne Bodenstaff (UT), *Managing Dependency Relations in Inter-Organizational Models.*

2010-16 Sicco Verwer (TUD), *Efficient Identification of Timed Automata, theory and prac-*

tice.

2010-17 Spyros Kotoulas (VU), *Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications.*

2010-18 Charlotte Gerritsen (VU), *Caught in the Act: Investigating Crime by Agent-Based Simulation.*

2010-19 Henriette Cramer (UvA), *People's Responses to Autonomous and Adaptive Systems.*

2010-20 Ivo Swartjes (UT), *Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative.*

2010-21 Harold van Heerde (UT), *Privacy-aware data management by means of data degradation.*

2010-22 Michiel Hildebrand (CWI), *End-user Support for Access to Heterogeneous Linked Data.*

2010-23 Bas Steunebrink (UU), *The Logical Structure of Emotions.*

2010-24 Dmytro Tykhonov, *Designing Generic and Efficient Negotiation Strategies.*

2010-25 Zulfiqar Ali Memon (VU), *Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective.*

2010-26 Ying Zhang (CWI), *XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines.*

2010-27 Marten Voulon (UL), *Automatisch contracteren.*

2010-28 Arne Koopman (UU), *Characteristic Relational Patterns.*

2010-29 Stratos Idreos(CWI), *Database Cracking: Towards Auto-tuning Database Kernels.*

2010-30 Marieke van Erp (UvT), *Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval.*

2010-31 Victor de Boer (UVA), *Ontology Enrichment from Heterogeneous Sources on the Web.*

2010-32 Marcel Hiel (UvT), *An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems.*

2010-33 Robin Aly (UT), *Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval.*

2010-34 Teduh Dirgahayu (UT), *Interaction Design in Service Compositions.*

2010-35 Dolf Trieschnigg (UT), *Proof of Concept: Concept-based Biomedical Information Retrieval.*

2010-36 Jose Janssen (OU), *Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification.*

2010-37 Niels Lohmann (TUE), *Correctness of services and their composition.*
2010-38 Dirk Fahland (TUE), *From Scenarios to components.*
2010-39 Ghazanfar Farooq Siddiqui (VU), *Integrative modeling of emotions in virtual agents.*
2010-40 Mark van Assem (VU), *Converting and Integrating Vocabularies for the Semantic Web.*
2010-41 Guillaume Chaslot (UM), *Monte-Carlo Tree Search.*
2010-42 Sybren de Kinderen (VU), *Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach.*
2010-43 Peter van Kranenburg (UU), *A Computational Approach to Content-Based Retrieval of Folk Song Melodies.*
2010-44 Pieter Bellekens (TUE), *An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain.*
2010-45 Vasilios Andrikopoulos (UvT), *A theory and model for the evolution of software services.*
2010-46 Vincent Pijpers (VU), *e3alignment: Exploring Inter-Organizational Business-ICT Alignment.*
2010-47 Chen Li (UT), *Mining Process Model Variants: Challenges, Techniques, Examples.*
2010-48 Withdrawn, .
2010-49 Jahn-Takeshi Saito (UM), *Solving difficult game positions.*
2010-50 Bouke Huurnink (UVA), *Search in Audiovisual Broadcast Archives.*
2010-51 Alia Khairia Amin (CWI), *Understanding and supporting information seeking tasks in multiple sources.*
2010-52 Peter-Paul van Maanen (VU), *Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention.*
2010-53 Edgar Meij (UVA), *Combining Concepts and Language Models for Information Access.*

2011

2011-01 Botond Cseke (RUN), *Variational Algorithms for Bayesian Inference in Latent Gaussian Models.*
2011-02 Nick Tinnemeier(UU), *Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language.*

2011-03 Jan Martijn van der Werf (TUE), *Compositional Design and Verification of Component-Based Information Systems.*
2011-04 Hado van Hasselt (UU), *Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference.*
2011-05 Base van der Raadt (VU), *Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline..*
2011-06 Yiwen Wang (TUE), *Semantically-Enhanced Recommendations in Cultural Heritage.*
2011-07 Yujia Cao (UT), *Multimodal Information Presentation for High Load Human Computer Interaction.*
2011-08 Nieske Vergunst (UU), *BDI-based Generation of Robust Task-Oriented Dialogues.*
2011-09 Tim de Jong (OU), *Contextualised Mobile Media for Learning.*
2011-10 Bart Bogaert (UvT), *Cloud Content Contention.*
2011-11 Dhaval Vyas (UT), *Designing for Awareness: An Experience-focused HCI Perspective.*
2011-12 Carmen Bratosin (TUE), *Grid Architecture for Distributed Process Mining.*
2011-13 Xiaoyu Mao (UvT), *Airport under Control. Multiagent Scheduling for Airport Ground Handling.*
2011-14 Milan Lovric (EUR), *Behavioral Finance and Agent-Based Artificial Markets.*
2011-15 Marijn Koolen (UvA), *The Meaning of Structure: the Value of Link Evidence for Information Retrieval.*
2011-16 Maarten Schadd (UM), *Selective Search in Games of Different Complexity.*
2011-17 Jiyin He (UVA), *Exploring Topic Structure: Coherence, Diversity and Relatedness.*
2011-18 Mark Ponsen (UM), *Strategic Decision-Making in complex games.*
2011-19 Ellen Rusman (OU), *The Mind ' s Eye on Personal Profiles.*
2011-20 Qing Gu (VU), *Guiding service-oriented software engineering - A view-based approach.*
2011-21 Linda Terlouw (TUD), *Modularization and Specification of Service-Oriented Systems.*
2011-22 Junte Zhang (UVA), *System Evaluation of Archival Description and Access.*
2011-23 Wouter Weerkamp (UVA), *Finding People and their Utterances in Social Media.*
2011-24 Herwin van Welbergen (UT), *Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior.*

2011-25 Syed Waqar ul Qounain Jaffry (VU)), *Analysis and Validation of Models for Trust Dynamics.*
2011-26 Matthijs Aart Pontier (VU), *Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots.*
2011-27 Aniel Bhulai (VU), *Dynamic website optimization through autonomous management of design patterns.*
2011-28 Rianne Kaptein(UVA), *Effective Focused Retrieval by Exploiting Query Context and Document Structure.*
2011-29 Faisal Kamiran (TUE), *Discrimination-aware Classification.*
2011-30 Egon van den Broek (UT), *Affective Signal Processing (ASP): Unraveling the mystery of emotions.*
2011-31 Ludo Waltman (EUR), *Computational and Game-Theoretic Approaches for Modeling Bounded Rationality.*
2011-32 Nees-Jan van Eck (EUR), *Methodological Advances in Bibliometric Mapping of Science.*
2011-33 Tom van der Weide (UU), *Arguing to Motivate Decisions.*
2011-34 Paolo Turrini (UU), *Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations.*
2011-35 Maaike Harbers (UU), *Explaining Agent Behavior in Virtual Training.*
2011-36 Erik van der Spek (UU), *Experiments in serious game design: a cognitive approach.*
2011-37 Adriana Burlutiu (RUN), *Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference.*
2011-38 Nyree Lemmens (UM), *Bee-inspired Distributed Optimization.*
2011-39 Joost Westra (UU), *Organizing Adaptation using Agents in Serious Games.*
2011-40 Viktor Clerc (VU), *Architectural Knowledge Management in Global Software Development.*
2011-41 Luan Ibraimi (UT), *Cryptographically Enforced Distributed Data Access Control.*
2011-42 Michal Sindlar (UU), *Explaining Behavior through Mental State Attribution.*
2011-43 Henk van der Schuur (UU), *Process Improvement through Software Operation Knowledge.*
2011-44 Boris Reuderink (UT), *Robust Brain-Computer Interfaces.*
2011-45 Herman Stehouwer (UvT), *Statistical Language Models for Alternative Sequence Selection.*
2011-46 Beibei Hu (TUD), *Towards Contextualized Information Delivery: A Rule-based*

Architecture for the Domain of Mobile Police Work.
2011-47 Azizi Bin Ab Aziz(VU), *Exploring Computational Models for Intelligent Support of Persons with Depression.*
2011-48 Mark Ter Maat (UT), *Response Selection and Turn-taking for a Sensitive Artificial Listening Agent.*
2011-49 Andreea Niculescu (UT), *Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality.*

2012

2012-01 Terry Kakeeto (UvT), *Relationship Marketing for SMEs in Uganda.*
2012-02 Muhammad Umair(VU), *Adaptivity, emotion, and Rationality in Human and Ambient Agent Models.*
2012-03 Adam Vanya (VU), *Supporting Architecture Evolution by Mining Software Repositories.*
2012-04 Jurriaan Souer (UU), *Development of Content Management System-based Web Applications.*
2012-05 Marijn Plomp (UU), *Maturing Interorganisational Information Systems.*
2012-06 Wolfgang Reinhardt (OU), *Awareness Support for Knowledge Workers in Research Networks.*
2012-07 Rianne van Lambalgen (VU), *When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions.*
2012-08 Gerben de Vries (UVA), *Kernel Methods for Vessel Trajectories.*
2012-09 Ricardo Neisse (UT), *Trust and Privacy Management Support for Context-Aware Service Platforms.*
2012-10 David Smits (TUE), *Towards a Generic Distributed Adaptive Hypermedia Environment.*
2012-11 J.C.B. Rantham Prabhakara (TUE), *Process Mining in the Large: Preprocessing, Discovery, and Diagnostics.*
2012-12 Kees van der Sluijs (TUE), *Model Driven Design and Data Integration in Semantic Web Information Systems.*
2012-13 Suleman Shahid (UvT), *Fun and Face: Exploring non-verbal expressions of emotion during playful interactions.*
2012-14 Evgeny Knutov(TUE), *Generic Adaptation Framework for Unifying Adaptive Web-based Systems.*

2012-15 Natalie van der Wal (VU), *Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes..*
2012-16 Fiemke Both (VU), *Helping people by understanding them - Ambient Agents supporting task execution and depression treatment.*
2012-17 Amal Elgammal (UvT), *Towards a Comprehensive Framework for Business Process Compliance.*
2012-18 Eltjo Poort (VU), *Improving Solution Architecting Practices.*
2012-19 Helen Schonenberg (TUE), *What's Next? Operational Support for Business Process Execution.*
2012-20 Ali Bahramisharif (RUN), *Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing.*
2012-21 Roberto Cornacchia (TUD), *Querying Sparse Matrices for Information Retrieval.*
2012-22 Thijs Vis (UvT), *Intelligence, politie en veiligheidsdienst: verenigbare grootheden?.*
2012-23 Christian Muehl (UT), *Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction.*
2012-24 Laurens van der Werff (UT), *Evaluation of Noisy Transcripts for Spoken Document Retrieval.*
2012-25 Silja Eckartz (UT), *Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application.*
2012-26 Emile de Maat (UVA), *Making Sense of Legal Text.*
2012-27 Hayrettin Gurkok (UT), *Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games.*
2012-28 Nancy Pascall (UvT), *Engendering Technology Empowering Women.*
2012-29 Almer Tigelaar (UT), *Peer-to-Peer Information Retrieval.*
2012-30 Alina Pommeranz (TUD), *Designing Human-Centered Systems for Reflective Decision Making.*
2012-31 Emily Bagarukayo (RUN), *A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure.*
2012-32 Wietske Visser (TUD), *Qualitative multi-criteria preference representation and reasoning.*
2012-33 Rory Sie (OUN), *Coalitions in Cooperation Networks (COCOON).*
2012-34 Pavol Jancura (RUN), *Evolutionary analysis in PPI networks and applications.*
2012-35 Evert Haasdijk (VU), *Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics.*
2012-36 Denis Ssebugwawo (RUN), *Analysis and Evaluation of Collaborative Modeling*

Processes.

2012-37 Agnes Nakakawa (RUN), *A Collaboration Process for Enterprise Architecture Creation.*

2012-38 Selmar Smit (VU), *Parameter Tuning and Scientific Testing in Evolutionary Algorithms.*

2012-39 Hassan Fatemi (UT), *Risk-aware design of value and coordination networks.*

2012-40 Agus Gunawan (UvT), *Information Access for SMEs in Indonesia.*

2012-41 Sebastian Kelle (OU), *Game Design Patterns for Learning.*

2012-42 Dominique Verpoorten (OU), *Reflection Amplifiers in self-regulated Learning.*

2012-43 Withdrawn, .

2012-44 Anna Tordai (VU), *On Combining Alignment Techniques.*

2012-45 Benedikt Kratz (UvT), *A Model and Language for Business-aware Transactions.*

2012-46 Simon Carter (UVA), *Exploration and Exploitation of Multilingual Data for Statistical Machine Translation.*

2012-47 Manos Tsagkias (UVA), *Mining Social Media: Tracking Content and Predicting Behavior.*

2012-48 Jorn Bakker (TUE), *Handling Abrupt Changes in Evolving Time-series Data.*

2012-49 Michael Kaisers (UM), *Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions.*

2012-50 Steven van Kervel (TUD), *Ontologogy driven Enterprise Information Systems Engineering.*

2012-51 Jeroen de Jong (TUD), *Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching.*

2013

2013-01 Viorel Milea (EUR), *News Analytics for Financial Decision Support.*

2013-02 Erietta Liarou (CWI), *MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing.*

2013-03 Szymon Klarman (VU), *Reasoning with Contexts in Description Logics.*

2013-04 Chetan Yadati(TUD), *Coordinating autonomous planning and scheduling.*

2013-05 Dulce Pumareja (UT), *Groupware Requirements Evolutions Patterns.*

2013-06 Romulo Goncalves(CWI), *The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience.*

2013-07 Giel van Lankveld (UvT), *Quantifying Individual Player Differences.*

2013-08 Robbert-Jan Merk(VU), *Making enemies: cognitive modeling for opponent agents in fighter pilot simulators.*
2013-09 Fabio Gori (RUN), *Metagenomic Data Analysis: Computational Methods and Applications.*
2013-10 Jeewanie Jayasinghe Arachchige(UvT), *A Unified Modeling Framework for Service Design..*
2013-11 Evangelos Pournaras(TUD), *Multi-level Reconfigurable Self-organization in Overlay Services.*
2013-12 Marian Razavian(VU), *Knowledge-driven Migration to Services.*
2013-13 Mohammad Safiri(UT), *Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly.*
2013-14 Jafar Tanha (UVA), *Ensemble Approaches to Semi-Supervised Learning Learning.*
2013-15 Daniel Hennes (UM), *Multiagent Learning - Dynamic Games and Applications.*
2013-16 Eric Kok (UU), *Exploring the practical benefits of argumentation in multi-agent deliberation.*
2013-17 Koen Kok (VU), *The PowerMatcher: Smart Coordination for the Smart Electricity Grid.*
2013-18 Jeroen Janssens (UvT), *Outlier Selection and One-Class Classification.*
2013-19 Renze Steenhuizen (TUD), *Coordinated Multi-Agent Planning and Scheduling.*
2013-20 Katja Hofmann (UvA), *Fast and Reliable Online Learning to Rank for Information Retrieval.*
2013-21 Sander Wubben (UvT), *Text-to-text generation by monolingual machine translation.*
2013-22 Tom Claassen (RUN), *Causal Discovery and Logic.*
2013-23 Patricio de Alencar Silva(UvT), *Value Activity Monitoring.*
2013-24 Haitham Bou Ammar (UM), *Automated Transfer in Reinforcement Learning.*
2013-25 Agnieszka Anna Latoszek-Berendsen (UM), *Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System.*
2013-26 Alireza Zarghami (UT), *Architectural Support for Dynamic Homecare Service Provisioning.*
2013-27 Mohammad Huq (UT), *Inference-based Framework Managing Data Provenance.*
2013-28 Frans van der Sluis (UT), *When Complexity becomes Interesting: An Inquiry into the Information eXperience.*
2013-29 Iwan de Kok (UT), *Listening Heads.*
2013-30 Joyce Nakatumba (TUE), *Resource-Aware Business Process Management: Anal-*

ysis and Support.
2013-31 Dinh Khoa Nguyen (UvT), *Blueprint Model and Language for Engineering Cloud Applications.*
2013-32 Kamakshi Rajagopal (OUN), *Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development.*
2013-33 Qi Gao (TUD), *User Modeling and Personalization in the Microblogging Sphere.*
2013-34 Kien Tjin-Kam-Jet (UT), *Distributed Deep Web Search.*
2013-35 Abdallah El Ali (UvA), *Minimal Mobile Human Computer Interaction.*
2013-36 Than Lam Hoang (TUe), *Pattern Mining in Data Streams.*
2013-37 Dirk Börner (OUN), *Ambient Learning Displays.*
2013-38 Eelco den Heijer (VU), *Autonomous Evolutionary Art.*
2013-39 Joop de Jong (TUD), *A Method for Enterprise Ontology based Design of Enterprise Information Systems.*
2013-40 Pim Nijssen (UM), *Monte-Carlo Tree Search for Multi-Player Games.*
2013-41 Jochem Liem (UVA), *Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning.*
2013-42 Léon Planken (TUD), *Algorithms for Simple Temporal Reasoning.*
2013-43 Marc Bron (UVA), *Exploration and Contextualization through Interaction and Concepts.*

2014

2014-01 Nicola Barile (UU), *Studies in Learning Monotone Models from Data.*
2014-02 Fiona Tuliyano (RUN), *Combining System Dynamics with a Domain Modeling Method.*
2014-03 Sergio Raul Duarte Torres (UT), *Information Retrieval for Children: Search Behavior and Solutions.*
2014-04 Hanna Jochmann-Mannak (UT), *Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation.*
2014-05 Jurriaan van Reijsen (UU), *Knowledge Perspectives on Advancing Dynamic Capability.*
2014-06 Damian Tamburri (VU), *Supporting Networked Software Development.*
2014-07 Arya Adriansyah (TUE), *Aligning Observed and Modeled Behavior.*
2014-08 Samur Araujo (TUD), *Data Integration over Distributed and Heterogeneous Data Endpoints.*

2014-09 Philip Jackson (UvT), *Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language.*
2014-10 Ivan Salvador Razo Zapata (VU), *Service Value Networks.*
2014-11 Janneke van der Zwaan (TUD), *An Empathic Virtual Buddy for Social Support.*
2014-12 Willem van Willigen (VU), *Look Ma, No Hands: Aspects of Autonomous Vehicle Control.*
2014-13 Arlette van Wissen (VU), *Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains.*
2014-14 Yangyang Shi (TUD), *Language Models With Meta-information.*
2014-15 Natalya Mogles (VU), *Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare.*
2014-16 Krystyna Milian (VU), *Supporting trial recruitment and design by automatically interpreting eligibility criteria.*
2014-17 Kathrin Dentler (VU), *Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability.*
2014-18 Mattijs Ghijsen (VU), *Methods and Models for the Design and Study of Dynamic Agent Organizations.*
2014-19 Vincius Ramos (TUE), *Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support.*
2014-20 Mena Habib (UT), *Named Entity Extraction and Disambiguation for Informal Text: The Missing Link.*
2014-21 Kassidy Clark (TUD), *Negotiation and Monitoring in Open Environments.*
2014-22 Marieke Peeters (UU), *Personalized Educational Games - Developing agent-supported scenario-based training.*
2014-23 Eleftherios Sidirourgos (UvA/CWI), *Space Efficient Indexes for the Big Data Era.*
2014-24 Davide Ceolin (VU), *Trusting Semi-structured Web Data.*
2014-25 Martijn Lappenschaar (RUN), *New network models for the analysis of disease interaction.*
2014-26 Tim Baarslag (TUD), *What to Bid and When to Stop.*
2014-27 Rui Jorge Almeida (EUR), *Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty.*
2014-28 Anna Chmielowiec (VU), *Decentralized k-Clique Matching.*
2014-29 Jaap Kabbedijk (UU), *Variability in Multi-Tenant Enterprise Software.*
2014-30 Peter de Cock (UvT), *Anticipating Criminal Behaviour.*
2014-31 Leo van Moergestel (UU), *Agent Technology in Agile Multiparallel Manufac-*

turing and Product Support.
2014-32 Naser Ayat (UvA), *On Entity Resolution in Probabilistic Data.*
2014-33 Tesfa Tegegne (RUN), *Service Discovery in eHealth.*
2014-34 Christina Manteli(VU), *The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems..*
2014-35 Joost van Ooijen (UU), *Cognitive Agents in Virtual Worlds: A Middleware Design Approach.*
2014-36 Joos Buijs (TUE), *Flexible Evolutionary Algorithms for Mining Structured Process Models.*
2014-37 Maral Dadvar (UT), *Experts and Machines United Against Cyberbullying.*
2014-38 Danny Plass-Oude Bos (UT), *Making brain-computer interfaces better: improving usability through post-processing..*
2014-39 Jasmina Maric (UvT), *Web Communities, Immigration, and Social Capital.*
2014-40 Walter Omona (RUN), *A Framework for Knowledge Management Using ICT in Higher Education.*
2014-41 Frederic Hogenboom (EUR), *Automated Detection of Financial Events in News Text.*
2014-42 Carsten Eijckhof (CWI/TUD), *Contextual Multidimensional Relevance Models.*
2014-43 Kevin Vlaanderen (UU), *Supporting Process Improvement using Method Increments.*
2014-44 Paulien Meesters (UvT), *Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden..*
2014-45 Birgit Schmitz (OUN), *Mobile Games for Learning: A Pattern-Based Approach.*
2014-46 Ke Tao (TUD), *Social Web Data Analytics: Relevance, Redundancy, Diversity.*
2014-47 Shangsong Liang (UVA), *Fusion and Diversification in Information Retrieval.*

2015

2015-01 Niels Netten (UvA), *Machine Learning for Relevance of Information in Crisis Response.*
2015-02 Faiza Bukhsh (UvT), *Smart auditing: Innovative Compliance Checking in Customs Controls.*
2015-03 Twan van Laarhoven (RUN), *Machine learning for network data.*
2015-04 Howard Spoelstra (OUN), *Collaborations in Open Learning Environments.*
2015-05 Christoph Bösch(UT), *Cryptographically Enforced Search Pattern Hiding.*

2015-06 Farideh Heidari (TUD), *Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes.*

2015-07 Maria-Hendrike Peetz(UvA), *Time-Aware Online Reputation Analysis.*

2015-08 Jie Jiang (TUD), *Organizational Compliance: An agent-based model for designing and evaluating organizational interactions.*

2015-09 Randy Klaassen(UT), *HCI Perspectives on Behavior Change Support Systems.*

2015-10 Henry Hermans (OUN), *OpenU: design of an integrated system to support life-long learning.*

2015-11 Yongming Luo (TUE), *Designing algorithms for big graph datasets: A study of computing bisimulation and joins.*