공학박사학위논문

대용량 의생물학 링크드 데이터를 위한 그래프 경로 탐색

Scalable Graph Path Discovery for Biomedical Linked Data

2017년 2월

서울대학교 대학원
치의과학과 의료경영과정보학
안진현

대용량 의생물학 링크드 데이터를 위한 그래프 경로 탐색

Scalable Graph Path Discovery for Biomedical Linked Data

지도교수 김홍기

이 논문을 공학박사 학위논문으로 제출함
2016년 10월

서울대학교 대학원
치의과학과 의료경영과정보학
안진현

안진현의 공학박사 학위논문을 인준함
2016년 12월

Abstract

Scalable Graph Path Discovery for Biomedical Linked Data

Jinhyun Ahn
Healthcare Management and Informatics
The Graduate School
Seoul National University

A drug could give rise to an adverse effect when combined with another particular drug. Addressing the underlying causes of the adverse effects is crucial for researchers to develop new drugs and for clinicians to prescribe medicine. Most existing approaches attempt to identify a set of target genes for which drugs are most effective, which provides insufficient information regarding these causes in terms of biological dynamics. Drugs should instead be considered as participants in activating a sequence of pathways that lead to some effects. I believe that the causes can better be understood by such linked pathways. Therefore, the purpose of this thesis is to develop algorithms and tools that can be used to discover a sequence of pathways that is activated by a particular drug combination. Furthermore, these algorithms are required to be scalable to manage massive biomedical Linked Data because up-to-date results of biomedical research are increasingly available in Linked Data.

My hypothesis is that for a drug combination, when a drug up-regulates particular pathways in one direction and another drug down-regulates the same pathways in an opposite direction, adverse effects may occur by the drug combination. In this regard, the problem of revealing the causes of adverse effects of drug combinations is cast into the problem of discovering paths of a sequence of linked pathways that begins and ends at the genes that the given drugs target. Therefore, the scalable graph path discovery and matching algorithms are devised such that they work with a distributed computing environment. A pathway graph model is defined to integrate diverse biomedical datasets and a visualization tool is implemented to provide biomedical researchers and clinicians with intuitive interfaces for revealing the causes of the adverse effects.

An algorithm for the shortest graph path discovery is proposed. An existing relational database approach is adapted to address the shortest graph path discovery in a distributed computing framework, in particular, Spark. The 2-hop reachability index is exploited to prune non-reachable paths during discovery computation. A vertex re-labeling technique is proposed to reduce the size of the 2-hop reachability index. Experimental results show that the proposed approach can successfully manage a large graph, which previous studies have failed to do.

The discovered shortest graph path can be transformed into a graph path query to find another similar graph path. To achieve this, a MapReduce algorithm for graph path matching, based on multi-way joins, is proposed. A signature encoding technique is devised to prune intermediate data that is not relevant to the given query. Experiments against RDF (Resource Description Framework) datasets show that SPARQL query processing is faster than the state-of-the-art approaches.

To adapt these algorithms into the problem of drug combinations causing adverse effects, a novel pathway graph model is proposed. In particular, a pathway relationship model is described; directed links between pathways are established using protein–protein interactions and up/down regulations between genes. A prototype system based on a visualization framework is implemented and applied to a pathway graph that is built on the basis of several biomedical Linked Data (e.g. Reactome, KEGG, BioGrid, STRING and etc). A list of candidate drug combinations is obtained using the proposed system, which is compared with known drug-drug combinations available in DrugBank.

A scalable graph path discovery solution is proposed in this thesis. Distributed computing frameworks and several index structures are exploited to efficiently handle massive graphs. A pathway graph model is defined and a prototype system for biomedical researchers is implemented to apply the algorithms to the problem of drug combinations causing adverse effects. In future works, the solution will be generalized to address the temporal organization of signaling pathways, thereby enabling the causes of adverse effects of drug combination to be better understood.

Keywords : Graph, Shortest Path Discovery, RDF, Linked Data, MapReduce, Spark

Student Number : 2014-30699

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

Diverse drugs are increasingly available. Combining several drugs in a pill is common these days to treat disease processes. Accordingly, it follows the risk caused by adverse drug combinations. We indicate an adverse drug combination if it does not act on disease processes as desired. For instance, a drug is effective when combined with a drug, while there could exist another drug that is not effective as desired when combined with the drug. Conventional approaches are to manually check if two drugs could yield adverse effects by examining chemical reactions or doing clinical trials. Obviously, these methods consume huge costs and time, furthermore, it could be involved in severe ethical problems. Therefore, computational approaches are required to address these issues.

We hypothesis that two drugs may cause adverse effects if one drug up-regulates a gene while another drug down-regulates the same gene. This way could make drug's intended effects canceled out. Relationships between biological entities need to be examined. Graph is a suitable data structure to model relationships between biological entities. Graph has successfully been applied to diverse domains such as social network, traffic modeling, natural language processing and etc.

In this thesis, we address adverse drug combination identification problem in terms of graph path discovery problem. Diverse biomedical datasets, such as drug, gene, protein and diseases, are integrated to build a graph. They are mapped via gene. Especially, drug-gene associations acquired from DrugBank are considered as drug's effects on genes. These are classified into positive and negative effects according to targeting types. Directed links are established between drug and gene based on positive and negative effects. These genes are linked to some pathways in which these genes participate. Links between pathways are established using our pathway mapping model.

In this way, we can define a path between two drugs linked via pathways. Specifically, we are interested in finding paths between drugs as they meet in a intermediate pathway that would cancel out the intended effects of two drugs. Furthermore, we assume that closer two pathways are, more influence they have each other. In the context of graph, this corresponds to shortest paths, a kind of graph path.

Existing shortest path discovery algorithms are not scalable as they work on a single machine. Volume of biomedical datasets keeps growing. To deal with massive graph, it is required to devise an algorithm working on a distributed environment. In this thesis, we propose Spark based shortest path discovery algorithm. The algorithm is optimized by exploiting reachability index. Furthermore, we propose a technique to reduce the size of reachability index.

Discovered shortest path can be viewed as a simple graph path. By substituting constants in a path with variables, a simple graph path query can be formed. Once we obtain a graph path query, another drug combination can be identified by matching with the graph. One of basic operations to process graph path queries is join operation. In terms of building join trees, there exist various join policies such as sequential, bush tree and multi-way. These approaches have its pros. and cons. We focus on overcoming the limitations of multi-way join approach. So far, multi-way join can hardly be applied in a distributed computing environment as it requires to exchange many redundant data. If we were able to reduce the redundant data, multi-way join could be one of suitable join strategies to be applied in a distributed computing environment. In this thesis, the signature encoding technique is proposed to prune data, thereby, dramatically reducing redundant data.

Existing adverse drug combination identification systems are limited in that they just report adverse drug combinations in terms of gene sets. During a medical treatment, clinicians might be interested in knowing pathways (instead of genes) that are relevant to a drug combination they try to prescribe. Moreover, few work is interested in showing drug-drug interactions in terms of linked pathways to researchers and clinicians.

In this regards, we propose an adverse drug combination identification system that is based on the algorithms mentioned above. The system is implemented as a plugin on top of Cytoscape, which is a popular visualization toolkit. The system is designed to interact with a cluster where Hadoop is installed. Computationally expensive tasks are carried out in a cluster. Results are shown in network display in Cytoscape. Not only biomedical researchers can identify adverse drug combinations, but also they are allowed to see clues represented in paths between pathways.

1.2 Contributions

1.2.1 Shortest Graph Path Discovery based on Reachability Index

Recently, RDB-based shortest path discovery algorithm has been proposed [3]. A graph data is stored in RDB tables. A shortest path is discovered by applying SQL queries in an iterative fashion. In this way, graph is not required to be loaded into memory. It means that the approach is able to deal with large graphs that cannot be loaded completely into memory. However, even if large graphs are loaded into tables, the performance of processing SQL queries could be in-efficient. In this regards, it still remains challenge to devise an approach that can deal with massive graph efficiently.

We devise a Spark based shortest path discovery algorithm. A graph data is loaded into RDD (Resilient Distributed Dataset) which is a distributed data structure. Spark operations are applied to the RDD to obtain a new RDD that records the shortest path. Experimental results showed that the proposed approach could successfully deal with large graphs while previous works failed. In this shortest path discovery framework, we observed that many intermediate paths are visited, which are not actually needed to visit. 2-hop reachability index is exploited to prune these intermediate paths. Experimental results show that using reachability index is efficient against dense graphs and tasks for searching long paths. To reduce the size of 2-hop reachability index, a vertex id assignment technique is proposed. Experimental results showed that the proposed approach can better reduce the index size compared to random assignment.

1.2.2 Graph Path Matching based on Signature Encoding

A chain graph path can be formed from shortest paths by substituting constants with variables. Recall that the shortest path conveys information how two drugs are related via proteins. Similar paths may exist in which some nodes are different with the original one. If drugs different from the original drugs in the shortest path are found in the new matched paths, we may expect that these two drugs have an adverse effect like the original drugs. In this regards, we address graph path matching problem.

We devise an efficient multi-way join algorithm. Join operation is one of important operations to process graph path matching queries. Existing join policies such as sequential [4], bush tree [5, 6] and multi-way [7] have its pros. and cons. Among them, we focus on multi-way join. Advantages of multi-way join is that it processes complex join in a single task while another join policies requires multiple tasks. Disadvantages of multi-way join is that it replicates target data to achieve such a single task processing. So far, multi-way join can hardly be applied in a distributed computing environment due to the second issue. If we were able to reduce the redundant data, multi-way join could be one of suitable join strategies to be applied in a distributed computing environment. In this thesis, the signature encoding technique is proposed to prune data, thereby, dramatically reducing redundant data. Experiments against RDF datasets showed that the proposed approach is faster than state-of-the-arts approach in terms of SPARQL query processing.

1.2.3 Application to Biomedical Linked Data

Existing adverse drug combination identification systems are limited in that they just list adverse drug combinations with confidence scores [8, 9, 10]. The reason why such adverse effects occur is more important than confidence score. In our framework, the reason can be better captured by shortest path between two drugs in a our novel pathway graph model.

Biomedical datasets relevant to adverse drug combination discovery are collected such as Drug-Gene (DrugBank), Protein-Protein Interaction (BioGrid), Pathway (Reactome, KEGG) and etc. These datasets are mapped using genes Drugbank contains drug-drug interactions that have been discovered in literatures. In our graph model, directed edges are established using our pathway relationship model; directed links between pathways are established using protein–protein interactions and up/down regulations between genes. A prototype system based on a visualization framework, Cytoscape, is implemented and applied to a pathway graph. Cytoscape is a popular network visualization tool. A list of candidate drug combinations with shortest paths is obtained using the proposed system, which is compared with known drug-drug combinations available in DrugBank. We hope that causes of the adverse effects can be better explained by these paths.

1.3 Thesis Organization

This paper is organized as follows. Chapter 2 explains preliminaries and related work. Chapter 3 describes scalable shortest path discovery algorithm based on reachability index. An optimization technique for reducing

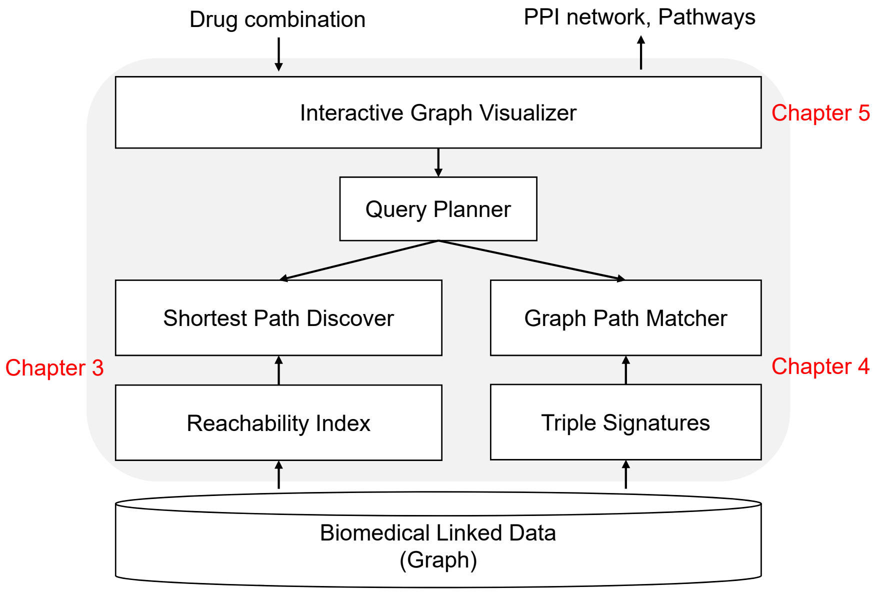


Figure 1: System architecture

reachability index size is then explained. Discovered shortest path can transformed into a chain graph path query. Chapter 4, in turn, describes graph path matching algorithm based on multi-way join. Signature encoding technique is proposed to reduce data redundancy which is one of limitations of multi-way join. These algorithms are integrated into a visualization toolkit in order to apply biomedical research, which is demonstrated in Chapter 5. The thesis is concluded in Chapter 6 with future works.

Chapter 2

Preliminaries and Related Work

2.1 Graph

Graph is a data structure to model a set of entities and their relationships [11]. Due to its intuitive structure, graph has been applied to diverse fields such as social network modeling, biological pathways, traffic modeling and etc. Formally, a graph G consists of a set V of vertices (or nodes) and a set E of edges. An edge $e = (v_i, v_j)$ is a pair with two vertices v_i and v_j. We can say v_i is linked to v_j via e. In a directed graph, edge is an ordered pair where the first vertex is designated as source and the second vertex as target. In an undirected graph, edge is a set with no order. See Figure 2, graphs are depicted where a circle represents a vertex, an arrow represents a directed edge and a line represents an undirected edge.

In particular, this thesis deals with a directed graph where edges have label and weight. Formally, edge-labeled weighted directed graph is $G(V, E, EL, EW)$ consists of a set V of vertices, a set E of ordered pairs of vertices, a map EL from E to a set L of edge-labels, and a map EW from E to a set $\Re$ of real numbers. In this thesis, EL and EW are omitted if the context is clear.

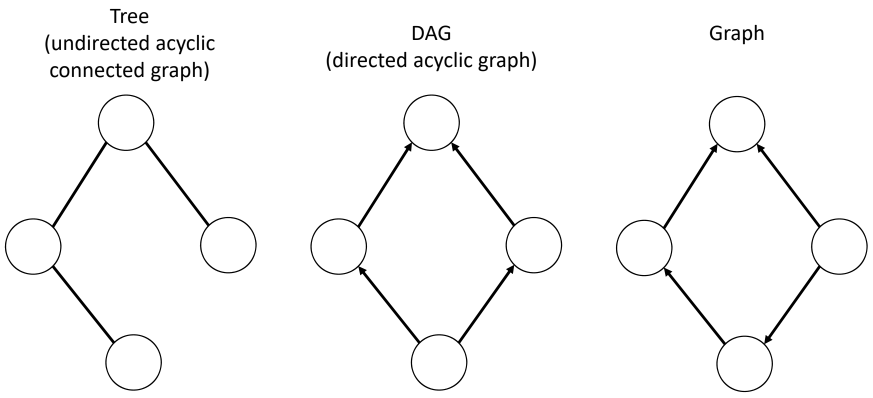


Figure 2: Graph

2.2 Graph Path

Graph path is an alternating sequence of vertices $P = v_0, v_1, ..., v_k$ where $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$. Formally, a graph G is acyclic if G has no graph path P such that $v_0 = v_k$. Shortest (graph) path between two vertices s and t in a graph G refers to the graph path $P = \{v_0 = s, v_1, v_2, ..., v_k = t\}$ whose weight sum $W = \sum_{i=0}^{k-1} w(e_{i,i+1})$ is the smallest where $w(e_{i,i+1})$ is the weight of an edge $e_{i,i+1}$ whose starting vertex is v_i and ending vertex is v_{i+1}. If the weight of every edge in G is equal to 1, shortest path is the path with fewest edges.

Dijkstra's algorithm is a well-known solution for shortest path discovery. An improved technique is to navigate both directions from source and target, which is called bi-directional strategy [12]. Recently, RDB (Relational database) based shortest path discovery algorithms is proposed [3]. It does not need to load the whole graph into memory. However, the scalability issues still remains challenges as RDB based algorithms work on a single machine.

2.3 Acyclic Transformation

DAG (Directed Acyclic Graph) is a directed graph with no cycle. Tree is a undirected graph with no cycle. A graph can be transformed into a DAG by dealing with cycles [13]. Acyclic transformation from G is to create a directed acyclic graph $DAG(V', E')$ by condensing every strongly connected components $SCC_i = \{v_i\}$ into a single vertex v'_{scc_i} and maintaining every original edges; if $(v_f, v_i) \in E$ then $(v_f, v'_{scc_i}) \in E'$ and if $(v_i, v_t) \in E$ then $(v'_{scc_i}, v_t) \in E'$.

2.4 Reachability

For a directed graph G and two vertices s and t, we may be interested in knowing if there exists a path starting at s and ending at t. For example, suppose a graph where vertex corresponds to cities and edges corresponds to roads between cites. Graph reachability is analogue to the existence of routes between two cites. Graph reachability can only be defined over DAGs. Reachability over a cyclic graph is meaningless because every vertex can be reached by every vertex. Formally, reachability relation R for a DAG $G = (V, E)$ is a set T of vertex pairs (s, t) where $s \in V$ and $t \in V$. For a pair (s, t), there exists a sequence $v_0 = s, v_1, v_2, ..., v_k = t$ such that $(v_i, v_{i+1}) \in E$ where $1 \leq i < k$. For a given two vertices s and t, if (s, t) is in R, we indicate $s \rightsquigarrow t$ which implies t is reachable from s and $s \not\rightsquigarrow t$ otherwise.

2.5 Distributed Computing Frameworks

MapReduce, a programming model in a distributed environment, consists of a map operation and a reduce operation [14]. The mapper distributes data to each machine by using a predefined key; the reducer applies some operations to the data collected in each machine. Before the map operation, the input file is split by the InputSplit class object and then allocated to each mapper. The allocated file is called InputSplit. MapReduce has been applied to machine learning [15], massive matrix computation [16], gnomic analysis [17] and etc.

Spark is a distributed computing framework on top of Hadoop eco system. RDD (Resilient Distributed Dataset) is designed to support various operations [18]. Spark differs with MapReduce in that Spark does not flush intermediate data into HDFS (Hadoop Distributed File System) if not needed while MapReduce stores intermediate data once a MapReduce job finishes. In other words, a sequence of operations can be applied to RDD without accessing disks. To achieve this, RDD is designed to be immutable.

2.6 RDF & SPARQL

Definition 1. (RDF graph) An RDF triple is a 3-tuple

$$(s,p,o) \in (U \cup B) \times (U) \times (U \cup B \cup L)$$

, where U is a set of URIs, B is a set of blank nodes, and L is a set of literals. s is called as subject, p as predicate or property, and o as object. A set of

RDF triples T is said to be a RDF graph.

An example of an RDF triple from DBPedia[1] is:

(<http://dbpedia.org/resource/Alan_Turing>,

<http://dbpedia.org/property/birthDate>,

"1912-06-23")

, which tells us that Alan Turning was born on 1912-06-23. An abbreviated form is also possible such as (`dbpedia:Alan_Turing, dbpprop:birthDate`, "1912-06-23"), where `http://dbpedia.org/resource` is replaced with `dbpedia` and `http://dbpedia.org/property` with `dbpprop`. The replaced string is called the namespace. SPARQL is a query language for RDF and its syntax is very similar to SQL in the domain of relational databases. A query for enumerating people with the same birth place as Turing can be represented in SPARQL as follows.

```
SELECT ?person
WHERE {
  dbpedia:Alan_Turing  dbpprop:birthPlace  ?birth_place .
  ?person  dbpprop:birthPlace  ?birth_place .
}
```

In the WHERE clause, there are two triple patterns, which are type of triples that may contain query variables. A set of triple patterns is called a Basic

[1] http://dbpedia.org

Graph Pattern (BGP). Note that `?birth_place` appears in both triple patterns, which means that `?birth_place` is a joining variable.

2.7 SPARQL Processing Engines

We review previous research into the area of SPARQL query processing for RDF data. These studies can be classified into two broad approaches, which are based on a single machine or clusters.

Single machine-based approaches Numerous techniques have been proposed for the storage of RDF data in relational databases using a single machine. For example, Jena [19] proposed the property table approach, which reduces the number of joins by allowing multiple triple patterns in a single property table. However, this method requires the identification of subjects that have the same set of properties. The vertical partitioning approach [20] is an alternative solution that partitions the triple table in a vertical manner according to the predicates of triples. In addition, much previous research has focused on the area of indexing techniques for RDF data. Hexastore [21] uses a main memory index that covers all possible patterns, such as `SPO`, `SOP`, `PSO`, `POS`, `OPS`, and `OSP`, where `S` stands for subject, `P` for predicate, and `O` for object. RDF-3X [22] uses clustered B+tree indices for all permutations of `S-P-O` triples. BitMat [23] is a bit-matrix index for RDF triples. Similarly, gStore [24] encodes RDF triples into bits strings and then places them in a tree structure and query processing is performed by bit-wise operations. However, because single machine-based systems are dependent on the main memory, they are not scalable as the volume of RDF data increases.

Cloud-based approaches Recently, cloud and distributed architectures have attracted much attention in the areas of SPARQL query processing. Cluster-based approaches include key-value stores and distributed file systems [25].

Methods dependent on key-value stores can be categorized into triple-based and graph-oriented approaches [25]. The first type treats RDF data as a set of triples and it places each triple permutation (i.e., `SPO`, `POS`, and `OSP`) in existing key-value stores, e.g., Rya [26] and $\mathsf{H_2RDF}$ [27] belong to this type. The second type deals with RDF data from a graph perspective. Trinity.RDF [28] is a graph-based RDF key-value store that splits RDF graph data into disjoint parts with several clusters. Other studies have focused more on ways in which a query is federated over back-end stores, e.g., N-hop replication [29], query workload replication [30, 31], and heterogeneous storage combinations [32]. However, although previous methods for distributed environments were designed for large-scale RDF data, these are limited because they are dependent on existing key-value stores such as data storage structures and join evaluation.

Several distributed file system based approaches employ the MapReduce framework [33]. SHARD [4] stores triples where the subject is the same as that in the distributed file. HadoopRDF [5] also stores RDF data in a vertical partitioning scheme, which partitions RDF data into smaller volumes according to their property values. Entity-based modeling is a different approach used in EAGRE [34], which focuses on the range queries and order constraints using the MapReduce framework. In particular, because MapReduce-based approaches focus on decreasing the number of jobs (i.e., the MapReduce jobs) that pass data through the disk accesses, join evaluation has become one of the main concerns in SPARQL query processing. SHARD [4] uses the sequential query plan whereas HadoopRDF [5] executes SPARQL queries using the bush query plan, which is suitable for parallel processing. However, the bush query plan still requires several Hadoop jobs with intermediate results exchanged through the disk accesses. We will discuss their limitations in details in the following section.

Chapter 3

Shortest Graph Path Discovery based on Reachability Index

3.1 Introduction

In this Chapter, a shortest path discovery algorithm based on distributed computing environment is described. Our goal is to overcome limitations of existing works. First, existing approaches are inefficient to deal with massive graph because these are designed to work on a single machine. Second, during computation, existing approaches visit vertices that are not reachable to destination vertex.

Our solution is to make use of distributed computing environment. Specifically, Spark is utilized here, which is one of popular distributed computing framework on top of Hadoop eco-system. Spark is more applicable than MapReduce. In MapReduce, disk I/O occurs between tasks which would slow down the discovery process. In Spark, no disk I/O is required between tasks, rather it tries to fully use memory. Shortest path discovery in our context is needed to be solved in real-time. One may argue that the shortest paths for all pair of drugs can be discovered in off-line in advance. This is infeasible because there exist too many drug combinations. As of Jan. 2017, DrugBank contains 8,257 drugs. If we assume that it takes one second to discover the shortest path for an ordered pair of drugs, it would take more than 2 years totally.

Existing approaches are inefficient for the case when some vertex have many outgoing edges. We cannot avoid expanding most of outgoing edges from a vertex. If we know that some outgoing edges does not have to be expanded, we can save much time. This can be achieved by utilizing reachability index, which conceptually contains a set of 3-tuple consisting of (source vertex, target vertex, reachable or not reachable). Before expanding an outgoing edge, we may access the reachability index. If it says no, we don't have to expand the edge, thereby, reducing the number of expansions.

However, the proposed approach could be inefficient if the size of reachability index is huge. To resolve the issue, we propose an optimization technique to reduce reachability index size. Graph statistics is exploited to assign ids to each vertex, which is then used to label each vertex. Section 3.2 demonstrates an optimization technique to reduce reachability index size. After then, Section 3.3 demonstrates Spark based shortest path discovery algorithm that utilizes the reachability index.

3.2 Space Reduction of Reachability Index

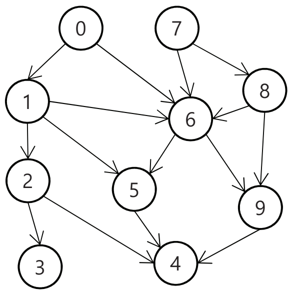
3.2.1 Introduction

Graph databases are increasingly used for RDF (Resource Description Framework) data management. Mining reachability relationships between resources is one of important building blocks for graph databases. Reachability relationships in a graph and its corresponding DAG (Directed Acyclic

Graph) are equivalent when we focus on reachability alone, which allows to focus on DAGs in this thesis. Diverse labeling schemes have been proposed to efficiently determine the reachability of DAGs. We focus on a state-of-the-art 2-hop labeling scheme that is based on a permutation of vertices to achieve a linear index size and reduce on-line searches that are required when the reachability cannot be answered by 2-hop labels only [35]. We observed that the approach can be improved to guarantee the minimized index size. Therefore, a way of reducing the 2-hop label size is proposed in this section with experimental results on real-world DAG datasets.

A 2-hop labeling scheme of a DAG is to label each vertex v with a pair $(L_{out}(v), L_{in}(v))$, where $L_{out}(v)$ is a set of vertices that v can reach and $L_{in}(v)$ is a set of vertices reachable from v [35]. In this thesis, we focus on a state-of-the-art variation of 2-hop labeling [36], where a permutation of vertices is used to allow $L_{out}(v)$ and $L_{in}(v)$ to keep at most k reachable vertices, which probabilistically guarantees reduction in on-line Depth-First-Search(DFS), a condition required when the reachability cannot be answered by these labels only. We briefly review the labeling scheme as follows.

2-hop Labeling based on Independent Permutation (IP) Let $G(V,E)$ be a DAG with V as a set of vertices and E as a set of edges pairing two vertices. $u \rightsquigarrow v$ is used to denote that u can reach v, and $u \not\rightsquigarrow v$, otherwise. A permutation of V is a bijection denoted as $\sigma : V \longrightarrow V$. Given G and a positive integer k, IP randomly generates σ and then outputs the 2-hop



v	$\sigma_1(v)$	$L_{out}(v)$	$L_{in}(v)$
0	7	0,2,3	7
1	4	0,2,3	4,7
2	9	3,8,9	4,7,9
3	3	3	3,4,7
4	8	8	0,1,2
5	0	0,8	0,1,4
6	5	0,2,5	1,4,5
7	6	0,1,2	6
8	1	0,1,2	1,6
9	2	2,8	1,2,4

$size\left(I^3_{\sigma_1}(V)\right) = 162$

v	$\sigma_2(v)$	$L_{out}(v)$	$L_{in}(v)$
0	0	0,1,2	0
1	7	1,2,3	0,7
2	1	1,2,8	0,1,7
3	8	8	0,1,7
4	2	2	0,1,2
5	6	2,6	0,3,5
6	3	2,3,4	0,3,5
7	5	2,3,4	5
8	9	2,3,4	5,9
9	4	2,4	0,3,4

$size\left(I^3_{\sigma_2}(V)\right) = 139$

Figure 3: The 2-hop index size of the same graph varies according to σ_1 and σ_2 when $k = 3$.

index $I^k_\sigma : V \longrightarrow H$, where H is a set of pairs $(L_{out}(v), L_{in}(v))$ defined as:

$$L_{out}(v) = min_k\{\sigma(u) | v \rightsquigarrow u\}$$

$$L_{in}(v) = min_k\{\sigma(u) | u \rightsquigarrow v\}$$

, where $min_k\{A\}$ is a sub-set of A, such that $A_i < A_j$ if $i < j$ and $|min_k\{A\}| \leq k$. In other words, $min_k\{A\}$ contains the k smallest integers in A.

In the original version of IP, σ is randomly generated using the Knuth shuffle algorithm. We argue that the randomized way of generating a permutation is unable to ensure that the minimized size of index is obtained. Therefore, we define the problem that generates a *space-efficient* permutation defined in Definition 2.

Definition 2. 2-Hop Size Minimization Given $G(V,E)$ and a positive integer k, output σ such that

$$\underset{\sigma}{\operatorname{argmin}}\ size(I_{\sigma}^{k}(V)) := \frac{1}{|V|}\sum_{v \in V} space(L_{out}(v)) + space(L_{in}(v))$$

, where $space(L)$ is the space requirements for a set L of integers, which can be defined according to the application requirements.

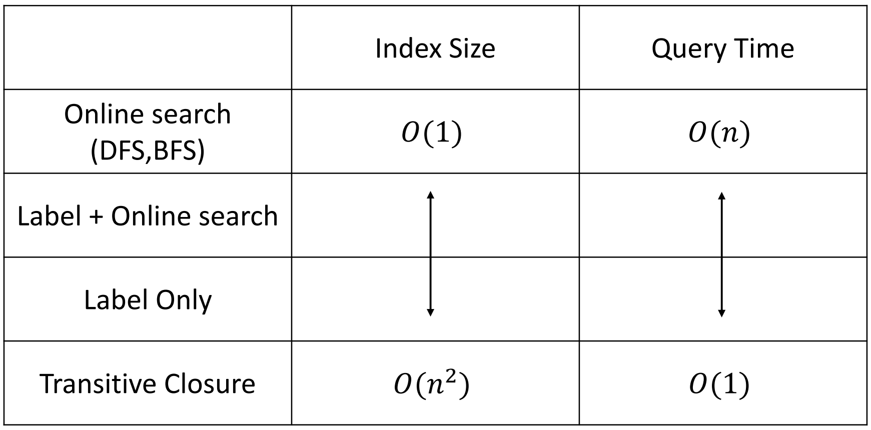
An illustrative example that shows the effect of σ on the index size is depicted in Figure 3. In order to see the index size difference for the very small DAG, we define space requirements as follows:

$$space(L) = \sum_{w \in L} w \tag{3.1}$$

We have a research question; Is it possible to deterministically choose a permutation σ of V that best minimizes the index size? Because the number of candidate permutations is $n!$, a brute-force algorithm is not feasible. One may choose a permutation by sorting V using a topological sort. Smaller numbers are assigned to vertices having more reachable vertices. It can be expected that this method allows $L_{in}(v)$ to have smaller numbers. However, conversely, $L_{out}(v)$ would have large numbers, offsetting smaller size of $L_{in}(v)$. A more sophisticated technique is required.

3.2.2 Related Work

Reachability relationships in a graph and its corresponding DAG are equivalent, since a graph can be transformed into a DAG by condensing every strongly connected component(SCC) into a single vertex and retaining



	Index Size	Query Time
Online search (DFS,BFS)	$O(1)$	$O(n)$
Label + Online search	↑	↑
Label Only	↓	↓
Transitive Closure	$O(n^2)$	$O(1)$

Figure 4: Trade-off in graph reachability processing

edges between vertices in SCC and the other vertices that are connected to SCC [13]. In terms of reachability relationships, Linked Open Data[1] (LOD) can be viewed as a real-world DAG dataset. A vast amount of knowledge is available in LOD, including social networks, biological networks, traffic networks, and software version management. Protein-protein interaction networks, for example, can be represented in DAGs. One may be interested in mining interactions (i.e., reachability relationships) between two proteins (i.e., vertices) [37]. Regarding RDF triple stores, reachability relationship identification helps process SPARQL queries efficiently [38].

There are extensive studies on labeling of DAGs for reachability queries processing [39]. The straightforward way is to pre-compute the edge transitive closures(TC) [40]. Even if this method provides an answer to a reachability query in $O(1)$ time, the index could be huge for even small dense graphs. On-line traversal, such as DFS and Bread-First Search(BFS), do not

[1]http://linkeddata.org

require any index but lead to slow query processing. See Figure 4, various approaches have been made to address trade-offs between the index size and the speed of query processing. Some authors have adapted the prime number labeling scheme to DAGs [41]. However, prime numbers quickly grow to large values. Tree Cover [42] labels a vertex v with a compressed TC of the subtree rooted at v. [43] present GRIPP(GRaph Indexing based on Pre- and Postorder numbering) that utilizes spanning trees based on an interval label scheme. GRAIL [44] is based on randomized multiple interval labeling. FERRARI [45] labels each vertex with a mixture of exact and approximate reachability intervals, where subsets of intervals are merged into approximate intervals. 2-hop labeling [35] labels each vertex v with a pair $(L_{out}(v), L_{in}(v))$.

Recently, a state-of-the-art variation of 2-hop labeling has been proposed in [36] that reports outstanding performance compared to existing approaches. A permutation of vertices is first generated randomly. MinHash is adapted to construct 2-hop labels; at most k reachable vertices are only maintained in 2-hop labels. For pairs that cannot be determined by k reachable vertices only, an on-line search is performed. Some heuristics are also presented to minimize the case that requires exhaustive on-line DFS searches.

An space optimization technique for tree labeling is proposed [1, 2]. Figure 5 depicts a label assignment reordering technique for unordered XML data. The authors observed that Asia node has fewer children than Europe node does. If Europe node has smaller label than Asia node, we can save the space requirements overall. For example, 10 is part of Asia node's label and

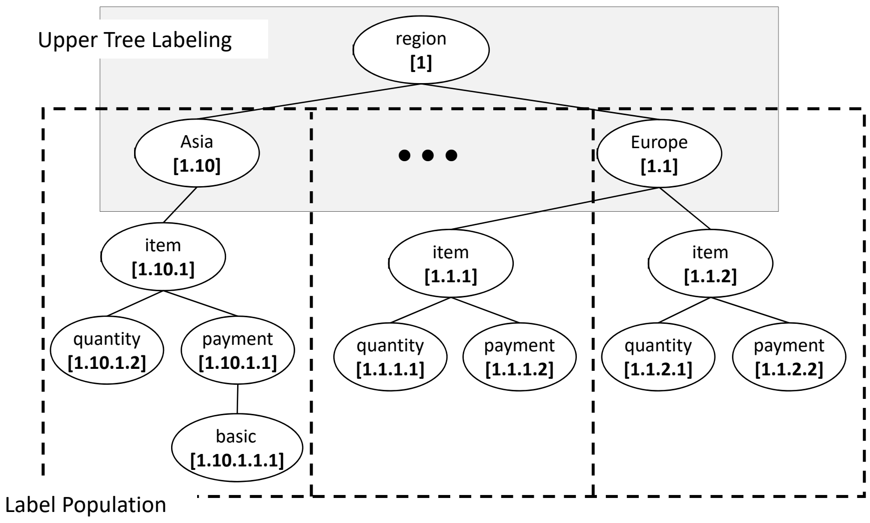


Figure 5: A prefix-based labeling of tree[1, 2]

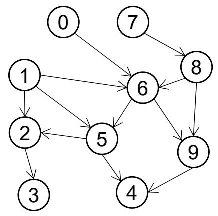
1 is part of Europe node's label. According to the prefix-based labeling strategy, 10 is used four times by Asia node's children and 1 is used six times by Europe node's children. If labels are assigned reversely, 10 will be used six times while 1 four times, which occupies more space than the proposed one. The approach motivates us to reduce reachability index size of DAGs.

3.2.3 The Proposed Approach

Regarding the research question mentioned in Section 3.2.1, our hypothesis is stated as follows.

- If smaller numbers are assigned to $\sigma(v)$ because v has more edges, then the 2-hop index size is reduced.

To prove the hypothesis, we first compute the degree of each of the vertices in V and then sort V in decreasing order by degree. The order is



v	degree	$\sigma(v)$
0	1	7
1	3	2
2	3	3
3	1	8
4	3	6
5	4	1
6	6	0
7	1	9
8	3	4
9	3	5

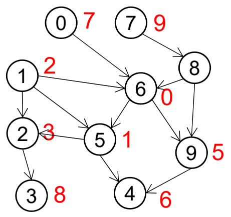


Figure 6: Permutation of vertices by degrees

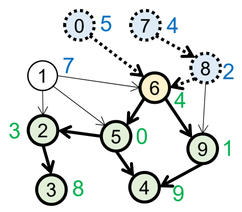
regarded as the desired permutation $\sigma(V)$ as follows.

$$\sigma(v) = i$$

, where v is the i-th element in V^{degree} such that V^{degree} is a sorted set of V, sorted by decreasing degree of v.

In other words, the more degrees v has, the smaller the number assigned to $\sigma(v)$. See Figure 6, for example, vertex 6 is assigned the smallest number ($\sigma(6) = 0$) because its degree is the largest among vertices. This is derived by an expectation that if v has many edges, v is likely to become a reachable vertex from another vertex. The overall 2-hop index size could be reduced by making $\sigma(v)$ smaller. See Figure 7, examples for each approach are shown; ids are assigned randomly, ids are not changed, and ids are assigned by the proposed approach, respectively. Note that the current approach does not take into account the parameter k. We plan to introduce the parameter k to guarantee a reduced index size for arbitrary k.

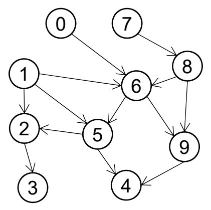
Random



v	$\sigma_1(v)$	$L_{out}(v)$	$L_{in}(v)$
0	5	0 1 3	5
1	7	0 1 3	7
2	3	3 8	0 2 3
3	8	8	0 2 3
4	9	9	0 1 2
5	0	0 3 8	0 2 4
6	4	0 1 3	2 4 5
7	6	0 1 2	6
8	2	0 1 2	2 6
9	1	1 9	1 2 4

131

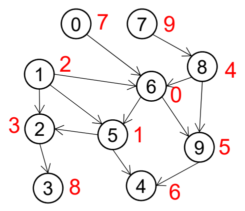
Identity



v	$\sigma_2(v)$	$L_{out}(v)$	$L_{in}(v)$
0	0	0 2 3	0
1	1	1 2 3	1
2	2	2 3	0 1 2
3	3	3	0 1 2
4	4	4	0 1 4
5	5	2 3 4	0 1 5
6	6	2 3 4	0 1 6
7	7	2 3 4	7
8	8	2 3 4	7 8
9	9	4 9	0 1 6

126

OUR



v	$\sigma_3(v)$	$L_{out}(v)$	$L_{in}(v)$
0	7	0 1 3	7
1	2	0 1 2	2
2	3	3 8	0 1 2
3	8	8	0 1 2
4	6	6	0 1 2
5	1	1 3 6	0 1 2
6	0	0 1 3	0 1 4
7	9	0 1 3	9
8	4	0 1 3	4 9
9	5	5 6	0 2 4

120

Figure 7: Diverse vertex permutations and the resultant index size

3.2.4 Theoretical Analysis

In this section, the relationships between degrees and the size of 2-hop index is theoretically analyzed. First, we propose a model for the size of 2-hop label.

3.2.4.1 2-Hop Index Size

We define the reachability matrix in Definition 3.

Definition 3. Reachability Matrix The reachability matrix R for a DAG $G(V,E)$ represents reachability relationships between two nodes.

$$R(i,j) = \begin{cases} 1 & \text{if } v_i \rightsquigarrow v_j \\ 0 & \text{if } v_i \not\rightsquigarrow v_j \end{cases}$$

, where $1 \leq i \leq |V|$ and $1 \leq j \leq |V|$

Definition 4. Permutation Vector For a DAG $G(V,E)$ and a permutation σ of V, the permutation vector P is defined as follows:

$$P = (\sigma(v_1), \sigma(v_2), ..., \sigma(v_n)) \tag{3.2}$$

, where $n = |V|$.

Definition 5. 2-Hop Label Size Let $L(G,P)$ be the simple label size for a DAG G and a permutation vector P, which is defined as follows:

$$L(G,P) = (R \times P^T)^T \times I + (R^T \times P^T)^T \times I \tag{3.3}$$

, where R is the reachability matrix and I is a n-dimensional column vector of 1s.

The first factor of Definition 5 represents the size of L_{out} and the second factor represents the size of L_{in}. Our goal is to minimize $L(G,P)$ with respect to P stated as follows.

$$\underset{P}{\operatorname{argmin}}\, L(G,P) \tag{3.4}$$

We expand the equation as follows.

$$\operatorname*{argmin}_P \begin{array}{l} \begin{bmatrix} \sigma(v_1)r(1,1)+...+\sigma(v_n)r(1,n) \\ \vdots \\ \sigma(v_1)r(n,1)+...+\sigma(v_n)r(n,n) \end{bmatrix}^T \times \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \\ + \begin{bmatrix} \sigma(v_1)r(1,1)+...+\sigma(v_n)r(n,1) \\ \vdots \\ \sigma(v_1)r(1,n)+...+\sigma(v_n)r(n,n) \end{bmatrix}^T \times \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \end{array}$$

The expression is reduced as follows.

$$\operatorname*{argmin}_P \sigma(v_1)(r(*,1)+r(1,*))+...+\sigma(v_n)(r(*,n)+r(n,*)) \tag{3.5}$$

, where $r(*,1)=\sum_{i=1}^{n} r(i,1)$ and $r(1,*)=\sum_{i=1}^{n} r(1,i)$.

Note that $r(*,i)$ is the number of vertices reachable to v_i such as $r(*,i) = |\{v_w : v_w \rightsquigarrow v_i\}|$ and similarly we have $r(i,*) = |\{v_w : v_i \rightsquigarrow v_w\}|$. In this regards, $L(G,P)$ can be reduced by making $\sigma(v_i)$ to be inverse-proportional to the number of reachable vertices, as stated in Lemma 1.

Lemma 1. The label size is minimized if we determine a permutation vector P as follows.

$$\sigma(v_i) \propto \frac{1}{r(*,i)+r(i,*)} \tag{3.6}$$

The expression suggests that smaller labels be assigned to vertices with large number of reachable vertices (in both directions).

3.2.4.2 Degrees and The number of reachable vertices

We showed that the minimum label size can be obtained as suggested by Lemma 1. However, in this Chapter, we have proposed an approach based on degrees. It is because that calculating the number of reachable vertices takes some time. Rather, we exploited degrees which can be counted easily. Now, we will show that degrees are proportional to the number of reachable vertices, which means that degrees can be used to reduce the label size. We first define l-length reachability.

Definition 6. ***l*-length reachability** The l-length reachability from v_i to v_j, denoted as $v_i \rightsquigarrow_l v_j$, is *true* if there is a path P from v_i to v_j such that P is composed of $l+1$ vertices. If l is 1, then v_i is adjacent to v_j.

We generalize the reachability matrix in Definition 3 to l-length reachability matrix as stated in Definition 7.

Definition 7. ***l*-length Reachability Matrix** The l-length reachability matrix R^l for a DAG $G(V,E)$ represents reachability relationships between two nodes as follows.

$$R^l(i,j) = \begin{cases} 1 & \text{if } v_i \rightsquigarrow_l v_j \\ 0 & \text{if } v_i \not\rightsquigarrow_l v_j \end{cases}$$

, where $1 \leq i,j \leq |V|$.

Corollary 3.2.0.1. For a DAG $G(V,E)$, R^1 is the same with the adjacent matrix A defined as follows.

$$A(i,j) = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{if } (v_i, v_j) \notin E \end{cases}$$

Two reachability matrices can be multiplied to obtain another reachability matrix using the reachability matrix multiplication operator, defined in Definition 8.

Definition 8. Reachability Matrix Multiplication Let $\odot$ be the reachability matrix multiplication operator between two reachability matrices R^w and R^w. R^{w+1} is obtained by $R^w \odot R^w$ which is defined as follows.

$$R^w \odot R^w = \begin{cases} 1 & \text{if } RR(i,j) \geq 1 \\ 0 & \text{if } RR(i,j) = 0 \end{cases}$$

, where $1 \leq i,j \leq |V|$ and $RR = R^w \times R^w$.

In terms of binary operators, $\odot$ can be viewed as replacing multiplication($\times$) and addition($+$) operators with AND($\wedge$) and OR($\vee$) operators, respectively. For example, consider $R^2 = R^1 \odot R^1$.

$$R^2 = \begin{bmatrix} R^1(1,1) \wedge R^1(1,1) \vee \ldots \vee R^1(1,n) \wedge R^1(n,1) & \ldots & R^1(1,1) \wedge R^1(1,n) \vee \ldots \vee R^1(1,n) \wedge R^1(n,n) \\ \vdots & \ddots & \vdots \\ R^1(n,1) \wedge R^1(1,1) \vee \ldots \vee R^1(n,n) \wedge R^1(n,1) & \ldots & R^1(n,1) \wedge R^1(1,n) \vee \ldots \vee R^1(n,n) \wedge R^1(n,n) \end{bmatrix} \quad (3.7)$$

Now, consider the n-dimensional column vector R^2_{row} by summing up rows of R^2.

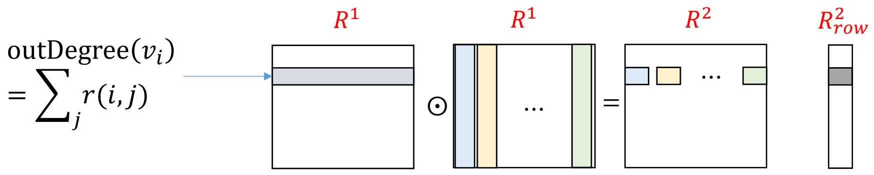


Figure 8: Degree contributes to counting the number of reachable vertices.

$$R^2_{row} = \begin{bmatrix} R^2(1,1) + \ldots + R^2(1,n) \\ \vdots \\ R^2(n,1) + \ldots + R^2(n,n) \end{bmatrix} = \begin{bmatrix} |\{v_j : v_1 \rightsquigarrow_2 v_j\}| \\ \vdots \\ |\{v_j : v_n \rightsquigarrow_2 v_j\}| \end{bmatrix} \tag{3.8}$$

See Figure 8 and Expression 3.7 and 3.8. The summation of values in ith row in R^1 corresponds to the out-degree of ith vertex. In order to obtain ith row in R^2_{row}, ith row in R^1 is taken into account j times where R^1 has j columns. The same process can be applied to $(R^1)^T$ that is related to in-degrees. The simplified observation leads us to state Corollary 3.2.0.2.

Corollary 3.2.0.2. For a vertex v_i, the number of reachable vertices from v_i and the number of vertices that can reach v_i can be measured approximately by out-degree and in-degree, respectively. Specifically, outDegree(v_i) $\propto |\{v_j : v_i \rightsquigarrow v_j\}|$ and inDegree(v_i) $\propto |\{v_j : v_j \rightsquigarrow v_i\}|$

3.2.5 Experimental Results

We implemented the proposed approach based on the source codes provided by the authors of IP[2]. The proposed approach (denoted as IP_adv) is

[2]`http://www1.se.cuhk.edu.hk/~hwei/source/IP.zip`

Table 1: Real-world DAG datasets

Name	Vertices	Edges	Average Degree (max.)	Median Degree
arxiv	6,000	66,707	22.2 (700)	14
go	6,793	13,361	3.9 (71)	3
pubmed	9,000	40,028	8.9 (432)	4
citeseer	340,945	312,282	1.8 (55,758)	1
citpatents	3,774,768	16,518,947	8.8 (793)	6
go-uniprot	6,967,383	34,769,339	10.0 (1,186,280)	4

evaluated compared with the original approach (IP) and baseline (IP_fix). IP generates a random permutation, which means that a different index is constructed for each run. IP_fix is based on the identity permutation such that $\sigma(v) = j$, where v is the j-th element in V. All the experiments were conducted on a machine with a 2.4 GHz CPU and 40 GB of RAM. We downloaded the real-world DAG datasets[3]. Statistics of the datasets are listed in Table 1.

The index sizes for each dataset are plotted in Figure 9. The index size by IP_adv and IP_fix are compared with the index sizes generated by 10 runs of IP. For each run, only IP showed the different index size due to its random nature. IP_adv showed the best performance for all datasets. For small datasets with large degrees such as `arxiv` and `pubmed`, relatively small improvements are observed. For large datasets, we observed relatively large improvements, except for `citeseer` and `yago`. It can be seen in Table 1 that these datasets have relatively small median degrees. Since IP_adv is based on degrees, too small or too large degrees would not significantly con-

[3]https://code.google.com/archive/p/ferrari-index/downloads

tribute to a reduction in the index size. In future works, we plan to utilize more graph metrics to work better against such datasets.

Labeling time is depicted in Figure 10. In most cases, IP_fix is faster than IP_adv. It is a natural result. IP_fix does not has the step for vertex id assignment like IP and IP_adv, which save labeling time. We argue that labeling is an off-line task whose consumed time should not be top priority to measure overall performance. Rather, we would like to focus on label size which is more related to query processing time in on-line task.

3.2.6 Conclusion and Future Work

It has remained a challenge to apply graph reachability indexing techniques to very large sets of LOD. In this Chapter, we showed that simple graph metrics can be exploited to reduce the index size. Specifically, a vertex id assignment technique is introduced, which utilizes vertex degrees. Experimental results showed that the proposed approach reduces 2-hop label size on real-world DAG datasets. In order to make 2-hop labeling to become more applicable to LOD, we suggest future research directions as follows.

2-Hop Label Update Linked Open Data (LOD) evolves over time [46]. When a DAG G_t changes into G_{t+1}, the straightforward way is to construct the 2-hop index against G_{t+1} from scratch. As this is not feasible for massive DAGs in LOD, we define a problem that generates an *update-efficient* permutation by which 1) a smaller number of existing labels are modified and 2) a smaller index size is maintained, as defined in Definition 9.

Definition 9. Update-efficient Permutation Assume that we already have $G_t(V_t, E_t)$ and its 2-hop index $I^k_\sigma(V_t)$. Given $G_{t+1}(V_{t+1}, E_{t+1})$, generate a new permutation σ_{t+1} such that

$$\underset{\sigma_{t+1}}{\operatorname{argmin}}\ \alpha \times |\{v \in V_t \bigcap V_{t+1} | I^k_{\sigma_t}(v) \neq I^k_{\sigma_{t+1}}(v)\}| + \beta \times size(I^k_{\sigma_{t+1}}(V_{t+1}))$$

, where $\alpha \geq 0$, $\beta \geq 0$.

Since there is a trade-off between these criteria, weights are used according to requirements. The first factor of the equation in Definition 9 represents the number of vertices in the current version V_{v+1} whose labels are not the same as in the previous version in $I^k_\sigma(v)$. When a new vertex v_{new} is added to a DAG $G_t(V_t, E_t)$ that has already been labeled by σ_t, which number should be assigned to $\sigma_{t+1}(v_{new})$? The simplest way is to make $\sigma_{t+1}(v_{new}) = max(\sigma_t(V_t)) + 1$. However, with this approach, the minimized index size is not always ensured. Sophisticated algorithm is required.

As a partial solution to the issue, we have some experience on scalable RDF change detection [47, 48, 49] that outputs added and removed triples for two given RDF dataset versions. By utilizing this system, it would be possible to update only a portion of an existing 2-hop index while avoiding re-labeling.

Scalable Algorithm To the best of our knowledge, there exists few work for 2-hop labeling algorithm working on distributed computing environment. Due to the not enough memory issue in a single machine, it would not be possible to do 2-hop labeling against massive graph like LOD. However, 2-hop labeling in parallel is not an easy task.

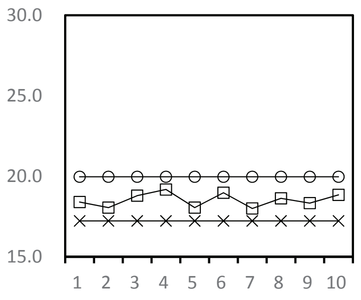
In order to do 2-hop labeling in a cluster, when vertices are distributed, how can we know reachable vertices from a vertex? Given a set M of machines, we may distribute a subset V_i of V to a machine M_i and then perform labeling independently. The challenge is to obtain $L_{in}(v)$ and $L_{out}(v)$ for a vertex $v \in V_i$ residing in M_i whereas some $o \in L_{in}(v) \bigcup L_{out}(v)$ have been distributed to the other machines M_j, such that $i \neq j$.

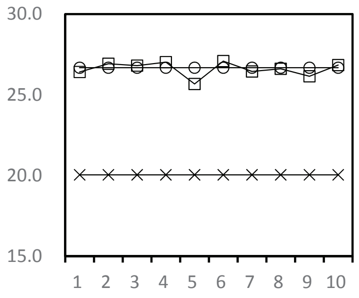
We are motivated by the approach in [50] that proposed cluster based labeling algorithms for trees. In that work, Mapper performs incomplete labeling and then Reducer completes the labels by referring to the offset table shared by all machine. The offset table is constructed based on information collected from each Mapper, which contains information needed to complete the labels in Reducers. We plan to adapt the idea in a manner that effectively deals with large DAGs in a cluster.

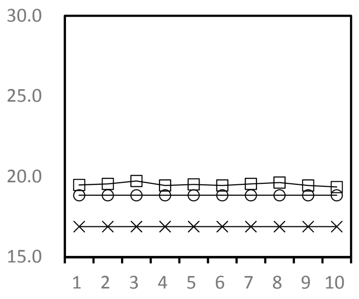
Extensive Experiments We plan to collect LOD datasets and then transform them into DAGs by condensing SCC. Synthetic DAG datasets will also be considered, varying graph metrics such as in/out-degree, diameter, and vertices with no ancestors or descendants. In particular, several releases of DBpedia datasets[4] will be collected in order to evaluate the performance of updating the index. To examine the pros and cons of our approach in greater details, we will compare it to a number of notable existing approaches such as [45, 44, 51]. Regarding our cluster-based algorithm, as there exist few works on cluster-based 2-hop labeling, we will modify existing cluster-

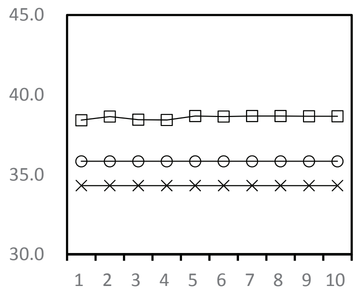
[4]`http://wiki.dbpedia.org/services-resources/datasets/previous-releases`

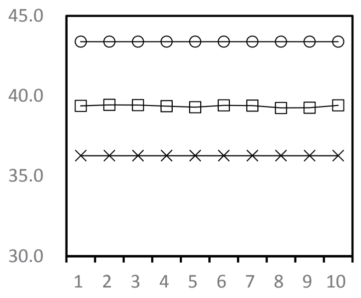
based tree labeling algorithms [50] and consider these as the baseline.

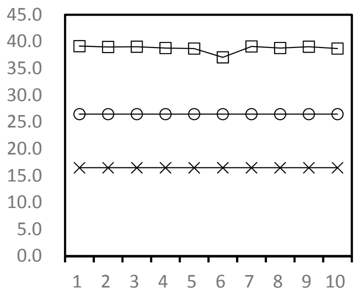

(a) arxiv

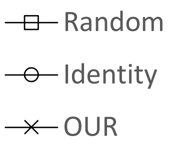

(b) go


(c) pubmed


(d) citeseer

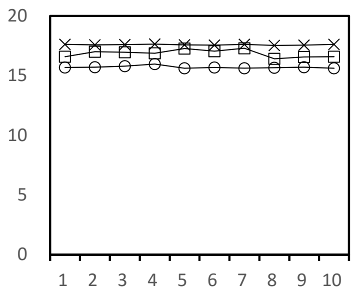

(e) citpatents


(f) go-uniprot

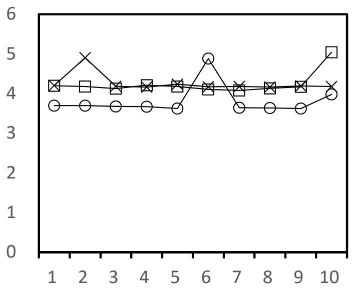


(g) Legend

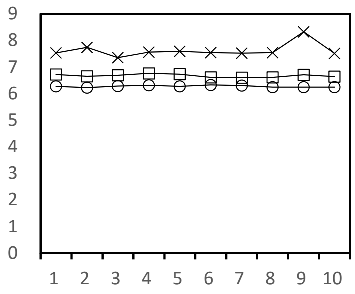
Figure 9: Label size



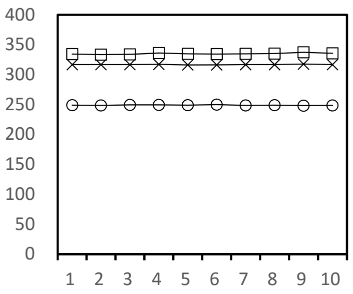
(a) arxiv



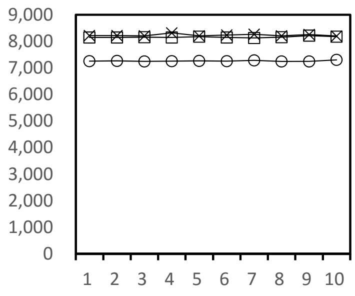
(b) go



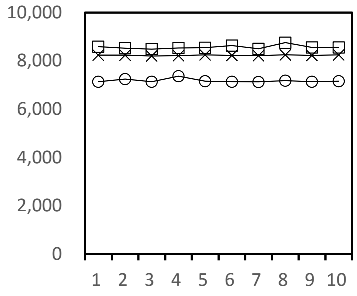
(c) pubmed



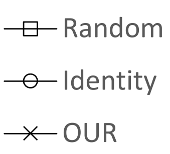
(d) citeseer



(e) citpatents



(f) go-uniprot



(g) Legend

Figure 10: Labeling time

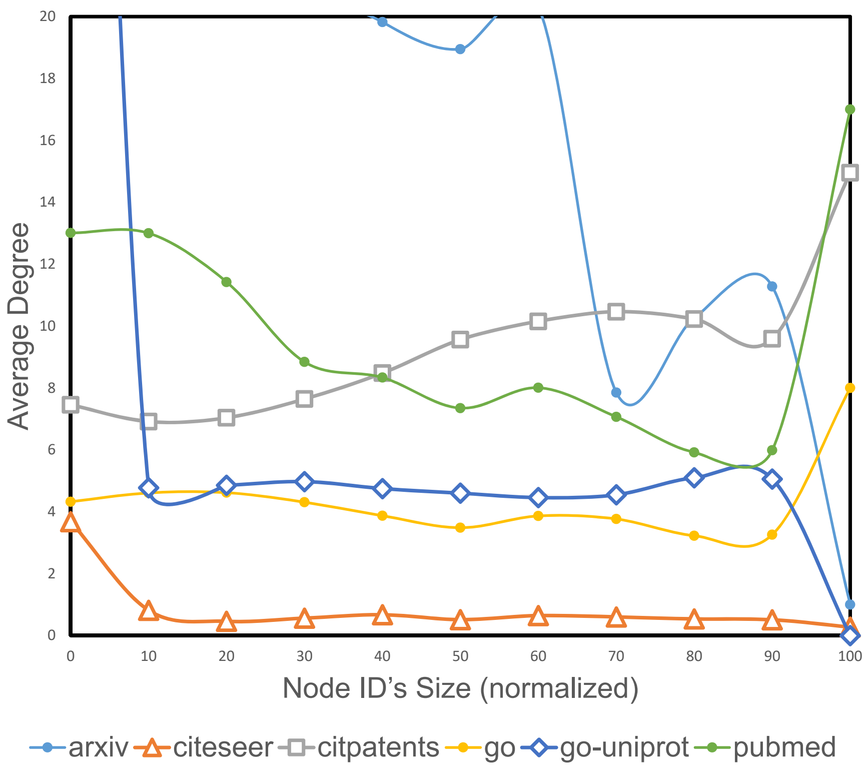


Figure 11: Average degrees of each size of vertex id.

3.3 Shortest Path Discovery

3.3.1 Introduction

Shortest path discovery is one of fundamental problems in computer science, which is defined in Definition 10 in our context.

Definition 10. Shortest Path Discovery Given a DAG $G(V,E,EW)$ and two distinct vertices $s = v_0$ and $t = v_k$, find a path $P = v_0, v_1, ..., v_k$ such that $\sum_{i=0}^{k-1} EW(v_i, v_{i+1})$ is the smallest, where EW is a map from edges to weights.

The meaning of *shortest* is defined according to requirements. Shortest path discovery has been applied to diverse domains. For example, cites and roads connecting two cites can be modeled in graphs. Shortest path is the route that visits smallest number of intermediate cites or whose sum of distance is the smallest.

A well-known solution for shortest path discovery is Dijkstra's algorithm. Bi-directional strategy is proposed to navigate both from source and target to reduce search space [12]. Recently, RDB based shortest path discovery algorithm is proposed, which is called FEM (Frontier-Expand-Merge) framework [3]. Although FEM framework utilizes RDB, it still remains a challenge to utilize distributed computing environment to solve shortest path discovery problem. In this chapter, a solution that utilizes distributed computing environment is proposed. By doing so, massive graph can be dealt with efficiently, where existing approaches work inefficiently.

The remainder of this Chapter is organized as follows. In Section 3.3.2, the state-of-the-art approach is briefly described. In Section **??**, the proposed approach called FEM-SR is demonstrated with two optimization techniques. Experimental results on real-world DAGs is shown in Section 3.3.5. Section 3.3.6 discusses the federated shortest path discovery which is a novel definition that is closely related to reachability. The Chapter is concluded in Section 3.4 with future directions.

3.3.2 FEM

FEM framework consists of three steps such as Frontier, Expansion and Merge. Figure 12 depicts the data-flow that finds shortest path from s to t in the top-left side graph. Edges of the graph are stored in TE table consisting of pairs (*src* = source vertex, *dst* = target vertex). For the simplicity, weight of edges is omitted here, which implies that every weight is set to 1. Visited vertex table A^i contains four items. *nid* represents vertex id, $d2s$ for distance to source vertex, $p2s$ for vertex id that has been visited before visiting *nid* and f for indicating whether *nid* has been expanded or not.

In initialization step, TE table is created from the input graph. When the user selects a source and target vertex, an iterative task begins. In round 1, the source vertex s is inserted into A^1, where $d2s$ is set to default value which is 0 and $p2s$ is set to itself. F operator selects the frontier node which has the smallest $d2s$ value and f is false. In this case s is selected. E operator in turn selects outgoing edges of the frontier node. Specifically, it selects *dst* from TE table such that *src* is s. In this case, a and b are selected. Accordingly, $d2s$ is set to 1 which is the sum of s's $d2s$ and a and b's weight. M operator merges A^1 and E^1 to obtain A^2 which is will be used for the

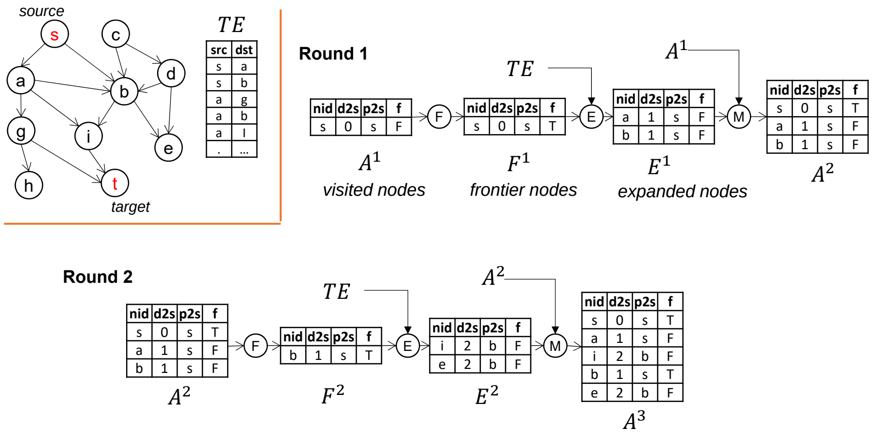


Figure 12: The data-flow diagram of RDB based shortest path discovery algorithm called FEM framework. Some part of the figure is borrowed from [3].

next round. This process continues until vising t. Shortest path is recovered by backtracking the last visited table. Two optimization techniques are proposed. Firstly, the process is carried out bi-directional way, that is, the similar process is performed from t. Whenever it meets an intermediate vertex, the process terminates. Secondly, in performing E operator, expanding to some outgoing edges is delayed if these edges have large weight.

3.3.3 FEM-SR

FEM-SR is a shortest path discovery system working on Spark. The name came from existing FEM framework because FEM-SR is also based on three stages such as Frontier, Expand and Mersge. Instead of using SQL queries, FEM-SR uses Spark operations, which means that intermediate data is maintained in Spark RDD. It should be noted that RDB tables are updatable while Spark RDD is immutable. This makes us to specify new

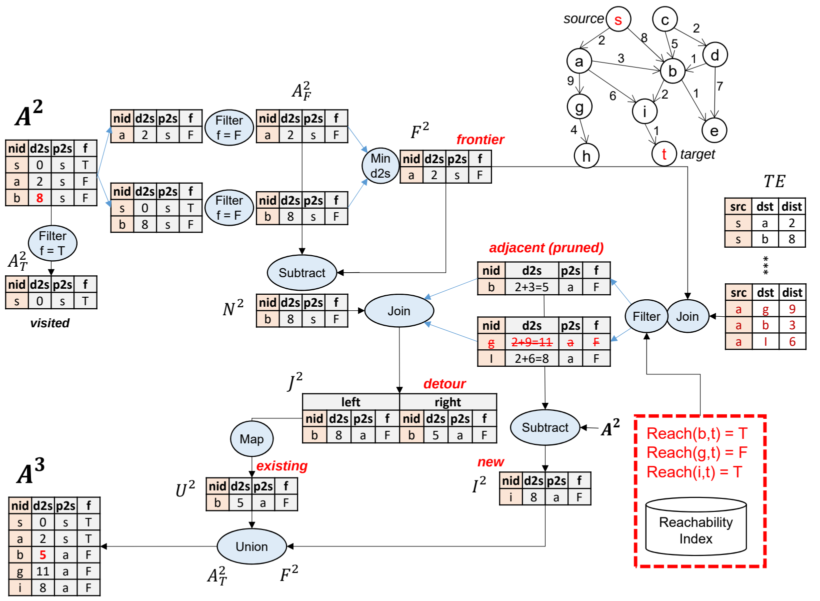


Figure 13: The data-flow diagram of the proposed approach called FEM-SR. The second round is depicted.

operations that works similar with SQL queries used in FEM.

The data-flow diagram of FEM-SR is depicted in Figure 13. For the simplicity, RDD is represented in table like RDB tables in Figure 12. In implementation level, RDD used here is a set of key-value pairs. *nid* becomes key and $(d2s, p2s, f)$ is value. This allows to distribute data by *nid* using a hash partitioner. Circles represents Spark operations. .

Only the second round is depicted in Figure 13, where shortest path from s to t is considerted. The vertex s in A^2 is marked as true since s has already been visited in the first round which is not presented here. The other vertices to be visited later are in false.

First step is to apply Filter operation in order to obtain an RDD A_F^2 with vertices whose f is false. The other already visited vertices are selected to create A^2_T. Then, Min operation is applied to A^2_F in order to select vertices that has the minimum $d2s$. F^2 is created from the frontier vertex and f is set to true to indicate that it has been visited.

Subtract operation is applied to A^2_F and F^2 to create N^2. It contains candidate vertices whose distance could be updated by detours. F^2 and TE are then joined on $nid = src$ to create D^2 that contains neighborhoods of the frontier node a. N^2 and D^2 are joined to create J^2 which consists of new vertices I^2 and existing vertices U^2. N^2 is replaced by U^2 if $d2s$ value in U^2 is smaller than the value in N^2. In other words, the distance of the new path s,a,b which is 5 is smaller than the distance of the old path s,b which is 8. Since shorter path is discovered, the visited table has to be updated.

The last step is to apply Union operation to A^2_F, F^2, U^2 and I^2 to create A^3 which is processed in the next round.

This process corresponds to forward navigation. We do the similar processing from t which is called backward navigation as proposed by the bi-directional strategy.

3.3.3.1 Exploiting Reachability Index

During Expand stage in FEM, every neighborhoods are considered, which often slow down the discovery process if too many neighborhoods exist. In the context of FEM-SR, this corresponds to D^2 in Figure 13. To overcome the limitation, we make use of reachability index to prune vertices that are not reachable to target vertex. Specifically, these not reachable vertices are not inserted into D^2.

The right-bottom side of Figure 13 depicts reachability index. The frontier node here is a and its neighborhood vertices are b, g, i according to edge RDD TE. Reachability queries are constructed having a neighborhood vertex as the source and t as the target, like $reach(b,t), reach(g,t), reach(i,t)$. Reachability index would say false for $reach(g,t)$ as can be seen in the graph where there is no path from g to t. In this way, we can prune g, which reduce the size of D^2.

3.3.3.2 Exploiting External Store

Edge list is maintained in TE, which means that TE is huge for huge graphs. In our scheme, TE is joined with F^i in ith round. Note that F^i contains one tuple. Join operation is not adequate operation in this skewed data. Rather, lookup operation would be efficient. Unfortunately, lookup operation is not efficient in Spark framework as it requires to collect data across worker machines. We decided to utilize external database such as key-value stores. RDB is not necessary in this case since no complex queries are needed but a simple lookup query is sufficient. In this thesis, MongoDB is used because there exist a MongoDB connector for Spark. Thus, the original Join operation is replaced by a MongoDB query. Specifically, edges are also stored in a MongoDB collection having the same column with TE. For a given frontier vertex, we issue a MongoDB query that ask to fetch documents where *src* is equal to the frontier vertex.

Utilizing MongoDB for all case is not a good choice. For the frontier vertex with many neighborhoods, utilizing MongoDB could be inefficient. MongoDB runs on a different process with Spark processes. Data fetched from MongoDB is fed into Spark processes through network communication. Moreover, the fetched data is only fed into Spark's driver process. This process does not take any benefits of distributed computing environment. In this regards, we use a heuristic that if the frontier vertex's number of neighborhoods is more than some threshold, TE is used and MongoDB is used otherwise.

Table 2: Real-world DAG datasets

Name	Vertices	Edges	Average Degree (max.)	Median Degree
`arxiv`	6,000	66,707	22.2 (700)	14
`citeseer`	340,945	312,282	1.8 (55,758)	1
`citeseerx`	3,774,768	16,518,947	8.8 (793)	6
`go-uniprot`	6,967,383	34,769,339	10.0 (1,186,280)	4

3.3.4 Theoretical Analysis

In this section, the correctness of FEM-SR is described. Then, we discuss the cost model of FEM-SR.

3.3.4.1 Correctness

In this section, we will show that the pruning technique based on the reachability index discovers the shortest path correctly.

Definition 11. ***k*-Level Structure** Given an edge-weighted DAG $G = (V, E, EW)$ and a root vertex s, $L_k(s)$ is k-level structure rooted at s that is a set of ver-

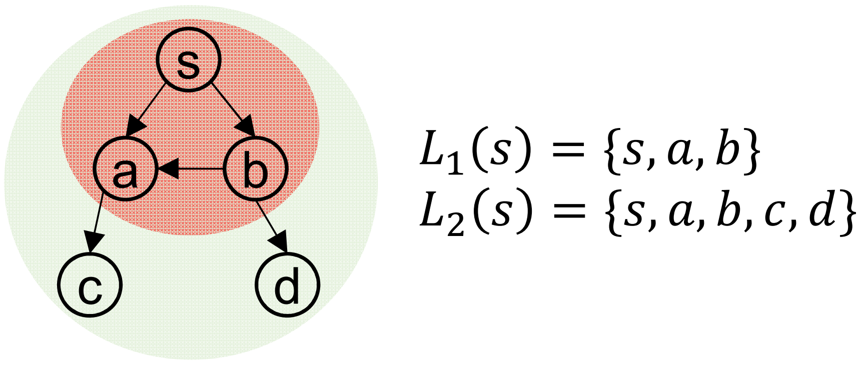


Figure 14: An example of level structures rooted at s.

tices as follows:

$$L_k(s) = \{v : dist(s,v) = k\} \cup L_{k-1}(s)$$

, where $i \geq 1$, $L_0(s) = \{s\}$ and $dist(s,v)$ is the maximum number of vertices in paths from s to v.

See Figure 14. For a given root vertex s. $L_1(s)$ is composed of s, a, b since a and b have one edge way from s.

We consider the correctness of FEM-SR based on level structures. $L_0(s)$ contains the source vertex s. $L_1(s)$ is composed of $L_0(s)$ and out-going vertices of $L_0(s)$. We may continue the process until all vertices are covered. It is trivial that the target vertex t is included in $L_k(s)$ at some k.

Theorem 3.3.1. Correctness of FEM-SR Given an edge-weighted DAG $G = (V, E, EW)$ and a shortest path (smallest weight sum) query (s,t) , FEM-SR (with pruning) discovers the shortest path correctly, in an assumption that the reachability index does not answer false positive such that it never

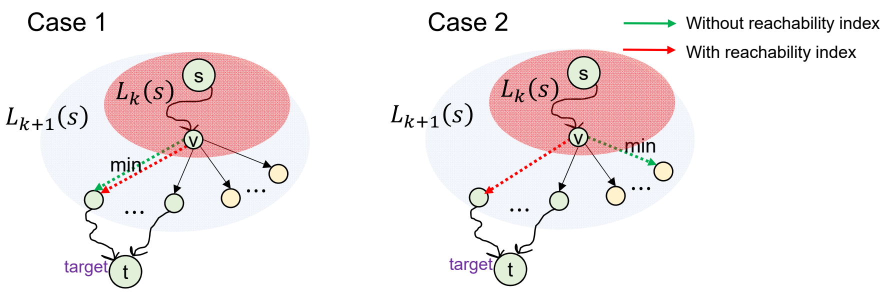


Figure 15: Two cases when expanding to adjacent vertices.

yield $Reach(v_i, v_j) = FALSE$ for the case when $v_i \rightsquigarrow v_j$.

Proof. We prove by mathematical induction. We have the base case $L_0(s) = \{s\}$, which is trivially the shortest path. We assume that FEM-SR discover shortest paths from s to every vertices in $L_k(s)$ correctly. We need to show that it also holds for the case $k+1$.

Let $O(v)$ be a set of out-going vertices for a vertex v. We may define $O(v)$ with respect to a target vertex t, such that $O(v) := O^r(v,t) \cup O^n(v,t)$ where $O^r(v,t)$ is a set of vertices that can reach t and $O^n(v,t)$ is a set of vertices that are not-reachable to t. See Figure 15, there two cases when we expand from a vertex v in $L_k(s)$ to an out-going vertex with the minimum edge-weight. The out-going vertex o_{min} with the minimum edge-weight could be either in $O^r(v,t)$ or in $O^n(v,t)$. Case 1 is when o_{min} is in $O^r(v,t)$ and Case 2 is when o_{min} is in $O^n(v,t)$. For Case 1, FEM-SR acts the same with FEM because o_{min} cannot be pruned. For Case 2, FEM-SR would choose a vertex o_s in $O^r(v,t)$ such that $EW(v,o_s)$ is the smallest among $EW(v,o_r)$ where $o_r \in O^r(v,t)$ because $o_{min} \in O^n(v,t)$ would be pruned. However, FEM will eventually backtrack to o_s because no paths exist toward t from any vertices in $O^n(v,t)$. This proves that FEM-SR and FEM find the same shortest path at $k+1$.

$\square$

3.3.4.2 Cost Model

FEM-SR exploits the reachability index while the baseline approach (i.e., FEM) does not utilize the index. In order to compare the cost of FEM-SR with the one of the baseline approach, the cost model of FEM-SR needs to incorporate two additional factors; the space requirements of the reachability index and the access cost of the index during computation.

Definition 12. Cost Model of FEM-SR Given a shortest path query (s,t), we have four types of vertices which are defined as follows: W is a set of relevant vertices, I is a set of not-relevant vertices, RT is a set of head of vertices that cannot reach t, and RN is a set of vertices that don't have to be visited because of being pruned. Let $cost_{visit}$ be the cost to visit a vertex and $cost_{reach}$ be the cost to access the reachability index for a vertex. $2HopSize$ denotes the space requirements defined in Definition 5. The cost models of FEM and FEM-SR are as follows:

$$cost_{\mathsf{FEM}} = cost_{visit}(|W| + |RT| + |RN|) \tag{3.9}$$

$$cost_{\mathsf{FEM\text{-}SR}} = cost_{visit}(|W| + |RT|) + cost_{reach}(|RT|) + 2HopSize \tag{3.10}$$

See Figure 16, an example of W, RT, RN, and I is depicted for a shortest path query (s,t). c and d are included in I because we don't have to visit

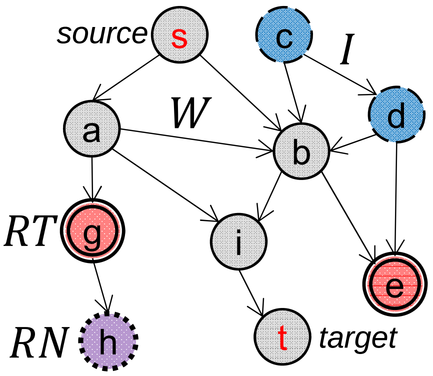


Figure 16: Vertices are classified into four types with respect to a shortest path query

these vertices when we start from *s*. We never visit *h* if *g* is pruned, which therefore belongs to *RN*. *e* is included in *RT* because even if *e* cannot reach *t* we have to check for the reachability of *e*.

Based on Definition 12, we derive an expression that represents the constraints where FEM-SR is more efficient than FEM in terms of query processing time and space requirements. The cost of FEM-SR should be smaller than the cost of FEM, $cost_{\text{FEM-SR}} < cost_{\text{FEM}}$, as re-expressed in Expression 3.11.

$$cost_{visit}(|W|+|RT|)+cost_{reach}(|RT|)+2HopSize-cost_{visit}(|W|+|RT|+|RN|)<0 \tag{3.11}$$

By placing the space requirements in the left-hand side, we get the following Expression 3.12.

$$2HopSize < cost_{visit}|RN| - cost_{reach}|RT| \tag{3.12}$$

$|RN|$ can be measured based on the pruning power, such as $|RN| \approx$

$P \cdot avg(RT_{child}) \cdot |RT|$ where P is the probability to prune vertices in RT and $avg(RT_{child})$ is the average number of children of vertices in RT.

By substituting $|RN|$, we get the final expression which is demonstrated in Lemma 2.

Lemma 2. Efficiency of FEM-SR FEM-SR is efficient than FEM if the following expression satisfies:

$$2HopSize < |RT|(cost_{visit} \cdot P \cdot avg(RT_{child}) - cost_{reach}) \tag{3.13}$$

Lemma 2 tells us that FEM-SR works better than FEM if i) there exist sufficient out-going vertices $|RT|$ that are used to prune vertices, ii) the pruning probability P is somehow big, iii) the cost to access to the reachability index is not high. There is a trade-off between the space requirements ($2HopSize$) and the number of visited vertices (right-hand side). $2HopSize$, $cost_{visit}$, and $cost_{reach}$ need to be estimated carefully according to application requirements to correctly compare the left-hand and right-hand side of Expression 2. We can see that for the small graph, which don't have sufficient vertices to be visited and pruned, FEM-SR could work worser than FEM.

3.3.5 Experimental Results

We performed experiments to measure effectiveness the proposed approach. Our system is divided into two systems FEM-S and FEM-SR. FEM-SR is the proposed approach and FEM-S is the same with FEM-SR except for that reachability index is not used. These systems are compared with the state-of-the-art shortest path discovery approach FEM [3].

Five machines equipped with 3.1 GHz CPU and 24 GB RAM were used. Ubuntu 14.04.4 is installed with Hadoop 2.7.1 and Spark 2.0.1. FEM-SR and FEM-S run in Yarn-Cluster mode. FEM is executed based on MySQL 5.5.49. All systems were implemented in Java.

Real-world DAG datasets[5] used in these experiments are listed in Table 2.

3.3.5.1 Computation Time

Figure 17 and 18 show the time required to compute shortest path varying path lengths from 0 to 6. Computation time varies according to systems and datasets. Overall, FEM-SR is faster for large and dense graph such as `gouniprot` and `citeseerx`. The other approaches are faster for small and sparse graph such as `arxiv` and `citeseer`. For large and dense graph, only FEM failed due to not enough memory problem. It is because that all expanded vertices has to be loaded into a table.

Shortest path query whose length is zero can be answered in constant time by using reachability index. This explains why FEM-SR takes very short time. In the case of query #1 of path length is zero in `arxiv`, FEM-SR is slower than FEM. This is because that reachability index cannot determine 100 % queries. For those query that cannot be answered by reachability index only, FEM-SR expands to all neighborhood vertices.

`citeseer` shows an interesting tendency in that FEM-SR is the worst

[5]`https://code.google.com/archive/p/ferrari-index/downloads`

and FEM is the best. The reason is that citeseer has almost same number of vertices and edges, which means that most vertices has one neighborhood vertex. In one iteration, only one new vertex is inserted. FEM must be the fastest since no network communication is required. However, FEM-SR and FEM-S requires to exchange data between machines even for the small data, which slow down the overall computation time. This is a natural consequence when using distributed computing environment. Processing small data is often inefficient in distributed computing environment than doing in a single machine.

For the case of citeseerx, FEM failed due to memory problem. Some vertices of citeseerx have more than 20 million neighborhood vertices. FEM failed because it cannot maintain a temporary table with such a huge number of rows. On the other hand, FEM-SR can successfully handle because of data distribution across machines.

For the case of gouniprot, FEM-SR is the best overall. Now, we can see the effectiveness of reachability index. For the small graph, FEM-S is faster than FEM-SR for some queries. For the large graph, FEM-SR is faster than FEM-S. Even if accessing reachability index takes some time, it would be more efficient, a condition required to expand to large number of neighborhood vertices.

3.3.6 Federated Shortest Path Discovery

In this section, we discuss our novel definition called federated shortest path discovery, in order to show applications of the proposed approach. To the best of our knowledge, there exist few literature that uses the term *federated* in the context of shortest path discovery.

Conventional shortest path discovery deals with a single target graph. Reachability index can be exploited to prune intermediate paths during computation. Unlike conventional counterpart, federated shortest path discovery is to find shortest path on multiple graphs, for instance, target and support graph. Even if a path exists in the target graph, it would be pruned if the path does not exist in the support graph, and vice versa. We give the definition of federated shortest path in Definition 13.

Definition 13. Federated Shortest Path Given a graph G_i, suppose that there exist no path from s to t, but there exists an edge (s_i, t_i) such that $s \rightsquigarrow s_i$ and $t_i \rightsquigarrow t$. In other word, for all i, we have $s_i \not\rightsquigarrow t_i$. If we have $s_i \rightsquigarrow t_i$ in another graph G_o, federated shortest path between s and t on G_i with G_o is $s \rightsquigarrow s_i \rightsquigarrow t_i \rightsquigarrow t$.

Figure 19 shows the conceptual view of conventional shortest path and federated shortest path. Conventional shortest path is based on a single graph. In other words, reachability index is built from the same graph. Thus, main purpose of the reachability index is to speed up the discovery process as propose in this thesis.

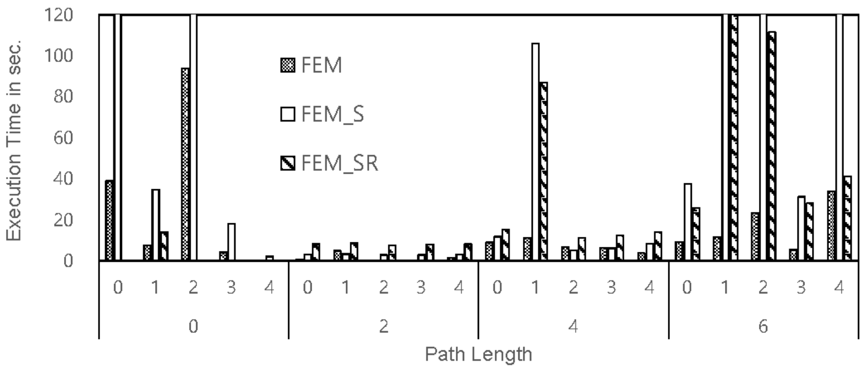
On the other hand, federated shortest path is based on two or more graphs. Shortest path discovery is basically performed against the target graph G_i. Reachability index is built from another graph G_j which is called here support graph. During the discovery process, reachability index for G_j is referenced to check paths exist or not. If there is no path in G_i but exist in G_j, then navigation keeps going. It can be viewed as a dynamic integration of two graphs.

Federated shortest path can be applied to biomedical domain, as depicted in the bottom part of Figure 19. Here, we assume that main target graph is constructed from Biogrid. We also have similar datasets called STRING. In the context of conventional shortest path discovery, Biogrid and STRING should be integrated to construct a single big graph. On the other hand, federated shortest path discovery does not require to have one graph. These datasets can exist separately. The usability of federated shortest path discovery can be understood by the fact that datasets evolve over time.

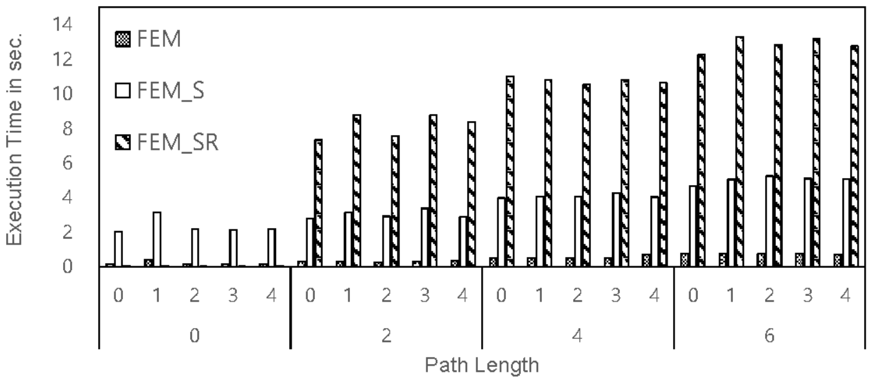
3.4 Conclusion

In this Chapter, we proposed a shortest path discovery algorithm. The purpose of the work is to make it possible to apply shortest path discovery to massive graph. Particularly, Spark, which is one of popular distributed computing frameworks, is utilized to carry out in distributed fashion. Reachability index is exploited to speed up the discovery by pruning not reachable paths. Furthermore, we proposed a approach that reduce reachability index size using vertices' degrees.

Future work is to consider k-shortest path discovery [52, 53, 54], which is generalized version of shortest path discovery discussed in this thesis. To the best of our knowledge, there exist few work on utilizing distributed computing environment to solve k-shortest path discovery. We plan to extend our Spark based approach to solve k-shortest path discovery. In addition, our approach is based on FEM framework. Required number of iteration is proportion to the path length. This slow down the proposed algorithm when dealing with long distance paths. We plan to devise an approach to reduce the number of iterations.

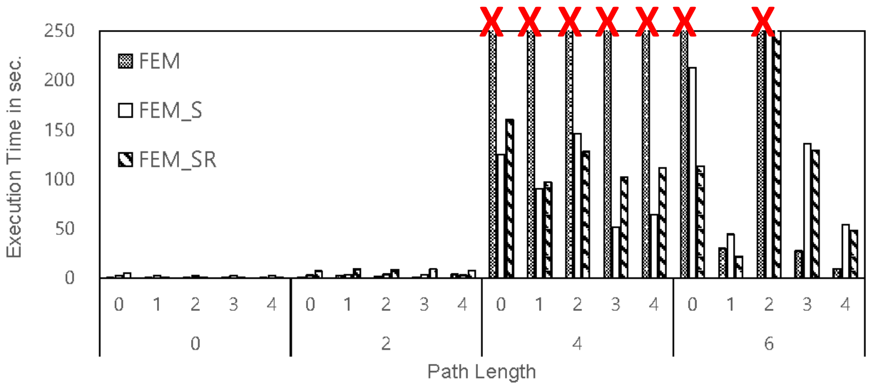


(a) arxiv

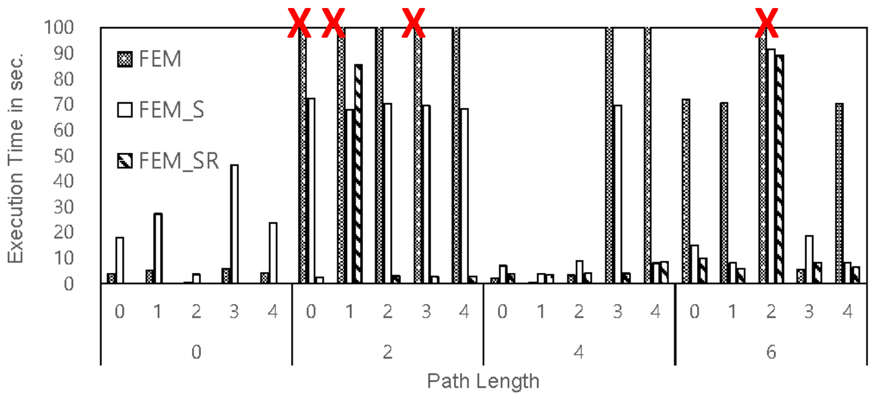


(b) citeseer

Figure 17: Time consumed to compute shortest path for the case of `arxiv` and `citeseer`. X-axis represents each query varying path lengths (0,2,4,6) and five queries (0,1,2,3,4). X indicates failure due to memory problem.



(a) citeseerx



(b) gouniprot

Figure 18: Time consumed to compute shortest path for the case of `citeseerx` and `gouniprot`. X-axis represents each query varying path lengths (0,2,4,6) and five queries (0,1,2,3,4). X indicates failure due to memory problem.

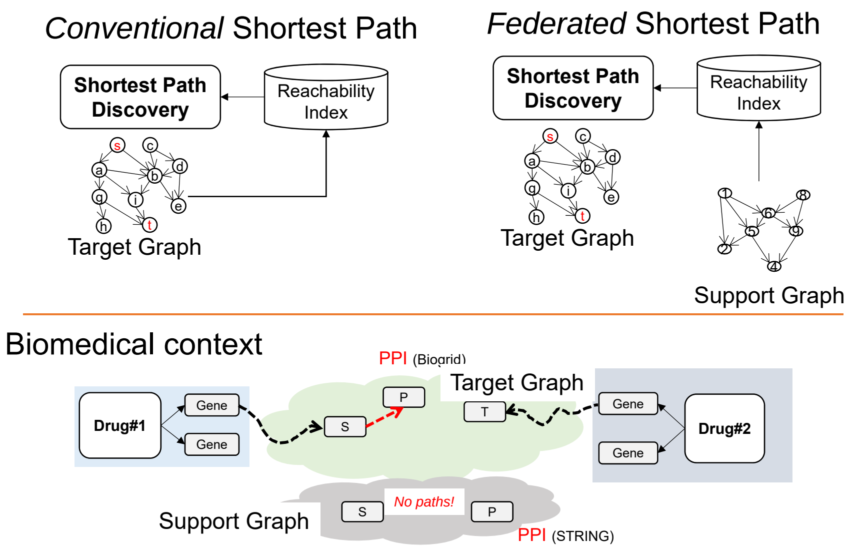


Figure 19: Federated shortest path discovery and its application to biomedical domain.

Chapter 4

Graph Path Matching based on Signature Encoding

4.1 Introduction

Large numbers of RDF (Resource Description Framework) triples are available in Linked Data which can grow exponentially. It makes SPARQL query processing engines infeasible on a single machine. To address this scalability issue, MapReduce framework-based SPARQL engines have been proposed, but we note that these methods are limited in terms of join evaluations. The two-way join based approach evaluates joins via a sequence of binary multiplications that require multiple MapReduce jobs, which involves costly disk accesses between MapReduce jobs. The multi-way join based approach combines multiple two-way join operations, which allows the simultaneous evaluation of joins during one MapReduce job. However, the size of data for the MapReduce job might increase exponentially if a complex query is given. In this study, we propose SigMR, a pruning method for multi-way join-based SPARQL query processing in MapReduce. In the proposed approach, a SPARQL query can be evaluated in a single MapReduce job, where the size of data is reduced dramatically by pruning based on our signature encoding technique, thereby overcoming the weaknesses of the

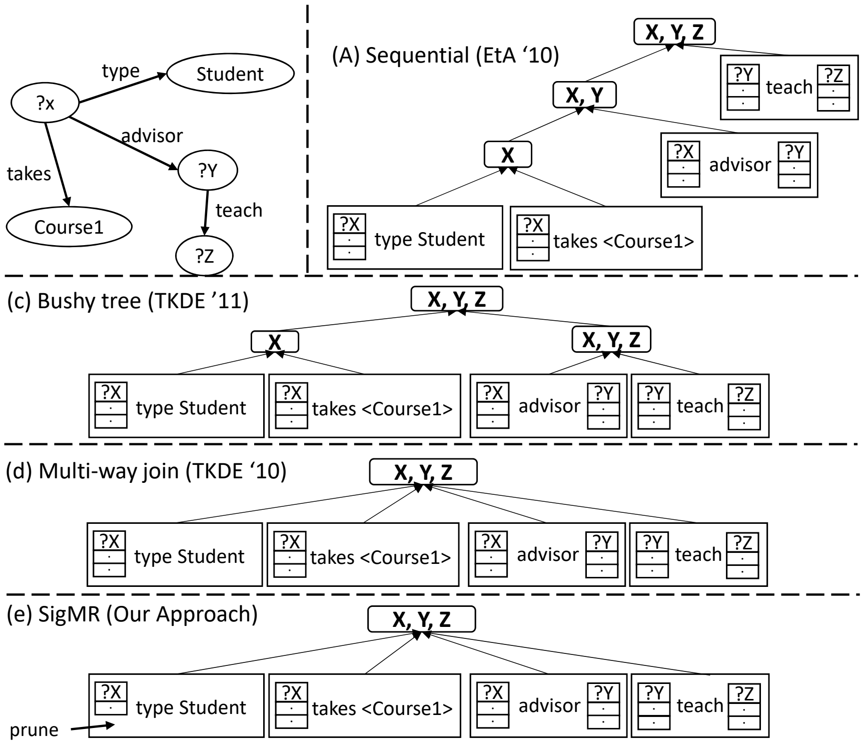


Figure 20: Join strategies

previous approaches. In experiments, we showed that the query processing time required was lower with our approach than existing MapReduce-based methods.

Large amounts of information are available on the Web. RDF (Resource Description Framework)[1] is a popular way of representing information on the Web, where a specific information fragment can be represented by an RDF triple, which comprises a subject, predicate, and object. The information demand against an RDF triple dataset mights be expressed in

[1]http://www.w3.org/RDF

SPARQL[2], which is a query language for RDF. Its nature of graph-based data model allows the user to navigate information in a more structured manner than is possible using traditional document search engines [55, 56]. Applications that make use of RDF datasets include DBpedia mobile [57], a cultural heritage guide [58], RDF browser [59], and a movie recommender [60]. These applications might be implemented upon an RDF triple management system (also known as a triples store or SPARQL engine) that supports the maintenance of RDF triples and the answering of SPARQL queries. SPARQL query processing can be viewed as graph path matching defined in Definition 14, since RDF data is a graph.

Definition 14. Graph Path Matching Given a DAG $G(V,E)$ and a graph path query $Q = v_0, ..., v_{i-1}, a_i, v_{i+1}, ..., v_k$ with a set $\{a_w\}$ of variables and a set $\{v_w\} \subset V$ of vertices, find a path $P = v_0, ..., v_{i-1}, v_i, v_{i+1}, ..., v_k$ where a_i is replaced with v_i such that $(v_{i-1}, v_i) \in E$ and $(v_i, v_{i+1} \in E)$ for all w. In terms of RDF, Q can be viewed as a set of triple patterns TP_i such that $TP_i := (s = a_i, p, o = v_{i+1})$ where p is the predicate associating the subject v_k and the object a_{k+1}. At least one of s, p, and o is a variable.

In connection with Chapter 3, a discovered graph path

$$P = c_0, ..., c_i, ..., c_j, ..., c_k$$

can be transformed into a graph path query

$$Q = c_0, ..., a_i, ..., a_j, ..., c_k$$

[2]http://www.w3.org/TR/rdf-sparql-query

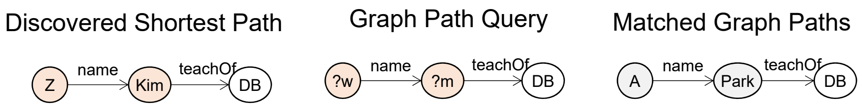


Figure 21: Graph path can be transformed into a graph path query.

such that at least one of constant c_w is replaced with a variable a_w. Depending on the choice of constants in P to be replaced with variables, different graph path queries can be obtained. Figure 21 depicts the relationship between graph path and graph path query. Z and Kim in the graph path is replaced with ?w and ?m, respectively, thereby, obtaining a graph path query. The goal of graph path matching is to find another constants that are matched to each variable, such as A and Park, respectively.

Notable RDF triple management systems include Sesame [61], Jena TDB [19], Virtuoso [62], and RDF-3X [22]. More triples are increasingly available on the Web, such as Linked Data[3], but these approaches have encountered a scalability problem. This issue has encouraged researchers to realize the full potential of distributed computing environments for SPARQL engines [63].

Distributed approaches for SPARQL engines can be categorized according to the systems upon which they are based, such as MapReduce [5, 34, 4], key-value stores [32, 26, 27, 28], and federation [30, 31]. It is difficult to state that a certain design is the best for a SPARQL engine [64], and thus we only consider MapReduce-based approaches in the present study. The MapReduce framework was proposed to facilitate the implementation of distributed algorithms that work on a cluster. Several SPARQL engines

[3]http://linkeddata.org

have been proposed based on the MapReduce framework [7, 6, 5], but these methods are limited in terms of the join evaluation strategy (see Figure 20). Left-deep tree planning (B in Figure 20) is a straightforward method for implementing a two-way join, but it is not feasible when the height of join evaluation tree is large. Bushy tree query planning, which is also a two-way join (C in Figure 20) and was described by [5, 6], decomposes a SPARQL query into several nonconflicting parts based on joining variables in order to simultaneously execute multiple MapReduce jobs for each nonconflicting part. This approach allows the parallel processing of SPARQL queries, but it requires multiple MapReduce jobs, which involve costly disk accesses. The multi-way join approach (D in Figure 20) was proposed by [7] for processing a query in a single MapReduce job, where the triples bounded by joining variables in a given SPARQL query are sent to multiple reduce jobs in a redundant manner. Unfortunately, this approach can also be inefficient when the data volume is huge because the number of triples required for MapReduce jobs can increase exponentially. Both of these issues are addressed in the present study, where the main contributions are summarized as follows.

- A multi-way join strategy was implemented in the MapReduce framework for SPARQL query processing. It allows to answer a SPARQL query using a single MapReduce job. This addresses the problem of costly disk accesses introduced by a family of two-way join approaches where an execution of multiple MapReduce jobs is required.

- To further improve the multi-way join strategy, we devised a novel index scheme based on signature encoding that helps reduce the size of data that are required originally by the join strategy. According to our index scheme, a triple is stored with bits strings that encode triples that are joined with the triple. It is especially utilized during a query processing in order to prune triples that are not joined according to a given SPARQL query (E in Figure 20).

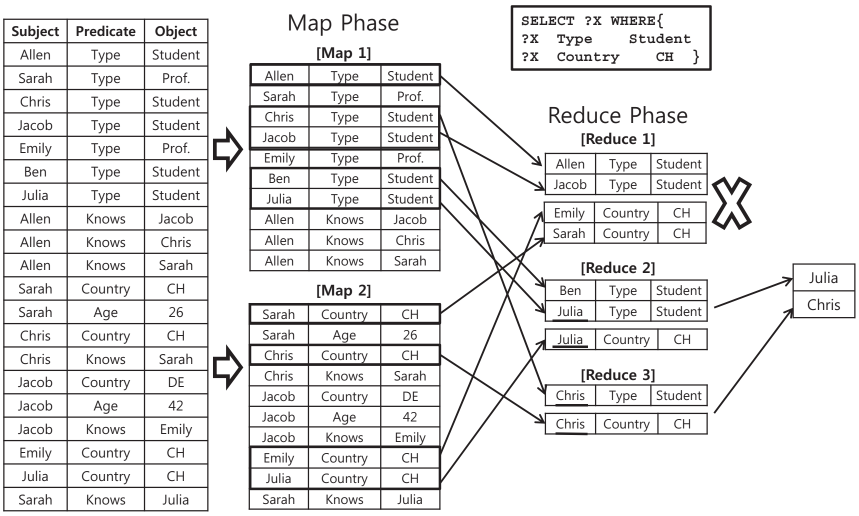
- We also devised a MapReduce based procedure for constructing our index from input RDF datesets, which enables to deal with a large number of triples in a distributed fashion.

- We performed experiments that demonstrated the effectiveness of the proposed approach compared with state-of-the-art systems. We also examined the performance of our approach after varying some parameters.

The remainder of this Chapter is organized as follows. Section 4.2 briefly reviews related work. The motivation of our research is discussed in Section 4.3, as well as some of the disadvantages of state-of-the-art approaches. Next, we explain the index structure in Section 4.5 and index building in Section 4.6. Query processing is described in Section 4.7. The experimental results are presented in Section 4.9. We give our conclusions and suggestions for future research in Section 4.10.

4.2 Related Work

MapReduce is a method for implementing a distributed algorithm on a cluster [33]. Apache Hadoop is one of the most popular MapReduce implementations [65]. Many other programming models exist for distributed algorithms, but MapReduce has attracted researchers due to its simplicity. A MapReduce program comprises Map and Reduce phases. In the Map phase, the input data are split by the Map key and then sent to worker nodes. In the Reduce phase, operations are applied to the received datasets to obtain the desired output. MapReduce framework has been utilized by researchers doing work in machine learning [15], massive matrix computation [16], genomic analysis [17], and etc. In particular, we focus on MapReduce-based SPARQL engines such as HadoopRDF [5]. Other approaches that are dependent on a distributed key-value store (HBase), such as H_2RDF [27], are beyond the scope of the present study. Figure 22 depicts a general method for processing a SPARQL query in the MapReduce framework. In the Map phase, each triple is loaded onto several Mappers, depending on the number of worker machines and triples. When they match with a given SPARQL query, the triples are sent to Reducers using a Map key. In this case, the string value in subject position is the Map key because the joining variable `?X` is in the subject position. In the Reduce phase, the remainder of the query is considered to find joined triples and emit the matched results.



Subject	Predicate	Object
Allen	Type	Student
Sarah	Type	Prof.
Chris	Type	Student
Jacob	Type	Student
Emily	Type	Prof.
Ben	Type	Student
Julia	Type	Student
Allen	Knows	Jacob
Allen	Knows	Chris
Allen	Knows	Sarah
Sarah	Country	CH
Sarah	Age	26
Chris	Country	CH
Chris	Knows	Sarah
Jacob	Country	DE
Jacob	Age	42
Jacob	Knows	Emily
Emily	Country	CH
Julia	Country	CH
Sarah	Knows	Julia

```
SELECT ?X WHERE{
?X  Type     Student
?X  Country     CH  }
```

Figure 22: General method for processing a SPARQL query in the MapReduce framework. Triples are sent to Reducers, which match joined triples in a given SPARQL query.

4.3 Limitations of MapReduce-based SPARQL engines

The limitations of previous MapReduce-based SPARQL engines can be considered in terms of the join evaluation strategies employed, i.e., two-way join (sequential plans and bushy tree plans) and multi-way join (refer to Figure 20). The sequential plan builds a query evaluation tree by placing each triple pattern on the left-hand side [4]. A drawback of this approach is that the query evaluation time depends on the height of the tree, which is proportional to the number of triple patterns in given SPARQL query. The bushy tree plan, which is employed by HadoopRDF [5, 6], tries to minimize the height of the tree by placing conflicting triple patterns into the same sub-tree whereas nonconflicting patterns are placed in different sub-trees. Nonconflicting join evaluations can be processed in a parallel manner but they still require several Hadoop jobs to complete the overall join evaluation. The multi-way join approach was proposed by [7, 66] in order to process a query using a single MapReduce job by sending the triples bounded by variables joined in a given SPARQL query to Reduce jobs in a redundant manner. Unfortunately, this redundancy cannot be handled easily if large volumes of triples are processed and the given SPARQL query has many joining variables.

Given all of these limitations, we suggest a more efficient join method, SigMR, which can improve the multi-way join strategy by our signature encoding technique that allows the pruning of triples during a query processing. It helps reduce the size of data required originally by the multi-way join strategy

4.4 SigMR

SigMR is a SPARQL engine based on the MapReduce framework. Figure 23 shows the input and output of SigMR, and its components. In the preprocessing step, which is shown in the upper section, N-Triple files are loaded to execute three MapReduce jobs, which generate the index catalog and the index stored in HDFS (Hadoop Distributed File System). In the query processing step, which is shown in the bottom section, a SPARQL query submitted by user is analyzed to load the created index and a multi-way join is then performed to find the matched results. To make this process

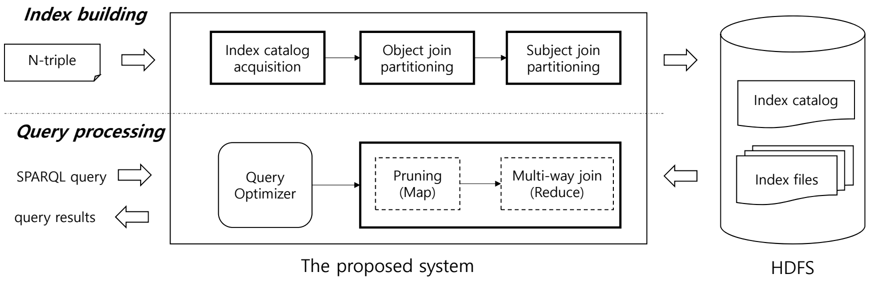


Figure 23: Architecture of SigMR based on the MapReduce framework. The bold rectangle represents a MapReduce job. Note that the query optimizer (depicted in a rounded rectangle) is not a MapReduce job, which means that query evaluation only requires a single MapReduce job.

easier to understand, our index structure is explained in the next section where we show how to store triples with encoded signatures. Each step is then demonstrated in the following sections.

4.5 Index Structure

The vertical partitioning scheme [20] is adapted for our index structure, where triples are partitioned by its predicates and stored in each different file. The scheme allows to reduce the query execution time by selectively scanning some index files that are associated with BGPs in a given query. In order to further reduce the query execution time, we add a novel index, called joined triples, which can be used for pruning triples during a multi-way join evaluation. In this paper, four kinds of joins are considered.

Definition 15. ***(SS Joined Triples)*** *Given a RDF graph T, a set of triples SS_t is called as SS (Subject-Subject) joined triples for a triple t if they share* *the same subject, which is formally defined as:*

$$SS_t = \{(s, p\prime, o\prime) | t = (s, p, o) \text{ and } (s, p\prime, o\prime) \in T \text{ for all } p\prime \text{ and } o\prime\}$$

Definition 16. ***(SO Joined Triples)*** *Given a RDF graph T, a set of triples SO_t is called as SO (Subject-Object) joined triples for a triple t if objects are the same with the subject in t, which is formally defined as:*

$$SO_t = \{(s\prime, p\prime, s) | t = (s, p, o) \text{ and } (s\prime, p\prime, s) \in T \text{ for all } s\prime \text{ and } p\prime\}$$

Definition 17. ***(OO Joined Triples)*** *Given a RDF graph T, a set of triples OO_t is called as OO (Object-Object) joined triples for a triple t if objects are the same with the object in t, which is formally defined as:*

$$OO_t = \{(s\prime, p\prime, o) | t = (s, p, o) \text{ and } (s\prime, p\prime, o) \in T \text{ for all } s\prime \text{ and } p\prime\}$$

Definition 18. ***(OS Joined Triples)*** *Given a RDF graph T, a set of triples OS_t is called as OS (Object-Subject) joined triples for a triple t if subjects are the same with the object in t, which is formally defined as:*

$$OS_t = \{(o, p\prime, o\prime) | t = (s, p, o) \text{ and } (o, p\prime, o\prime) \in T \text{ for all } p\prime \text{ and } o\prime\}$$

One index file consists of lines of six items, i.e., four signatures, subject, and object, each of which is concatenated by a delimiter. Since each line corresponds to a triple, subject and object are from the triple. The signatures are generated from four joined triples of the triple. A formal definition of the signatures is as follows.

Definition 19. ***(Signatures)*** *Given a triple t, its four signatures are defined as*

$$Sig_b(SS_t)|Sig_b(SO_t)|Sig_b(OO_t)|Sig_b(OS_t)$$

where $|$ is a delimiter and $Sig_b(SS_t)$ is a signature that encodes SS joined triples of t, $Sig_b(SO_t)$ for SO joined triples and so on, which is defined in Definition 21.

The signature in our context refers to a bits string, which is a sequence of “0” or “1”. A *b*-bits string is a bits string whose length is *b*. Details of encoding joined triples into bits strings are explained in the following section.

4.5.1 Encoding Joined Triples

We first explain the string encoder that converts URI strings or literals or blank nodes into a bits string.

Definition 20. ***(String Encoder)*** *Given a string representation u of URI or literals or blank nodes, a b-bits base string encoder Bit_b is a map from u into b bits. A hash function is used to convert a string into a number which is then represented in bits.*

To exploit the nature of a URI string, we split a URI string with a comma character. For example,

http://www.Department0.University0.edu/GraduateCourse2

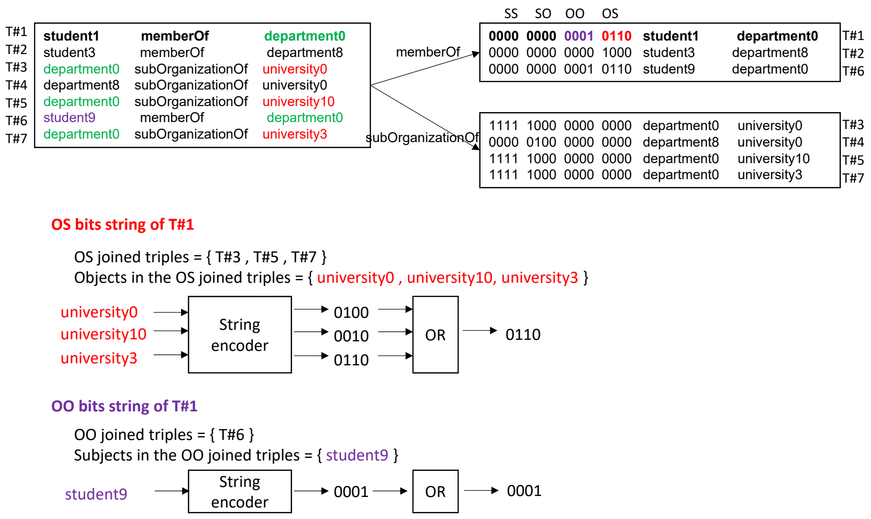

T#1
T#2
T#3
T#4
T#5
T#6
T#7
student1 memberOf department0
student3 memberOf department8
department0 subOrganizationOf university0
department8 subOrganizationOf university0
department0 subOrganizationOf university10
student9 memberOf department0
department0 subOrganizationOf university3
memberOf
subOrganizationOf
SS SO OO OS
0000 0000 0001 0110 student1 department0 T#1
0000 0000 0000 1000 student3 department8 T#2
0000 0000 0001 0110 student9 department0 T#6
1111 1000 0000 0000 department0 university0 T#3
0000 0100 0000 0000 department8 university0 T#4
1111 1000 0000 0000 department0 university10 T#5
1111 1000 0000 0000 department0 university3 T#7
OS bits string of T#1
OS joined triples = { T#3 , T#5 , T#7 }
Objects in the OS joined triples = { university0 , university10, university3 }
university0
university10
university3
String encoder
0100
0010
0110
OR
0110
OO bits string of T#1
OO joined triples = { T#6 }
Subjects in the OO joined triples = { student9 }
student9
String encoder
0001
OR
0001

Figure 24: Example showing the data flow when appending bits strings to each triple. The triples are actually stored in separate files according to predicate, as explained in section 4.5. In the cases of T#1, T#2, and T#6, no triple is joined with subject. As a result, we have “0” bits strings. In the cases of T#3, T#4, T#5, and T#7, there are no `OO` and `OS` joined triples.

is split into four strings such as

```
http://www,
Department0,
University0,
edu/GraduateCourse2
```

A hash function is applied to each string to compute an integer value, where 16-bit characters can be used to represent the integer value. By concatenating four of the 16-bit bits strings, we obtain a bits string with 64 bits. For example, a bits string that is twice as large is obtained by splitting each substring into two further substrings. For literals and blank nodes, we don't care about comma characters.

Using the string encoder, joined triples, which is a set of triples, are also encoded into a bits string by using OR operators.

Definition 21. ***(Joined Triples Encoder)*** *Given a joined triples W, a b-bits base joined triples encoder Sig_b is a map from W into a b-bits string.*

$$Sig_b(W) = \begin{cases} Bit_b(sub(w_0)) \diamond Bit_b(sub(w_1)) \diamond \ldots \diamond Bit_b(sub(w_n)) \\ \qquad \text{if } W \text{ is a OO or SO joined triples} \\ Bit_b(obj(w_0)) \diamond Bit_b(ob(w_1)) \diamond \ldots \diamond Bit_b(obj(w_n)) \\ \qquad \text{if } W \text{ is a SS or OS joined triples} \end{cases}$$

where $w_i \in W$, $\diamond$ stands for OR operator for two sequences of bits, $sub(w_i)$ is the subject of w_i, and $obj(w_i)$ is the object of w_i.

The method used to obtain `OO` and `OS` join bits strings is depicted in Figure 24. We assume that there are seven triples, as shown on the left-hand side. Our goal here is to obtain `OO` and `OS` bits strings for each triple, as shown on the right-hand side. In the case of the `OS` bits string for T#1, we have three `OS` joined triples: T#3, T#5, and T#7, which are all joined by object of T#1, i.e., `department0`. Only the URI strings in object position in these `OS` joined triples are passed to the URI encoder. To obtain a bits string, an OR operator is applied to the set of resultant integers, which are assumed to be binary numbers. The final bits string generated is `0110`, which carries information about these three URI strings, such as `university0`, `university10`, and `university3`. This type of procedure is motivated by the Bloom filter [67]. Note that given a URI string such as `university3`, which is assumed to be converted into a bits string `0010`, it is easy to check whether `university3` is in the bits string `0110` using a bit-wise operation. During query processing, a triple with a negative membership check would be pruned, which will be explained in Section 4.7. According to this schema, for a given URI string, a membership check returns that it exists “possibly” or “definitely not” in the set of URI strings represented actually in a bits string. The latter case would occur if a bits string of `university80` is also `0010` and a membership check returns `true`, which is not actually correct. Even if it is not possible to filter out 100 % like these cases, it is useful to benefit from the space-efficient structure of this method that does not require to store URI strings explicitly.

In the case of the `OO` bits string for T#1, we have one `OO` joined triple T#6. The `OO` bits string can be obtained from `student9` in subject position. Therefore, we are able to obtain `OO` and `OS` bits strings for T#1. The same procedure is applied to each triple. Note that the `OO` and `OS` bits strings of T#6 are the same as those of T#1. This is naturally given that T#6 and T#1 have the same object. This observation allows us to design a MapReduce-based approach for obtaining the bits strings in a parallel manner, which will be explained in Section 4.6. The `SS` and `SO` join bits strings are obtained in a similar manner. For example, in the case of the `SS` join, triples with the same subject as a given triple are considered. The `SS` bits string is obtained from a set of URI strings in object position of the `SS` joined triples.

We have shown how to obtain bits strings, but it should be noted that we only require subject or object to generate a bits string. For example, in the case of the `OS` bits string of T#1 in Figure 24, we ignore predicates such as `subOrganizationOf`. This is because it is meaningless to place predicates in bits strings given that predicate can be determined by a domain or range in a joined triple. Let us consider triples joined with T#1 by object. It would follow automatically from T#1's predicate `memberOf` that triples having predicate `memberOf` have certain predicates whose domain is `Department`, which is actually equal to the range of `memberOf`. In this case, we can automatically know that there must exist a triple having `subOrganizationOf` in predicate position. However, this is not always the case because these triples could be missing in a dataset for a department that does not yet belong to any university. In the present study, we do not address this issue because predicates must also be considered in order to generate bits strings which is a time consuming task. However, our approach can easily be generalized to consider joined triples on predicates. A straightforward way is to extend Definition 21 to also take predicates when generating a bits string for a triple.

4.6 Index Building

Index building is the process used to create the index (Definition 19). The MapReduce framework is utilized to benefit from a distributed environment in order to handle a large number of triples in an efficient manner. Building the index based on the MapReduce framework can be achieved in three MapReduce jobs. In the first job, the domains and ranges of predicates used in the input triples are identified. In the second job, the triples are split according to the ranges identified for predicates, and `OS` and `OO` bits strings are also computed. In the third job, the split triples are split again according to their domain, and `SS` and `SO` bits strings are also computed. After performing these three jobs, we obtain a set of index files that are partitioned vertically in the context of the domains and ranges of the predicates. We explain each job in a greater detail in the following sections.

4.6.0.1 Index Catalog Acquisition

Index catalog acquisition is a MapReduce process that takes triples as inputs and it outputs a catalog for the index. The index catalog includes the domains and ranges of the predicates and namespace strings. Figure 25 shows an example of the data flow in this process. In the following, we describe the process in an informal manner, but the interested reader is referred to Algorithm 1 and Algorithm 2 in Appendix for fuller details of the algorithms employed by the Map and Reduce functions.

In the Map phase, each triple is sent twice to two different Reduce jobs. The Map Keys are subject and object. The value for a Reduce job corresponds to the triple. For example, the first triple (`s1`, `type`, `Student`) goes to both of the Reduce jobs, i.e., `s1` and `Student`.

In the Reducer phase, a list of URIs and the domains and ranges of the predicates are identified. We denote `subject-key triple` as a triple with an subject that is equal to the Reduce Key. Similarly, `object-key triple` is denoted as a triple with an object that is equal to the Reduce

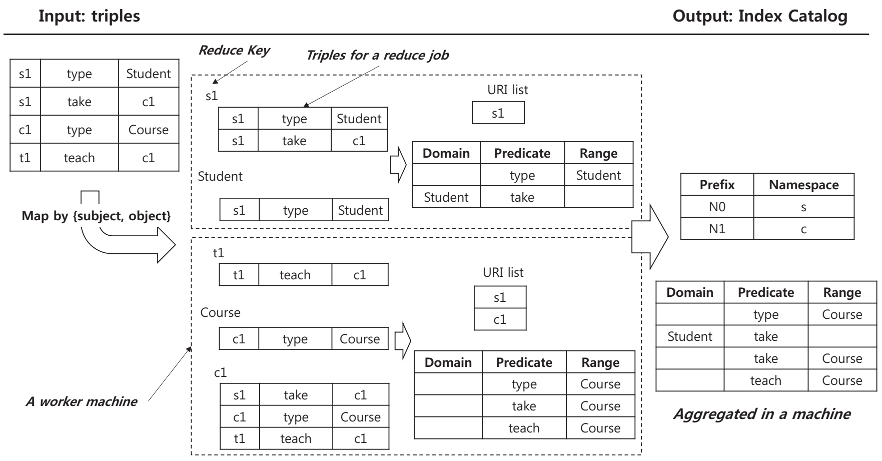
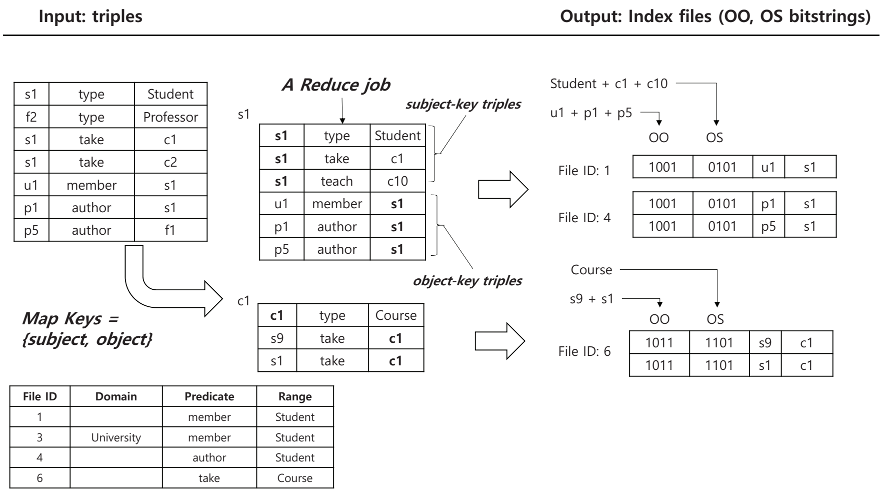


Figure 25: Example showing the data flow during an index catalog acquisition job. A triple is sent two times to Reducers based on subject and object. Each Reducer outputs a URI list and the predicates that are then aggregated on a single machine.

Key. See the Reduce job for `c1` shown in Figure 25. Here, we have one `subject-key triple`, i.e., (`c1`, `type`, `Course`), and two `object-key triples`, i.e., (`s1`, `take`, `c1`), (`t1`, `teach`, `c1`).

In the first Reduce job called `s1` in Figure 25, we have a `subject-key triple`, i.e., (`s1`, `take`, `c1`). If another `subject-key triple` exists where predicate is `type`, we know that the resource (`s1`) in subject belongs to the class with the name in object (`Student`). Based on this fact, the domains of the predicates of the other triples in `subject-key triples` can be identified automatically. In this case, we identified the domain of `take` as `Student`. The ranges of the predicates of the triples in `object-key triples` can be identified in a similar manner.

During this procedure, we also maintain distinct URI strings and their frequencies. For a URI string with a frequency that is above some threshold, we take its prefix string and this will be used for a namespace later. The final step is to aggregate these files into a single file.



s1	type	Student
f2	type	Professor
s1	take	c1
s1	take	c2
u1	member	s1
p1	author	s1
p5	author	f1

s1	type	Student
s1	take	c1
s1	teach	c10
u1	member	**s1**
p1	author	**s1**
p5	author	**s1**

c1	type	Course
s9	take	**c1**
s1	take	**c1**

1001	0101	u1	s1

1001	0101	p1	s1
1001	0101	p5	s1

1011	1101	s9	c1
1011	1101	s1	c1

File ID	Domain	Predicate	Range
1		member	Student
3	University	member	Student
4		author	Student
6		take	Course

Figure 26: Example showing the data flow in an object join partitioning job.

The aggregation step can be performed using a single machine because the number of predicates is generally not high, and thus they can be accommodated in the memory. However, this is not the case when we consider a list of URI strings. Counting the frequency of URI string during the Reduce phase requires the maintenance of URI strings in the memory and the frequency must be updated whenever a URI string is encountered. In most cases, the memory is not sufficient with large datasets. Fortunately, we do not have to acquire an exact index catalog without missing any data. The absence of some predicates does not necessarily mean that triples with predicates are omitted in the index because it would be sufficient to store the triples in an index file such as “unknown predicate.” In addition, some frequent URI strings could be missing. Thus, in that case, triples with a URI string need to be stored in a fully qualified form rather than an abbreviated form with a namespace prefix. This observation allows us to design a faster version of the index catalog acquisition process. We need to allow some triples to be skipped randomly when reading input triples during the Map phase. In the MapReduce framework, some lines can be skipped by investigating the line numbers in the given files. In terms of the input for the Map phase, the input key is the line number and the input value is a triple in the line (refer to Algorithm 1 for details).

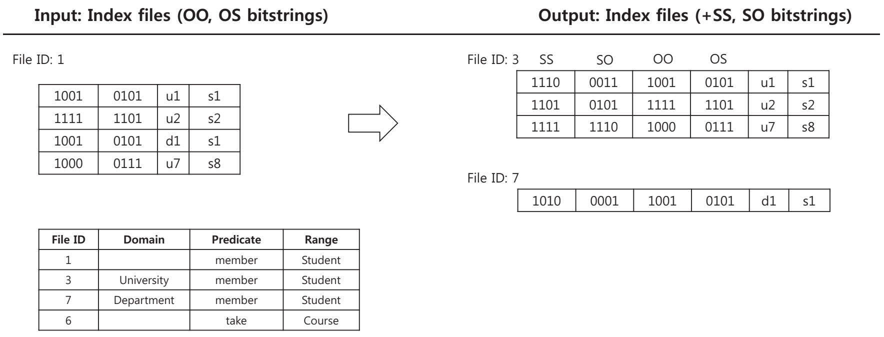
4.6.0.2 Object Join Partitioning

Object join partitioning is a MapReduce process that takes triples as its inputs and it then creates index files. By referring to the index catalog collected from the first job, we can store triples in a vertically partitioned manner. The input for this job is the same as that with the first job, i.e., triples in N-Triple formats. Figure 26 shows an example of the data flow in this process. During the Map phase, each triple is sent twice to two different Reduce jobs where subject and object are taken as the Map Key. In the Reduce phase, whenever `subject-key triples` are encountered, the URI strings in object are taken into the signature encoder. For example, we have three `subject-key triples` in the Reduce job called `s1`. The URI strings in object position of these triples are `Studnet`, `c1` and `c10`. By taking these URI strings as inputs, the signature encoder generates an `OS` bits string (`0101`). The bits string is then only appended to `object-key triples`. This allows each `object-key triple` to contain information about the `OS` joined triples. The bits strings and triples are stored in a corresponding file, which is determined by its predicate. To determine the index file where the triple (u1,member,s1) should be stored, the index catalog is scanned to find the match where predicate is member and the range is Student. Thus, the file ID 1 is found. Note that the domain of member cannot be determined in this step. Therefore, we must store the triples in index files with an empty domain, which indicates any domain. In the next job explained in Section 4.6.0.3, this will be split further by its domain.

It should be noted that in a Reduce job, the OO and OS bits strings that are the same as each other are appended to each object-key triple. This is based on the fact that each object in object-key triples is the same as that of the others, which is the Reduce key. Automatically, this means that OS joined triples are the same. It should also be noted that in this job, subject-key triples are not emitted to certain index files whereas object-key triples with bits strings were emitted and stored in their corresponding index files. However, this does not mean that some triples are not stored in any index files. For example, the triple (s1, take, c1) is ignored in the s1 Reduce job but it is emitted from the c1 Reduce job. For further details, please refer to Algorithm 1 and Algorithm 3 in the Appendix.

4.6.0.3 Subject Join Partitioning

Subject join partitioning is a MapReduce process that takes the entire index file created from the previous job (object join partitioning) as its input and it creates another index file. Figure 27 shows the input and output of this job. In the input index files, the triples have been stored based on the


Input: Index files (OO, OS bitstrings)

File ID: 1

1001	0101	u1	s1
1111	1101	u2	s2
1001	0101	d1	s1
1000	0111	u7	s8

File ID	Domain	Predicate	Range
1		member	Student
3	University	member	Student
7	Department	member	Student
6		take	Course

Output: Index files (+SS, SO bitstrings)

File ID: 3

SS	SO	OO	OS		
1110	0011	1001	0101	u1	s1
1101	0101	1111	1101	u2	s2
1111	1110	1000	0111	u7	s8

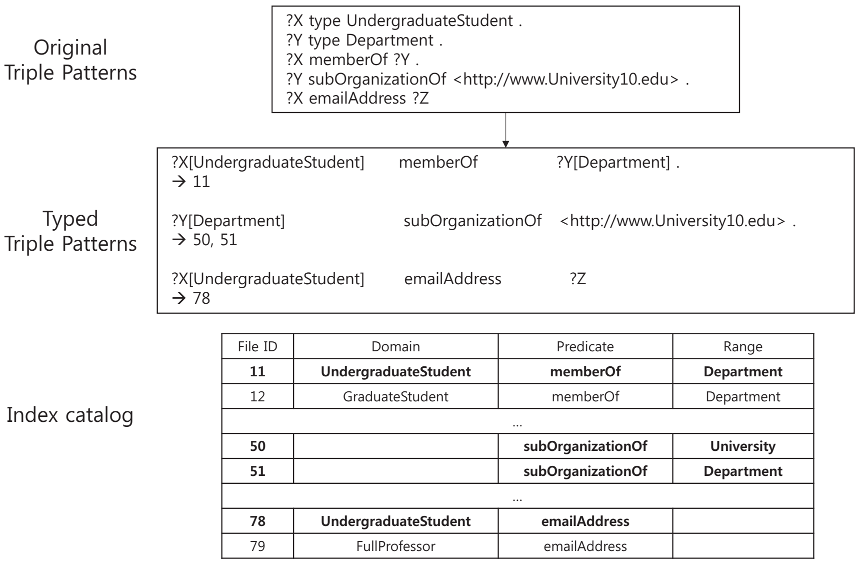
File ID: 7

1010	0001	1001	0101	d1	s1

Figure 27: Example showing the data flow in a subject join partitioning job. Only the input and output are depicted. The detailed procedure is similar to the object join partitioning so we omit the details.

range of predicate with the `OO` and `OS` bits strings. For example, the index file ID 1 contains triples where predicate is `member` and the type of object is `Student`. Subject join partitioning is intended to split the triples based on the domain of predicate and it also computes the bits string on the `SS` and `SO` join. In this case, it will create two index files (File ID 3 and 7 in Figure 27) where the domains correspond to `University` and `Department`, respectively. The `SS` and `SO` bits strings computed during this job are also appended to each triple. We will not explain how to compute the `SS` and `SO` bits strings in detail because this procedure is similar to that employed in the object join partitioning.

Let us give a brief description of subject join partitioning. The `object-key triples` are considered instead of the `subject-key triples` to obtain the `SS` and `SO` bits strings. The bits string is then appended to the `subject-key triples` instead of the `object-key triples`. Note that the `subject-key triples` already have the `OO` and `OS` bits strings


Original
Triple Patterns
?X type UndergraduateStudent .
?Y type Department .
?X memberOf ?Y .
?Y subOrganizationOf <http://www.University10.edu> .
?X emailAddress ?Z
Typed
Triple Patterns
?X[UndergraduateStudent] memberOf ?Y[Department] .
→ 11
?Y[Department] subOrganizationOf <http://www.University10.edu> .
→ 50, 51
?X[UndergraduateStudent] emailAddress ?Z
→ 78
Index catalog

File ID	Domain	Predicate	Range
11	**UndergraduateStudent**	**memberOf**	**Department**
12	GraduateStudent	memberOf	Department
…			
50		**subOrganizationOf**	**University**
51		**subOrganizationOf**	**Department**
…			
78	**UndergraduateStudent**	**emailAddress**	
79	FullProfessor	emailAddress	

Figure 28: Example showing the data flow during query analysis, which can be performed on a single machine without MapReduce jobs.

appended from the previous job, which allows this job to obtain four bits strings for each triple. For further details, please refer to Algorithm 5 and Algorithm 6 in the Appendix.

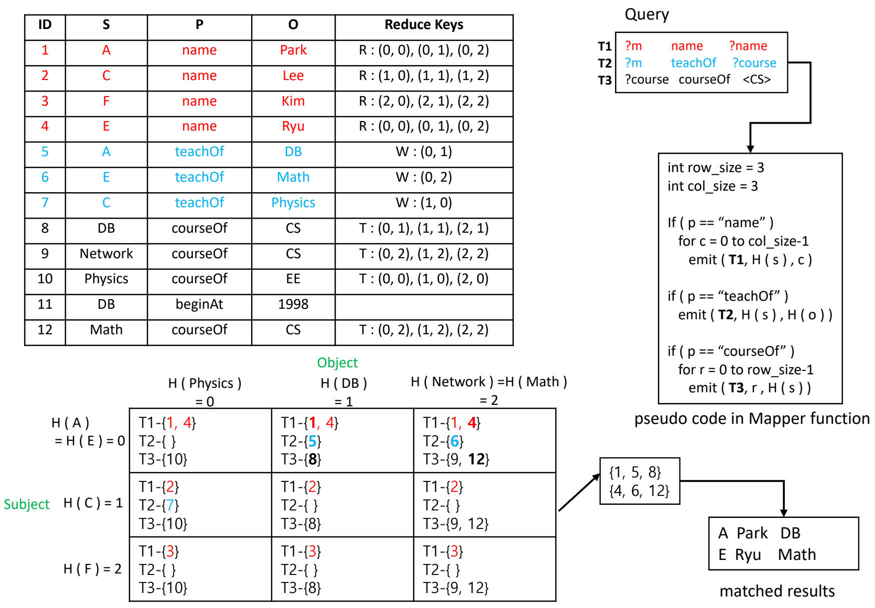
4.7 Query Processing

Query processing involves two steps such as query analysis and multi-way join. During the query analysis step, which is performed on a single machine, the domains and ranges of predicates appearing in triple patterns in input SPARQL query are identified in order to load the index files selectively. During the multi-way join step, the triples read from the index files are sent to Reduce jobs, which emits the matched results.

Query Analysis The aim of this step is to select index files that are relevant to the input SPARQL query. Figure 28 depicts an example of the data flow during query analysis. There are five triple patterns, which comprise two triple patterns with `type` predicate. The object in the `type` triple patterns is bound to a variable in another triple pattern, which corresponds to the subject position. For example, from the first triple pattern, we know that the type of the variable `?X` in the 3rd and 5th triple patterns is `UndergraduateStudent`. Thus, we have three `typed` triple patterns, as shown in the second box in Figure 28. The next step is to find the corresponding index files for each `typed` triple pattern by accessing the index catalog. In the case of the second triple pattern, the type of object is unknown which is actually a constant URI string. The file IDs of 50 and 51 match the triple pattern because, in order to find triples that match with the triple pattern, we have to scan each index file where predicate is `subOrganizationOf` regardless of its domain and range.

Multi-way Join The selected index files are taken as inputs to perform a MapReduce job for query processing. Figure 29 shows the overall MapReduce job process for query processing based on a multi-way join. To make this easier to understand, we first introduce the working principle of query processing based on a multi-way join [7]. We then explain our pruning schema in the next subsection.

Suppose that we have 12 triples and the input SPARQL query has three triple patterns with two joining variables, i.e., `?m` and `?course`. A two-dimensional `join matrix` helps us to understand how a multi-



ID	S	P	O	Reduce Keys
1	A	name	Park	R : (0, 0), (0, 1), (0, 2)
2	C	name	Lee	R : (1, 0), (1, 1), (1, 2)
3	F	name	Kim	R : (2, 0), (2, 1), (2, 2)
4	E	name	Ryu	R : (0, 0), (0, 1), (0, 2)
5	A	teachOf	DB	W : (0, 1)
6	E	teachOf	Math	W : (0, 2)
7	C	teachOf	Physics	W : (1, 0)
8	DB	courseOf	CS	T : (0, 1), (1, 1), (2, 1)
9	Network	courseOf	CS	T : (0, 2), (1, 2), (2, 2)
10	Physics	courseOf	EE	T : (0, 0), (1, 0), (2, 0)
11	DB	beginAt	1998	
12	Math	courseOf	CS	T : (0, 2), (1, 2), (2, 2)

Query

```
T1  ?m       name      ?name
T2  ?m       teachOf   ?course
T3  ?course  courseOf  <CS>
```

```
int row_size = 3
int col_size = 3

If ( p == "name" )
  for c = 0 to col_size-1
    emit ( T1, H ( s ) , c )

if ( p == "teachOf" )
  emit ( T2, H ( s ) , H ( o ) )

if ( p == "courseOf" )
  for r = 0 to row_size-1
    emit ( T3, r , H ( s ) )
```

pseudo code in Mapper function

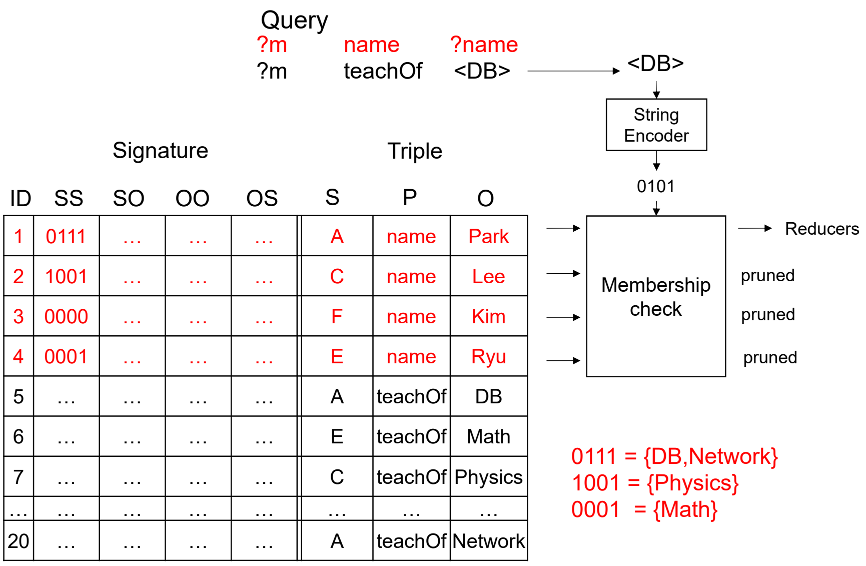
Subject \ Object	H (Physics) = 0	H (DB) = 1	H (Network) =H (Math) = 2
H (A) = H (E) = 0	T1-{1, 4} T2-{ } T3-{10}	T1-{1, 4} T2-{5} T3-{8}	T1-{1, 4} T2-{6} T3-{9, 12}
H (C) = 1	T1-{2} T2-{7} T3-{10}	T1-{2} T2-{ } T3-{8}	T1-{2} T2-{ } T3-{9, 12}
H (F) = 2	T1-{3} T2-{ } T3-{10}	T1-{3} T2-{ } T3-{8}	T1-{3} T2-{ } T3-{9, 12}

Figure 29: Example showing the data flow in the MapReduce framework when a multi-way join is performed for a given SPARQL query with two joining variables. H denotes a hash function that takes a string as input.

way join works within the MapReduce framework, where the dimension is attributable to the number of joining variables. The size of the `join matrix` is predetermined, i.e., a 3×3 `join matrix` in the example depicted in Figure 29. In the experimental section, we demonstrate how the size parameter affects the performance. The joining variables correspond to the column and row, i.e., `?course` and `?m`, respectively. The integer number in brackets in each cell in the `join matrix` indicates the ID of the triple. The existence of three triple patterns means that each cell has three different tags for triples, which are denoted by `R`, `W`, `T`. Each cell of the `join matrix` corresponds to a Reduce job, each of which checks whether the triples are joined during Reduce phase.

During the Map phase, according to the MapReduce framework, only one triple in index files is read each time line by line, but without any information about the other triples (before or after). For each triple, a set of Reduce Keys that comprises two integer elements (`row`, `column`) is generated by considering the triple patterns in the input query. For example, #1 triple (`A`, `name`, `Park`) is matched with the first triple pattern by its `name` predicate. Because `?m` is a joining variable, we can map to a row by applying a hash function to `A`. In this case, we have `0`, which is mapped to the 0-th row. Because `?name` is not a joining variable, `Park` should not be mapped to any specific column. Instead, `Park` is distributed to every column. Thus, three Reduce Keys are generated, i.e., (`0, 0`), (`0, 1`), (`0, 2`). Next, we explain another case of #5 triple (`A`, `teachOf`, `DB`). The second triple pattern is matched with the triple. Because the triple pattern has two joining variables, exactly one cell can be assigned, by mapping `A` to the 0-th row and `DB` to the 1-th column. Applying the same procedure to every triple, nine Reduce jobs will be created.

Let us now demonstrate the work-flow in Reduce jobs that corresponds to one cell in the `join matrix`. Because each triple is assigned a tag (`R`, `W` or `T` in this example), we know that no matched data will be identified in the case where at least one of the tags is missing. For example, in the first cell (`0, 0`), we have two triples tagged with `R`, one with `T`, but none with `W`. Without any further processing, we can terminate the Reduce job immediately because no triple is available that matches with the second triple pattern, i.e., (`?m teachOf ?course`). Next, we consider the cell (`0`,`1`),



	Signature				Triple		
ID	SS	SO	OO	OS	S	P	O
1	0111	…	…	…	A	name	Park
2	1001	…	…	…	C	name	Lee
3	0000	…	…	…	F	name	Kim
4	0001	…	…	…	E	name	Ryu
5	…	…	…	…	A	teachOf	DB
6	…	…	…	…	E	teachOf	Math
7	…	…	…	…	C	teachOf	Physics
…	…	…	…	…	…	…	…
20	…	…	…	…	A	teachOf	Network

Figure 30: Part of the data flow from Figure 29 related to the pruning of triples. Triples are read one by one and their bits strings are compared with the object in joined triple patterns. The unmatched triples will be pruned (in this case, the triple of ID 2,3,4).

which is a different case, where every tag is assigned to at least one of the triples. In this case, we need to check whether the triples are indeed joined. For example, there are two possibilities: (#1, #5, #8) and (#4, #5, #8), i.e., a sequence of triple IDs from `R`, `W`, `T`, respectively. Triples #1 and #5 share the same subject whereas #4 and #5 do not, thereby allowing us to decide that (#1, #5, #8) is a valid result. The checking criteria is based on the fact that the first and second triple pattern in the query are joined in the subject position (`?m`).

Pruning During the Map Phase In this section, we provide details of how to prune triples that have not been addressed by previous related works.

Figure 30 shows part of the procedure related to pruning, which corresponds to the scanning of four triples read from an index file containing triples having `name` predicate. For the readability, we show predicates in the figure, which are not actually included in the index files. The first triple pattern, i.e., (`?m name ?name`), is taken into account here. The second triple pattern is also considered because it is the `SS` joined with the first triple pattern. A bits string is obtained from the URI string (`DB`) in object in the `SS` joined triple pattern, which is denoted as a query bits string. Whenever a triple is read, the `SS` bits string is taken and compared with the query bits string to check whether the `SS` bits string is present in the query bits string. If it is present, the triple is sent to the Reduce jobs. If it is not present, the triple is pruned. In this case, the triples #2,#3 and #4 are pruned because the query bits string (`0101`) is not included in any of these `SS` bits strings. On the other hand, the query bits string is included in the `SS` bits string of the triple #1 (`0101`), which is sent to reducers. This way allows to reduce the number of triples that are unnecessarily sent to reducers.

4.8 Theoretical Analysis

In this section, the efficiency of the proposed approach is analyzed by proposing a cost model that is defined in the context of the size of signatures, `join matrix`, and the number of pruned triples. Then, we discuss the correctness of the pruning technique.

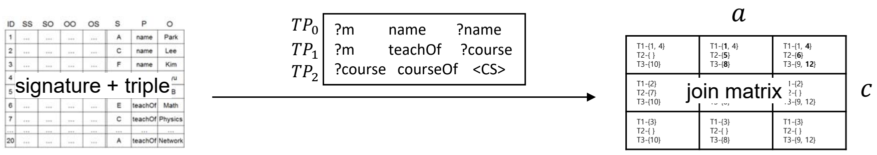


Figure 31: Factors that are relevant to the cost of SigMR

4.8.1 Cost Model

The performance is theoretically analyzed in this section based on a multi-way join analysis proposed in [7]. Consider the example query Q in Figure 31. Let G be the target RDF data. Let $M(TP_1)$ be a set of triples that are matched to the first triple pattern (?m,`name`,?`name`), $M(TP_2)$ for the second triple pattern (?m,`teachOf`,?`course`), and $M(TP_3)$ for the third triple pattern (?`course`,`courseOf`,<`CS`>). Let k be the size of the $a \times b \times c$ `join matrix`, which means that $abc = k$. b will be set to 1 for the query Q since it has two joining values ?`m` and ?`course`. A hash function is used to map values in subject to one of a rows and values in object to one of c columns in the `join matrix`.

In addition, we have the parameters associated with the pruning technique. Let $P(TP_1, G)$ be the proportion of remaining triples to the number of original triples matched to TP_1. If $P(TP_1, G) = 1$, then no triple is pruned. If it is 0.5, then 50% triples are pruned. To prune triples during map phase, signatures are read for each triple. Thus, we need to take into account the cost for reading signatures, denoted as s.

We define the cost of multi-way join as the summation of the number of triples that are read during map phases and the number of triples that are sent to reducers, which is defined in Definition 4.1.

$$\begin{aligned} cost_{base} = {} & (M(TP_1) + M(TP_2) + M(TP_3)) \\ & + aM(TP_1) + bM(TP_2) + cM(TP_3) \end{aligned} \tag{4.1}$$

The cost of SigMR includes additional factors as defined in Definition 4.2.

$$\begin{aligned} cost_{\mathsf{SigMR}} = {} & (M(TP_1) + M(TP_2) + M(TP_3)) + s(M(TP_1) + M(TP_2) + M(TP_3)) \\ & + aM(TP_1)P(TP_1, G) + bM(TP_2)P(TP_2, G) + cM(TP_3)P(TP_3, G) \end{aligned} \tag{4.2}$$

SigMR is efficient than the conventional one if the following Expression holds.

$$cost_{\mathsf{SigMR}} < cost_{base} \tag{4.3}$$

Expression 4.3 is re-written as follows.

$$\begin{aligned} & (M(TP_1) + M(TP_2) + M(TP_3)) + s(M(TP_1) + M(TP_2) + M(TP_3)) \\ & + aM(TP_1)P(TP_1, G) + bM(TP_2)P(TP_2, G) + cM(TP_3)P(TP_3, G) \\ & - (M(TP_1) + M(TP_2) + M(TP_3)) - (aM(TP_1) + bM(TP_2) + cM(TP_3)) < 0 \end{aligned} \tag{4.4}$$

If we reduce the formula, we get the following expression.

$$\begin{aligned}(s+a(P(TP_1,G)-1))M(TP_1) \\ +(s+b(P(TP_2,G)-1))M(TP_2) \\ +(s+c(P(TP_3,G)-1))M(TP_3)<0\end{aligned} \tag{4.5}$$

If we prefer to strict constraints, the following conditions are derived.

$$\begin{aligned} s &< a(1-P(TP_1,G)) \\ s &< b(1-P(TP_2,G)) \\ s &< c(1-P(TP_3,G)) \end{aligned} \tag{4.6}$$

The expression suggests that if we increase the size of signatures, which corresponds to s, we also need to increase the size of `join matrix` in order to obtain smaller cost than the baseline cost. On the other hand, we cannot increase k much due to the restricted number of worker machines. Therefore, if we want to keep k smaller, then we have to increase the pruning power $P(T_i,G)$ to satisfy the expression for a given s.

The expression is generalized to arbitrary query as stated in Lemma 3.

Lemma 3. SigMR is efficient than the conventional one if

$$s < d_i(1-P(TP_1,G)) \text{ for all } i$$

, where i is the number of joining values in query and d_i is the number of slots in *i*th dimension of join matrix.

4.8.2 Correctness

In the repetitive multi-way join strategy employed in the proposed approach, mappers send replicated triples to reducers. Equi-join operations are then performed by reducers to obtain resultant tuples matched to the query. Signatures are considered by mappers to prune triples. We first need to prove that signatures does not yield a false positive answer. For the readability, we only prove that the bit string for a set of strings does not answer a false positive for a membership check of a string.

Theorem 4.8.1. Suppose we obtain a bit string $Bit(S)$ from a set S of strings as follows:

$$Bit(S) = h(s_0) \diamond h(s_1) \diamond ... \diamond h(s_{n-1})$$

, where $s_i \in S$, $h(s)$ is a hash function that maps a string s to a bit sequence, and $\diamond$ is a bit-wise OR operator for two bit sequences.

We can define a membership check function *isMember* as follows:

$$isMember(Bit(S), w) = \begin{cases} \text{TRUE} & \text{if } Bit(S) \diamond w = Bit(S) \\ \text{FALSE} & \text{if } Bit(S) \diamond w \neq Bit(S) \end{cases}$$

isMember basically answers TRUE for $w \in S$ and FALSE for $w \notin S$. It sometimes answers TRUE for $w \notin S$, but will never answer FALSE for $w \in S$.

Proof. We prove by contradiction. We assume $isMember(Bit(S), t) = FALSE$

for a string $t \in S$. By definition, we know that $Bit(S) = h(s_0) \diamond ... \diamond h(t) \diamond ... \diamond h(s_{n-1})$. It is trivial that $Bit(S) \diamond w = Bit(S)$. This prove that it will never answer FALSE for $w \in S$.

$\square$

Now we prove that our pruning technique produce the correct resultant tuples as Theorem 4.8.2.

Theorem 4.8.2. Suppose a query Q is given, which consists of n triple patterns $\{T_1, ..., T_n\}$. Let $M(T_i)$ be a set of triples matched to T_i. Let $M^P(T_i)$ be a set of triples matched to T_i subtracted by a set of triples pruned by a pruning technique P. We showed that the pruning technique P is correct in Theorem 4.8.1. In addition, we assume that the equi-join operator $\bowtie$ is sound and complete, then we have

$$M(T_1) \bowtie M(T_2) \bowtie ... \bowtie M(T_n) \Leftrightarrow M^P(T_1) \bowtie M^P(T_2) \bowtie ... \bowtie M^P(T_n)$$

Proof. Let A be $M(T_1) \bowtie ... \bowtie M(T_n)$ and B be $M^P(T_1) \bowtie ... \bowtie M^P(T_n)$. Suppose we have two triples $t_0(x,y,z) \in M(T_1)$ and $t_1(x,s,t) \in M(T_2)$. Suppose that T_1 is joined with T_2 in subject. The other joining cases can be proved similar ways.

The left to right direction ($\Rightarrow$): We assume a resultant tuple $u_0(x,y,z,s,t,...)$ obtained by joining these two triples exist in A. By contradiction, we assume that $u_0 \notin B$. Since $\bowtie$ is correct, there are two cases, where $t_0(x,y,z) \notin M^P(T_1)$ and $t_1(x,s,t) \notin M^P(T_2)$. Both cases is impossible since P is sound and complete.

The right to left direction ($\Leftarrow$): We assume a resultant tuple $u_0(x,y,z,s,t,...)$ obtained by joining these two triples exist in B. By contradiction, we assume that $u_0 \notin A$. Since $\bowtie$ is correct, there are two cases, where $t_0(x,y,z) \notin M(T_1)$ and $t_1(x,s,t) \notin M(T_2)$. Both cases is impossible since $M(T_1)$ and $M(T_2)$ must contain all triples associated with resultant tuples.

$\square$

4.9 Experiments

We performed experiments to demonstrate that our approach reduces the query execution time compared with existing approaches. In addition, we examined the effectiveness of some of the parameters used in our approach.

Systems Compared We selected systems that could be compared with our approach, which is based on MapReduce without any other capabilities. We wanted to isolate the impact of our techniques as much as possible, i.e., signature encoding-based multi-way join. Thus, we selected `HadoopRDF`, which is a state-of-the-art SPARQL query processing system based on MapReduce. We also compared our system with MultiMR, which is a variant of our approach but without signature encoding. These systems are all based on the MapReduce framework. Because any implementation of the MapReduce framework is possible, we used Apache Hadoop, which is one of the most popular MapReduce implementations. In particular, we used Hadoop version 2.4.0. Unfortunately, HadoopRDF was implemented using a previous version of Apache Hadoop. Thus, to ensure a fair comparison, we modified some of the Hadoop version-dependent codes in HadoopRDF to allow it to work with Hadoop 2.4.0.

Hardware Settings We used five machines running the 64-bit version of CentOS 6.5 with a Quad-Core 2.80 GHz CPU, 10 GB RAM, and 2 TB disk. We assigned 8 GB RAM to each slave for Apache Hadoop. We used the 64-bit version of Java Virtual Machine 1.8 to build and run the systems.

Datasets Used `LUBM` (Lehigh University Benchmark) datasets[4] were used in the experiments. To perform the experiments with different volumes of data, the `LUBM` dataset generator was used to obtain `LUBM` datasets where the number of universities varied, i.e., `LUBM10`, `LUBM100`, and `LUBM1000` contains approximately 1 million, 13 million, and 130 million triples, respectively.

4.9.1 Index Building Time and Space Requirements

The space requirements and index building time are shown in Table 3. HadoopRDF required the least space whereas SigMR required the most space. This is because the index structure of SigMR has additional bits strings that are not required by HadoopRDF. MultiMR also required more space than HadoopRDF, although MultiMR does not maintain bits strings. This is because vertical partitioning methodologies are managed slightly

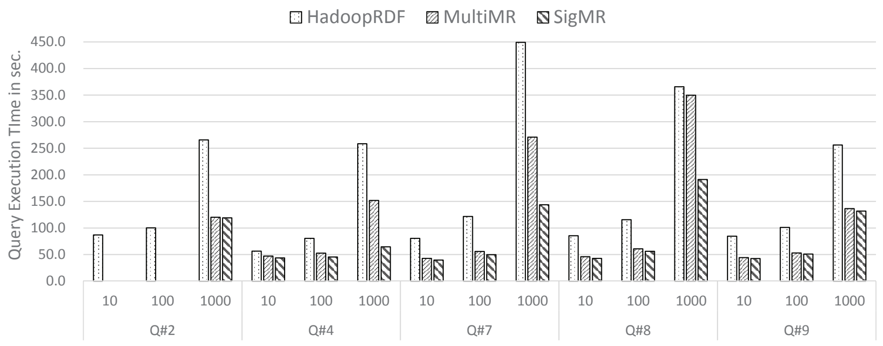
[4]http://swat.cse.lehigh.edu/projects/lubm

Table 3: Index building time and space requirements. Bits indicates the size of signature.

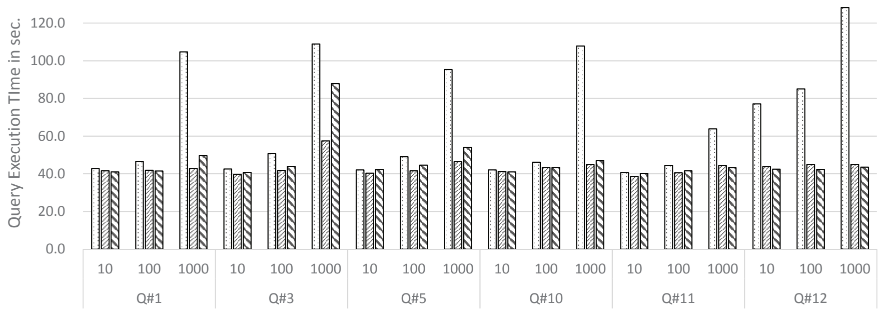
		LUBM10		LUBM100		LUBM1000	
	Bits	Time (sec)	Space (GB)	Time (sec)	Space (GB)	Time (sec)	Space (GB)
SigMR	64	239	0.2	618	3.2	5,171	33.0
	128	241	0.4	724	5.1	6,645	51.9
	256	251	0.7	709	8.8	6,196	89.3
MultiMR	-	284	0.1	553	1.4	4,007	14.0
HadoopRDF	-	150	0.1	296	1.3	2,424	12.9

differently, i.e., MultiMR splits triples by range and then again by domain whereas HadoopRDF splits only by range. Thus, MultiMR stores more triples because some URI resources have multiple types. For example, in the LUBM datasets, a resource called `GS0` could have two types: `GraduateStudent` and `TeachingAssistant`. In MultiMR, a triple (`GS0`, `teachingAssistantOf`, `Course11`) is stored in two files with distinct domains, i.e., `GraduateStudent` and `TeachingAssistant`. By contrast, in HadoopRDF, a triple is stored in a file only once because it does not distinguish the domain of subject.

Although SigMR and MultiMR were not more efficient than HadoopRDF in terms of their index building time and space requirements, their faster query execution time, as shown in the next section, compensated for these disadvantages. After the index is built during a preprocessing stage, we are no longer concerned with the index building time. In addition, the price of disks is becoming cheaper with time, thereby making the query execution time more crucial for real-world applications.



(a) LUBM query #2,#4,#7,#8,#9



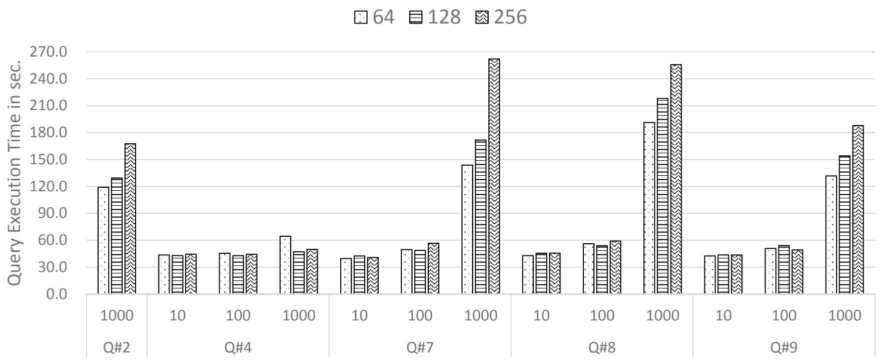
(b) LUBM query #1,#3,#5,#10,#11,#12

Figure 32: Query execution times averaged over three experiments against `LUBM10`, `LUBM100`, and `LUBM1000` datasets. Two figures are provided to improve the readability. The parameter settings used were as follows: number of bits = 64, the size of `join matrix` = 3.

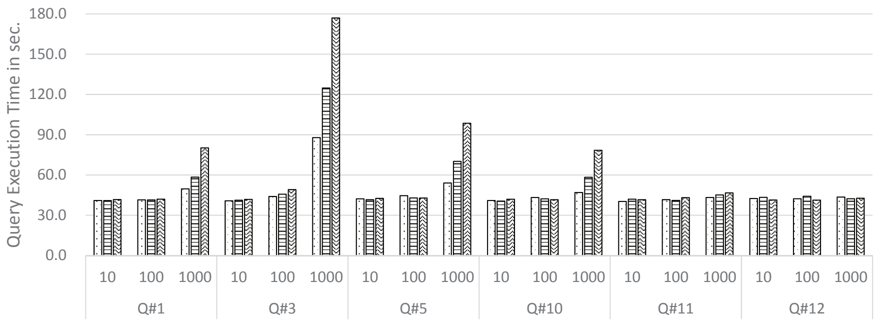
4.9.2 Query Execution Time

The query execution time is defined as the time from when a system receives a query until it finishes saving results in a local file. The SPARQL queries used in the present study were `LUBM` queries. Of the 14 `LUBM` queries, we only included queries having joins. In addition, in order to exclude the inference performance, which is outside the scope of the present study, the URI resources in the queries were replaced with the most specific ones. For example, if `Person` was present in a query such as `(?s rdf:type Person)` and `Person` is not the most specific class, it was replaced with `GraduateStudent`, which is the most specific class and a subclass of the `Person` class. The queries used are listed in Appendix 6.

Figure 32 shows the query execution times. For all of the queries, SigMR performed better than HadoopRDF. The effect of our pruning schema can be seen for Q#4, Q#7 and Q#8 which are rather of complex queries. For Q#3, Q#5 and Q#10, MultiMR performed better than SigMR. The existence of only one or two triple patterns in these queries accounts for these results. In other words, the pruning didn't help for these simple queries. Rather, it increased the running time because of that SigMR needs to check bits strings that is not the case in MultiMR. Note that there are no query execution time for Q#2 on `LUBM10` and `LUBM100`. It can be understood by their index structure: HadoopRDF has to scan some index files, whereas SigMR and MultiMR do not have to scan any index files. Splitting the triples further by domain in the index files allowed SigMR and MultiMR to determine that there was no result by looking only at the index catalog.



(a) LUBM query #2,#4,#7,#8,#9



(b) LUBM query #1,#3,#5,#10,#11,#12

Figure 33: Query execution time of SigMR against LUBM10, LUBM100, and LUBM1000 datasets. The query execution time is averaged over three experiments. The results are shown for variable numbers of bits in the bits strings. The parameter settings used were as follows: the size of join matrix = 3.

4.9.3 Effect of Signature Encoding

The query execution times with SigMR when using different numbers of bits in the bits strings are shown in Figure 33. We used three different number of bits, i.e., 64, 128, and 256. In most cases, 64 bits obtained the best performance, except for Q#4, which had the worst performance with 64 bits. Figure 34 helps us to understand why this was the case, i.e., in the case of Q#4, fewer triples were pruned with 64 bits than those with 128 and 256 bits. The lower number of triples pruned meant that the Reduce jobs received more triples, thereby increasing the query execution time. Thus, when fewer bits were used, the number of pruned triples was lower. However, this was not the case for Q#8 and Q#12. For Q#8, the URI string in the query, i.e., `<http://www.University10.edu>`, was too general and simple in terms of the degree of variations in generated bits string. The URI string was also simple in Q#4 but it had four triple patterns, excluding `type` predicate, whereas Q#8 had three triple patterns. For Q#7, there was a specific URI string, which allowed many triples to be pruned for 64 bits was used.

4.9.4 Effect of the Size of Join Matrix

The query execution time for SigMR also depends on the number of Reduce jobs and the number of triples sent to one Reduce job. The `join matrix` shown in Figure 29 is used to help us understand these issues. Each cell of the `join matrix` corresponds to a Reduce job. A larger `join matrix` leads to more Reduce jobs but with a smaller number of triples for each Reduce job. We define the size of `join matrix` as the number

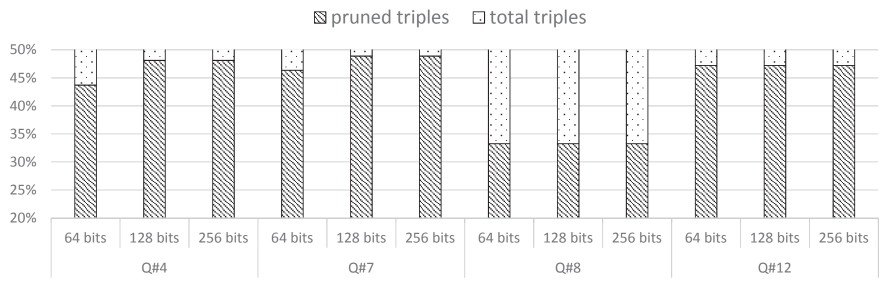
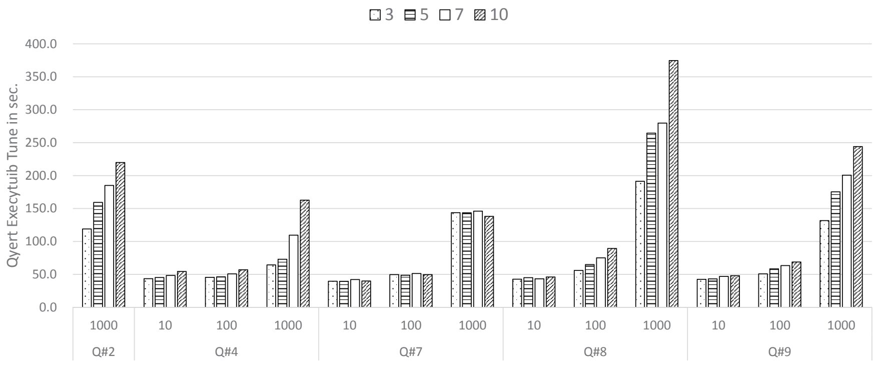
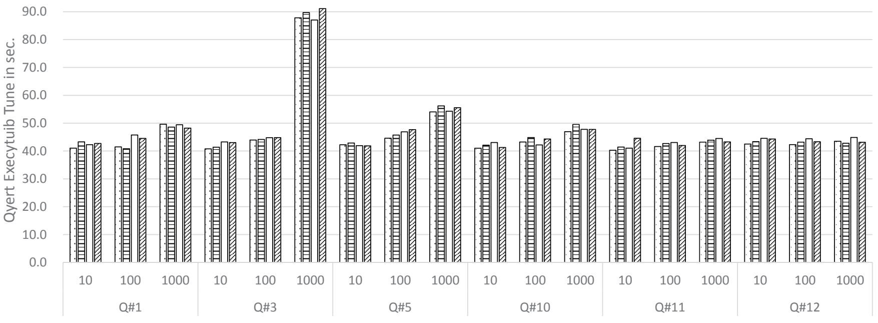


Figure 34: The proportion of pruned triples in the total triples that are originally considered in a multi-way join against `LUBM1000` dataset. The results are shown for three different bits string sizes. For example, 44% of triples were pruned out of the total triples (which is always 100% in this figure) when 64 bits were used and 47% of the triples were pruned when 128 bits were used.

of its rows and columns. A square matrix is used for `join matrix` in the present study. We set the size parameter as 3, 5, 7, and 10, as shown in Figure 35. In most cases, the 3 by 3 `join matrix` were sufficient to achieve the best performance. Significant differences were obtained for Q#2, Q#4, Q#8, and Q#9, which were relatively complex queries with many triple patterns. Note that the dimension of the `join matrix` is equal to the number of triple patterns. Thus, the size of the `join matrix` increases for complex queries, thereby creating many Reduce jobs. We found that many Reduce jobs with small number of triples did not help to reduce the execution time for complex queries. However, a larger `join matrix` could help to reduce the query execution time for simple queries. The best choice for the size of `join matrix` depends on the characteristics of the datasets and queries.



(a) LUBM query #2,#4,#7,#8,#9



(b) LUBM query #1,#3,#5,#10,#11,#12

Figure 35: Query execution time with SigMR averaged over three runs against each LUBM10, LUBM100, and LUBM1000 datasets. The results are shown for variable size of join matrix. The parameter settings used were as follows: number of bits = 64.

4.10 Conclusion

In this study, we proposed a SPARQL query processing engine based on the MapReduce framework, where a signature encoding technique is employed to prune triples when performing a multi-way join. The aim of the proposed method is to overcome the limitations of the join evaluation strategies employed in previous approaches, i.e., the two-way join approach involves costly disk accesses and the multi-way join approach is not scalable. We performed experiments to demonstrate that our proposed approach reduces the query execution time. We also showed that our signature encoding method helps to reduce the query execution time. We examined different parameter settings using our approach in order to show effects of parameters with respect to queries and datasets. Our interesting future research may include compressing the bits strings to minimize the space requirements and supporting the full-text search expressed in filter clauses. In addition, real-time processing frameworks such as storm [68] can be integrated in order to improve query processing performance [69, 70].

Chapter 5

Application to Biomedical Linked Data

5.1 Introduction

This chapter discusses an way of applying the proposed algorithms to biomedical Linked Data.

Firstly, we propose a way of integrating biomedical Linked Data, especially, for adverse drug combination discovery. Our model is designed to discover pathways associated with a given drug combination. The approach is different from previous works in that existing works focus on finding relevant genes whereas our work focus on finding a path linked through pathways. Our approach not only shows more complex information but also helps clinicians understand how a drug combination yield some adverse effects.

Secondly, we describe CyHadoop, a Cytopscape plugin that works with Hadoop cluster running on multiple machines. The input data given by user through Cytoscape is automatically fed into Hadoop cluster and its result is in turn shown in the Cytoscape visualizer. It also provides graphical user interfaces that allows users to check the status of Hadoop cluster's. We argue that availability of both visualization and bigdata processing in a software, provides convenience to biologist.

5.2 Related Work

To the best of our knowledge, few Cytoscape plugin exists that can handle large volume of data. Most the plugins run on a single machine which is hard to be scalable due to the memory and CPU limitation. Big data in biology domain includes UniProtKB/Swiss-Prot (2.6 GB), GeneOntology (4.7 GB) and etc. Capabilities of processing such a bigdata are becoming an increasingly important role in biology [71].

Apache Hadoop, a bigdata processing framework, is a distributed computing framework that implements MapReduce [14], which parallelizes data processing tasks on a cluster with multiple machines. Pre-configured Hadoop clusters are provided by Amazon Web Services (AWS) that gives an opportunity for biologits to make use of distributed computing environment without detailed knowledge.

Researchers have tried to use Hadoop to address heavy computation problem in biology. For example, SeqPig [72] extends Apache Pig, a script for executing Hadoop tasks, to make it an easy task to run a sequence alignment task in Hadoop. MR-Tandem ([73]) is a peptide search engine written in a Python script that works with Hadoop cluster. BlueSNP [74] is a R package performing a statistical test of genome-wide association study on Hadoop cluster.

Even though Hadoop has already been used in biology domain in various ways, few researcher has attempted to enable visualization tools to work with Hadoop. We could get the benefit both of faster processing of bio big-data by using Hadoop and easy checking of the processing result by using Cytoscape.

Diverse Cytoscape plugins are available in Cytoscape App Store[1]. We review Cytoscape plugins that are related to deal with biological network in terms of graph processing, which is listed in Table 4.

Table 4: Cytoscape plugins related to biological network and graph processing

Name	Main Features	Cluster Support	Update
ShortestPath	Visualize shortest path between two nodes	NO	2007
PathExplorer	Highlight paths	NO	2012
StrongestPath	Find confident paths in PPI network	NO	2015
KeyPathwayMiner	Extract all maximal connected sub-network	NO	2016
Vital AI Graph Visualization	Full-text search for nodes by its name using WordNet	YES (cluster provided by the company)	2016
CyHadoop	Find shortest paths that are associated with adverse drug combinations	YES (Hadoop Cluster)	2016

Most plugins does not support cluster. In other words, internal algorithms run on a single machine in an in-memory fashion. If memory is not enough, large graph cannot be handled. `Vital AI Graph Visualization` should not be considered to fully support clusters because the users have to connect to the company's cluster. In addition, it does not provide function-

[1]`http://apps.cytoscape.org`

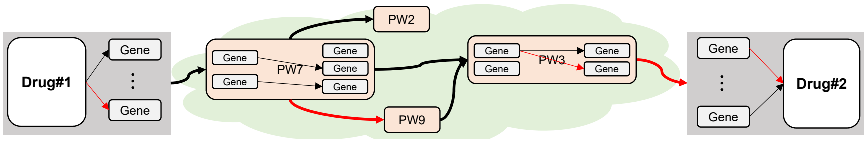


Figure 36: Conceptual diagram of integrating biomedical datasets related to adverse drug combination. The tick lines indicates links established by the proposed approach. The plain lines indicates existing links. The red line represents the inhibition (or down-regulation).

alities of graph processing, rather, it focus on full-text support based on WordNet.

5.3 Data Model

This section discusses how to integrate biomedical datasets to address the adverse drug combination problem. Figure 36 depicts our model.

In the left and right hand sides, we have two drugs, respectively. Each drug is linked to target genes. In the middle, we have pathways linked through genes. Within this model. a directed path can be discovered from a drug to another drug. Several pathways would be on the path, which reflects how two drugs interact each other. Note that a drug has target genes, which is modeled in directed edges from a drug to each gene. However, we also have edges from genes to a drug in the right-hand side if genes from a pathway down-regulates these genes. Using such a trick, we are able to define directed path from a drug to another drug. When a discovered path is shown to users, the direction of these edges can be reversed to give a correct information.

Biomedical data pertaining to adverse drug combination discovery is modeled in an edge-labeled directed graph as follows:

$$BG(D, G, PW, DG, GPW, PWPW, PWG, GD) \tag{5.1}$$

Notations are defined in Table 5.

Table 5: Notations

Notation	Meaning
D	a set of drugs
G	a set of genes
P	a set of proteins
PP	a set of relationships between proteins such that (p_i, up, p_j) or $(p_i, down, p_j)$ where $p_i, p_j \in P$
PW	a set of pathways
DG	a set of relationships from drugs to theirs' target genes such that $(d_i, target, g_j)$ where $d_i \in D$ and $g_j \in G$
GD	a set of relationships from gene to drugs such that $(g_j, inverse\_target, d_i)$ where $g_j \in G$ and $d_i \in D$
GPW	a set of relationships from gene to pathways such that (g_j, in, pw_i) where $g_j \in G$ and $pw_i \in PW$
PWG	a set of relationships from pathways to genes such that (pw_j, out, g_i) where $pw_j \in PW$ and $g_i \in G$
$PWPW$	a set of relationships between pathways such that (pw_i, up, pw_j) or $(pw_i, down, pw_j)$ where $pw_i, pw_j \in PW$

Basic building blocks of the graph BG are drugs, genes and pathways. We firstly define the pathway as stated in Definition 22.

Definition 22. Pathway (PW) A pathway pw is a *nested* directed hypergraph that consists of a set P of proteins and a set $PCPC$ of hyperedges (PC_s, PC_t) with a set PC_s of source proteins and a set PC_t of target proteins. Specifically, PC_s indicates to the protein complex consisting of a set of proteins that participate together in an interaction. The term *nested* is

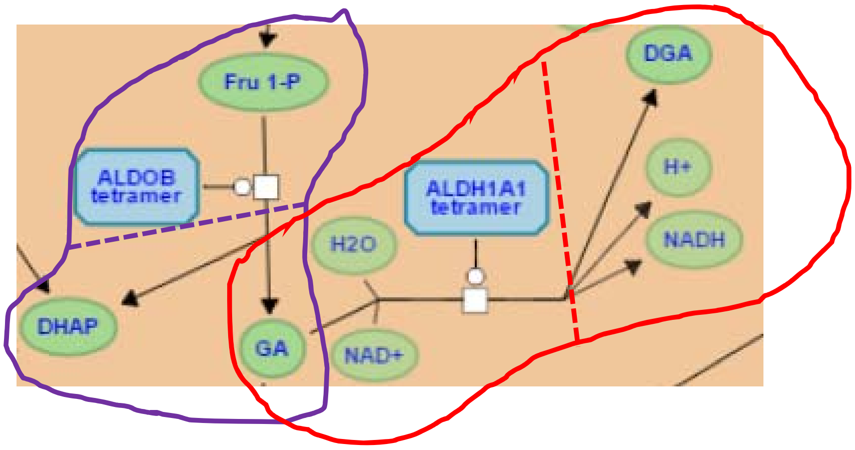


Figure 37: An illustrative example of input and output of a pathway.

used because a pathway as the whole becomes a node of *BG*. To simplify the model of *BG*, relationships between two protein complexes are grouped into two relationships, *up* and *down*. *PCPC* can be obtained from a pathway database.

To establish relationships between pathways, we need to define the input and output of pathways. We assume that signals are sent from a pathway to another pathway via interactions between proteins. Specifically, signals from a pathway pw_s is sent to another pathway pw_t if the output of pw_s is mapped into the input of pw_t. The formal definition of the input and output of a pathway is stated in Definition 23.

Definition 23. Pathway Input & Output The input $I(pw)$ of a pathway pw is a set of proteins in the source of hyperedges defined as

$$I(pw) = \{p : p \in PC_s \text{ such that } (PC_s, PC_t)\} \tag{5.2}$$

, where $pw = (P, PCPC)$ and $PCPC = \{(PC_s, PC_t)\}$ Similarly, the output $O(pw)$ is defined as

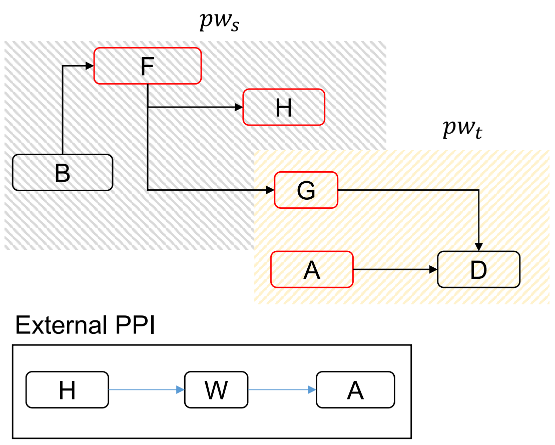
$$O(pw) = \{p : p \in PC_t \text{ such that } (PC_s, PC_t)\} \tag{5.3}$$

Figure 37 is a screenshot of part of a pathway taken from Reactome[2]. For the simplicity, we assume that the figure depict the whole of a pathway, consisting of two reactions. In the left-side, we have a reaction whose input is `Fru 1-P` and `ALDOB tetramer`, and whose output is `DHAP` and `GA`. In the right-side, we have another reaction whose input is `GA`, `H2O`, `NAD+` and `ALDH1A1 tetramer`, and whose output is `DGA` and `H+` and `NADH`. From these reactions, we identified input and output of the pathway. Input is `Fru 1-P, ALDOB tetramer, ALDH1A1 tetramer, GA, H2O, NAD+` and output is `DHAP, GA, DGA,H+,NADH`. Note that `GA` participates in both input and output of two reactions.

Edges between two pathways are established based on the degree to which how many protein-protein relationships are there. The protein-protein relationship is formally defined in Definition 24.

Definition 24. ***n*-Hop Protein-Protein Relationship** A protein p_s in a pathway pw_i has a 0-hop mapping with a protein p_t in a pathway pw_j, denoted as $p_s \xrightarrow{0} p_t$, if p_s and p_t are the same protein and $p_s \in O(pw_i)$ and $p_t \in I(pw_j)$. For a positive integer $n > 0$, p_s has a n-hop mapping with p_t, denoted as $p_s \xrightarrow{n} p_t$, if p_s and p_t are not the same one and $p_s \in O(pw_i)$ and $p_t \in I(pw_j)$,

[2]http://www.reactome.org/



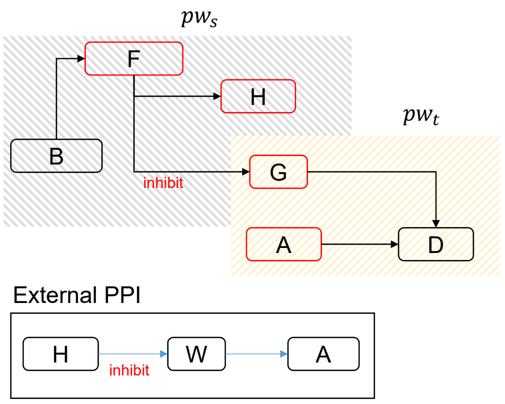
$O(pw_s)$	$I(pw_t)$	Hop
F	G	1
F	A	
H	G	2
H	A	
G	G	0
G	A	

Figure 38: An illustrative example of n-hop protein-protein relationship between two imaginary pathways.

but it satisfies one of two condition; ① (direct) there exists a n-length path from p_s to p_t in pw_i or pw_j. ② (indirect) there exists a set PI of proteins $\{p_k, ..., p_{k+n}\}$ where interactions (p_s, p_k), (p_k, p_{k+1}),..., (p_{k+n-1}, p_{k+n}), and (p_{k+n}, p_t) in an external protein-protein interaction database and $|PI| = n-1$.

See Figure 38, where the input and output of two imaginary pathways are shown. `G` is in both $O(pw_s)$ and $I(pw_t)$, which means that we have `G` $\xrightarrow{0}$ `G`. We have `F` $\xrightarrow{1}$ `G` because there exists a path from `F` to `G` in pw_s and the length is 1. An indirect relationship is found from `H` to `A` through external PPIs consisting of `H`, `W`, and `A`, shown in the bottom part of the figure.

The types of protein-protein relationships are distinguished by the way in which the source protein affects the target protein. Although there exist various ways of interaction between proteins, we classify into two interaction types such as *up* and *down*, as defined in Definition 25.



$O(pw_s)$	$I(pw_t)$	Hop	Type
F	G	1	DOWN
F	A		
H	G	2	DOWN
H	A		
G	G	0	UP
G	A		

Figure 39: An illustrative example of n-hop protein-protein relationship classification, extended from Figure 38

Definition 25. ***n*-Hop Protein-Protein Relationship Classification** (*PP*) Given a n-hop protein relationship $p_s \xrightarrow{n} p_t$, we classify it into two relationships $p_s \xrightarrow[up]{n} p_t$ and $p_s \xrightarrow[down]{n} p_t$. $p_s \xrightarrow{n} p_t$ is classified into $p_s \xrightarrow[down]{n} p_t$ if there exists at least one (p_k, p_{k+1}) in the path from p_s to p_t, where p_k down-regulates p_{k+1} according to a protein-protein interaction database. $p_s \xrightarrow{n} p_t$ is classified into $p_s \xrightarrow[up]{n} p_t$, otherwise. If n is 0, we assume $p_s \xrightarrow{n} p_t$ is always classified into $p_s \xrightarrow[up]{n} p_t$.

See Figure 39, where types of protein-protein relationships are shown. The relationship $\mathtt{F} \xrightarrow{1} \mathtt{G}$ is classified into *down* because F inhibit G in pw_s. Likewise, $\mathtt{H} \xrightarrow{2} \mathtt{G}$ is classified into *down* because one of PPIs in the path is *down* (i.e., H inhibit W).

We can calculate the proportion of the number of *up* and *down* protein-protein relationships to the total number of pairs between proteins in two pathways, as defined in Definition 26.

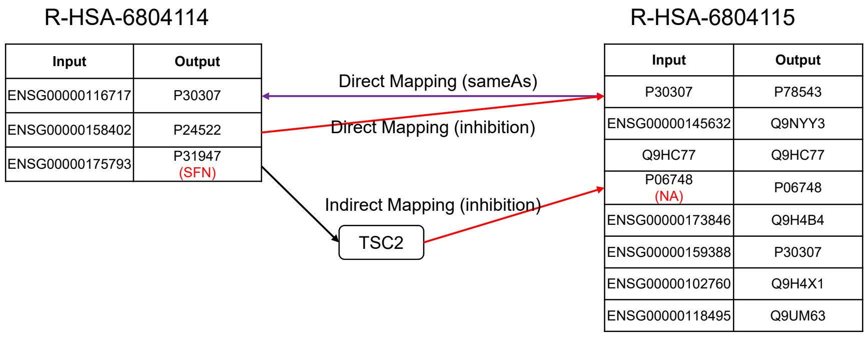


Figure 40: An illustrative example of establishing links between two real-world pathways from Reactome.

Definition 26. Pathway-Pathway Relationship ($PWPW$) We have two pathways pw_i and pw_j with $p_s \in O(pw_i)$ and $p_t \in I(pw_j)$. The weight of each n-hop protein-protein relationship is defined as $w_n = \frac{1}{n+1}$. We compute the following scores.

$$Up(pw_i, pw_j) = \frac{\sum w_n \times |p_s \xrightarrow[up]{n} p_t|}{|O(pw_i)| \times |I(pw_j)|} \tag{5.4}$$

$$Down(pw_i, pw_j) = \frac{\sum w_n \times |p_s \xrightarrow[down]{n} p_t|}{|O(pw_i)| \times |I(pw_j)|} \tag{5.5}$$

$$Link(pw_i, pw_j) = Up(pw_i, pw_j) + Down(pw_i, pw_j) \tag{5.6}$$

We establish an edge from pw_i to pw_j if $Link(pw_i, pw_j) > t$, where t is the threshold. The label of the edge between two pathways is assigned *up* if $Up(pw_i, pw_j) > Down(pw_i, pw_j)$ and *down*, otherwise.

See Figure 40, where protein-protein relationships are shown between two pathways. Because the proportion of *down* protein-protein relationships is greater than the proportion of *up* ones, we will assign *down* to the edge between two pathways.

So far, we have described the way to model relationships between pathways. We will discuss the way to model relationships between genes and pathways. Genes can be connected to pathways via proteins. The relationship between genes and pathways is formally defined in Definition 27.

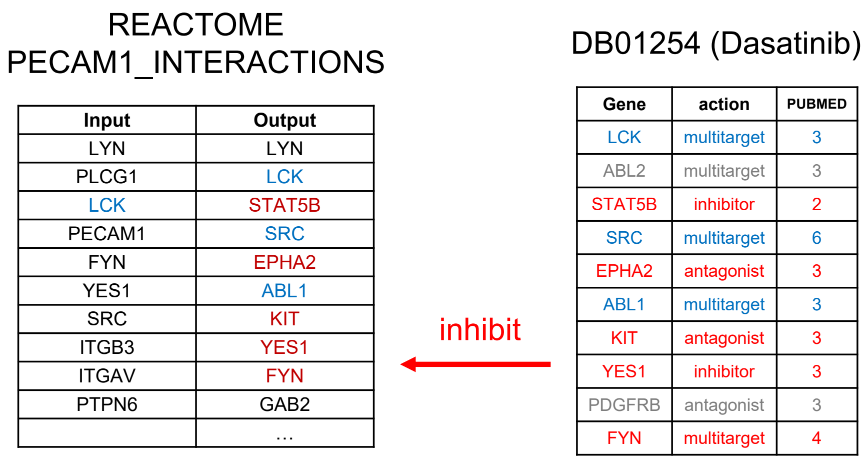
Definition 27. Gene-Pathway and Pathway-Gene Relationship (*GPW* and *PWG*) For a pathway pw and a gene g, an edge is created from g to pw if g is involved in producing $p \in I(pw)$. Similarly, an edge is created from pw to g if g is involved in producing $p \in O(pw)$.

A relationship from a gene to a pathway is established through proteins in the input of the pathway. Inversely, a relationship from a pathway to a gene is established through proteins in the output of the pathway.

We assume that information about drugs and target genes is available. Drugs can now be linked to genes to build a graph *BG*. Relationships between gene and drug in two directions are defined in Definition 28.

Definition 28. Drug-Gene and Gene-Drug Relationships (*DG* and *GD*) For a drug $d_i \in D$, we have a set $TGT(d_i)$ of target genes. Automatically, we create edges from a drug $d_i \in D$ to a target gene $g_j \in TGT(d_i)$, which is in turn linked to a pathway pw_k by creating an edge (g_j, pw_k) according to Definition 27. Opposite edges from genes to drugs are created using the same data.

Based on Definition 27 and 28, we can establish edges from pathways



Input	Output
LYN	LYN
PLCG1	LCK
LCK	STAT5B
PECAM1	SRC
FYN	EPHA2
YES1	ABL1
SRC	KIT
ITGB3	YES1
ITGAV	FYN
PTPN6	GAB2
	…

Gene	action	PUBMED
LCK	multitarget	3
ABL2	multitarget	3
STAT5B	inhibitor	2
SRC	multitarget	6
EPHA2	antagonist	3
ABL1	multitarget	3
KIT	antagonist	3
YES1	inhibitor	3
PDGFRB	antagonist	3
FYN	multitarget	4

Figure 41: An illustrative example of establishing a link from a drug to a pathway.

to drugs and vice versa. Note that the types of relationships between drugs and pathways is classified in the same way in which the types of relationships between pathways is classified. See Figure 41, an edge from a drug `DB01254` to a pathway called `PECAM1_INTERACTIONS` is established. Since most of genes in the drug side inhibit genes in the pathway side, the edge is classified into *down*.

5.4 CyHadoop

Visualizing biological data plays an important role in biology research. Many data visualization tools are available to help users capture significant features in biological data. Cytoscape ([75]) is one of popular data visualization tools in biology domain. It allows biologists to interactively navigate biological network through network displays. The functionalities of the tool

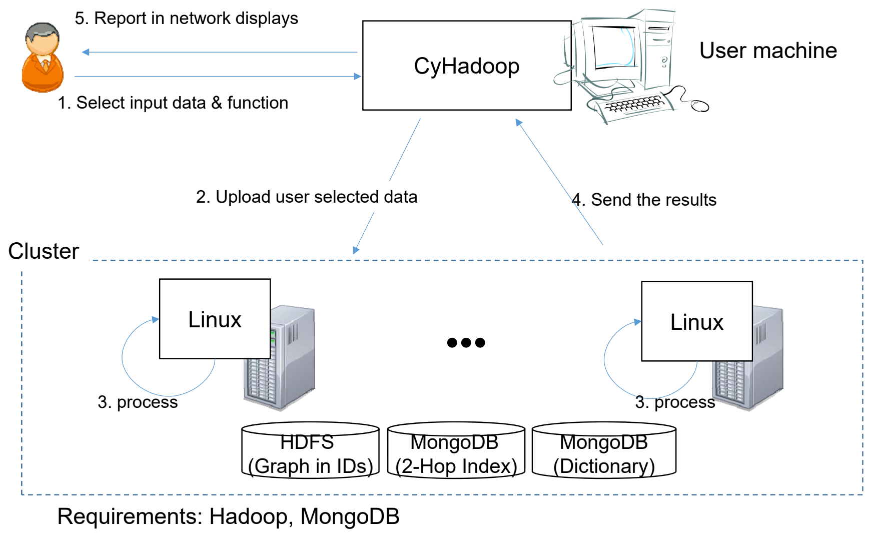


Figure 42: CyHadoop Architecture

can be extended by installing third-party plugins. A number of plugins for Cytoscape has being developed to equip it with data processing functionalities. However, most the plugins are designed to run on a single computer that is not capable of efficiently handling large volume of data.

CyHadoop runs inside Cytoscape user-interface and to work with Hadoop cluster through network communication. Figure 42 depicts CyHadoop's architecture consisting of an user machine and several worker machines. In the user machine, CyHadoop is installed as a plugin into Cytoscape. Hadoop is configured in the worker machines on which an `Linux` operation system has been installed. All login information for work machines is specified in CyHadoop in order for the user machine to freely communicate with work machines (See Figure 43). To do so, some prerequisites are `expect`, `sshd` and etc. These are required to enable the user machine to interact with the

Table 6: Biomedical dataset

Contents	Sources	Entities	Size
Pathway	Reactome[3], KEGG[4]	Pathways Genes per pathway (avg/max/min) Unique genes in pathways	860 58.7 / 933 / 9 24,017
Drug	DrugBank[5]	Drugs Genes per drug (avg/max/min)	179,380 58.7 / 933 / 9
PPI	BIND, BioGrid DIP, HPRD, IntAct MINT, STRING	Gene-Gene Pairs	651,113

worker machines through SSH communications.

CyHadoop deals with a graph in the edge list format, which consists of a set of 4-tuples (*source*, *label*, *target*, *weight*). The format represents a directed weighted labeled graph. We have an edge (*source*, *target*) and its label *label* with the weight *weight*.

Shortest Path Discovery between Two Drugs The system allows to users to input two drugs in either id or name (See Figure 44). If the user clicks the discovery button, it will show shortest path between two drugs (See Figure 46). The user can adjust the diameter that determines how far additional vertices from the discovered path is shown. If the diameter is set to 1, one additional vertices adjacent to vertices in the discovered path are shown. In this way, the user is able to see some contextual information relevant to the discovered shortest path.

[3]`http://www.reactome.org/`
[4]`http://www.genome.jp/kegg/`
[5]`http://www.reactome.org/`

5.5 Scenario

In this section, we will show an imaginary scenario that describes the work-flow of the system. Suppose that an user wants to know if two drugs of interests have adverse effects or not. If it has, the user then wants to further examine its causes.

See Figure 43, when Cyhadoop is launched, the hadoop configuration panel appears. The user has to configure the hadoop cluster by specifying ip address, login id, password and etc. The user then check if the cluster is working by clicking the status button. Then, the user moves to the query panel (See Figure 44). In that panel, the user inputs names of two drugs in this case, `DB02716` and `DB02546`. In addition, the system allows the user to specify the diameter value that determine *k* hop expansions. Suppose that the user select 1. By clicking the `Discover` button, a shortest path discovery task is executed in the hadoop cluster. Logs are displayed in the textbox together with progress. After a while, the task is finished. The user clicks the `View` button. See Figure 46, in the right hand side, a graph is visualized that depicts 1 hop expanded shortest path from `DB02716` to `DB02546`. There exist five vertices between source and target vertices. Several vertices are attached to the path which is shown because the user specify diameter as 1. The users are able to see a nested graph (See Figure 47) that contains cycles. In addition, the system allows to see a list of drugs interacted with a given drug available in DrugBank (See Figure 45). The feature helps compare discovered interactions with existing interactions.

Table 7: Drug-Drug interactions in Drugbank

Total Drugs	8,226
Drugs that have at least one interaction	2,214
Interactions	489,405
Drugs per drug (avg/max/min)	242.6 / 1208 / 1
Adverse drugs per drug (avg/max/min)	90.8 / 455 / 1

5.6 Preliminary Results

In this section, we present some preliminary results that we can get from integrated biomedical datasets. Our goal is to discover drug-drug interactions that might occur adverse effects. Discovered interactions need to be compared with existing drug-drug interaction database such as Drugbank listed in Table 7.

See Figure 48 for example, a drug "Pirfenidone" has some drug interaction with interaction description. Unfortunately, that information is all we can obtain from Drugbank, which means that there is no explicit way to classify positive effect and negative effect by a drug combination. We do very simple classification based on interaction description. When an interaction description contains the phrase "adverse effect", we classify it as an adverse drug combination. We classify as a positive drug combination, otherwise.

See Figure 49, a list of drug-gene interaction types is shown. In order to identify left and right side drugs, we classify each drug-gene interaction types into UP and DOWN by taking into account the label of types. The red one are classified into DOWN and the other ones are into UP. Using the simple heuristic, we obtain statistics as shown in Table 8.

See Figure 50, gene-gene interactions in Reactome are shown. We clas-

Table 8: Drug-Gene interactions in Drugbank

Total Drugs	8,226
Drugs with target genes	6,763
Genes per drug(avg/max)	2.5 / 57
Up-Gene per drug(avg/max)	0.6 / 56
Down-Gene per drug(avg/max)	1.9 / 31
Unique Genes	1,627

sify gene-gene interactions based on Annotation. See Figure 51, annotations are listed. We classify each gene-gene interaction type into UP and DOWN by taking into account the annotation of types. The red one are classified into DOWN and the other ones are into UP.

Drug-Drug interactions discovered by the proposed approach is listed in Table 9. Note that some interactions has already been reported in Drugbank and some are new by our approach. For those that has already been reported in Drugbank, we need to compare the discovered path with literatures mentioning the interaction. For those that has not been reported in Drugbank, clinical experiments are required to verify the correctness.

5.7 Future Directions

We discuss limitations of the prototype system and some ways to improve the issues, in order to be used in real-world problems.

Input Data Collection and Conversion The prototype system assumes that input graph data has already been prepared in the edge list format. No converting modules are available in the plugin. Automatically integrating biomedical datasets is beyond the scope of the thesis. Integration strategies for biomedical datasets vary according to requirements. It is not easy task to develop a general purpose integration tool. Moreover, biomedical datasets changes frequently over time. Once a graph data is created, it cannot reflect up-to-date datasets available.

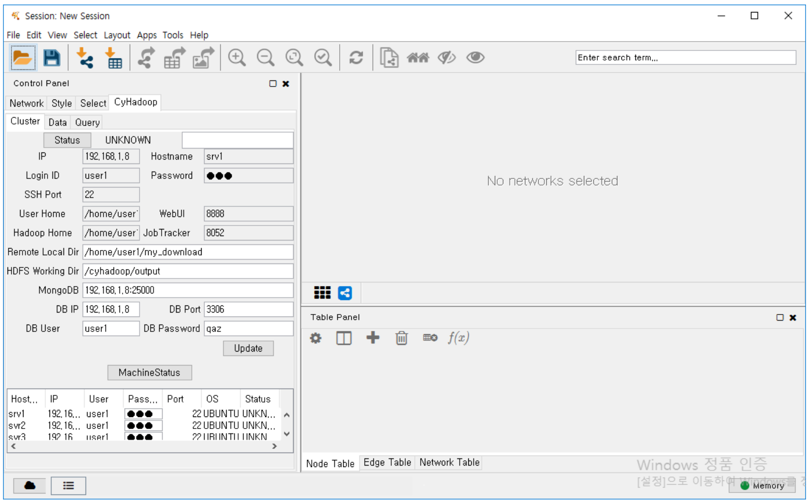
In addition, even if such a graph data has been prepared, the user must upload it into HDFS manually. No GUI interface is included in the current system that allows users to select a file in local machine and to upload it into HDFS.

Graph to DAG The prototype system only deals with DAGs. In the scenario presented in the thesis, we integrated several biomedical datasets and created a graph data. We implemented a simple in-memory program that converts a graph into corresponding DAG by substituting each SCC into a single vertex. However, in this way, massive graph data cannot be converted into DAG. We plan to implement a distributed algorithm that supports the functionalities.

Visualization One of most important features of the plugin is to visualize large graph in the screen in a way that human is able to easily capture the intended meaning. In the prototype version, different colors are used to distinguish source and target vertices. Genes and drugs are distinguished by shapes. Another important aspect is to place shapes in appropriate positions, which are completely omitted in the current version. We plan to cooperate with human-computer interaction researchers to address the issue.

Table 9: A portion of drug-dug interactions discovered by the proposed approach. Drugbank indicates whether the drug combination has aleady been reported by Drugbank or not. Path shows the discovered path from Drug#1 to Drug#2.

Drug#1	Drug#2	Drugbank	Path
Carvedilol	Pentoxifylline	T	GENESET#NPPB_GJA1_VCAM1_ADRB1, KEGG_MAPK_SIGNALING_PATHWAY, GENESET#NT5E_PDE4B
Carvedilol	Ibudilast	F	GENESET#NPPB_GJA1_VCAM1_ADRB1, KEGG_MAPK_SIGNALING_PATHWAY, GENESET#PDE4A
Carvedilol	Gallium nitrate	F	GENESET#NPPB_GJA1_VCAM1_ADRB1, KEGG_MAPK_SIGNALING_PATHWAY, GENESET#RRM2
Buspirone	Indapamide	F	GENESET#HTR1A, REACTOME_SEROTONIN_RECEPTORS, REACTOME_POTASSIUM_CHANNELS, GENESET#KCNQ1
Methylphenobarbital	Dutasteride	F	GENESET#GABRA2_GABRA4_GABRA3, REACTOME_GAP_JUNCTION_DEGRADATION, REACTOME_ENDOGENOUS_STEROLS, GENESET#SRD5A2_SRD5A1


Session: New Session

Control Panel — Network | Style | Select | CyHadoop — Cluster | Data | Query

Status	UNKNOWN		
IP	192.168.1.8	Hostname	srv1
Login ID	user1	Password	●●●
SSH Port	22		
User Home	/home/user	WebUI	8888
Hadoop Home	/home/user	JobTracker	8052
Remote Local Dir	/home/user1/my_download		
HDFS Working Dir	/cyhadoop/output		
MongoDB	192.168.1.8:25000		
DB IP	192.168.1.8	DB Port	3306
DB User	user1	DB Password	qaz

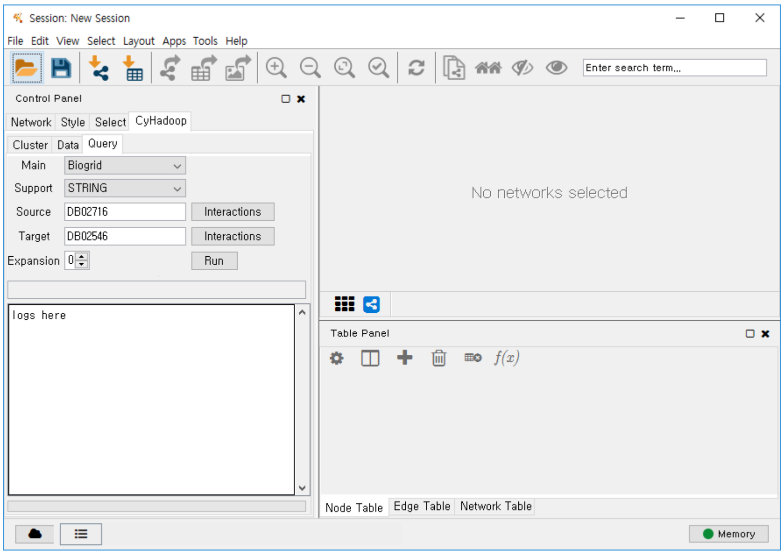
Update

MachineStatus

Host...	IP	User	Pass...	Port	OS	Status
srv1	192.16...	user1	●●●	22	UBUNTU	UNKN...
svr2	192.16...	user1	●●●	22	UBUNTU	UNKN...
svr3	192.16	user1	●●●	22	UBUNTU	UNKN

No networks selected

Figure 43: Cluster setup


Session: New Session

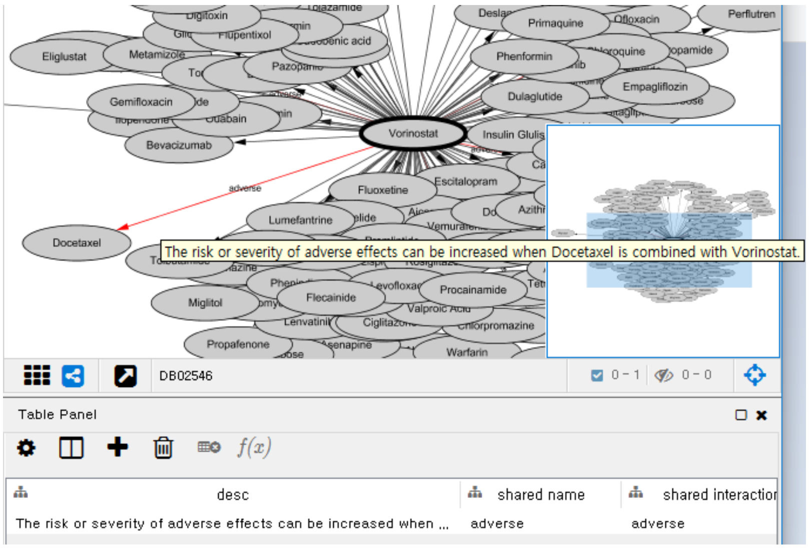
Control Panel — Network | Style | Select | CyHadoop — Cluster | Data | Query

Main	Biogrid	
Support	STRING	
Source	DB02716	Interactions
Target	DB02546	Interactions
Expansion	0	Run

```
logs here
```

No networks selected

Figure 44: The user inputs two drugs of interests (DB02716 and DB02546)

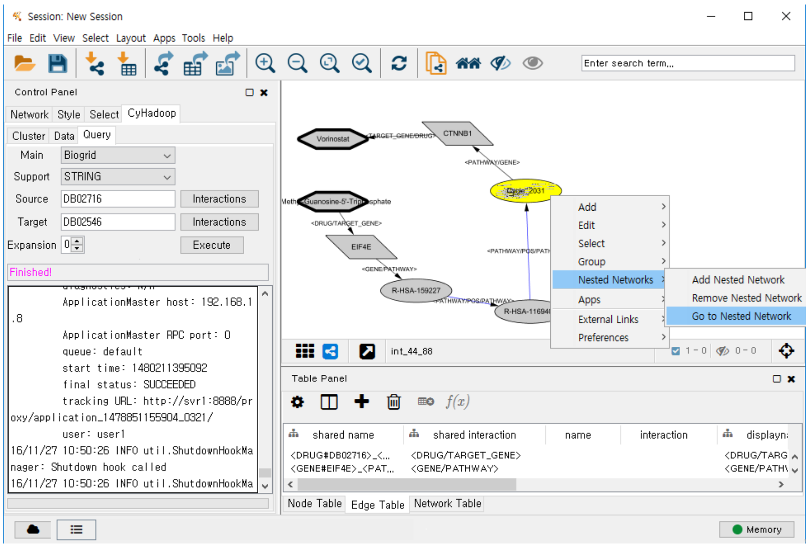


The risk or severity of adverse effects can be increased when Docetaxel is combined with Vorinostat.

Table Panel

desc	shared name	shared interaction
The risk or severity of adverse effects can be increased when ...	adverse	adverse

Figure 45: Known durg-drug interaction from DrugBank



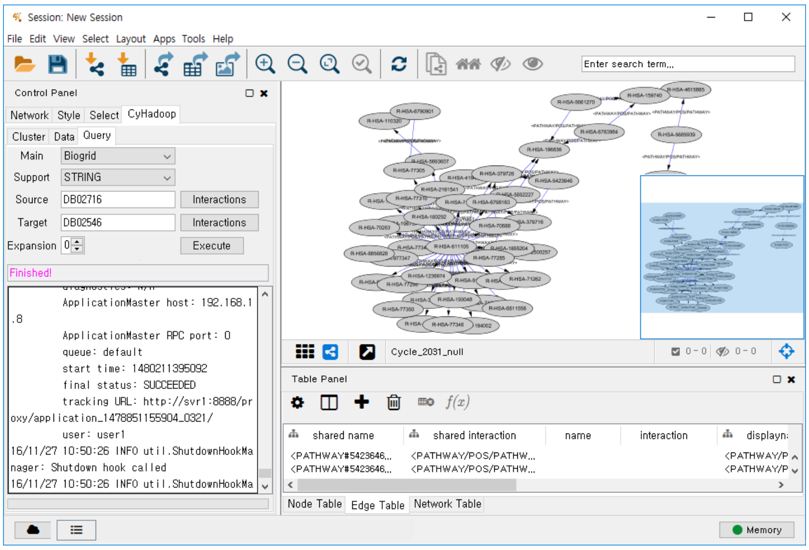
Main: Biogrid
Support: STRING
Source: DB02716
Target: DB02546
Expansion: 0

Finished!

```
        Diagnostics: N/A
        ApplicationMaster host: 192.168.1
.8
        ApplicationMaster RPC port: 0
        queue: default
        start time: 1480211395092
        final status: SUCCEEDED
        tracking URL: http://svr1:8888/pr
oxy/application_1478851155904_0321/
        user: user1
16/11/27 10:50:26 INFO util.ShutdownHookMa
nager: Shutdown hook called
16/11/27 10:50:26 INFO util.ShutdownHookMa
```

shared name	shared interaction	name	interaction	displayn
<DRUG#DB02716>_<...	<DRUG/TARGET_GENE>			<DRUG/TARG
<GENE#EIF4E>_<PAT...	<GENE/PATHWAY>			<GENE/PATH

Figure 46: Discovered shortest path between two drugs.



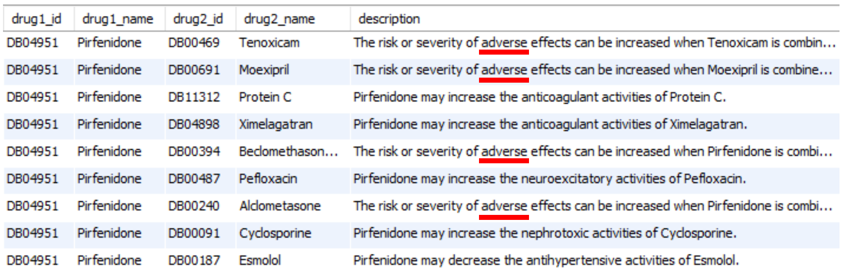
Main: Biogrid
Support: STRING
Source: DB02716
Target: DB02546
Expansion: 0

Finished!

```
        ApplicationMaster host: 192.168.1
.8
        ApplicationMaster RPC port: 0
        queue: default
        start time: 1480211395092
        final status: SUCCEEDED
        tracking URL: http://svr1:8888/pr
oxy/application_1478851155904_0321/
        user: user1
16/11/27 10:50:26 INFO util.ShutdownHookMa
nager: Shutdown hook called
16/11/27 10:50:26 INFO util.ShutdownHookMa
```

shared name	shared interaction	name	interaction	displayn:
<PATHWAY#5423646...	<PATHWAY/POS/PATHW...			<PATHWAY/P
<PATHWAY#5423646...	<PATHWAY/POS/PATHW...			<PATHWAY/P

Figure 47: Nested graph that is contained in a cycle node in Figure 46



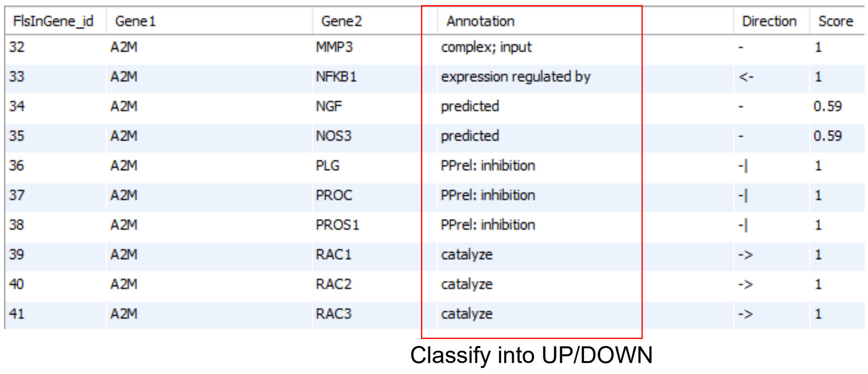
drug1_id	drug1_name	drug2_id	drug2_name	description
DB04951	Pirfenidone	DB00469	Tenoxicam	The risk or severity of adverse effects can be increased when Tenoxicam is combin...
DB04951	Pirfenidone	DB00691	Moexipril	The risk or severity of adverse effects can be increased when Moexipril is combine...
DB04951	Pirfenidone	DB11312	Protein C	Pirfenidone may increase the anticoagulant activities of Protein C.
DB04951	Pirfenidone	DB04898	Ximelagatran	Pirfenidone may increase the anticoagulant activities of Ximelagatran.
DB04951	Pirfenidone	DB00394	Beclomethason...	The risk or severity of adverse effects can be increased when Pirfenidone is combi...
DB04951	Pirfenidone	DB00487	Pefloxacin	Pirfenidone may increase the neuroexcitatory activities of Pefloxacin.
DB04951	Pirfenidone	DB00240	Alclometasone	The risk or severity of adverse effects can be increased when Pirfenidone is combi...
DB04951	Pirfenidone	DB00091	Cyclosporine	Pirfenidone may increase the nephrotoxic activities of Cyclosporine.
DB04951	Pirfenidone	DB00187	Esmolol	Pirfenidone may decrease the antihypertensive activities of Esmolol.

Figure 48: A portion of interactions for a drug in Drugbank.

action	count(*)
	9847
inhibitor	1875
antagonist	1395
agonist	848
potentiator	413
binder	237
cofactor	106
other/unknow	101
inducer	67
activator	64
other	57
positive all	36
partial agon	34
antibody	33
ligand	33
negative mod	29
product of	26
unknown	25
intercalatio	23
allosteric m	22
adduct	21
cross-linkin	20
chelator	17
modulator	17
incorporatio	14

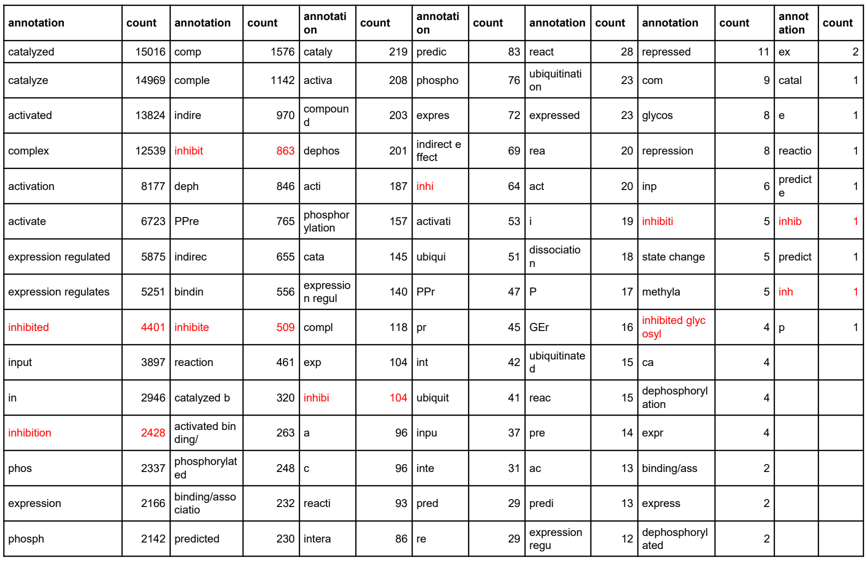
action	count(*)
multitarget	9
cleavage	8
stimulator	8
inverse agon	7
inhibitor, c	5
binding	4
blocker	4
chaperone	4
metabolizer	4
suppressor	4
neutralizer	3
partial anta	2
Ryanodine re	2
substrate	2
acetylation	1
component of	1
desensitize	1
Nonstructura	1
positive mod	1
Progesterone	1
reducer	1
Voltage gate	1

Figure 49: Drug-gene interaction types in Drugbank



FIsInGene_id	Gene1	Gene2	Annotation	Direction	Score
32	A2M	MMP3	complex; input	-	1
33	A2M	NFKB1	expression regulated by	<-	1
34	A2M	NGF	predicted	-	0.59
35	A2M	NOS3	predicted	-	0.59
36	A2M	PLG	PPrel: inhibition	-|	1
37	A2M	PROC	PPrel: inhibition	-|	1
38	A2M	PROS1	PPrel: inhibition	-|	1
39	A2M	RAC1	catalyze	->	1
40	A2M	RAC2	catalyze	->	1
41	A2M	RAC3	catalyze	->	1

Figure 50: Gene-gene interaction in Reactome



annotation	count	annotation	count	annotati on	count	annotati on	count	annotation	count	annotation	count	annot ation	count
catalyzed	15016	comp	1576	cataly	219	predic	83	react	28	repressed	11	ex	2
catalyze	14969	comple	1142	activa	208	phospho	76	ubiquitinati on	23	com	9	catal	1
activated	13824	indire	970	compoun d	203	expres	72	expressed	23	glycos	8	e	1
complex	12539	inhibit	863	dephos	201	indirect e ffect	69	rea	20	repression	8	reactio	1
activation	8177	deph	846	acti	187	inhi	64	act	20	inp	6	predict e	1
activate	6723	PPre	765	phosphor ylation	157	activati	53	i	19	inhibiti	5	inhib	1
expression regulated	5875	indirec	655	cata	145	ubiqui	51	dissociatio n	18	state change	5	predict	1
expression regulates	5251	bindin	556	expressio n regul	140	PPr	47	P	17	methyla	5	inh	1
inhibited	4401	inhibite	509	compl	118	pr	45	GEr	16	inhibited glyc osyl	4	p	1
input	3897	reaction	461	exp	104	int	42	ubiquitinate d	15	ca	4		
in	2946	catalyzed b	320	inhibi	104	ubiquit	41	reac	15	dephosphoryl ation	4		
inhibition	2428	activated bin ding/	263	a	96	inpu	37	pre	14	expr	4		
phos	2337	phosphorylat ed	248	c	96	inte	31	ac	13	binding/ass	2		
expression	2166	binding/asso ciatio	232	reacti	93	pred	29	predi	13	express	2		
phosph	2142	predicted	230	intera	86	re	29	expression regu	12	dephosphoryl ated	2		

Figure 51: Annotations of gene-gene interactions in Reactome

Chapter 6

Conclusion

In this thesis, we proposed graph path discovery algorithms, and a prototype tool was implemented to address biomedical problems using the developed algorithms. The tool can be used by biomedical researchers and clinicians to identify adverse drug combination and its causes in terms of graph made of drug, gene, and pathways. Firstly, shortest path discovery algorithm was proposed to find linked pathways involved in a drug combination. The algorithm was implemented on top of Spark, which guarantees scalability. Reachability index was exploited to speed up the discovery process. Federated shortest path discovery was discussed in terms of dynamic integration of biomedical datasets. We discussed that discovered shortest path can be transformed into a simple graph path query by substituting constants into variables. Secondly, efficient multi-way join processing algorithm was proposed to find another drug combinations that are associated with the matched graph path. Signature encoding technique was developed to prune redundant data required by conventional multi-way join processing approaches. Thirdly, a Cytoscape plugin was implemented in order for biomedical researchers to use the proposed graph path discovery algorithms against biomedical graphs. The novelty can be found in allowing Cytoscape to interact with clusters. In addition, it helps biomedical researchers and clinicians to discover interacted pathways that are involved in drug combinations.

The future directions are as follows.

1) Reachability index building should also be carried out in a cluster. To the best of our knowledge, there exists few work that support massive graph reachability index building using a cluster. In addition, more complex graph statics should be taken into account to further reduce the size of reachability index.

2) Federated shortest path discovery was defined in an abstract level in this thesis. We will give a concrete definition with real-world examples. Moreover, applicability to biomedical research should also be addressed.

3) Applications of shortest path discovery is limited as it gives us only one path. However, one mights want to know second-best, third-best, and k-best path that would give us another interpretation for some biological phenomena. We plan to devise a scalable algorithm for k shortest path discovery. It is expected that the proposed shortest path discovery algorithm can be extended to deal with k shortest path discovery. By doing so, our framework is able to cover diverse biomedical research.

4) The main drawback of signature encoding technique is that signatures occupy huge space. We plan to adopt compressing techniques to reduce space requirements of signatures.

5) In this thesis, we assume that biomedical datasets has already been converted into graphs. We plan to devise an automatic way to integrate existing biomedical datasets into graphs. Then, our tool can be used when underling datasets change.

References

[1] J. Ahn, D.-H. Im, and H.-G. Kim, "A mapreduce-based approach for preffix-based labeling of large xml data," in *Joint International Semantic Technology Conference*, IEEE, 2016.

[2] J. Ahn, D.-H. Im, T. Lee, and H.-G. Kim, "A dynamic and parallel approach for repetitive prime labeling of xml with mapreduce," *The Journal of Supercomputing*, pp. 1–27, 2016.

[3] J. Gao, J. Zhou, J. X. Yu, and T. Wang, "Shortest path computing in relational dbmss," *IEEE Trans. on Knowl. and Data Eng.*, vol. 26, pp. 997–1011, Apr. 2014.

[4] K. Rohloff and R. E. Schantz, "High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store," in *Programming Support Innovations for Emerging Distributed Applications*, p. 4, ACM, 2010.

[5] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. Thuraisingham, "Heuristics-based query processing for large rdf graphs using cloud computing," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 23, no. 9, pp. 1312–1327, 2011.

[6] J. Myung, J. Yeon, and S.-g. Lee, "Sparql basic graph pattern processing with iterative mapreduce," in *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC '10, (New York, NY, USA), pp. 6:1–6:6, ACM, 2010.

[7] F. N. Afrati and J. D. Ullman, "Optimizing multiway joins in a mapreduce environment," *IEEE Trans. on Knowl. and Data Eng.*, vol. 23, pp. 1282–1298, Sept. 2011.

[8] H. Yu, S. Choo, J. Park, J. Jung, Y. Kang, and D. Lee, "Prediction of drugs having opposite effects on disease genes in a directed network," *BMC systems biology*, vol. 10, no. 1, p. 17, 2016.

[9] S. Yoon, J. Jung, H. Yu, M. Kwon, S. Choo, K. Park, D. Jang, S. Kim, and D. Lee, "Context-based resolution of semantic conflicts in biological pathways," *BMC medical informatics and decision making*, vol. 15, no. 1, p. 1, 2015.

[10] A. J. Golubski, E. E. Westlund, J. Vandermeer, and M. Pascual, "Ecological networks over the edge: Hypergraph trait-mediated indirect interaction (tmii) structure," *Trends in ecology & evolution*, vol. 31, no. 5, pp. 344–354, 2016.

[11] A. Dickson, "Introduction to graph theory," 2006.

[12] D. Wagner and T. Willhalm, "Speed-up techniques for shortest-path computations," pp. 23–36, 2007.

[13] R. R. Veloso, L. Cerf, W. Meira Jr, and M. J. Zaki, "Reachability queries in very large graphs: A fast refined online search approach.," in *EDBT*, pp. 511–522, Citeseer, 2014.

[14] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[15] X. Cui, P. Zhu, X. Yang, K. Li, and C. Ji, "Optimized big data k-means clustering using mapreduce," *The Journal of Supercomputing*, vol. 70, no. 3, pp. 1249–1259, 2014.

[16] J. Myung and S.-g. Lee, "Exploiting inter-operation parallelism for matrix chain multiplication using mapreduce," *The Journal of Supercomputing*, vol. 66, no. 1, pp. 594–609, 2013.

[17] J.-h. Um, H. Choi, S.-k. Song, S.-p. Choi, H. Yoon, H. Jung, and T.-h. Kim, "Development of a virtualized supercomputing environment

for genomic analysis," *The Journal of Supercomputing*, vol. 65, no. 1, pp. 71–85, 2013.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012.

[19] B. McBride, "Jena: Implementing the rdf model and syntax specification.," in *SemWeb*, 2001.

[20] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pp. 411–422, VLDB Endowment, 2007.

[21] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple indexing for semantic web data management," *Proc. VLDB Endow.*, vol. 1, pp. 1008–1019, Aug. 2008.

[22] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB Journal*, vol. 19, pp. 91–113, Feb. 2010.

[23] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix bit loaded: a scalable lightweight join query processor for rdf data," in *Proceedings of the 19th international conference on World wide web*, pp. 41–50, ACM, 2010.

[24] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: answering sparql queries via subgraph matching," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 482–493, 2011.

[25] Z. Kaoudi and I. Manolescu, "Rdf in the clouds: a survey," *The VLDB Journal*, pp. 1–25, 2014.

[26] R. Punnoose, A. Crainiceanu, and D. Rapp, “Rya: a scalable rdf triple store for the clouds,” in *Proceedings of the 1st International Workshop on Cloud Intelligence*, p. 4, ACM, 2012.

[27] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris, “H2rdf: adaptive query processing on rdf data in the cloud.,” in *Proceedings of the 21st international conference companion on World Wide Web*, pp. 397–400, ACM, 2012.

[28] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, “A distributed graph engine for web scale rdf data,” in *Proceedings of the VLDB Endowment*, vol. 6, pp. 265–276, VLDB Endowment, 2013.

[29] J. Huang, D. J. Abadi, and K. Ren, “Scalable sparql querying of large rdf graphs,” *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[30] K. Hose and R. Schenkel, “Warp: Workload-aware replication and partitioning for rdf,” in *4th International Workshop on Data Engineering meets Semantic Web (DESWeb 2013)*, (Brisbane, Australia), 2013.

[31] L. Galárraga, K. Hose, and R. Schenkel, “Partout: A distributed engine for efficient rdf processing,” in *Proceedings of the companion publication of the 23rd international conference on World wide web companion*, pp. 267–268, International World Wide Web Conferences Steering Committee, 2014.

[32] A. Aranda-Andújar, F. Bugiotti, J. Camacho-Rodríguez, D. Colazzo, F. Goasdoué, Z. Kaoudi, and I. Manolescu, “Amada: Web data repositories in the amazon cloud,” in *CIKM 2012*, (Maui, États-Unis), 2012.

[33] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[34] X. Zhang, L. Chen, Y. Tong, and M. Wang, “Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud,” in *Data Engineering*

(ICDE), 2013 IEEE 29th International Conference on, pp. 565–576, IEEE, 2013.

[35] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM Journal on Computing*, vol. 32, no. 5, pp. 1338–1355, 2003.

[36] H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: An independent permutation labeling approach," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1191–1202, 2014.

[37] H. Gabr and T. Kahveci, "Signal reachability facilitates characterization of probabilistic signaling networks," *BMC Bioinformatics*, vol. 16, no. Suppl 17, p. S6, 2015.

[38] A. Gubichev, S. J. Bedathur, and S. Seufert, "Sparqling kleene: fast property paths in rdf-3x," in *First International Workshop on Graph Data Management Experiences and Systems*, p. 14, ACM, 2013.

[39] J. X. Yu and J. Cheng, "Graph reachability queries: A survey," in *Managing and Mining Graph Data*, pp. 181–215, Springer, 2010.

[40] K. Simon, "An improved algorithm for transitive closure on acyclic digraphs," *Theoretical Computer Science*, vol. 58, no. 1, pp. 325–346, 1988.

[41] G. Wu, K. Zhang, C. Liu, and J. Li, "Adapting prime number labeling scheme for directed acyclic graphs," in *Database Systems for Advanced Applications*, pp. 787–796, Springer, 2006.

[42] R. Agrawal, A. Borgida, and H. V. Jagadish, *Efficient management of transitive relationships in large data and knowledge bases*, vol. 18. ACM, 1989.

[43] S. Trißl and U. Leser, "Fast and practical indexing and querying of very large graphs," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 845–856, ACM, 2007.

[44] H. Yıldırım, V. Chaoji, and M. J. Zaki, "GRAIL: a scalable index for reachability queries in very large graphs," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 21, no. 4, pp. 509–534, 2012.

[45] S. Seufert, A. Anand, S. Bedathur, and G. Weikum, "Ferrari: Flexible and efficient reachability range assignment for graph indexing," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pp. 1009–1020, IEEE, 2013.

[46] T. Käfer, J. Umbrich, A. Hogan, and A. Polleres, "Towards a dynamic linked data observatory," *LDOW at WWW*, 2012.

[47] J. Ahn, D.-H. Im, J.-H. Eom, N. Zong, and H.-G. Kim, "G-Diff: A Grouping Algorithm for RDF Change Detection on MapReduce," in *Semantic Technology*, pp. 230–235, Springer, 2014.

[48] D.-H. Im, S.-W. Lee, and H.-J. Kim, "A version management framework for RDF triple stores," *International Journal of Software Engineering and Knowledge Engineering*, vol. 22, no. 01, pp. 85–106, 2012.

[49] D.-H. Im, S.-W. Lee, and H.-J. Kim, "Backward inference and pruning for RDF change detection using RDBMS," *Journal of Information Science*, pp. 238–255, 2012.

[50] H. Choi, K.-H. Lee, and Y.-J. Lee, "Parallel labeling of massive xml data with mapreduce," *The Journal of Supercomputing*, vol. 67, no. 2, pp. 408–437, 2014.

[51] A. D. Zhu, W. Lin, S. Wang, and X. Xiao, "Reachability queries on large dynamic graphs: a total order approach," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 1323–1334, ACM, 2014.

[52] J. Y. Yen, "An algorithm for finding shortest routes from all source nodes to a given destination in general networks," *Quarterly of Applied Mathematics*, pp. 526–530, 1970.

[53] D. Eppstein, "Finding the k shortest paths," *SIAM Journal on computing*, vol. 28, no. 2, pp. 652–673, 1998.

[54] D. Eppstein *et al.*, "k-best enumeration.," 2015.

[55] M. Arenas, B. Cuenca Grau, E. Evgeny, S. Marciuska, and D. Zheleznyakov, "Towards semantic faceted search," in *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, WWW Companion '14, (Republic and Canton of Geneva, Switzerland), pp. 219–220, International World Wide Web Conferences Steering Committee, 2014.

[56] T. Berners-Lee, J. Hendler, O. Lassila, *et al.*, "The semantic web," *Scientific american*, vol. 284, no. 5, pp. 28–37, 2001.

[57] C. Becker and C. Bizer, "Dbpedia mobile: A location-enabled linked data browser," in *WWW 2008 Workshop: Linked Data on the Web*, 2008.

[58] C. Van Aart, B. Wielinga, and W. R. Van Hage, "Mobile cultural heritage guide: location-aware semantic search," in *Knowledge engineering and management by the masses*, pp. 257–271, Springer, 2010.

[59] T. Berners-Lee, Y. Chen, L. Chilton, D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, and D. Sheets, "Tabulator: Exploring and analyzing linked data on the semantic web," in *Proceedings of the 3rd International Semantic Web User Interaction Workshop*, vol. 2006, 2006.

[60] J. Koren, Y. Zhang, and X. Liu, "Personalized interactive faceted search," in *Proceedings of the 17th international conference on World Wide Web*, pp. 477–486, ACM, 2008.

[61] E. Minack, L. Sauermann, G. Grimnes, C. Fluit, and J. Broekstra, "The sesame lucene sail: Rdf queries with full-text search," Technical Report 2008-1, NEPOMUK Consortium, February 2008.

[62] "http://virtuoso.openlinksw.com."

[63] Faye, O. Cure, and Blin, "A survey of rdf storage approaches," *ARIMA Journal*, vol. 15, pp. 11–35, 2012.

[64] G. Aluç, M. T. Ozsu, and K. Daudjee, "Workload matters: Why rdf databases need a new design," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, 2014.

[65] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, IEEE, 2010.

[66] T. Lee, D.-H. Im, H. Kim, and H.-J. Kim, "Application of filters to multiway joins in mapreduce," *Mathematical Problems in Engineering*, vol. 2014, 2014.

[67] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, July 1970.

[68] "http://storm.apache.org."

[69] L. T. X. Phan, Z. Zhang, B. T. Loo, and I. Lee, "Real-time mapreduce scheduling," in *Technical Report No. MS-CIS-10-32, University of Pennsylvania*, 2010.

[70] X. Dong, Y. Wang, and H. Liao, "Scheduling mixed real-time and non-real-time applications in mapreduce environment," in *ICPADS*, pp. 9–16, IEEE, 2011.

[71] V. Marx, "Biology: The big challenges of big data," *Nature*, vol. 498, no. 7453, pp. 255–260, 2013.

[72] A. Schumacher, L. Pireddu, M. Niemenmaa, A. Kallio, E. Korpelainen, G. Zanetti, and K. Heljanko, "Seqpig: simple and scalable scripting for large sequencing data sets in hadoop," *Bioinformatics*, vol. 30, no. 1, pp. 119–120, 2014.

[73] B. Pratt, J. J. Howbert, N. I. Tasman, and E. J. Nilsson, "Mr-tandem: parallel x! tandem using hadoop mapreduce on amazon web services," *Bioinformatics*, vol. 28, no. 1, pp. 136–137, 2012.

[74] H. Huang, S. Tata, and R. J. Prill, "Bluesnp: R package for highly scalable genome-wide association studies using hadoop clusters," *Bioinformatics*, vol. 29, no. 1, pp. 135–136, 2013.

[75] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker, "Cytoscape: a software environment for integrated models of biomolecular interaction networks," *Genome research*, vol. 13, no. 11, pp. 2498–2504, 2003.

Appendix

Algorithms

Algorithm 1 Map function for both of Index Catalog Acquisition and Object Join Partitioning

```
1: procedure MAP($key, value$)        ▷ $key$ is the line number of input RDF file.
2:                                    ▷ $value$ is the triple in current line read.
3:     $skip \leftarrow key \bmod k$  ▷ k depens on the number of triples in the file.
4:     if $skip \neq 0$ then
5:         return
6:     end if
7:     EMIT($value.subject$, $value$)
8:     if $value.object$ is an URI string then    ▷ URI string begins with "http://"
9:         EMIT($value.object$, $value$)
10:    end if
11: end procedure
```

Algorithm 2 Reduce function of Index Catalog Acquisition

```
1:  procedure REDUCE($key, values$)    ▷ $key$ is an URI string which is subject or object.
2:                                     ▷ $values$ is a list of triples.
3:      $uri\_type \leftarrow$ a map (key=uri, value=type) ▷ type of subject or object URI
4:      $uri\_freq \leftarrow$ a map (key=uri, value=frequence) ▷ frequence of URI.
5:      $list \leftarrow$ a list of triples
6:      for each triple $t$ in $values$ do
7:          if $t.predicate$ == rdf:type then
8:              put ($t.subject, t.object$) into $uri\_type$
9:          else
10:             add $t$ to $list$
11:         end if
12:         $uri\_freq(t.subject) \leftarrow uri\_freq(t.subject) + 1$
13:         if the number of keys in $uri\_freq(t.subject) >$ some threshold then
14:             for each triple $t$ in $uri\_freq$ do
15:                 EMIT(uri_freq, $uri\_freq$)
16:             end for
17:             clear $uri\_freq$ from the main memory    ▷ since memory is limited.
18:         end if
19:     end for
20:     for each triple $t$ in $list$ do
21:         if $t.subject == key$ then
22:             EMIT(predicate_domain, $t.predicate|uri\_type(t.subject)$) ▷ | delimeter
23:         else
24:             EMIT(predicate_range, $t.predicate|uri\_type(t.object)$)
25:         end if
26:     end for
27:     EMIT(uri_freq, $uri\_freq$)
28: end procedure
```

Algorithm 3 Reduce function of Object Join Partitioning

```
1: procedure REDUCE($key, values$)    ▷ $key$ is an URI string which is subject or object.
2:                                     ▷ $value$ is a list of triples.
3:     $ot$ ← an empty list of object-key triples.
4:     $os\_bits$ ← an empty bitstring for OS join.
5:     $oo\_bits$ ← an empty bitstring for OO join.
6:     $uri\_type$ ← a map (key=uri, value=type)  ▷ type of subject or object URI
7:     for each triple string $t$ in $values$ do
8:         if $t.predicate$ == rdf:type then
9:             put ($t.subject$, $t.object$) into $uri\_type$
10:        end if
11:        if $t.subject == k$ then
12:            $o$ ← generate a bitstring from $t.object$
13:            add $o$ to $os\_bits$
14:        else if $t.object == key$ then
15:            $s$ ← generate a bitstring from $t.subject$
16:            add $s$ to $oo\_bits$
17:            add $t$ to $ot$                ▷ for the iteration below
18:        end if
19:    end for
20:    for each triple string $t$ in $ot$ do    ▷ iterate over object-key triples
21:        $o\_types$ ← get type list from $uri\_type(t.object)$
22:        $filenames$ ← GETINDEXFILENAMES(empty_list,$t.predicate$,$o\_types$)    ▷ refer to Algorithm 4
23:        for each $filename$ in $filenames$ do
24:            EMIT($filename$, $oo\_bits|os\_bits|t$) ▷ | delimiter ▷ triples are emited to files determined by its predicate and types of object.
25:        end for
26:    end for
27: end procedure
```

Algorithm 4 Generate index file names for a triple

```
procedure GetIndexFileNames($subject\_types, predicate, object\_types$)
    ▷ $subject\_types$ is a list of URI strings which are types of a certain subject.
    ▷ $object\_types$ is a list of URI strings which are types of a certain object.
    $filenames$ ← an empty list
    if $subject\_types$ is empty and $object\_types$ is empty then
        add "$*|predicate|*$" to $filenames$        ▷ *: indicate to any type
    else if $subject\_types$ is not empty and $object\_types$ is not empty then
        for each URI string $s$ in $subject\_types$ do
            for each URI string $o$ in $object\_types$ do
                add "$s|predicate|o$" to $filenames$
            end for
        end for
    else if $subject\_types$ is not empty and $object\_types$ is empty then
        for each URI string $s$ in $subject\_types$ do
            add "$s|predicate|*$" to $filenames$
        end for
    else if $subject\_types$ is empty and $object\_types$ is not empty then
        for each URI string $o$ in $object\_types$ do
            add "$*|predicate|o$" to $filenames$
        end for
    end if
    return $filenames$
end procedure
```

Algorithm 5 Map function of Subject Join Partitioning. This takes as input the index files emitted from the Object Join Partitioning job (Algorithm 3).

```
1: procedure MAP($key, value, filename$)    ▷ $key$ is the line number
2:                  ▷ $value$ consists of OO/OS bitstring, subject and object
3:          ▷ $filename$ is the name of input index file that it is reading.
4:     $itmes \leftarrow$ a string array made by splitting $filename$ by the delimiter
   |                                                ▷ refer to Algorithm 4
5:     $object\_type \leftarrow itmes[2]$          ▷ the type of object of this triple
6:     $predicate \leftarrow itmes[1]$               ▷ the predicate of this triple
7:     $value\_for\_reduce \leftarrow$ "$predicate|object\_type|value$"   ▷ | delimiter
8:     EMIT($value.subject$, $value\_for\_reduce$)
9:     if $value.object$ is an URI string then
10:        EMIT($value.object$, $value\_for\_reduce$)
11:    end if
12: end procedure
```

Algorithm 6 Reduce function of Subject Join Partitioning

```
procedure REDUCE($key, values$)    ▷ $key$ is an URI string which is subject or object.
                                   ▷ $value$ is a list of $value\_for\_reduce$ from Algorithm 5.
    $st$ ← an empty list of subject-key triples.
    $ss\_bits$ ← an empty bitstring for SS join.
    $so\_bits$ ← an empty bitstring for SO join.
    $uri\_type$ ← a map (key=uri, value=type)    ▷ type of subject or object URI
    for each triple string $t$ in $values$ do
        if $t.predicate$ == rdf:type then
            put ($t.subject$, $t.object$) into $uri\_type$
        end if
        if $t.subject == k$ then
            $o$ ← generate a bitstring from $t.object$
            add $o$ to $ss\_bits$
            add $t$ to $st$    ▷ for the iteration below
        else if $t.object == key$ then
            $s$ ← generate a bitstring from $t.subject$
            add $s$ to $so\_bits$
        end if
    end for
    for each triple string $t$ in $st$ do    ▷ iterate over subject-key triples
        $o\_type$ ← get object type from $t$
        $predicate$ ← get the predicate from $t$
        $s\_types$ ← get type list from $uri\_type(t.subject)$
        $filenames$ ← GETINDEXFILENAMES($s\_types$,$predicate$,$o\_type$)
        for each filename $filename$ in $filenames$ do
            EMIT($filename$, $ss\_bits|so\_bits|t$)    ▷ | delimiter ▷ triples are emited to index files
        end for
    end for
end procedure
```

LUBM Queries

Prefixes

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

Query #1

```
SELECT ?X

WHERE {

  ?X rdf:type ub:GraduateStudent .

  ?X ub:takesCourse <http://www.Department0.University10.edu/GraduateCourse1>


}
```

Query #2

```
SELECT ?X ?Y ?Z

WHERE {

  ?X rdf:type ub:GraduateStudent .

  ?Y rdf:type ub:University .

  ?Z rdf:type ub:Department .

  ?X ub:memberOf ?Z .
```

```
   ?X ub:undergraduateDegreeFrom ?Y
}
```

Query #3

```
 SELECT ?X
 WHERE {
   ?X rdf:type ub:Publication .
   ?X ub:publicationAuthor <http://www.Department0.University10.edu/FullProfessor0


}
```

Query #4

```
 SELECT ?X ?Y1 ?Y2 ?Y3
 WHERE {
   ?X rdf:type ub:FullProfessor .
   ?X ub:worksFor <http://www.Department0.University10.edu> .
   ?X ub:name ?Y1 .
   ?X ub:emailAddress ?Y2 .
   ?X ub:telephone ?Y3
}
```

Query #5

```
SELECT ?X
WHERE {
   ?X rdf:type ub:UndergraduateStudent .
   ?X ub:memberOf <http://www.Department0.University10.edu>
}
```

Query #7

```
SELECT ?X ?Y
WHERE {
   ?X rdf:type ub:UndergraduateStudent .
   ?Y rdf:type ub:Course .
   ?X ub:takesCourse ?Y .
   <http://www.Department0.University10.edu/AssociateProfessor0> ub:teacherOf ?Y

}
```

Query #8

```
SELECT ?X ?Y ?Z
WHERE {
   ?X rdf:type ub:UndergraduateStudent .
   ?Y rdf:type ub:Department .
   ?X ub:memberOf ?Y .
   ?Y ub:subOrganizationOf <http://www.University10.edu> .
   ?X ub:emailAddress ?Z
}
```

Query #9

```
SELECT ?X ?Y ?Z
WHERE {
   ?X rdf:type ub:GraduateStudent .
   ?Y rdf:type ub:FullProfessor .
   ?Z rdf:type ub:GraduateCourse .
   ?X ub:advisor ?Y .
   ?Y ub:teacherOf ?Z .
   ?X ub:takesCourse ?Z
}
```

Query #10

```
SELECT ?X
WHERE {
  ?X rdf:type ub:GraduateStudent .
  ?X ub:takesCourse <http://www.Department0.University10.edu/GraduateCourse10>

}
```

Query #11

```
SELECT ?X
WHERE {
  ?X rdf:type ub:ResearchGroup .
  ?X ub:subOrganizationOf <http://www.Department0.University10.edu>

}
```

Query #12

```
SELECT ?X ?Y
WHERE {
  ?X rdf:type ub:FullProfessor .
```

```
    ?Y rdf:type ub:Department .

    ?X ub:worksFor ?Y .

    ?Y ub:subOrganizationOf <http://www.Department0.University10.edu>


}
```

초 록

의약품의 종류가 많아지면서 부작용이 발생할 수 있는 의약품 조합을 예측하고 그 원인을 규명하려는 연구의 중요성이 커지고 있다. 이는 신약을 개발하는 연구원이나 의약품을 처방하는 임상의에게 중요한 연구이다. 기존 대부분의 연구는 의약품에 가장 많은 영향을 받는 유전자 목록을 식별하는 데에 그치고 있다. 이러한 방법론은 의약품 조합에 따른 부작용의 원인을 생물체 내의 요소간 화학반응 체계(대사경로)의 관점에서 규명하는 데에 한계가 있다. 부작용의 원인은 대사경로들의 연결관계에 의해 더 직관적으로 이해될 수 있으므로 의약품은 대사경로의 구성원으로서 다뤄져야 한다. 따라서, 본 학위논문의 목적은 주어진 의약품 조합에 의해 활성화되는 대사경로들의 연결관계를 탐색하는 도구를 개발하는 데에 있다. 특히, 최신의 의생물학 연구결과는 링크드 데이터로 공개되고 있고 그 양이 폭발적으로 증가하고 있으므로 대용량 의생물학 링크드 데이터를 효율적으로 다룰 수 있도록 알고리즘들이 개발돼야 한다.

본 논문의 가정은 다음과 같다; 주어진 의약품 조합의 한 의약품이 특정 대사경로들에 대해 상승-조절효과를 일으키고 다른 의약품이 같은 대사경로들에 대해 반대로 하강-조절효과를 일으킨다면, 의약품 조합이 당초 의도한 효과가 상쇄되거나 이상한 효과가 발생할 수 있다. 이와 같은 관점에서의 부작용의 원인을 밝히는 문제는 의약품이 조절하는 유전자들을 시작과 끝으로 삼는 대사경로들의 연결관계(경로)를 찾는 문제로 정의될 수 있다. 따라서, 본 논문에서는 분산환경에서 작동하는 최단거리경로 탐색과 그래프 경로 매칭 알고리즘을 고안한다. 또한, 다양한 의생물학 데이터를 통합한 대사경로 그래프 모델을 제안하고 의생물학 연구자나

임상의들이 사용할 수 있도록 시각화 기반 도구를 개발한다.

대용량 그래프에 대한 최단거리경로를 탐색하는 알고리즘을 제안했다. 기존 관계형 데이터베이스 기반 알고리즘을 확장하여 분산컴퓨팅 프레임워크의 일종인 Spark에서 작동하도록 구현했다. 기존 그래프 도달 가능성 인덱스를 활용하여 경로 탐색 도중 불필요한 경로를 제외시킴으로서 탐색 시간을 줄이는 기법을 고안했다. 또한, 그래프 도달 가능성 인덱스의 크기를 줄이기 위한 노드 아이디 재부여 기법을 고안했다. 실험결과 기존 방법에서 다루지 못했던 대용량 그래프도 효율적으로 다룰 수 있었다.

최단거리경로는 그래프 경로 질의로 변환될 수 있고 이는 최단거리 경로의 패턴과 유사한 또 다른 그래프 경로를 찾는 데에 활용할 수 있다. 이를 위해, 멀티-웨이 조인 기법을 활용한 맵리듀스 기반의 그래프 경로 매칭 알고리즘을 제안했다. 경로 매칭 도중 그래프 경로 질의와 관계없는 데이터를 제외시키기 위한 시그니쳐 인코딩 기법을 고안했다. RDF (Resource Description Framework) 데이터에 대한 SPARQL 질의 처리 속도에 대한 실험결과 기존 방법보다 제안한 방법이 빨랐다.

의약품 조합에 관련된 다양한 의생물학 데이터(e.g. Reactome, KEGG, BioGrid, STRING 등)를 대사경로 관점에서 통합하기 위한 대사경로 그래프 모델을 제안했다. 단백질-단백질 상호작용 및 유전자 조절관계에 기반하여 대상경로간의 직접/간접 유방향 연결관계를 도출했다. 의약품 조합 부작용 관련 연구자들이 이러한 대사경로 그래프에 대해 상기 알고리즘들을 쉽게 적용할 수 있도록, 시각화 도구인 Cytoscape의 플러그인을 개발했다. 본 도구를 이용해 식별된 의약품 조합 목록을 DrugBank의 의약품 조합 목록과 비교하였다.

본 학위논문에서는 대용량 그래프 경로 탐색을 지원하는 도구가 제안됐다. 분산컴퓨팅 프레임워크와 인덱스 구조를 활용하여 대용량 그래프

를 효율적으로 다룰 수 있게 하였다. 의약품 조합에 따른 부작용의 원인을 밝히는 데에 적용하기 위해, 대사경로 그래프 모델을 제안했고 시각화 기반 도구를 개발했다. 향후 연구에서는, 시간에 따라 변화하는 대사경로의 특성을 반영하기 위해 대사경로 그래프 모델 및 그래프 경로 탐색 알고리즘에 시간 요소를 추가할 예정이다.

주요어 : 그래프, 최단거리경로 탐색, 그래프 패턴 매칭
학번 : 2014-30699