공학박사학위논문

An Indexing Framework for Improving Data Consistency of Triple Database

트리플 데이터베이스의 데이터 일관성 향상을 위한 인덱싱 프레임워크

2013 년 8 월

서울대학교 대학원

전기컴퓨터공학부

강승석

Ph.D. Dissertation

An Indexing Framework for Improving Data Consistency of Triple Database

트리플 데이터베이스의 데이터 일관성 향상을 위한 인덱싱 프레임워크

August 2013

School of Computer Science and Engineering

The Graduate School

Seoul National University

강승석

Abstract

As more data are provided in Semantic Web, processing large amounts of data with flexible format, and interlinking the applications with utilization have become important. In relational databases, a user must acquaint with the schema information to execute certain query on database. Triple is a well-knows flexible data representation format in Semantic Web. If we represent the content in relational database in triple data format, system can utilize the enterprise data with flexibility for various purposes. To guarantee the reliability of triple database, the enforcement of integrity constraints on triple database is required. Integrity constraints are retrieved from the relational database, and translated into triple database with exact same meaning. Triple database can get reliability and consistency by adapting the concept of the enforcement of integrity constraints. Not only representing content by triple data format without the loss of information, but also organizing triples efficiently is important to use triple database practically. However, most existing triple index techniques suffer from data duplication and the problem of large index sizes. In the thesis, we analyze the drawback of existing triple indexing methods from the viewpoint of the reliability and effectiveness of a triple database. We also consider the issues that need to be addressed to build a triple index for the management of relational database-based triple data. As a result, we propose Tridex: a lightweight B+-tree triple index, designed to facilitate efficient processing of triple database. Tridex is beneficial in reduced size of index tree and less data redundancy. In addition, we propose the enhanced shortcut selection methods in triple database. Triples are

commonly represented as a directed graph. With a given triple graph, retrieving data by particular paths can be very expensive due to the self-join problem in triple database. To reduce the self-join operations during query execution, we extend the concept of shortcut, a direct path between specific nodes. By adding appropriate shortcuts in triple database, self-join operations in triple database can be reduced. We propose a reduced candidate shortcut selection considering the maintenance of triple database. The experimental evaluations compare our approach with the state-of-the-art approaches and show adequate performance with less building time in terms of effectiveness and efficiency.

Keywords: Triple database, Semantic Web, Integrity Constraint, Index Structure, Shortcut Selection, Query Optimization

Student Number: 2005-21319

Contents

List of Figures

List of Tables

Chapter 1. Introduction

With the effort of the W3C (World Wide Web Consortium), The Semantic Web enables integration and sharing of data across different applications. Semantic web supports semantic interoperability between programs exchanging data. In Semantic Web, Any kind of information can be represented by a flexible data format such as RDF (Resource Description Framework) [55] – a collection of triple (subject-predicate-object) where each triple encodes the binary relation predicate between subject and object. Every triple delegates the single knowledge fact of information. But most significant data in the enterprise reside in non-triple database such as relational database. Generally RDB (Relational Database) supports enterprise data management with reliable consistency option and strong performance for the application. Every databases based on relational database has their own database schema design, which lead to make difficult to exchange valuable information between principals of information. With this gap, it will be crucial for many real-world Semantic Web applications to be able to access the enterprise database with more flexible data format.

One area in which the Semantic Web community differs from relational database community is the choice of data model. In Semantic Web, every data is represented as statement about resources using a triple. The advantage of this triple-based approach is representation of data and service by the application. Flexible data representation provides the general ways to represent any kind of data regardless of the domain. In addition, it has a possibility to enhance the quality of service by semantically-enriched

information such as inference. There are many researches which have struggled to map and transform the enterprise data in the relational database into triple database. However, there is no evident success of triple database for the practical application despite of the sophistication of existing researches until now. We believe that this failure result from the lack of deliberate consideration of consistency guarantee of traditional database in the Semantic Web region. In other word, there are some problems of current RDB-to-RDF mapping researches [31] for enterprise databases in the viewpoint of practical application of triple database.

In this thesis, we aim to realize practical triple database systems with the fundamental aspect of triple-based data model which can accommodate the entire information in the relational database, as well as the additional features of relational databases such as index structure and query optimization. In Section 1.1, we explain our research motivation. Section 1.2 overviews our contributions of research. In Section 1.3, we propose our research outline of the rest of the thesis.

1.1 Research Motivation

As we mentioned in Section 1, there are a need to transform relational databases to more flexible data format. The first reason why enterprise information should be re-constructed into triple database is data integration. With the standardized triple data format such as RDF, different types of data are interlinked each other to achieve the advanced service such as search and recommendation. On the other hand, there are amount of frequent changes of database by the purpose of management of information. RDB provides strict

and firm verification for checking Integrity constraints while those transactions are executed. We summarized the motivation of our research with two keywords: Flexibility and Utilization.

Flexibility. Flexible data representation is the main advantage of triple-based approach. It is important to exploit heterogeneous data with unified data representation format. Any information can be represented by triple-based data model without regard to their data types of schema information. Figure 1 shows an example of unified data representation with triple database which are retrieved from separated enterprise information with relational databases.

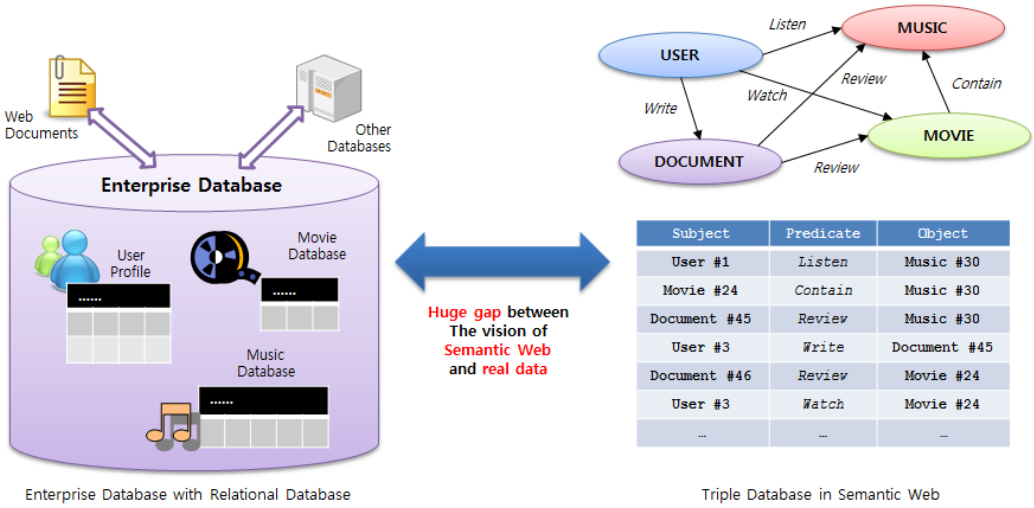


Figure 1: An example of unified data representation with triple database

Utilization. With flexible data, triple database may enhance the quality of service and enable advanced content services such as recommendation. Figure 2 and Figure 3 denote an example of content-based music recommendation by schema information. For recommend songs of

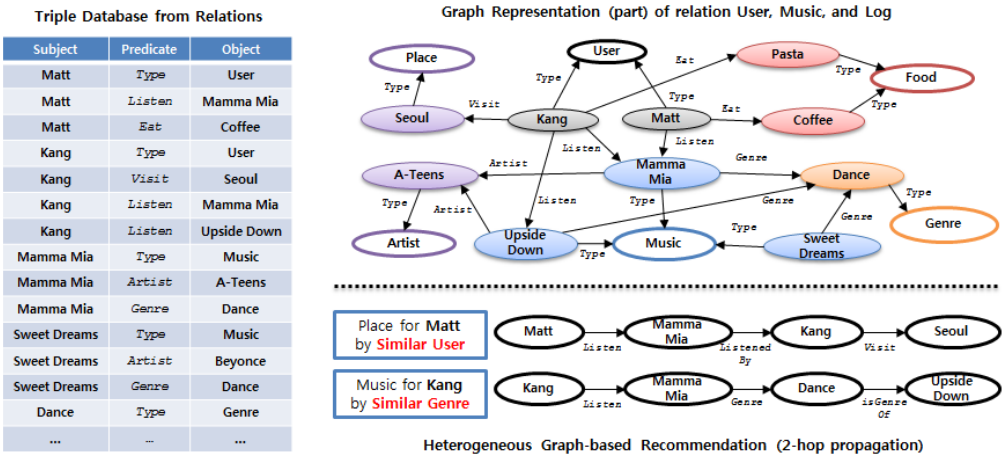


Figure 2: An example of recommendation with triple database

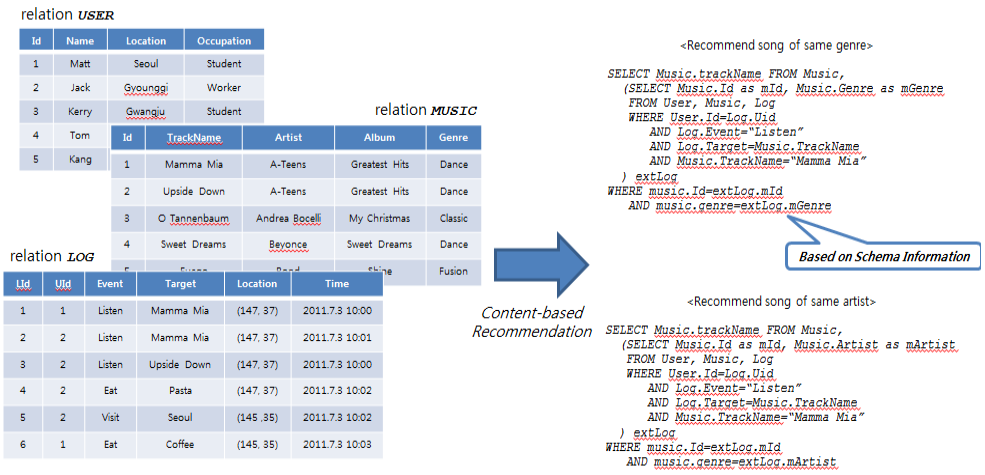


Figure 3: An example of recommendation with relational database

same genre to user with given relation USER, MUSIC, and LOG, content providers should understand the schema information for such as properties of attributes and connections between relations. If the schema information has been changed, request queries have to be rewritten by the changes of schema information. On the other hand, with triple database and heterogeneous graph which is derived from a set of triples, recommendation can be accomplished without regard to schema information since the recommendation is built on

the characteristics of graph structure such as hop propagation. Even though the design of original database has been changed, content provider need not to consider re-design of service system. In detail, we describe the difference between relational database and triple database in Table 1.

Table 1: Difference between relational database and triple database

Relational Database	**Triple Database**
"**Syntax**" is important for knowledge representation	"**Semantic**" is important for knowledge representation
Recommend **homogeneous** features (without schema information)	Recommend **heterogeneous** features (based on rule propagation and graph search)

So far, there are many attempts to use triple databases for practical applications. There can be many categorizations of triple database systems; however, we categorized the approaches to triple database as three ways.

Firstly, relational database can use triple database as redundant data storage. In other word, there are two duplicated data inside relational database and triple database. This RDB-with-Triple approach has some merits such as fast query response and concrete and safe system organization. On the other hand, redundant data storage requires complex system integration for synchronization. Ontology search system for PPS (Public Procurement System) [36] is based on this approach

Secondly, system uses relational database as a primary storage while triple database resides as a view of database. In this case, triple database is a simple view of relational database. *Virtuoso RDF view* [26] and some other approaches [13, 59] follow the principles of this approach. With RDB and the

view of triple representation, database administrator can organize systems simpler. In addition, there is no need of consideration of database update since triple database is only exists as a virtual table. Applicability is guaranteed because the legacy application only use existing relational database while advanced application is connected with the view of database. However, it does not provide enough flexibility of data because the representation of data still follows the way of traditional relational database. Various extension of service extension is also limited by the lack of physical storage of triple database

Finally, there can be triple database as a primary storage which adapts the essential features of relational database, while relational database has a supporting role to make a point of contact between legacy applications and triple databases. In this approach, a set of triples is transformed from relational database and stored into the physical storage with a given simple schema. System adapts the fundamental features of relational databases such as relationships and constraints. Applications use triples as the main data source with the support of triple database systems. This is a main idea of several researches which focus the *transformation of triples* [8, 14]. With this approach, system obtains flexibility of data representation without loss of feasibility of application. The organization of system is also simplified since it does not contain any data duplication between relational database and triple database. The demerits of this approach are applicability and reliability of database. Although many researches attempt to use Semantic Web into practical application, we cannot explain the successful case of practical solutions of Semantic Web until now. In addition, it has to keep additional information to guarantee the same expressiveness and reliability of relational

database. Without the enforcement of integrity constraints, triple database cannot provide the complete operations of relational database.

To evaluate and compare these three approaches, we defined the main criteria into six categories:

1. Legacy Application Support (Supporting legacy applications from traditional database)
2. Service Extension (Adapting semantic service with triple database)
3. Database Consistency (Enforcement of integrity constraints)
4. Performance (Applicability of database)
5. Flexibility (Degrees of flexibility of data mode)
6. Feasibility (Execution of conventional database operation).

Table 2 shows the summary of comparison of three approaches with these criteria. By this result, we strongly believe that triple database can replace the primary storage if there are an adequate enforcement process of integrity constraint for database consistency and some optimization techniques for performance.

Table 2: A summary of comparison of three approaches

	RDB with TDB: TDB as a Redundant Storage	**RDB is a primary storage: TDB as a View**	**TDB is a primary storage: RDB as a view**
Legacy Application Supplement	++ (Support existing service with a RDB while extending semantic service with a TDB)	++ (using RDB as a database for existing service: triple can support additional view of application)	+ (TDB can support existing application with specific conditions such as integrity constraints)
Service Extension	0 (triple database can be applied for service extension with synchronization with RDB)	-- (only triple view support semantic application: limited service extension)	++ (Triple database is a primary storage: easily adapt or extend semantic application with the triple database schema)
Database Consistency	- (Synchronization is needed for every database transaction: data redundancy problem)	++ (Traditional RDB provides powerful transaction management: No need for consider database consistency)	0 (Enforcement of integrity constraint should be support for database transaction within triple database)
Performance	0 (RDB operations is very fast: TDB operations are relatively slow for synchronizations of update operations)	++ (same as the performance of RDB: relatively high performance)	- (Triple database is not suitable for heavy database transactions without optimizations until now)
Flexibility	0 (redundant storage of triples give flexible data model for applications)	-- (based on strict data model of RDB: low flexibility)	++ (base data model is triple: high flexibility)
Feasibility	- (feasibility is guaranteed with base RDB: triple database should have same power for execution of queries of RDB)	+ (based on the feasibility of RDB operations such as database transactions)	0 (Triple database should be guarantee some reasonable feasibility such as select and update operation)

1.2 Our Contributions

While there are several approaches to transforming relational database to triple database, there are some problems in data integration scenarios in existing researches.

Information Preservation. Generally triple mapping scheme focuses to map RDF triples in a single giant triple table with generic attribute such as subject, predicate, and object. Generally it requires more sophisticate rules to preserve its entire information including relational schema information. Most of the constraints (e.g. primary key constraint and foreign key constrain) that capture the semantics of relational database rely on metadata of database. However, many researches of triple database do not consider the preservation of constraint information which is prerequisite of data transformation. Several approaches have introduced basic mapping rules for constraints on schema level [18, 48].

Nevertheless, the process of enforcement of integrity constraints is mostly relying on automatic reasoning process of inference engine. Since the limitation of inference engine, automated process using reasoning engine would be very slow, since as the number of triples in the database scales to verity the integrity of databases. As a result, we believe that the current approaches about RDB-to-RDF mapping do not provide enough solution for preserving semantic information of conventional relational databases.

Applicability. If we assume that enterprise would decide to use triple database as their main data repository instead of relational database, there

should be a number of transactions among database operations. There are two problems while transactions are executed on triple database; In case of transaction in SELECT statement, it is necessary to perform the n-ways subject-object self joins over triples [1]. In addition, in case of transaction in UPDATE statement, every update operation can harm the consistency of triple database. In other word, it should be reflect the change of triples in RDF as well as the enforcement of integrity constraints of conventional databases. In the viewpoint of these problems, most of current triple-based mapping techniques do not consider issues of applicability. We believe that well-designed index structure and optimized query execution will be help to enhance the performance of current triple-related approaches.

Data Redundancy and Synchronization. As one of the solutions for guaranteeing information preservation and applicability, both relational database and triple database can be applied concurrently in the database system. In particular, relational database holds the entire information so that legacy application is connected with relational database without any changes of applications, while triples are used as secondary facility for enriching the operation. With this scheme, data redundancy problem is certainly occurred between two different databases. System should be kept same data for two databases without any violation of database consistency. Every operation for database update should be applied within two databases by synchronous or asynchronous processes. However, synchronization process between two databases is a kind of ad-hoc solution so that it should be corrected according to the changes of update queries. In addition, it should support the complex and difficult synchronization process when synchronizing process is failed

by external causes. Example of three problems of PPS ontology search system is depicted in Figure 4.

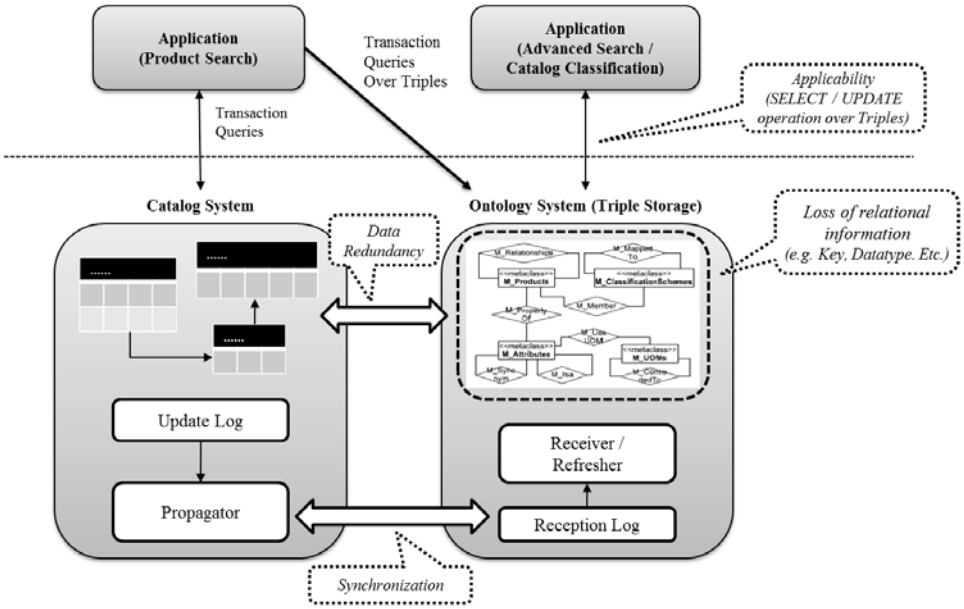


Figure 4: PPS ontology system

PPS ontology system has two separated systems, such as catalog system based on relational database and ontology system based on triple database. Despite of the complex synchronization modules between two databases, irreparable errors caused from data redundancy often happened during service time. By these problem statements, we propose simple approach using a triple as the main database for enterprise information with considering the enforcement of integrity constraint in the triple database. Basic transformation process of relational database follows the result of traditional triple transformation researches such as triple storing technique [42, 60] while we extend the coverage of triple database within the metadata of relation by separating the information of constraints of original database in the small table. In other word, triples only represent the content of target

database while metadata information is referenced during the enforcement of integrity constraint for the execution of database creation and update. The proposed approach contains support features to speed-up the operations which are related with enforcement of integrity constraints on triple database. Triple index structure will be one of the solutions, which can exploit the equivalent potentiality for the practical applications compared to the relational databases. We analyze effective index structure on triple database with lightweight index tree and payload bucket. Lastly, for solving the applicability problem of triple database, we propose enhanced shortcut selection for reducing self-joins in triple databases. Not only integrity checking process, but also general query execution process can utilize these two additional features which are originally derived from the relational database.

In Section 3, as background of our research, we formally define the principles and terminologies of triple data model. It contains the basic rules of design of triple database based on the problem definition. Then, we describes a transformation process from relational database to the set of triples, which is brute-force method to build a triple database by transforming the relations, attributes, and value of tuples in relational databases. For preserving the semantic information of relational database such as the relationships among relations, tables for metadata of database is generated within the triple database, separated from triples which denotes the contents of database. General triple representation scheme of our research also describes in this Section.

With the definitions of triple database and triple data mode, we focus on the integrity constraints on triple database in Section 4. We propose the

general definition of integrity constraint on tripe database. Actually the conditions which decide the validation of database state are come from the traditional relational database. We adapt the core concepts of integrity constraint of relational database, and transformed those conditions fitting to our triple data model. There are six major integrity constraints in relational database, such as primary key constraint, functional dependency, referential integrity as known as foreign key constraint, not null constraint, unique constraint, and user-defined domain constraint. We describe these six integrity constraints from the viewpoint of triple database, and analyze the process for guaranteeing database consistency without and violation of transformed constraint conditions. Performing operations for checking violations of integrity constraints on triple database is another process besides the logical definition of integrity constraints. Depending on given conditions, processing of constraint violation check would be compromised.

In Section 5, we describe the index structure for integrity constraint of triple database. Actually the process of constraint violation check can be fairly inefficient process for scanning entire triple database. Because the overhead of scanning triple database for every check process, adequate index structure can be substantial if the triple database is often updated or changed. In particular, the violation of integrity constraints is examined using the index structure of a set of attributes in relational databases. Similarly, there should be a tree index structure for a triple database to derive the equivalent state of the database. in this Section, we propose a new index structure called Tridex for indexing triples based on the less heightened B+-tree and payload bucket for remaining information. The index structure of Tridex allows us to efficiently find the elements of triples given by the condition of integrity

constraint check process. The separated role of each triple is indexed within one large B+-tree. In addition, every data (leaf) node of B+-tree has unique payload for additional data, which does not appear in the index nodes. The difference between Tridex and other triple index approaches is avoidance of data duplication. In general, at least six accesses of the index tree is required for one process of triple pattern finding in existing triple index solutions. Since there are data duplications among multiple index trees, it causes redundant storage and unnecessary query processing costs. The experimental result shows that Tridex can give better performance for query response time and index update time with less memory consumption, compared to existing solutions.

In Section 6, we focus on the actual query execution on triple database. Majority of triple database is three-column table which can be presented in directed graph, where nodes are concepts and edges are relationships between concepts. Despite of firm description of triple database and triple index structure, querying path expression using triple database is still expensive [1]. Generally path expression queries require $n-1$ subject-object joins where *n* is the length of the path. Nevertheless triples are stored in different schemes according to the policy of triple store, self-joins between triple table is one of the unavoidable problem which leads to performance degrade of triple database.

Figure 5 shows an example of triple database represented by the graph with simple triple pattern finding query. For example, starting from the mode '*Kang*', if we want to find all genres of songs that user '*Kang*' listened, we can get the names of songs '*Mamma Mia*' and '*Upside Down*' where the genre of two songs is 'Dance'. We can intuitively understand that this simple path-following query is actually executed by the join operation between two triples. Firstly, query engine scans the triples in triple database whose pattern satisfy the given triple pattern $<Kang\ Listen\ ?x_1>$. Likewise, query engine also try to find the connected triple with the next given condition $<x_1\ Genre\ y_1>$ using retrieved set of triples $\{(Kang, Listen, Mamma\ Mia), (Kang, Listen, Upside\ Down)\}$. Every query plan to retrieve results contains subject-object join to connect given two triple patterns. If triples are stored in one giant triple table, it can be drawback of decreasing performance drastically. If we could shrink path expression using adequate techniques,

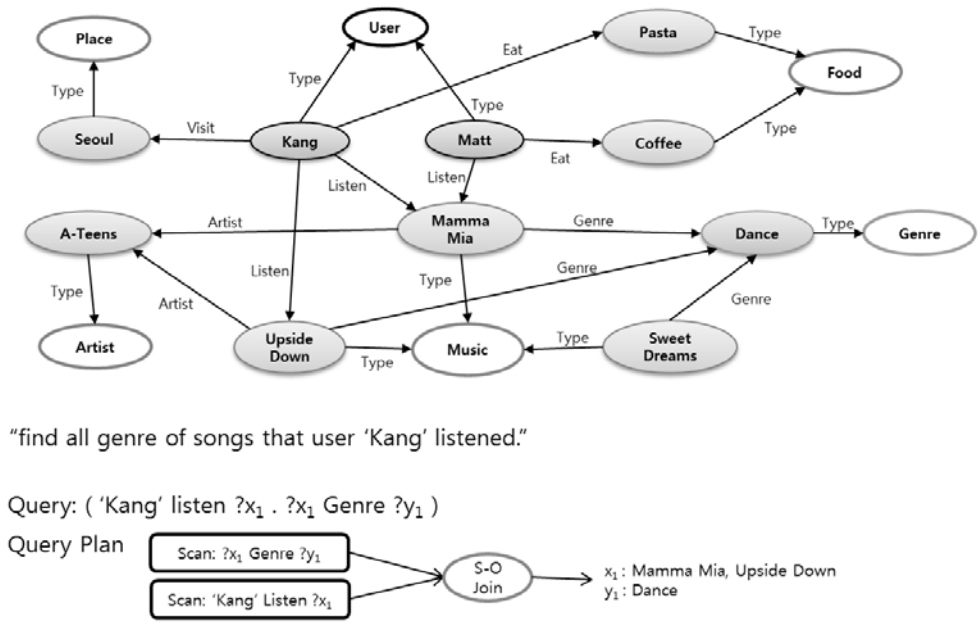


Figure 5: An example of triple database with self-join query

this self-joins could be eliminated. For achieving better performance in formulating and executing frequent path queries, concept of "*Shortcut*" is introduced. Shortcut is a path that corresponds to frequently accessed paths by augmenting the schema and data graph of a triple repository with additional triples [23]. The problem is that how we can choose adequate set of shortcuts within limited time and space to maximize the benefit of shortcuts. Thus, we propose a novel technique for eliminating self-joins by shortcut selection with shortcut ranking measure. Our measurement of shortcut selection is focused on three parameters of improvement of query performance [43]; query processing cost, shortcut building cost, and space and time limitation. Since our triple is based on relational databases, we can optimize shortcut selection using schema information of relational database. In addition, whenever a triple database is changed, existing shortcuts have to be changed in order to up-to-date query results. We apply the concept of database update into the measurement of shortcut selection, so that proper set of shortcuts which are less affected by update operations can get better benefits than update-sensitive shortcuts. To validate our proposed methods, we performed experiments on two datasets, and the experimental results show that our approach outperforms existing solutions in the viewpoint of query response time and space limitation.

The main contributions of the thesis can be summarized as follows:

- We explain the fundamental rule of triple database based on the triple transformation. Our approach has advantages in that triple data format is flexible and representable for heterogeneous data without any understand of database schema design.

- We suppose the essential conditions to guarantee the valid state of triple database. The enforcement of six integrity constraints on triple database is the prerequisite conditions for using triple database practically.
- We propose a novel triple index structure named Tridex, which is defined on triple database, and explain how to exploit the validation of triple database using triple index. By reducing the tree size and data duplication, our index structure has novelty for efficient triple pattern finding problem.
- We present a shortcut selection technique for reducing self-joins on triple database, and explain how to decide the adequate set of shortcut for triple database. By adapting update factors and schema information, result of shortcut selection can be more precise.

Following picture gives an overview of system which contains the comparison with relational database system and triple database system. Essential features for operating relational database are mapped to equivalent features of triple database, and denoted triple database contains four contributions of our system.

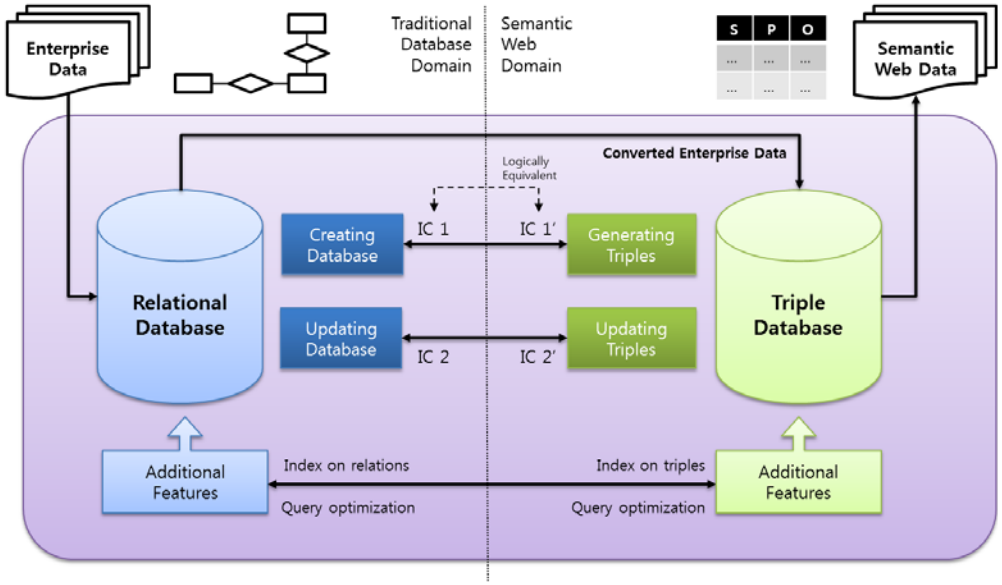


Figure 6: System overview

1.3 Outline

The rest of the thesis is organized as follows. In Chapter 2, we review the related work. Chapter 3 describes the background knowledge which needs to understand the fundamental aspects of triple database. Then, we explain integrity constraint on triple database and database operations for guaranteeing consistency of triple database in Chapter 4. In Chapter 5, we propose an index structure for integrity constraints on triple database. Chapter 6 shows a reduced candidate shortcut selection for reducing self-joins in triple database. In Chapter 7, we summarized our work and conclude the thesis.

Chapter 2. Related Work

Our work aims to build practically usable triple database by considering the essential features of relational database. The range of work covers wide and various research categories, including triple data representation, data format, and triple storing and querying framework. In this chapter, we introduce current researches about triple mapping and querying, integrity constraints on triple database, and additional structures supporting triple database.

This chapter is organized as follows. Section 2.1 and 2.2 gives explanation about existing approaches to achieve mapping and storing triples from relational database. Then, we cover current researches about triple ndex structure on triple database in Section 2.3. Section 2.4 summarizes the representative approaches mentioned in this chapter.

2.1 Triple Mapping

With the requirement of exchange data between applications and enterprise, the research about triple mapping and storing in Semantic Web has been made to apply triple data representation for the content of relational database. RDB-to-RDF mapping is one of the representative approaches for describing enterprise information with triple data format. Since RDF is the standard model for data interchange on the Web [55], W3C RDF Working Group [54] becomes the leader to map the content of relational database to RDF. In Direct Mapping of relational data to RDF by W3C RDB2RDB Working Group [5], data in a relational database is taken as input for defining an RDF

graph representation with a set of common datatypes. The speciation of direct mapping is described by mapping language, named R2RML [22]. R2RML is RDB-to-RDF mapping language that allows the creation of customized mapping from relational data to RDF. Direct mapping approach uses the relational tables to classes in an RDF vocabulary and the attributes of the tables to properties in the vocabulary. The goal of direct mapping is to expose a RDB on Semantic Web with exact statement transformation by pre-defined given rules. For example, the following SQL DDL (Data Definition Language) which means the creation of two tables with single-column primary key and one foreign key reference between them:

```
CREATE TABLE "Addresses" (
     "ID" INT, PRIMARY KEY("ID"),
     "city" CHAR(10),
     "state" CHAR(2)
)

CREATE TABLE "People" (
     "ID" INT, PRIMARY KEY("ID"),
     "fname" CHAR(10),
     "addr" INT,
     FOREIGN KEY("addr") REFERENCES "Addresses"("ID")
)

INSERT INTO "Addresses" ("ID","city","state") VALUES (1,'Boston','MA')
INSERT INTO "People" ("ID","fname","addr") VALUES (7,'Bob',1)
INSERT INTO "People" ("ID","fname","addr") VALUES (8,'Sue',NULL)
```

By given a base IRI http://www.example/, the direct mapping of the example database is represented by a RDF.

```
@base <http://www.example/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .


<People/ID=7> rdf:type <People> .
<People/ID=7> <People#ID> 7 .
<People/ID=7> <People#fname> "Bob" .
<People/ID=7> <People#addr> 1 .
<People/ID=7> <People#ref-addr> <Addresses/ID=18> .
<People/ID=8> rdf:type <People> .
<People/ID=8> <People#ID> 8 .
<People/ID=8> <People#fname> "Sue" .

<Addresses/ID=18> rdf:type <Addresses> .
<Addresses/ID=18> <Addresses#ID> 1 .
<Addresses/ID=18> <Addresses#city> "Boston" .
<Addresses/ID=18> <Addresses#state> "MA" .
```

Direct mapping is simple and concrete mapping scheme between relational database and RDF. Its simplicity helps users to understand and implement the language. However, it does not fully support any of the additional features such as integrity constraints and static metadata. Therefore, additional mapping languages which support additional features have been proposed [11, 15, 38, 10, 31].

Virtuoso RDF Views [27, 46] is another representative framework for RDB-to-RDF mapping. It is developed by Openlink Software, where it focuses on generation of mapping between RDB and RDF automatically. Virtuoso RDF Views provides the RDF Views by following rules: Definition of RDF class IRI for each table, Construction of a subject IRI for each primary key column value, Construction of a predicate IRI for each non-key column. At the most basic level, Virtuoso RDF views rely heavily on the specified SQL SELECT query to transform the result into a set of triples. Thus, it has major drawbacks when the semantics of databases are hidden in SQL string so that it is not easily accessible without the knowledge of target database.

Besides, there are other approaches using mapping languages to map relational database into triple database. Astrova et al. supposed the rules for mapping SQL relational databases to OWL ontologies [7][8][9]. It focuses on generating ontology by syntactical components (e.g. OWL vocabularies) with transformation rules. For example, a table in relational database is mapped to a class unless all its columns are foreign keys to two other tables. Foreign key of table is denoted by object property of OWL ontology. Similarly, a column is mapped to a data type property with a given

cardinality. If the column has to have at most one value for each row in the table, the maximum cardinality of mapped property is 1. In this way, rule-based transformation is executed, while ontology is produced from relational databases. Figure 7 illustrates the basic idea of *rule-based transformation*. A relational database is transformed to an ontology using a set of rules called *mapping rules*.

The main drawback of this approach is that it only considered the

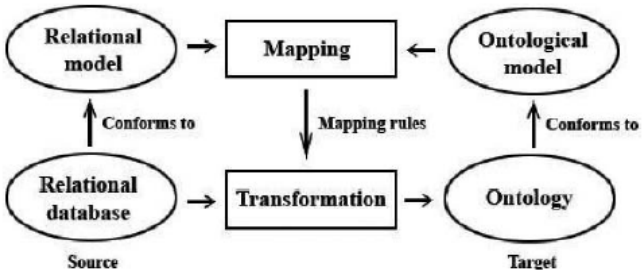


Figure 7: Transformation of relational databases to ontologies

representation of relational database without any chance of database update. Unless the mapping process is completed successfully, ontology should be rewritten by mapping rules if triple database is updated at least once. In this manner, for verifying the integrity of triple database, System should use inference engine for checking the validation of mapped data. For example, if system administrator adds one instance to the mapped database, whole ontology validation process should be executed for every change of database by given mapping rules. We design a small experiment for ontology validation for database consistency using inference engine. System holds a dumped DBPedia [15] knowledge base which contains about 275 million RDF triples with more than 2.6 million entities. For verifying integrity of loaded RDF data, we check the data loading time and validation time

according to the increase of the number of triples. As we depicts in Figure 8, whole ontology validation process takes enormous time for small database update. As a result, we believe that there should be a generous and efficient validation scheme for triple database without any rule-based integrity check process.

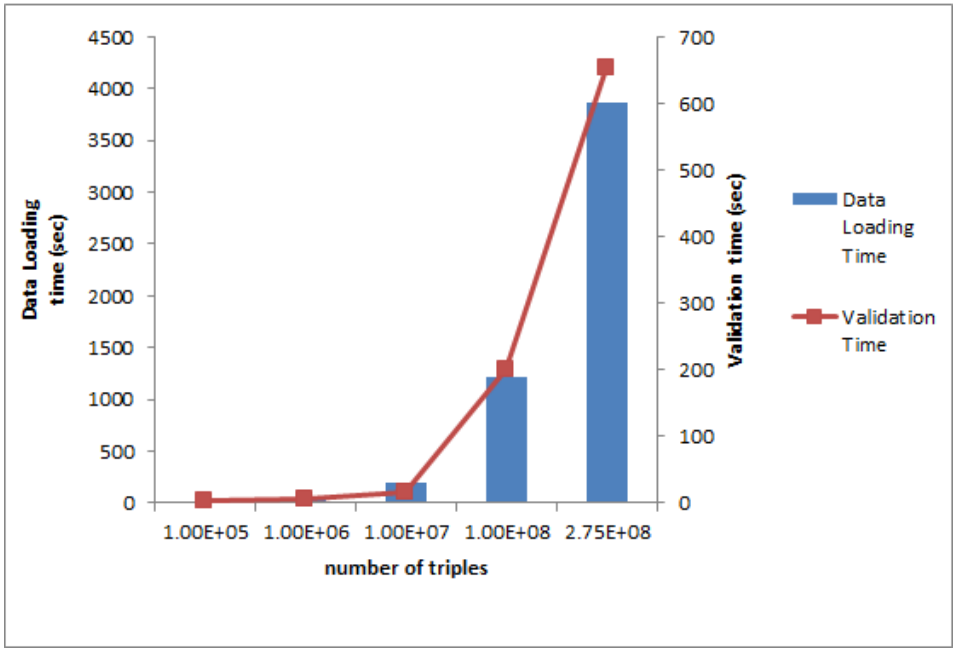


Figure 8: Ontology validation by inference engine

2.2 Triple Storing

To management of increasing volume of Semantic Web Data, various RDF storage methods were proposed to improve the efficiency of triple database. Generally, most approaches of triple storing rely on the direct mapping. In fact, Tim Berners-Lee described a relationship between the Semantic Web and relational databases and supposed a brute-force approach for direct mapping the content of relational database into RDF: [22]

- A record is an RDF node;
- The field (column) name is RDF propertyType;
- The record field (table cell) is a value.

In general, most of academic researches which are related to connect RDB and triple are rely on this concept of mapping. Despite of the variation of representation which is derived in the characteristic of information domain, main concept of mapping technique is still based on the same idea of brute-force approach.

Most notable techniques for storing triple data are is *Triple Store* [16], *Vertical Partitioning* [2], and *Property Table* [58]. In Triple Store, all triples are stored in a single giant table with three columns. Each columns corresponds to the roles of triples, such as *subject*(*S*), *predicate*(P), and *object*(O). Therefore, the space requirement is determined by the number of triples in triple database. For evaluating queries on triple database, the triple table is joined once for each triple in the graph pattern, where variables in the query are bound to their values when they encounter the slot in which the variable appears. Thus, the more query is extended; the more performance of triple store is decreased rapidly. It is the main drawback of Triple Store. However, it is referenced and implemented as the main storage for triple database due to its simple data structure and flexible representation format. Figure 9 and Figure 10 depict a sample triple graph and its relational representation of the given triple graph. *RDF-3X* [44] proposed by Neumann et al. and *Hexastore* [56] proposed by C. Weiss et al. are representative RDF storage scheme which adapt the Triple Store technique.

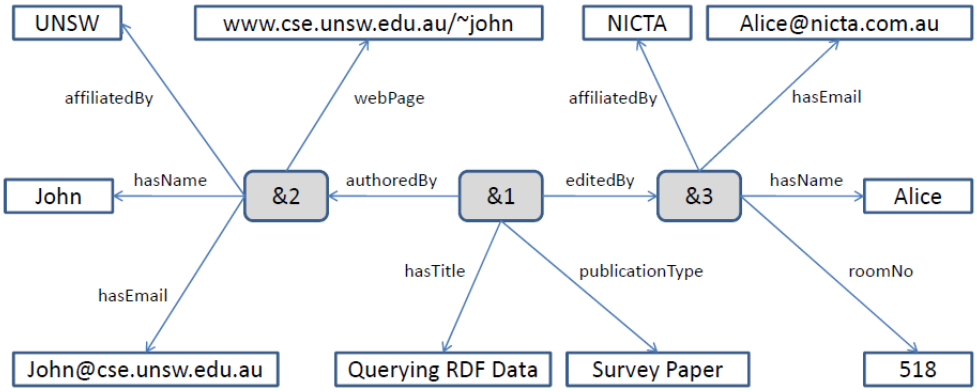


Figure 9: An example of triple graph

Subject	Predicate	Object
Id1	publicationType	Survey Paper
Id1	hasTitle	Querying RDF Data
Id1	authoredBy	Id2
Id2	hasName	John
Id2	affiliatedBy	UNSW
Id2	hasEmail	John@cse.unsw.edu.au
Id2	webPage	www.cse.unsw.edu.au/~john
Id1	editedBy	Id3
Id3	hasName	Alice
Id3	affiliatedBy	NICTA
Id3	hasEmail	Alice@nicta.com.au
Id3	roomNo	518

Figure 10: Relational representation of Triple Store

Abadi et al. [2] have presented *SW-Store* as a new DBMS which stores triple data using a fully decomposed storage model called vertical partitioning. In vertical partitioning, a separate table is created for each distinct predicate to store all triples that features this predicate. Triple table is rewritten into *n* two-column tables where *n* is the number of unique properties in the data. The first column of separated table denotes the subjects that define that property, where the second column contains the

object values for those subjects. Consequently, the value of predicate is omitted and only the values of the subject and object are stored in table. Every partitioned table is sorted by the value of subject, so that the specified set of subjects can be located very quickly. The space requirement of vertical-partitioned table is proportional to the number of triples, as well as the number of unique predicates in the triple database. One advantage of this approach is that while one advantage of this approach is that while property tables need to be carefully constructed so that they are wide enough but not too wide to independently answer queries, the algorithm for creating tables in the vertically partitioned approach is straightforward and need not change over time. Moreover, it has an advantage for fast merge-joins among the tables since every table is sorted by subject. Bandwidth utilization and data compression is also improved due to the locality (e.g. same predicate) of the database. On the other side, vertically partitioned table have the important disadvantages, such as increased cost of inserts and increased tuple reconstruction cost. Since every table is separated from one three-column triple table, multiple distinct tables have to be updated for each inserted triple. Figure 11 illustrates the example of vertical partitioning with the triple database in Figure 9.

In property table, triple data are stored in relations in traditional relational databases whose schema carries the semantics of the residential data. The main objective of property table is to reduce self-joins involved with the triple database. In principle, applications typically have access specific patterns in which certain subject and predicates are accessed together. For reducing self-joins during triple pattern accessing, Property tables clustered the same set of predicates which have the same subjects into a

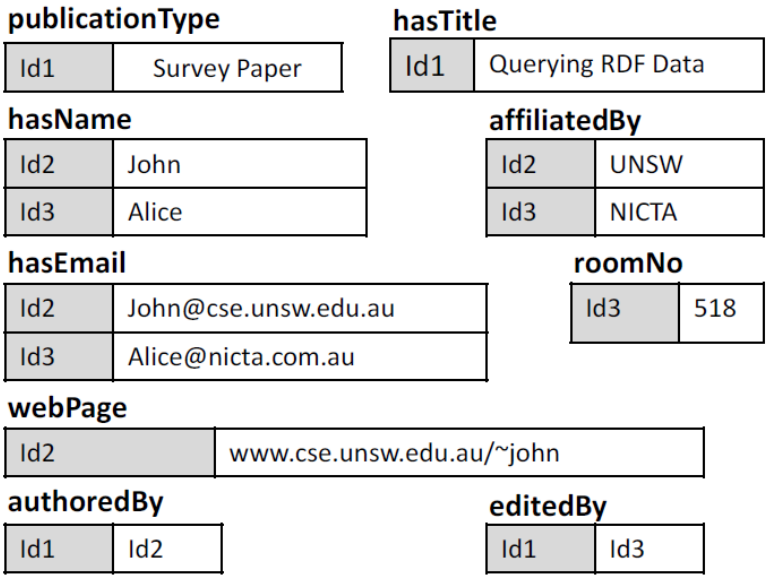


Figure 11: An example of vertical partitioning

property table. (Clustered Property Table). Vice versa, table is created for each class and store information about all triples of that class, while the attributes of created relations are fulfilled with single-value predicates of selected triple (Property-class Table). Unlike the first type of property table, a property may exist in multiple property-class tables. Both property table store all instances of the predicate of triple database where that predicates does not appear in any other table used for the triple database. The space requirement for property table consist of two parts: the space required for storing the property table, and the space required for storing the additional information which are not stored in the property table (leftover table). *Jena2* [57] and *Oracle* [19] adapt the property table for their triple storage scheme. Using the knowledge of the frequent access patterns to construct the property-tables and influence the underlying database storage structures can provide a performance benefit and reduce the number of join operations during the query evaluation process. However, there are trade-off problem

with property table. If property table are made narrow, the average value density of the table increased and the table is less sparse. On the other hand, if many properties are included in a single giant property table, the number of *NULL* values in the property table is increased, that lead to space wasting. Figure 12 depicts an example of property table which represent the triples in Figure 9.

Publication

ID	publicationType	hasTitle	authoredBy	editedBy
Id1	Survey Paper	Querying RDF Data	Id2	id3

Person

ID	hasName	affiliatedBy	hasEmail	webPage	roomNo
Id2	John	UNSW	John@cse.unsw.edu.au	www.cse.unsw.edu.au/~john	
Id3	Alice	NICTA	Alice@nicta.com.au		518

Figure 12: An example of property table

2.3 Index Structures for Triple Database

Validating the consistency of triple database with given constraint condition is time-consuming process as mentioned in Section 1.2. To facilitate efficient consistency checking process as well as query answering, multiple indexing techniques for triple database were proposed.

With the definition of a triple representation scheme, there are some conventional solutions to triple indexing. Real-world SPARQL queries usually consist of multiple triple access patterns such as <*?p ?q ?r*>. Because

most simple triple access patterns require sequential lookup of atoms, triple should be indexed so that the lookup process can be scaled to a real-time query process. Generally, a B+-tree is used for the indexing scheme of a triple database, similar to a relational database. Much effort has been invested in designing index trees to facilitate efficient query answering. There are three state-of-the-art solutions for multiple indexing techniques: *multiple access pattern (MAP)*, *HexTree*, and *TripleT.*

MAP [29] is a tree data structure that builds six clustered B+-tree indices on six triple store tables. Each tree is structured with one permutation of subject, predicate, and object: i.e., SPO, SOP, PSO, POS, OSP, OPS. As a result, all triples are indexed six times repeatedly on the leaf nodes of index trees. For example, triple (*id1*,*Name*,*Kim*) in Figure 5 is reassembled into six different forms of index key, where # is a predefined separator: {(*id1#Name#Kim*), (*id1#Kim#Name*), (*Name#id1#Kim*), (*Name#Kim#id1*), (*Kim#id1#Name*), and (*Kim#Name#id1*)}. Several triple store systems, such as Virtuoso [26] and RDF-3X [44], have adopted this approach. Figure 13 illustrate an example of relations (a), transformed triple database (b), and its graph representation (c). From the triple database, MAP retrieves the combination of six permutations of triples, and generates of six clustered B+-trees as depicted in Figure 14.

HexTree [56], also known as Hexastore, also builds six clustered B+-trees with combinations of triple ordering, similarly to MAP. The difference between HexTree and MAP is given by the value of the index key. HexTree uses only two of the three roles of the triple: i.e., SP, PS, SO, OS, PO, and OP. Leaf nodes of symmetric roles, e.g., SP and PS, share the same payload.

Figure 13. An example of relations and triple transformation

r_1

Name	Age	Gender
Kim	20	M
Lee	30	M
Choi	30	F

r_2

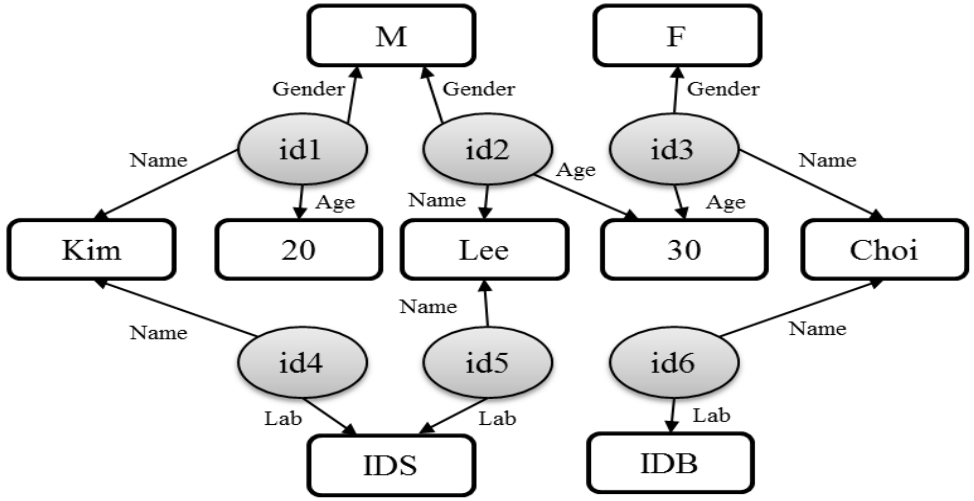
Name	Lab
Kim	IDS
Lee	IDS
Choi	IDB

(a)

T

S	P	O
id1	Name	Kim
id1	Age	20
id1	Gender	M
id2	Name	Lee
id2	Age	30
id2	Gender	M
id3	Name	Choi
id3	Age	30
id3	Gender	F
id4	Name	Kim
id4	Lab	IDS
id5	Name	Lee
id5	Lab	IDS
id6	Name	Choi
id6	Lab	IDB

(b)



(c)

For join queries, the SP index is queried to obtain *<a b ?x>* while another PO index is queried to execute *<?x c d>* for a simple access pattern. Then the two results are joined together to get a final result. HexTree has some advantages over MAP: concise and efficient handling of multi-valued resources, a reduction of data locality, and avoidance of *NULL* values. However, these two index schemes are inadequate to satisfy the requirement of reducing data duplication originating from multiple B+-trees. In the worst case, triple data are duplicated five times in HexTree and six times in MAP. This data duplication generally causes space overhead for storing the index and extra time costs for updating the index tree. Figure 15 depicts an example of HexTree index structure.

Rather than using multiple B+-trees, TripleT [28] uses only one B+-tree to index all distinct values, regardless of the role. The index key of TripleT can be subject, predicate, or object. The leaf node of B+-tree represents the distinct role of the triple. In addition, each leaf node in TripleT has an

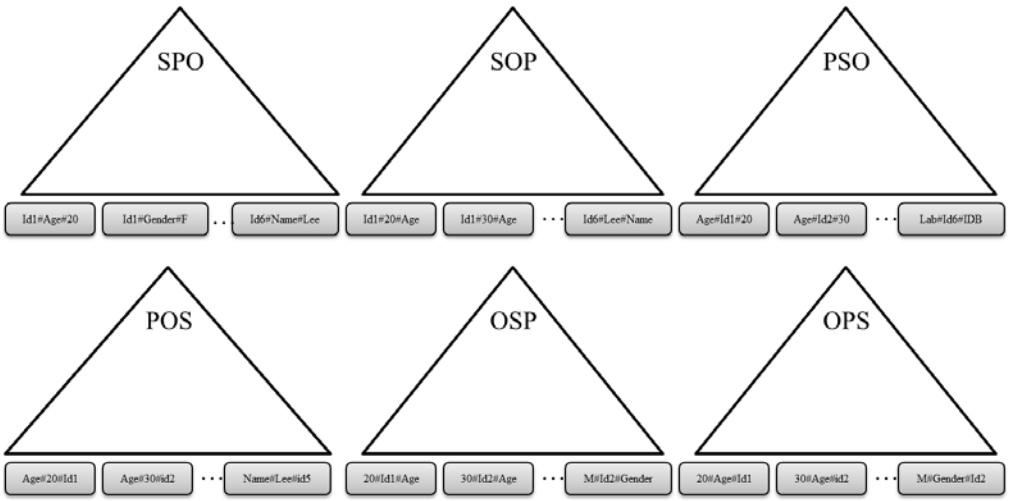


Figure 14. An example of MAP index structure

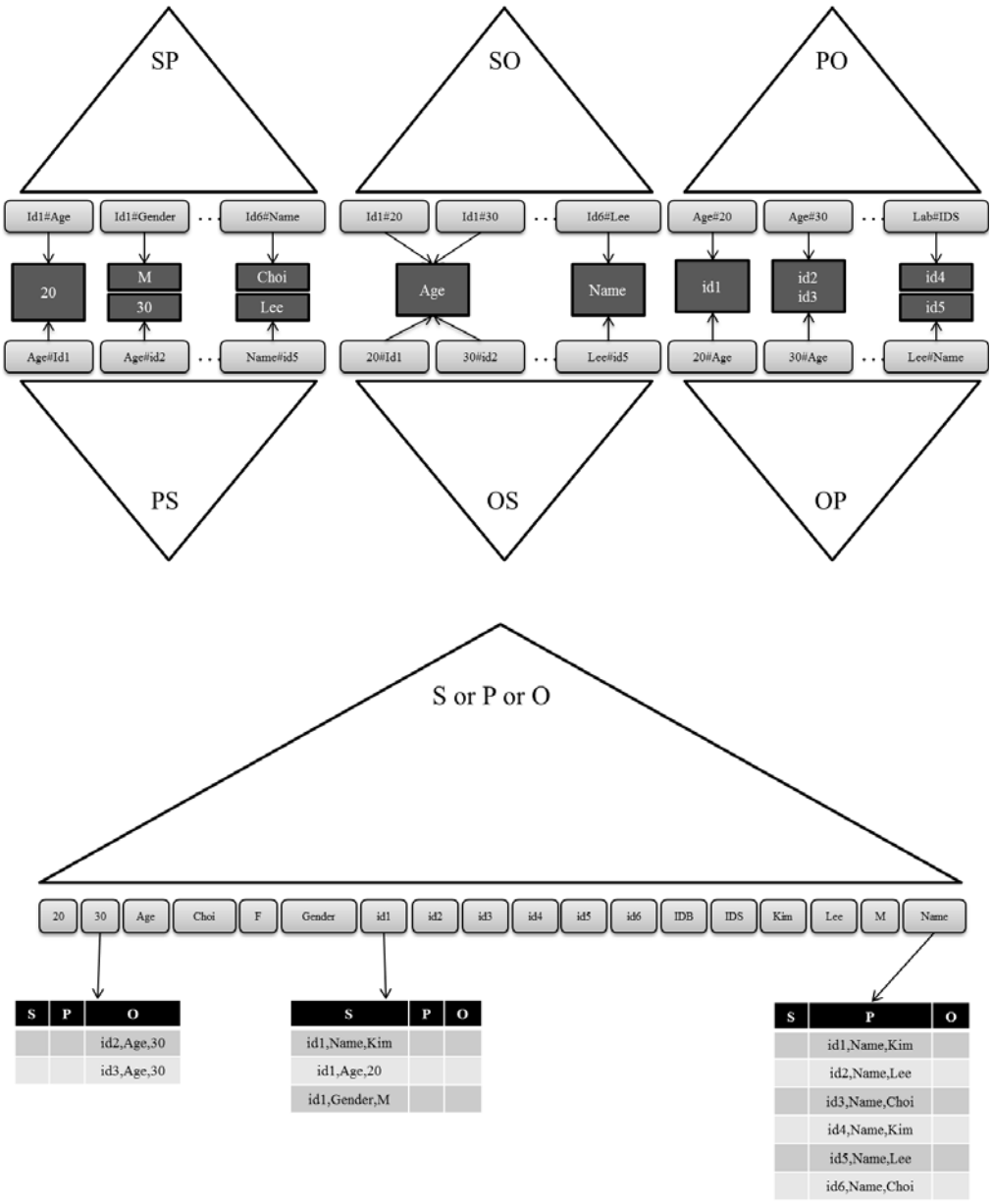


Figure 15: An example of HexTree and TripleT index structure

adequate payload that is split into three buckets, where each bucket contains the list of related atoms. As a result, additional space is required for the payload bucket. For example, triple (*id1,Name,Kim*) in Figure 5 is dissolved with three components: *id1*, *Name*, and *Kim*. The leaf node that contains

value *id1* has payload bucket [*id1*,*Name*,*Kim*] for the subject. Because no predicate and object has value *id1*, the predicate and object payload of *t* is empty in this case. Each triple in the database is stored three times in TripleT. A major drawback of TripleT is the oversized index tree, because all distinct values of triples are stored in a single B+-tree.

There are a number of existing studies that provide the fundamentals of triple database concerning various research issues. In this section, we summarized the representative researches explained in this chapter. Table 3 illustrates the representative approaches in an organized form of research dimension we presented in previous section.

Research Group	Name	Description	Applications
Triple Mapping	Direct Mapping	● Defines an RDF Graph representation of the data in any relational database ● Direct mapping takes as input a relational database and generates an RDF graph	-
	Virtuoso RDF View	● Focus on generation of mapping between RDB and RDF automatically ● Generating and publishing RDF views of relational data	Openlink Virtuoso DBMS
	Rule-based Transformation	● Focus on generating ontology by syntactical components with transformation rules	-
Triple Storing	Triple Store	● Each RDF triple is stored directly in a three-column table ● Quite simple and general, easy to manipulate	3Store RDF storage RDF-3X x-RDF-3X
	Vertical Partitioning	● n two-columns tables for triples ● Fast merge-join among the tables	RDFMATCH Path-based relational RDF database
	Property Table	● Clustering properties for speed up queries over self-joins	SW-Store
Triple Indexing	MAP	● Indexing scheme that utilizes six indices ● Every permutation of possible ordering of S-P-O consists one key in the tree	RDF-3X
	HexTree	● Only use two roles (e.g. SP, SO, etc.) for each triple index tree ● Payloads are shared between indexed with symmetric ordering	Hexastore
	TripleT	● Use only one big B+tree for indexing entire triple ● Leaf node of tree represents the atom of triple ● Each leaf node holds a payload	CHex

Table 3: A summary of representative researches of triple database

Chapter 3. Background

In this chapter, we formally define the terminologies and principles of triple data model, as the preliminaries for remained chapters. In addition, we outline the process of triple representation according to defined triple data model.

3.1 Terminologies and Design Principle

Basically we follow the traditional representation scheme of relational database. For the set of relations, there is relation schema $\mathrm{R} = \{r_1, r_2, \ldots, r_n\}$. Each symbol r denotes the corresponding relation. $r \subset D_1 \times D_2 \times \ldots \times D_n$ where D is a domain of database. Furthermore, there are set of attributes $\mathrm{A} = \{a_1, a_2, \ldots, a_k\}$ where $a(r)$ is a set of attributes of relation r. A tuple $\mu \in r_i$ where $\mu(a)$ denotes the value of the attribute a of the tuple μ in relation r_i.

In fact, there is intimate relationship between triple data format and relational database model. In our definition, triple representation strictly follows the fundamental concept of E-R (Entity-Relationship) Model. Since the relational database is logical data representation of conceptual data model, it is trivial that every concept of relational database can be mapped into the feature of logical data model of triple database based on the concept of E-R model. For example, with the E-R model of bank account, assume that "*customer*" entity has a set of attributes "*customer-id*", "*customer-name*", "*customer-street*", and "*customer-city*". In relational database, relation

"*customer*" is defined from the entity "*customer*", while the columns of relation is derived from the set of attributes of entity "*customers*". Similarly, there is one-to-one mapping relationship between E-R model and triple representation. Since predicate defined the association between subject and object, attributes of entity has been mapped into the predicates of triples, while the value of subject and object is obtained from the instance of entity "*customer*". In this way, we can obtain the basic rule of triple representation from relational database. Figure 16 shows the relationship between relational database and triple database. Every component of two databases are connected each other in the level of conceptual data model using E-R diagram.

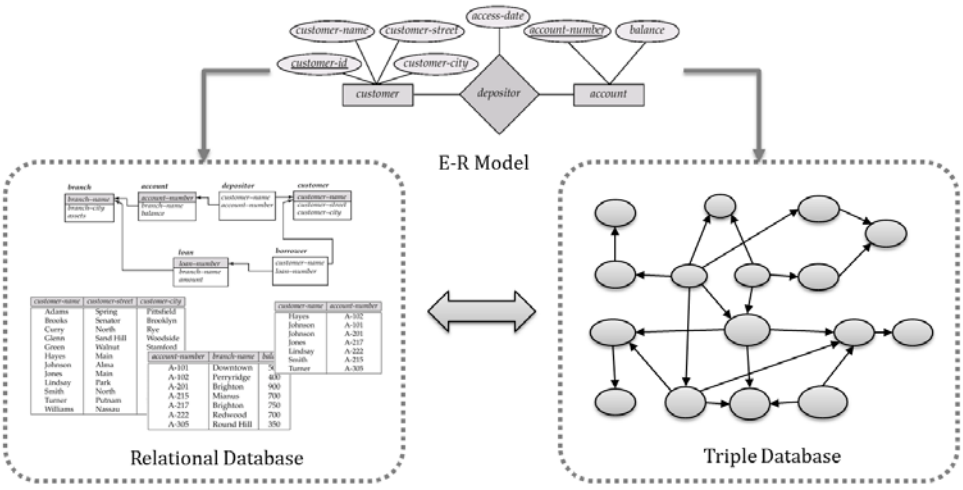


Figure 15: Relational database, E-R Model, and triple database

3.2 Triple Data Model

As we discussed in Section 2.2, our triple data model is based on Tim Berners-Lee's definition of triple database called Brute-force approach. In

general, most of academic researches which are related to connect RDB and triple are rely on this mapping concept. Despite of the variation of representation which is derived in the characteristic of information domain, main concept of mapping technique is still based on the same idea of brute-force approach. We also believe that it is one and only trustworthy reference to achieve the complete conversion between two database until now. As a result, we adapt the main idea of W3C transformation from relational representation to a triple transformation based on brute-force approach. Meanwhile, for securing the originality of our work as well as secure the contribution of consideration of integrity constraint, we define our triple model with extension of metadata of database such as key constraints based on the W3C's triple representation as follows.

3.2.1 Design of Triple Database

Definition 1. ***Triple and Triple Database****. Let Z be an enumerable set of atoms (e.g. keywords, URIs), then a triple* t *is defined by a set of three components (subject, predicate, object)* $\in \{Z \times Z \times Z\}$*. Triple database T is a set of triples* $\{t_1, t_2, \dots, t_n\}$.

Basically, triple $t = (s_1, p_1, o_1)$ means that the value of predicate p_1 in subject s_1 is o_1. *t(s)*, *t(p)*, and *t(o)* with triple t denotes the value of given role of t. For instance, *t(s)*=s_1 for previous example. In tuple μ, $\mu(a_i)$ denotes the value of attribute a_i of the tuple μ. Thus, key-value pairs $(a_i, \mu(a_i))$ can be mapped with the combination of predicate a_i and object $\mu(a_i)$. Since every tuple has discriminator such as primary key due to the

uniqueness of tuple, subject can be used for represent discriminator of tuple. Let tuple $\mu \in r_k$ and attribute set $A = \{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m\} = a(r_k)$, where $\{a_1, a_2, \dots, a_n\}$ is the set of primary key of relation r_k. Then, the concatenation of the value of primary key columns $\mu(a_1)|\mu(a_2)| \dots |\mu(a_n)$ can be discriminator of tuple μ so that the value of subject of triple t is $\mu(a_1)|\mu(a_2)| \dots |\mu(a_n)$. With this assumption, we describe a tuple μ in relation r_k by the set of triples.

Definition 2. ***Triple Representation.*** *Let tuple* $\mu \in r_k$ *and attribute set* $A = \{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m\} \in a(r_k)$ *where* $\{a_1, a_2, \dots, a_n\}$ *is the set of primary key columns of relation* r_k*. Then a tuple* μ *is represented by the set of triples* $\left\{ \begin{matrix} (\mu(a_1)|\mu(a_2)| \dots |\mu(a_n), a_1, \mu(a_1)), \\ (\mu(a_1)|\mu(a_2)| \dots |\mu(a_n), a_2, \mu(a_2)), \\ \dots, \\ (\mu(a_1)|\mu(a_2)| \dots |\mu(a_n), b_m, \mu(b_m)) \end{matrix} \right\}$.

3.2.2 Triple Transformation

Definition of triple and triple representation provides the essential condition for representing tuples with triples: guarantee of completeness of entire information of relational database (we already mention of information preservation in Section 1.2). The following lemma shows the completeness of triple database which contains entire tuples of relational database.

Lemma 1. ***Complete Transformation of Relational Database.*** *Assume that there is relation* $r = \{\mu_1, \mu_2, \dots, \mu_n\}$ *with a set of attribute* $A = \{a_1, a_2, \dots, a_n\}$ *where the primary key is* a_1*, and a set of triples T*

which are derived from r. *Then, any tuple* $\mu_k \in r$ *for* $1 \leq k \leq n$ *can be represented by a set of triples* $T' = \{t_1, t_2, \dots, t_n\}$ *if and only if* $t_p(o) = \mu_k(a_p)$ *for every* p $(1 \leq p \leq n)$.

Proof. *A tuple* μ_k *is decomposed to a set of key-value pairs* $\{(a_1, \mu_k(a_1)), (a_2, \mu_k(a_2)), \dots, (a_n, \mu_k(a_n))\}$. *Every tuple in relation* r *can be distinguished among the other tuples with a set of primary key columns; a value of primary key* $\mu_k(a_1)$ *can be used for distinguishing tuple* μ_k *in relation r. By definition 2, tuple* μ_k *is decomposed into the set* $\{(\mu_k(a_1), a_1, \mu_k(a_1)), (\mu_k(a_1), a_2, \mu_k(a_2)), \dots, (\mu_k(a_1), a_n, \mu_k(a_n))\}$. *If a set of object values* $t_p(o)$ $(1 \leq p \leq n)$ *of triple set T' is equivalent to the value of attribute* a_p *of tuple* μ_k, *then any tuple* r *is can be represented by a set of triples where the subject of tin Every component of three-column arrays is aggregated into one table horizontally which is represented by a set of triple t where the value of subject is* $\mu_k(a_1)$. *Vice versa, a set of triple T =*$\{(s_1, p_1, o_1), \dots, (s_1, p_o, o_o)\}$ *can compose a tuple* $\mu_k = \{(p_1, o_1), (p_2, o_2), \dots, (p_o, o_o)\}$ *where* $\{p_1, \dots, p_o\}$ *is the set of attribute A of relation r and* $\{o_1, \dots, o_o\}$ *is the value of A of relation r if and only if* s_1 *holds the same value* $\mu_k(a_1)$. *Table holds every key-value pairs of tuple* $\mu_k = \{(a_i, \mu_k(a_i)\}$ *where* $1 \leq i \leq n$ *with discriminator* $\mu_k(a_1)$.

With definition 2 and lemma 1, we can say that every tuple in relation can convert into a set of triples in triple database. following picture illustrates the simple transformation process of our work when the table has five attributes {*a1*, *a2*, *b1*, *b2*, *b3*} where primary key of table is a set of attributes {*a1*, *a2*}.

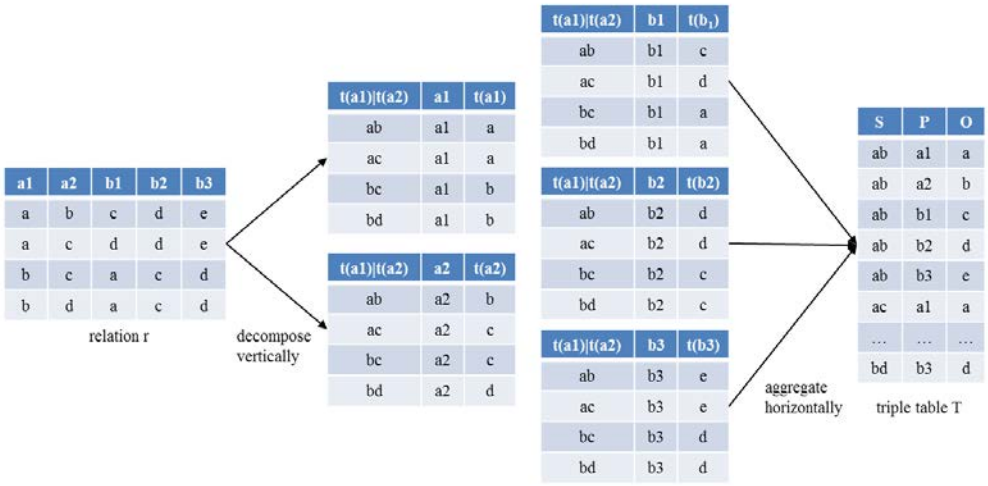


Figure 16: An example of triple transformation

Target relation is decomposed vertically with tuple's discriminator, attribute, and the value of attribute. Each component is mapped into the role of triples. Finally, triple table is constructed with horizontal aggregation of decomposed relation. In detail, there are some functions that hold intermediate result. For example, function *Attribute(r) A* with relation *r* returns a set of attributes *a(r)* of given relation *r*, where *keyAttrivute(r) kA* returns an array of a set of primary key of given relation *r*. With the intermediate results from function *A* and *kA*, function *keyValue(μ, A) kV* with tuple μ and the set of attribute *a(r)* returns an array of key-value pairs (attribute and the value of attribute) of μ. In this way, six functions are

defined within triple transformation. Table 4 shows the summary of each function. In Figure 18, logical process is described for building triple table from one relation with defined set of functions

Table 4: A Summary of functions for triple transformation

Function	Description	Result
attribute(r) A	returns a set of attribute a(r) of relation r	$\{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m\}$
keyAttribute (r) kA	returns a set of primary key attributes of relation r	$\{a_1, a_2, \dots, a_n\}$
keyValue (μ, A) kV	returns an array of key-value pairs of tuple μ	$(a_1, \mu(a_1)),$ $(a_2, \mu(a_2)), \dots,$ $(b_m, \mu(b_m))$
discriminator (kA) D	makes discriminator of triple using a string of concatenation of result of kA	$a_1 \vert a_2 \vert \dots \vert a_n$
keyValueTable (kV, D) kVT	returns a table with discriminator with key-value pairs kV	$(\mu(a_1) \vert \mu(a_2) \vert \dots \vert \mu(a_n), a_1, \mu(a_1))$
tripleTable (kVT) T	returns a triple table with aggregated keyValueTables	$\{(\mu(a_1) \vert \mu(a_2) \vert \dots \vert \mu(a_n), a_1, \mu(a_1)),$ $(\mu(a_1) \vert \mu(a_2) \vert \dots \vert \mu(a_n), a_2, \mu(a_2)),$ $\dots,$ $(\mu(a_1) \vert \mu(a_2) \vert \dots \vert \mu(a_n), b_m, \mu(b_m))\}$

3.2.3 Triple Table and Meta Table

For keeping the consistency of triple database, our system gathers the schema information of transformed relations from relational database. Since triple table in triple database only manipulate the set of triple with three columns for the schema-free characteristic, there should be space for managing the schema information which is not described with triple table. We called it *Meta Table*, and Meta Table expresses whole information of

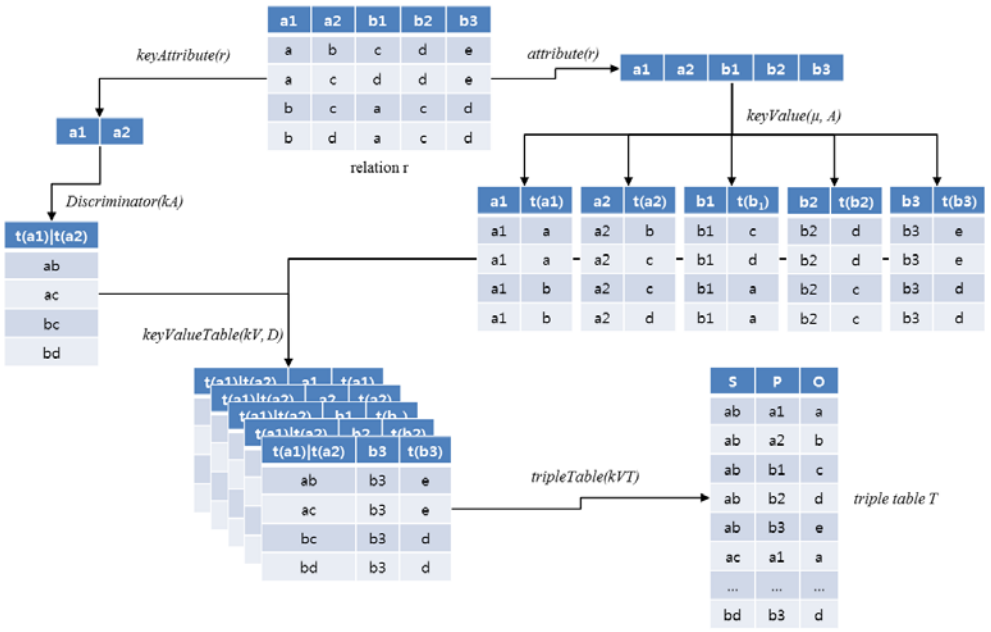


Figure 17: Logical process of triple transformation

existing database schema. It contains a list of tables and columns in table, relationship between tables, datatype, length of columns, and constraints on columns. This information is gathered by external module during the triple transformation, and saved in small XML files with triple table. In detail, for relational database schema $R = \{r_1, r_2, \dots, r_n\}$, Meta Table M is defined by $\{r_k, a(r_k), \phi(a(r_k)), v(a(r_k))\}$ for $1 \leq k \leq n$, where $\phi(a(r_k))$ is a set of data type and length of attribute a, and $v(a(r_k))$ is a set of constraint conditions in which attribute a holds.

Generally Meta Table describes additional information for database manipulation. Thus, it is referenced for every data manipulation of triple database for checking any violations while triples are inserted or deleted. For example, when a tuple has been inserted in relation, Triple table should handle the update of triple database with the set of triples which are matched with inserted tuple. During update of triple database, Meta Table is

referenced to get key attributes and additional constraints for checking any violations while triple database is manipulated. With the two data structure such as triple table and Meta Table, we will discuss the enforcement of integrity constraints on triple database in next chapter.

We introduced a straight-forward method for triple transformation with triple data model. Our triple data model is consolidated from the mapping between relational database, and we suggested the process for transforming tuples in relation and triples in triple database without any loss of information. By using the triple transformation with schema information, we now have triple database with full of triples which has been ready to manipulate.

However, it still has a limitation to adapt triple database with practical applications. Firstly, it does not guarantee any reliability during data manipulation. Although we have the information about original schema design in Meta Table, Entire triples should be utilized with solid and firm process of enforcement unless triple database becomes meaningless data duplication. Secondly, contents of relational database which would be transformed into triple representation are often too large. The large size of relation can negatively affect the triple database in terms of data processing time, as well as triple transformation time. Note that one process for checking uniqueness of tuples in one relation requires full table scan if there is no index on relation. If the size of relation is quite large, process for enforcement of integrity constraints will be overhead, which can make the triple database useless thing. To resolve such limitation of current triple database, we propose the definition and process of enforcement of integrity constraints in Chapter 4.

Chapter 4. Integrity Constraints on Triple Database

In this chapter, we describe the basic feature of enforcement of integrity constraint on triple database. Validation of integrity constraint conditions is important for the practical use of a triple database in Semantic Web. We describe the basic characteristics of the integrity constraint in a triple database in viewpoint of the equivalence between relational database and triple database. We assumed that there are six basic categories of integrity constraints: *primary key*, *functional dependency*, *referential integrity*, *not null*, *unique*, and *user-defined domain constraint*. Each process for the enforcement of integrity constraints are given by the logical interpretations of relational algebra which describe the transformed triples. In addition, we discuss about the real operations over triple database in the viewpoint of data definition and manipulation. SQL DDL and SQL DML execution on relational database cause the requirement of update of triple database consequently. We summarized the relationship between considered integrity constraints and the DDL and DML operations in relational database to provide the guideline for setup the validation of triple database.

The rest of chapter is organized as follows. Section 4.1 describes the basic definition of integrity constraints on triple database shortly. In Section 4.2, we present the detailed features of the enforcement of integrity constraints. Section 4.3 explains the database operations which are derived from deducted conditions for consistency of triple database. Section 4.4 summarized and concludes the chapter.

4.1 Definition of Integrity Constraints on Triple Database

There are two cases for which the integrity constraint should be considered. The former case is transformation itself. When the triple database is constructed, every initial condition such as key and referential integrity should be kept into a set of triples. In other word, there are no meaningless triples in the triple database which means the violation of initial relational database. The latter case is about database update. After the database is transformed into triples, triple database should handle entire operation of database update such as insertion of tuples. For using triple database as a primary storage for the service, conventional database's constraint as well as a domain constraint should be maintained during the update of triples. We describe the enforcement of integrity constraints for database creation in this Section.

As many researches have defined the definition of integrity constraint, we follow their explanation of constraints of relational database. Integrity constraint γ on relation schema R, Boolean function $\gamma(r)$ that associates with each relation $\gamma \in R$, r satisfies γ if $\gamma(r) = true$, and violates γ if $\gamma(r) = false$.

Theorem 1. ***Integrity Constraints.*** *A set of constraints* $\{\gamma_1, \gamma_2, \dots, \gamma_n\}$ *in relation r is converted into a set of transformed constraints* $\{\gamma'_1, \gamma'_2, \dots, \gamma'_n\}$ *in a triple database T equivalently.*

Note that a set of integrity constraints on relational database is transformed into triple database without any change of meaning of loss or

information in Figure 6. Similar to complete transformation of relational database, logical definition of integrity constraints on triple database also means the essential conditions of triple database for guaranteeing the minimal consistency of data. Indeed, it is equivalent translation and complete mapping of constraints on conventional database using relational algebra.

4.2 Enforcement of Integrity Constraints

We adapted and translated the traditional definition of integrity constraints from relational databases into triple databases, and categorized those into six conditions: Primary key constraint (PK), Functional dependency (FD), Referential Integrity, also known as Foreign Key constraint (FK), Not Null constraints (NN), Unique constraint (UQ), and Other domain constraint that defined by user (UD). All but the user-defined domain constraint are essential features for guaranteeing the valid state of a database. That is, the five integrity constraints should be preserved in a triple database if the triple database is to be used instead of a relational database for general applications with reliability of data.

We can assume the scenario which leads the enforcement process in triple database integrity: "*When triple database T is mapped from relation r with primary key attribute 'Name', then how can we say that primary key constraint is also satisfied in transformed triple database T?*". In this scenario, same value of subject means that it is originated from same tuple. Thus, if two triple have the same value for the subject, , the object values of two triples which are correspond with predicate '*Name*' should be different, Since it is only way to satisfy the uniqueness of key attribute property. In

other word, for every triple $\{t_1, t_2, \dots, t_n\}$ in triple database *T*, if $p(t_k)$ $(1 \leq k \leq n)$ is '*Name*', then every $t_p(\mathrm{o})$ $(1 \leq p \leq n)$ should have different value from other object value $t_q(\mathrm{o})$ $(1 \leq q \leq n)$ if $t_p(s) \neq t_q(s)$.

In addition, the consistency of triple database can be harmed during the dynamic operations, such as the change of data source and update of triple database. Similarly, we can assume another scenario which is caused by the changes of data: *"When a new triple t = (s,p,o) is added to T, does triple t affect the valid state of the triple database T after the insertion?"* to answer this scenario, some conventions to represent the essential condition of triple database to guarantee the consistency of database. We called this process the enforcement of integrity constraint on triple database. Figure 19 depicts the example of validation checking process with unique constraint over predicate 'Name' from triple database T. every triple is filtered with specific predicate which holds the unique constraint. Then two subset of filtered table is joined by object value. Finally, count the number of tuples grouped by the value of object value from joined result. If all the count of grouped object 1, then unique constraint on triple table T is satisfied.

The following lemmas show the process of enforcement of six integrity constraints of relational database on the triple database.

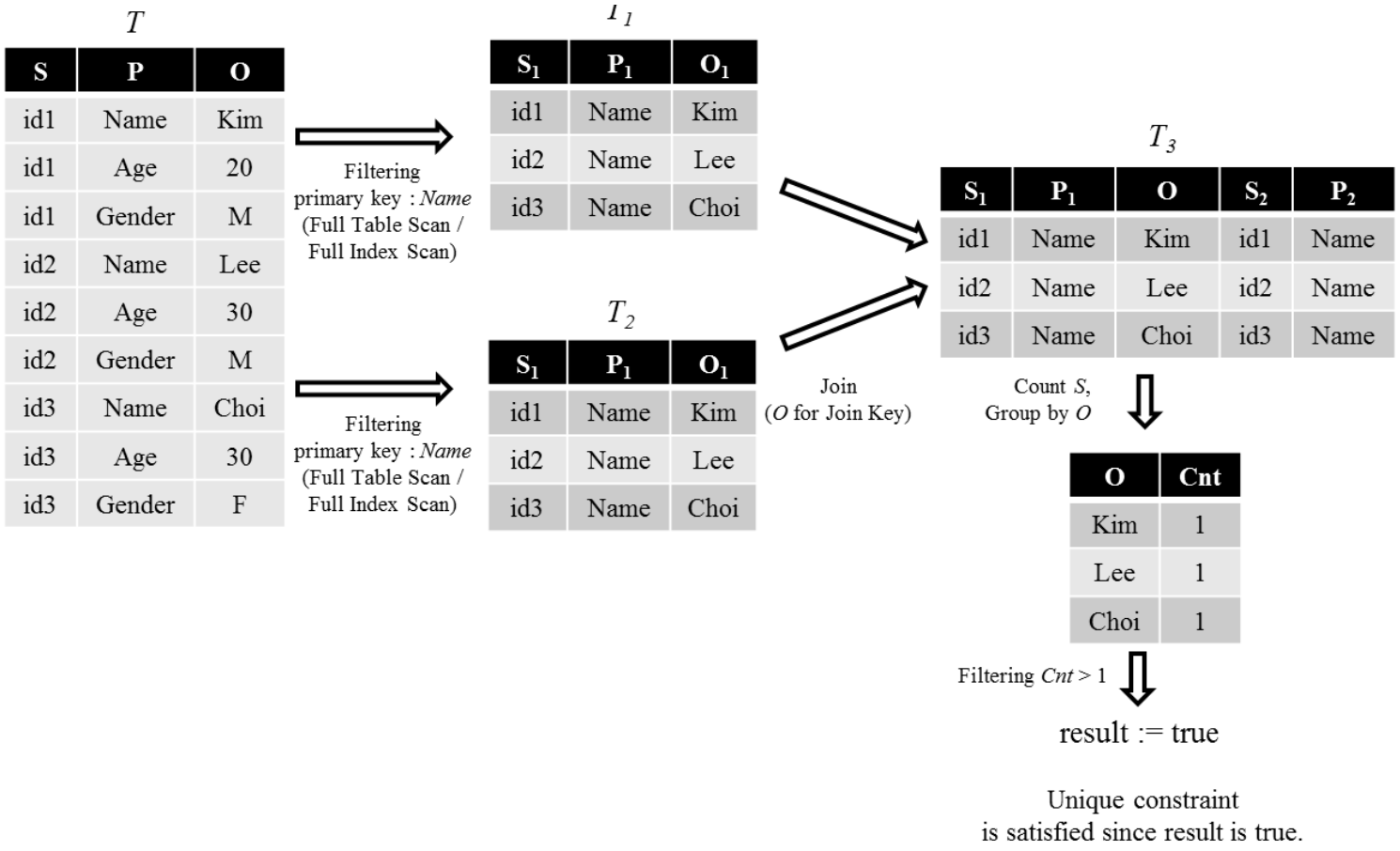


Figure 18: Process of checking unique constraint on triple database

4.2.1 Primary Key Constraint

Lemma 2. ***Primary Key Constraint.*** *Let K be a set of predicates which means superkey of r. For every triple* $t_1, t_2 \in T$ *where* $t_1(p), t_2(p) \in K$, *if* $\{t_n(o)\} \neq \emptyset$ *and* $t_1(p) = t_2(p)$ *and* $t_1(o) = t_2(o)$, *then* $t_1(s) = t_2(s)$.

Proof. *In relational database, primary key constraint is defined as "No two distinct tuples in r have the same value on all attributes in a set of superkey K". Assume that a set of triple* $q_1, q_2, \ldots, q_n$ *is derived from tuple* μ_1, *and* $r_1, r_2, \ldots, r_n$ *is derived from* μ_2, *then* $q_1(s)=q_2(s)=\ldots=q_n(s)$ *and* $r_1(s)=r_2(s)=\ldots=r_n(s)$. *By the definition of primary key constraint, we can say that primary key constraint on triple table T is satisfied if No two distinct triples in T have the same object value on all triples when two triples have same subject and predicate value. Thus, it is satisfied if* $q_n(p)=r_n(p) \wedge q_n(o)=r_n(o) \rightarrow q_n(s)=r_n(s)$ *is true for a set of predicate* $q_n(p), r_n(p) \in K$ *and all pairs of* q_k *and* r_k *where* $1 \leq k \leq n$.

The time complexity for processing the primary key constraint on triple table is a multiplication of *O(n)* for scanning the subject of *n* triples and *O(m)* for comparing the *m* distinct values of object of tuples; *O(nm)*.

4.2.2 Functional Dependency

Lemma 3. ***Functional Dependency.*** *Let Y be a set of predicates which are functionally determined by another set of predicate X. For every triple*

$t_1, t_2, t_3, t_4 \in T$ *where* $t_1(p), t_2(p) \in X$ *and* $t_3(p), t_4(p) \in Y$. *if* $t_1(s) = t_3(s)$ *and* $t_2(s) = t_4(s)$ *and* $t_1(p) = t_2(p)$ *and* $t_1(o) = t_2(o)$, *then* $t_3(p) = t_4(p)$ *and* $t_3(o) = t_4(o)$.

Proof. *Assume that two triples t_1, t_3 on triple table T are derived from μ_1, and another two triples t_2,t_4 are derived from μ_2, then $t_1(s)=t_3(s)$, $t_2(s)=t_4(s)$. If a predicate Y is functionally determined by another predicate X, then $\mu_1(Y)= \mu_2(Y)$ if $\mu_1(X)= \mu_2(X)$. Let $t_1(p)$ and $t_2(p)$ are subset of X, where $t_3(p)$ and $t_4(p)$ are subset of Y. Because Y is dominated by X, $t_3(o)$ is determined by $t_1(o)$ and $t_4(o)$ is determined by $t_2(o)$. Thus, with given $t_1(o)$ and $t_2(o)$, if $t_1(o)=t_2(o)$ then $t_3(o)=t4(o)$ where $t_1(s)=t_3(s)$ and $t_2(s)=t_4(s)$. Similarly, if $t_1(p)=t_2(p)$ and $t_1(o)=t_2(o)$ then t3(p)=t4(p). In conclusion, functional dependency on triple table T is satisfied if and only if* $q_n(p) = q_m(p) \wedge q_n(o) = q_m(o) \rightarrow r_n(p) = r_m(p) \wedge r_n(p) = r_m(p)$ *is true for a set of triples* $q_k(s) = r_l(s)$ *and* $q_k(s) = r_l(s)$ *where* $1 \leq k \leq n$ *and* $1 \leq l \leq m$.

Actually functional dependency is a general case of primary key constraint. The time complexity for processing the functional dependency constraint on triple table is a multiplication of *O(n)* for scanning the subject of *n* triples and squares of *O(m)* for scanning the *m* distinct values of object of triple and comparing the *m* distinct values of object of tuples; $O(nm^2)$.

4.2.3 Referential Integrity

Lemma 4. ***Referential Integrity: Foreign Key Constraint.*** *Let a set of triples t_n and t_m from relation r_1 and r_2 with primary keys K_1 and K_2, respectably. Let*

a subset α of r_2 be a foreign key referencing K_1 in relation r_1. For every set of triples $T_n, T_m \subseteq T$ and triple $t_1 \in T_n$ and $t_2 \in T_m$, if $t_2(p) \in \alpha$ and $t_1(p) \in K_1$, then $T_m(o) \subseteq T_n(o)$.

Proof. *Assume that a set of triples T_1, T_2 is derived from relation r_1, r_2, respectably. If r_2 references r_1 with a foreign keys F, we can get a subset of triples $T_n \subseteq T_1$, $T_m \subseteq T_2$ where $t_n \in T_n$ and $t_n(p) \in K_1$ and $t_m \in T_m$ and $t_m(p) \in F$ ($F \subseteq K_1$). If there is a pair of triples $t_1 \in T_n$ and $t_2 \in T_m$ where $t_1(o) \neq t_2(o)$ and $t_1(p) \in K_1$, there should be $t_2(s)$ where $t_2 \in T_m$ and $t_2 \notin T_2$. However, there cannot exist such $t_2(s)$ because T_m is a subset of T_2. Thus, referential integrity is satisfied if $t_2(p) \in \alpha \wedge t_1(p) \in K_1 \rightarrow T_m(o) \subseteq T_n(o)$ is true for every triple $T_n, T_m \subseteq T$.*

The time complexity for processing the functional dependency constraint on triple table is divided into two cases. When the tuple is inserted or updated, there need to *O(n)* for scanning *n* triples in the table and *O(m)* for comparing the *m* distinct values which means referenced attribute: *O(nm)*. On the other hand, *O(n)* for scanning for referenced attribute for cascading on *n* triple table is needed for deletion of tuple.

4.2.4 Not Null Constraint

Lemma 5. ***Not Null Constraint.*** *Let a set of triples T from tuple μ in relation r with a set of attributes A. There is subset of attributes A' ⊆A that* $\mu(a_k)$ *is not null value where* $a_k \in A'$*. For every triple* $t \in T$*, if* $\{t(s)\} \neq \emptyset$ *and* $t(p) \in A'$ *then* $\{t(o)\} \neq \emptyset$*.*

Proof. *Assume that a set of attributes* $A'=\{a_1, a_2, .., a_n\}$ *in relation r holds not null constraint. Then* $\mu(a_k)$ *of tuple* μ *for* $1 \leq k \leq n$ *should have some values. By the definition of triple model, tuple* μ *can be decomposed into a set of triples* $\{t_1,t_2, \ldots, t_m\}$ *where* $t_k(p)=a_k$ *and* $t_k(o)= \mu(a_k)$ *for* $1 \leq k \leq n$ *. Thus, not null constraint is satisfied if* $\{t_k(s)\} \neq \emptyset \wedge t_k(p) \in A' \rightarrow \{t_k(o)\} \neq \emptyset$ *is true for every triple* $t_k \in T$*.*

4.2.5 Unique Constraint

Lemma 6. ***Unique Constraint.*** *Let K be a set of predicates which are derived from attributes of relation r. For every* $t_n(p) \in K$*, If triple* $t_1, t_2 \in T$ *and* $t_1(p) = t_2(p)$ *and* $t_1(o) = t_2(o)$*, then* $t_1(s) = t_2(s)$*.*

Proof. *It is trivial to proof the unique constraint on triple database unique constraint is a general case of primary key constraint except not null constraint (See Lemma 2 and 5). Basically, if two triples have same predicate and object value, subject value of two triples is also same since two triples are derived from one tuple when unique constraint is enforced with triple*

database.

4.2.6 User-Defined Domain Constraint

Finally, for the enforcement of user-defined domain constraint, We define additional two functions named *cast()* for retrieving data types of predicate of triple *t*, and *check()* for representing specific domain constraints on relational database. Actually not null constraint and unique constraint is a subset of domain constraint. For the convenience of use, we separate those two constraints from domain constraint, so that domain constraint is more focused on user-defined condition.

Definition 3. ***Casting Types.*** *cast(t(p), d) is a function over triple database which returns Boolean result according to given data type of attribute and the predicate value of triple. It returns true (or 1) when the value of t(p) is same as a given data type d, or returns false (or 0) when the value of t(p) is distinct from a.*

Definition 4. ***Checking values.*** *Check(t(o), C) is a function over triple database which returns the Boolean result according to given conditions and the object value of triple. It returns true (or 1) when t(o) is satisfied with the given constraints* $C=\{c_1, \ldots, c_n\}$*, or returns false (or 0) when t(o) violates certain constraints.*

With these definitions, we design the following lemmas about domain constraint.

Lemma 7. ***Domain Constraint.*** *Let A be a set or attributes which hold the set of domain constraints* $\boldsymbol{C}=\{c_1, c_2, \ldots, c_n\}$ *and domain types d in relation r. For an attribute* $a \in A$ *and every triple* $t \in T$, *if* $t(p) \in A$ *then cast(t(p),d) and check(t(o),* $\boldsymbol{C}$*)=1*

Proof. *Assume that a set of attribute* $A=\{a_1, a_2, \ldots, a_n\}$ *in relation r. Since every triple is retrieved from r,* $t(p) \in A$ *and* $t(o) \in \mu(a_k)$ for $1 \leq k \leq n$. *If attribute* a_k *is defined with specific domain type d, corresponding predicate t(p) also have same domain type so that cast(t(p), d) returns 1. Similarly, if every tuple value* $\mu(a_k)$ *satisfies given condition C, t(o) from* $\mu(a_k)$ *also satisfy the user-defined condition. Thus, check(t(o), C) returns 1 when the domain constraint is satisfied with triple database.*

With these results, we can guarantee the enforcement of integrity constraint of triple database during the creation and manipulation of triple database Table 5 depicts a summary of result for enforcement of integrity constraints discussed above.

4.3 Database Operations for Integrity Constraints

As we mentioned in Section 4.2, integrity constraint should be enforced continuously after the triple database is created. For example, when a new triple is inserted into triple database, we can consider the enforcement of unique constraint and not null constraint preferentially, since a set of new triple can affect the previous condition of unique and not null constraint. In addition, if new triple contains a predicate which denotes the key attributes in

relational database, system also consider about the violation of primary key constraint and foreign key constraint for triple insertion. In conventional database system, database administrator manipulates relation instances as well as relation schema. For entire operations which can be occurred in traditional database, triple database should be ready to decide which constraint enforcement process is needed and process the adequate triple manipulation. We describe the essential considerable violations on triple database according to the operations of relational database in Table 6. A set of database operations contain both manipulation of relation data schema and instance.

Table 5: A summary of integrity constraint on triple database

Constraint on Conventional Database	Integrity Constraint (Relational Database)	Integrity Constraint (Triple Database)	Reference
Primary Key	$\forall \mu_1, \mu_2 \in r(\mu_1(K) = \mu_2(K) \rightarrow \mu_1 = \mu_2)$	$\forall t_1, t_2 \in T, \forall t_1(p), t_2(p) \in K$ $(\{t_n(o)\} \neq \emptyset \wedge t_1(p) = t_2(p) \wedge t_1(o) = t_2(o)$ $\rightarrow t_1(s) = t_2(s))$	Lemma 2
Functional Dependency	$\forall \mu_1, \mu_2 \in r$ $(\mu_1(X) = \mu_2(X) \rightarrow \mu_1(Y) = \mu_2(Y))$	$\forall t_1, t_2, t_3, t_4 \in T, \forall t_1(p), t_2(p) \in X, \forall t_3(p), t_4(p) \in Y$ $(t_1(s) = t_3(s) \wedge t_2(s) = t_4(s) \wedge t_1(p) = t_2(p)$ $\wedge t_1(o) = t_2(o) \rightarrow t_3(p) = t_4(p) \wedge t_3(o) = t_4(o))$	Lemma 3
Referential Integrity	$\forall r_1, r_2 \in R(\Pi_\alpha(r_2) \subseteq \Pi_K(r_1))$	$\forall T_n, T_m \subseteq T, \forall t_1 \in T_n, \forall t_2 \in T_m$ $(t_2(p) = \alpha \wedge t_1(p) = K \rightarrow T_m(o) \subseteq T_n(o))$	Lemma 4
Not Null Constraint	$\forall \mu \in r(\{\mu(A')\} \neq \emptyset)$	$\forall t \in T, \forall a \in A'(\{t(s)\} \neq \emptyset \wedge t(p) = a$ $\rightarrow \{t(o)\} \neq \emptyset)$	Lemma 5
Unique Constraint	$\forall \mu_1, \mu_2 \in r(\mu_1(K) = \mu_2(K) \rightarrow \mu_1 = \mu_2)$	$\forall t_1, t_2 \in T, \forall t_1(p), t_2(p) \in K$ $(t_1(p) = t_2(p) \wedge t_1(o) = t_2(o) \rightarrow t_1(s) = t_2(s))$	Lemma 6
Domain Constraint	$\forall \mu \in r, \forall a \in A(cast(a, d) = 1)$ $\forall \mu \in r, \forall a \in A(check(\mu(A), C) = 1)$	$\forall t \in T, \forall a \in A(t(p) = a \rightarrow cast(t(p), d) = 1$ $\forall t \in T, \forall a \in A(t(p) = a \rightarrow check(t(o), C) = 1$	Lemma 7

Table 6: Constraints and database operations

Database Operations			**Integrity Constraints**		
			PK	FD	FK
Database Schema	ADD RELATION		X	X	X
	DELETE RELATION		X	X	O (deletion of referenced relations)
	UPDATE RELATION / ADD ATTRIBUTE		X	X	X
	UPDATE RELATION / DELETE ATTRIBUTE		O (deletion of key attributes)	O (deletion of determinant set)	O (deletion of referenced attributes)
	UPDATE RELATION / RENAME RELATION		X	X	O (rename of referenced relations)
Database Instance	Relational Database	INSERT TUPLE	O (duplicate tuples)	X	X
		DELETE TUPLE	X	O (deletion of determinant set)	O (deletion of referenced tuples)
		UPDATE TUPLE	O (duplicate tuples)	O (update of determinant set)	O (deletion of referenced tuples)
	Triple Database	INSERT TRIPLE	O (duplicate triples)	X	X
		DELETE TRIPLE	X	O (update of determinant triple)	O (deletion of referenced triples)
		UPDATE TRIPLE	O (duplicate triples)	O (update of determinant triple)	O (deletion of referenced triples)

Database Operations			Integrity Constraints		
			NN	UQ	UD
Database Schema	ADD RELATION		X	X	X
	DELETE RELATION		X	X	X
	UPDATE RELATION / ADD ATTRIBUTE		X	X	X
	UPDATE RELATION / DELETE ATTRIBUTE		X	X	X
	UPDATE RELATION / RENAME RELATION		X	X	X
Database Instance	Relational Database	INSERT TUPLE	O (insertion of NULL)	O (duplicate tuples)	O (out of domain type or out of data range)
		DELETE TUPLE	X	X	X
		UPDATE TUPLE	O (update to NULL)	O (duplicate tuples)	O (out of domain type or out of data range)
	Triple Database	INSERT TRIPLE	O (insertion of triples with NULL)	O (duplicate triples)	O (out of domain type or out of data range)
		DELETE TRIPLE	X	X	
		UPDATE TRIPLE	O (update triples including NULL)	O (duplicate triples)	O (out of domain type or out of data range)

With this considerable violation, we introduce the comparison of database operations between relational database and triple database in the viewpoint of listed integrity constraints. For instance, insertion of tuple affects both relational database and triple database. Since a set of triples reformulate one tuple, operation for tuple insertion is also decomposed into operations of several triple insertion. Vice versa, insertion of triple is interpreted with the insertion of tuple with NULL value. In this way, every operations of database are mapped from relational database to triple database according to the rule we discussed in Section 4.3.

We categorized the manipulations of database into 11 cases; 5 cases for database schema manipulation, and 6 cases for database instance

manipulation. Table 7 summarizes the description of operation in triple database according to the type of triple manipulation which can be occurred by database update.

Table 7: Comparison of operations between RDB and TDB

Database Operations			**RDB**	**TDB**	**Const raints**
Database Schema	ADD RELATION		$R \leftarrow R \cup r_1(A_1, A_2, \ldots, A_n)$	$M \leftarrow M \cup m(r_1(A_1, A_2, \ldots, A_n))$	-
	DELETE RELATION		$R \leftarrow R - r_1(A_1, A_2, \ldots, A_n)$	$M \leftarrow M - m(r_1(A_1, A_2, \ldots, A_n))$	FK
	UPDATE RELATION / ADD ATTRIBUTE		$R \leftarrow (R - r_1(A_1, A_2, \ldots, A_n)) \cup r_1(A_1, A_2, \ldots, A_{n+1})$	$M \leftarrow (M - m(r_1(A_1, A_2, \ldots, A_n)) \cup r_1(A_1, A_2, \ldots, A_{n+1})$	-
	UPDATE RELATION / DELETE ATTRIBUTE		$R \leftarrow (R - r_1(A_1, A_2, \ldots, A_n)) \cup r_1(A_1, A_2, \ldots, A_{n-1})$	$M \leftarrow (M - m(r_1(A_1, A_2, \ldots, A_n)) \cup r_1(A_1, A_2, \ldots, A_{n-1})$	PK, FD, FK
	UPDATE RELATION / RENAME RELATION		$R \leftarrow (R - r_1(A_1, A_2, \ldots, A_n)) \cup r_2(A_1, A_2, \ldots, A_n)$	$M \leftarrow (M - m(r_1)) \cup r_2$	FK
Database Instance	Relational Database	INSERT TUPLE	$r \leftarrow r \cup \{\mu_1(A_1), \mu_1(A_2), \ldots \mu_1(A_n)\}$	$T \leftarrow T \cup \{(\mu_1(A_1) \mid \mu_1(A_2) \mid \ldots \mid \mu_1(A_n), A_1, \mu_1(A_1)), \ldots, (\mu_1(A_1) \mid \mu_1(A_2) \mid \ldots \mid \mu_1(A_n), A_n, \mu_1(A_n))\}$	PK, NN, UQ, UD
		DELETE TUPLE	$r \leftarrow r - \{\mu_1(A_1), \mu_1(A_2), \ldots \mu_1(A_n)\}$	$T \leftarrow T - \{(\mu_1(A_1) \mid \mu_1(A_2) \mid \ldots \mid \mu_1(A_n), A_1, \mu_1(A_1)), \ldots, (\mu_1(A_1) \mid \mu_1(A_2) \mid \ldots \mid \mu_1(A_n), A_n, \mu_1(A_n))\}$	FD, FK
		UPDATE TUPLE	$r \leftarrow \prod_{F1, F2, \ldots, Fn}(r)$	$T \leftarrow \prod_{Sk, P1, O1}(T) \cup \prod_{Sk, P2, O2}(T) \cup \ldots \cup \prod_{Sk, Pn, On}(T)$	PK, FD, FK, NN, UQ, UD
	Triple Database	INSERT TRIPLE	$r \leftarrow r \cup \{null, \ldots, \mu_1(A_k), \ldots, null\}$	$T \leftarrow T \cup \{\mu_1(A_1) \mid \mu_1(A_2) \mid \ldots \mid \mu_1(A_n), A_k, \mu_1(A_k)\}$	PK, NN, UQ, UD
		DELETE TRIPLE	$r \leftarrow r - \{null, \ldots, \mu_1(A_k), \ldots, null\}$	$T \leftarrow T - \{\mu_1(A_1) \mid \mu_1(A_2) \mid \ldots \mid \mu_1(A_n), A_k, \mu_1(A_k)\}$	FD, FK
		UPDATE TRIPLE	$r \leftarrow \prod_{Fk}(r)$	$T \leftarrow (T - \{\mu_1(A_1) \mid \mu_1(A_2) \mid \ldots \mid \mu_1(A_n), A_k, \mu_1(A_k)\}) \cup \{\mu_2(A_1) \mid \mu_2(A_2) \mid \ldots \mid \mu_2(A_n), A_k, \mu_2(A_k)\}$	PK, FD, FK, NN, UQ, UD

In this chapter, we introduced the fundamental properties for using triple

database which is designed by rules of triple transformation discussed in Chapter 3, and discussed the enforcement integrity constraints according to the relationship between relational database and triple database. Building triple database for advanced service is not able to be achieved without rigid consistency and reasonable performance. The main contribution of this chapter is the logical process of enforcement of integrity constraints which guarantee the consistency of database. A set of constraints is converted into another set of transformed constraints without any change meaning or loss of information. We conduct the six categories of integrity constraint using relational algebra, and summarized database operation for achieving listed constraints by comparing database manipulations of relational database and triple database. In next chapter, we will discussed about the additional component of our system, which supports the better performance by reducing overhead for checking integrity constraints and query response time for executing triple queries over triple database.

Chapter 5. Tridex: Triple Index Structure for Integrity Constraints

As more data are provided in Semantic Web, processing large amounts of data with triple-format triple-data, and interlinking the applications with triples, has become important for a variety of applications. Assurance of database consistency of database is prerequisite for applying triple database into practical service with vast Semantic Web data. Integrity constraints ensure that changes of database do not result in a loss of data consistency. Nevertheless, processes for checking integrity constraints may be costly to apply, since every validation process requires one of multiple accesses to triple database itself. As more contents are transformed into triple database from enterprise data, indexing a triple database efficiently has become an important challenge. In addition, not only for the validation of integrity, the increasing amount of data also calls for efficient storage management of triple database. Much research has focused on storing and querying triple data. However, existing triple storing and indexing techniques often do not consider the maintenance of triple data. Consider, for example, a bank maintenance application where each customer's information is stored in a triple database: customer account information at a bank is very sensitive, so it should be handled carefully during the execution of transition queries, with no loss of information. If we consider triple as the representation scheme for bank information, the entire triple database should always be maintained in the consistent state. That is, for reliability, the database should maintain equivalent conditions, such as the integrity constraints of conventional

databases.

In particular, the violation of integrity constraints is examined using the index structure of a set of attributes in a relational database. Similarly, there should be a tree index structure for a triple database to derive the equivalent state of the database. As we introduced in Section 2.3, several works on index systems for the triple data format have been proposed: *MAP* [29], *HexTree* [56], and *TripleT* [28]. However, these approaches have inadequate structures for maintaining efficient triple index querying as well as the validation of database consistency.

We argue that there are two main requirements for an efficient triple index from the viewpoint of the practical use of triple data: a data retrieval process (e.g., transactional query executions) and a data management process (e.g., validation of database integrity). We developed a novel technique for building a lightweight triple index structure, named *Tridex*, to consider enforcement of the integrity constraint of a triple database. Our work can be applied to a relational database-based triple database, which requires reliability and trustworthiness of data: for example, bank account information.

The remainder of this chapter is organized as follows. Section 3.1 presents the motivation and problem definition of our triple index research with the example of triple database. In Sections 3.2, we present an analysis of the problems of conventional solutions using a triple index and comparing it to a query set and a query pattern. We also describe the logical model and qualitative analysis of Tridex. Section 3.3 presents the implementation of our index structure and experimental results. Finally, Section 3.4 summarized and concludes the chapter.

5.1 Motivation and Problem Definition

By the definition of triple database and the process of triple transformation in Chapter 3, transformed triple $t_1 = (s_1, p_1, o_1)$ in our triple database is a part of a specific tuple of a relation where the value of attribute p_1 of tuple s_1 is o_1. One tuple comprises several triples with the same subject values. Any unique value can be used for distinguishing a distinct tuple, so that we choose an arbitrary distinct keyword (e.g., a concatenated value of a set of primary key attributes) as a value of the subject. $\mu(a_1)$ denotes the value of attribute a_1 of the μ and the key-value pair $(a_1, \mu(a_1))$ is represented by the combination of predicate and object. For instance, tuple *{Kim,20,M}* in relation *r* in Figure 20 is mapped to a set of triples {(*id1*,*Name*,*Kim*), (*id1*,*Age*,*20*), (*id1*,*Gender*,*M*)} in a triple database *T*, where *id1* is an identifiable keyword, such as a combination of key values of relation *r*.

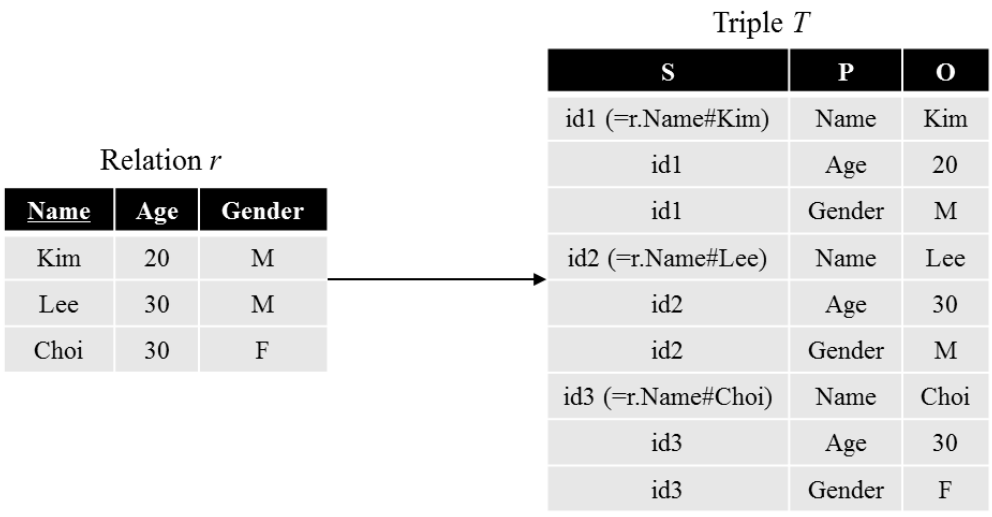


Figure 19: An example of triple table representation

Let *r* be a relation with *n* tuples and *m* attributes. Then, triple database *T*

from r consists of {(s_1, p_1, o_{11}), (s_1, p_2, o_{12}), …, (s_1, p_m, o_{1m}), (s_2, p_1, o_{21}), …, (s_2, p_m,o_{2m}), …, (s_n, p_m, o_{nm})}. To enforce the integrity constraint, several processes are required for scanning the values of the roles of triples. Figure 19 in previous chapter describes a basic process for validating unique constraints in T where attribute Name is a primary key of the original relation. The worst case in terms of time complexity for checking the unique constraint in Figure 19 is $O(nm)$ for scanning n tuples with m attributes. That is, every triple in T should be scanned once, minimally, to validate the unique constraint. Without an indexing method, every process for checking the consistency of a triple database takes a significant amount of time.

Current triple indexing solutions can be applied for the enforcement of integrity constraints. However, there are some limitations, mentioned in Section 2, for achieving efficient validation of a triple database and data retrieval query execution. To address these limitations, we propose several query sets for a triple database in two categories: data retrieval and constraint checks. The data retrieval query can be traditional SQL or SPARQL queries for retrieving a specific set of triples according to the requests of users. A constraint check identifies the essential conditions to guarantee the integrity of the triple database. Every query set for a constraint check can be achieved by combining queries of specific triple query patterns. We believe that a triple index structure should support both types of query set. In this Section, we discuss the problems of conventional triple index solutions using an example.

With regard to the relationship between graph pattern analysis in SPARQL and the process of integrity constraint enforcement, we argue that query analysis of a triple database should be a specific triple pattern-finding

problem. Jalali et al. summarized triple patterns using 10 different *simple graph patterns* (*SGP*) and variants with connected graphs, such as *Single Join Graph Pattern* (*SJGP*) and *Chain Shape Pattern* (*CSP*). [34]. In detail, the process for finding duplicate values of an object with a full-table scan exactly matches the SGP-finding problem, the target pattern of which is the SJGP with *object-object* join. For example, with a primary key constraint, the system tries to find a duplicate object value with a given predicate: *Name* in Figure 20. If there are redundant objects the value of which is *Kim* in *T*, the algorithm counts the number of triples that have duplicate values of the object by scanning the *object-object* joined triple patterns, such as $<s_1\ p_1\ ?o\ .\ s_2\ p_2\ ?o>$. Despite this apparently simple process for the query pattern-finding problem, there are some challenging issues in applying existing index approaches directly.

Data Duplication. Duplicated atoms of triples that are stored separately in MAP and HexTree are not helpful in executing one process for finding SGPs. That is, MAP and HexTree need six accesses of the index tree for one process of triple pattern querying, because there are six different index trees, depending on the permutations of the triple roles. Data duplication also leads to redundant storage and unnecessary query processing costs (e.g., a simple join on a single pair of triples can require multiple index scans to retrieve the join key). We believe that a triple index should have simpler architecture than MAP or HexTree. The key idea behind reducing data duplication is that each distinct role of a triple appears exactly once in the tree structure. Also, additional information is kept in an external structure, such as payload, which is dangled from the leaf node of the index tree. These concepts can

markedly reduce the size of the entire triple index tree.

Irrelevant Index Key Sequences. Similar to data duplication, MAP and HexTree keep six clustered B+-tree indices. However, certain index trees are useless for the constraint-checking process. For example, *SPO*, *SOP*, *OSP*, and *OPS* tree indices in MAP are not used to check the primary key condition because queries for checking violations of the primary pey constraint reside in the SGP-finding problem. Only the *PSO* and *POS* index trees are used to enforce the primary key constraint. Most index trees in HexTree can also be ignored in the constraint-checking process for the reasons mentioned above. Hence, TripleT may be the best index structure for SJGP with *object-object* joins because all distinct values in a triple database are indexed in one index tree. For example, we can find the payload for index key Name in the predicate bucket of TripleT, such as {[*id1*,*Kim*],[*id2*,*Lee*],[*id3*,*Choi*]}, with one tree access in the example in Figure 5. Logically, the time complexity of one primary key check process can be decreased to $O(log_m n|U|)$ in the worst case, where $|U|$ is the number of indexed triples.

Oversized Index Tree. Given frequent changes to a database, we believe that a triple index scheme should consider efficiency and a complete index-update process, based on enforcement of the integrity constraint. Update cost is generally proportional to the size of the index tree and additional structures of the index structure. From the viewpoint of scalability, TripleT has one important drawback: each distinct value of a triple database, regardless of role, is stored in one large tree. This can create unpredictable

situations and lead to low performance when large amounts of data are converted in a triple database. For every repeated select query execution and update operation, TripleT has additional query execution costs even though it is a SGP-finding problem. To date, there is no complete solution for selecting and updating processes in a triple database. We developed a method to provide an equivalent validation of the state of a triple database from the features of a relational database, such as integrity constraints and index structure, regardless of the data characteristics or storage solution. We adapted the basic concept of an index structure in a relational database to build our novel triple index structure, named Tridex.

5.2 Tridex: A Lightweight Triple Index Structure

In this Section, we introduce Tridex, our lightweight triple index structure for integrity constraints on triple database.

5.2.1 Query Set and Query Pattern

One of the important goals of our triple index system is covering the entire set of triples to support every query set, which is processed by a conventional database solution. To facilitate the possible query set in the triple database, we adapted the categorization of query pattern from previous research [34] and expanded the definition of triple patterns to two query set types: data retrieval and constraint checks.

First, there are *simple triple patterns* (*STP*) based on the triple atom itself. For example, querying a specific object value with a given subject and

predicate value can be described with one STP <*s p ?o*>. Generally, we consider six triple patterns, <*s p ?o*>, <*s ?p o*>, <*?s p o*>, <*?s p ?o*>, <*?s ?p o*>, and <*s ?p ?o*>. We can ignore the case of <*?s ?p ?o*> because this STP query only requests a full-table scan operation, regardless of index type.

It is called a *connected graph pattern* (*CGP*) if two or more STPs are connected with a common variable; if the CGP consists of exactly two STPs, it is called an *simple join graph pattern* (SJGP). There are six patterns of SJGP by join type. For example, query <*?s* p_1 o_1 . *?s* p_2 o_2> is SJGP with *subject-subject* join. If the CGP has multiple joins, we call those queries a *complex join pattern* (*CJP*).

Generally, each complex query pattern can be decomposed into STPs. This means that if a triple index handles a specific type of triple pattern such as STP or SJGP, then the index structure provides sufficient availability for processing an SPARQL query. This assumption can also be made regarding enforcement of the integrity constraint. For example, finding a violation of the primary key constraint is part of the predefined query processing with query pattern <s_1 p_1 *?o* . s_2 p_2 *?*o_2>. With this assumption, we can map the query sets of two categories into defined query patterns. Table 8 summarizes comparisons with the query set using a specific query pattern, as mentioned above.

Table 8: A Summary of query set and query pattern

Query Set			**Query Pattern**	**Cate gory**
Data Retrieval	Select	Without Condition	*?s p ?o*	STP
		Range Search	*?s p ?o* *?s p_1 o_1 . ?s p_2 o_2*	STP, SGJP
		Exact Match	*?s p ?o* *?s p_1 o_1 . ?s p_2 o_2*	
	Join	Single Join	*s_1 p_1 ?o . s_2 p_2 ?o*	SSP
		Multiple Join	*?s p_1 o_1 . ?s p_2 o_2* *s_1 p_1 ?o . s2 p_2 ?o*	
Data Manage-ment	primary key constraint		*s_1 p_1 ?o . s_2 p_2 ?o*	
	foreign key constraint		*?s_1 p ?o_1 . ?s_2 p ?o_2*	
	not null constraint		*?s p ?o*	STP
	unique constraint		*s_1 p_1 ?o . s_2 p_2 ?o*	SSP
	user-defined domain constraint		-	CGP

5.2.2 Description of Tridex

We adapted the B+-tree structure from relational database indexing, while new features, such as additional data space for storing triple data, were attached. Tridex is a hierarchical multiple B+-tree consisting of index (non-leaf) nodes and data (leaf) nodes, similar to MAP, HexTree, and TripleT. Among the existing triple index structures, Tridex has the unique feature of storing triple data in an index node and a data node; it keeps three distinct index trees, based on the type of index key (e.g., subject, predicate, and object). For example, keyword *id1* of triple (*id1*,*Name*,*Kim*) is a value of subject role; thus, only the subject index tree has an index key *id1* in B+-tree

index nodes. Because every value of roles, extracted from the roles of the triple, appears exactly once in the index tree, Tridex can decrease the volume of a triple index structure.

In addition, every data (leaf) node of Tridex has its payload bucket for additional triple data, which does not appear in the index nodes. The structure of the payload bucket of Tridex is similar to that of HexTree and TripleT; however, Tridex has payloads of different values. In detail, keywords *Name* and *Kim* in the above example are not indexed in the subject tree. To process a point query with keyword *id1*, such as "find the name of the person of triple *id1*," Tridex specifies the remaining values of the triple, and stores them in a payload bucket that is dangled from the leaf node. As a result, data node *id1* in the subject tree has a unique payload with predicate value *Name* and object *Kim*. Every value of roles of the triple database appears up to twice, in either the data node or the payload bucket. Thus, Tridex can keep a minimized structure, versus TripleT, with no loss of information. In addition, because most of the processes for checking the integrity constraint of a triple database consist of combinations of point queries, such as <*?s p o*>, the payload structure of Tridex is suitable for processing enforcement of the integrity constraint.

In detail, a Tridex index structure is a multi-tree multi-level index. Figure 21 shows a typical leaf node structure of Tridex index tree. It contains *n-1* key values $K_1, K_2, \ldots, K_{n-1}$, where *K* is one of the roles of the triple, and *n* pointers $P_1, P_2, \ldots, P_n$, and additional n-1 pointers $Q_1, Q_2, \ldots, Q_{n-1}$ which points each payload buckets dangled with index key. It follows the traditional structure of a B+-tree except for the payload bucket. Using two pointers and one index key, Tridex consist of three distinct B+-trees with payload bucket.

$K_p \in \{S\}$ or $\{P\}$ or $\{O\}$ $(1 \le p \le n-1)$

Figure 20: A typical node structure of Tridex

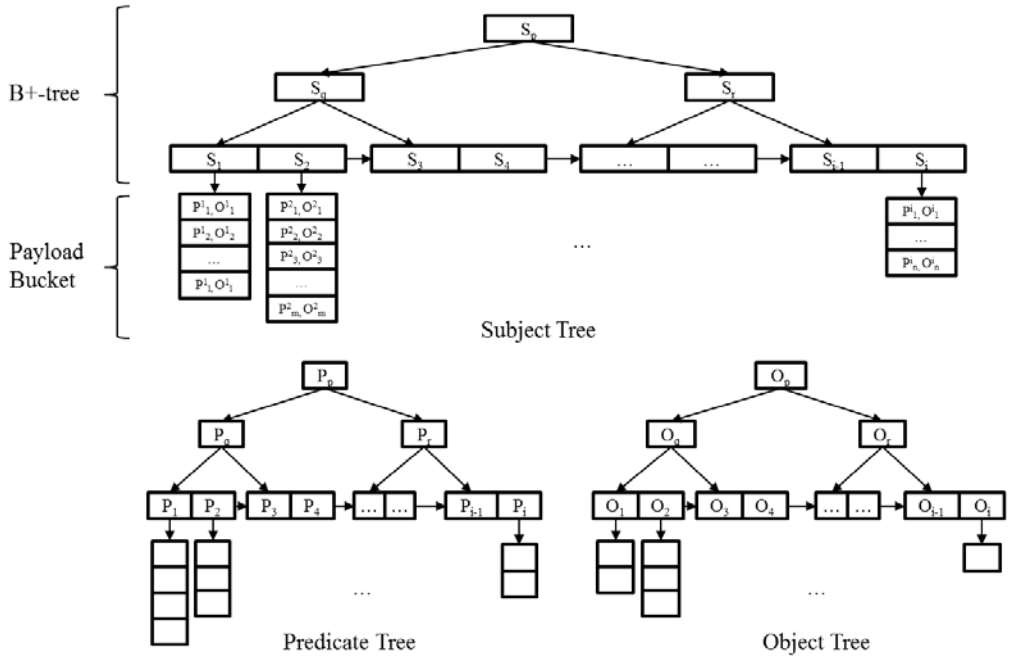


Figure 21: Basic structure of Tridex

Figure 23(a) shows a complete tree of the subject for triple database T. There are two additional trees for the predicates and objects. These examples of three index trees are all balanced in that the length of every path from the root to a leaf node, which contains a link to the payload bucket, is the same. Each triple can be searched by a concatenation of the index key and the value of the payload bucket. Figure 23(b) shows the payload bucket of leaf node which contains index key id1 and id2.

T

S	P	O
id1	Name	Kim
id1	Age	20
id1	Gender	M
id2	Name	Lee
id2	Age	30
id2	Gender	M
id3	Name	Choi
id3	Age	30
id3	Gender	F
id4	Name	Kim
id4	Lab	IDS
id5	Name	Lee
id5	Lab	IDS
id6	Name	Choi
id6	Lab	IDB

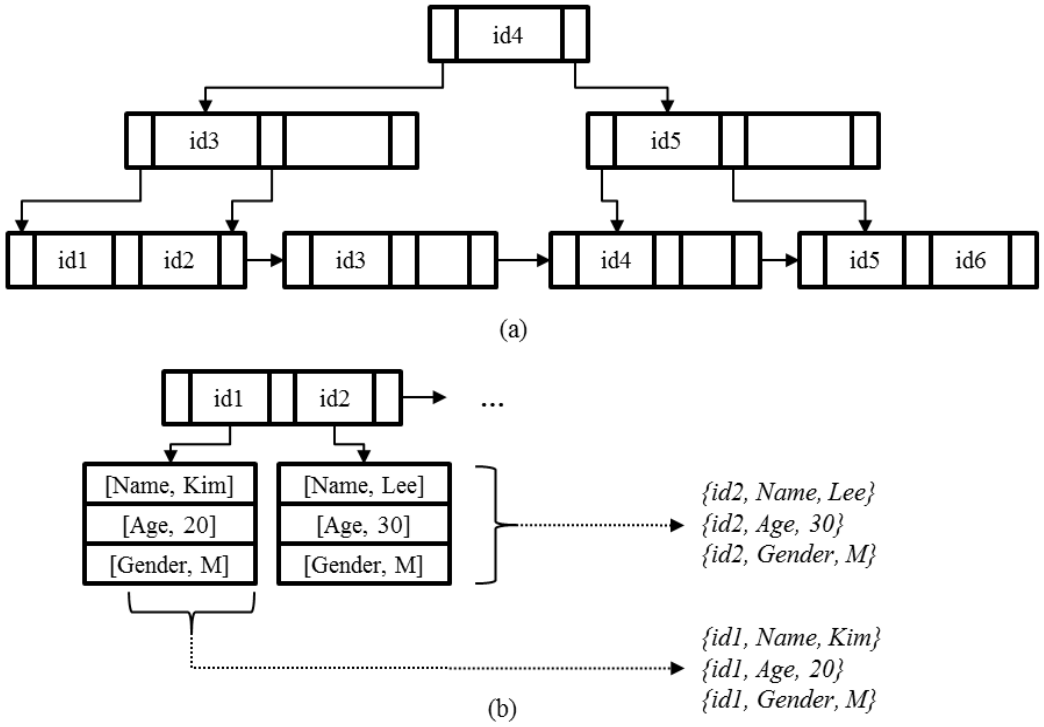


Figure 22: An example of Tridex with triple database

(a) A complete subject tree for a given triple database (n=3)
(b) A leaf node with payload bucket for subject tree index (n=3)

As an example to explain the above description, Figure 24 includes a database example, triplized data from a relational database, and its index tree structure. By the logical definition of Tridex, there are three index trees *idx*(*S*), *idx*(*P*), and *idx*(*O*). The idx(S) tree uses a set of subject values as a value of the index node. idx(P) is a predicate index tree, so that it keeps every set of attributes of the relational database. Finally, every object value is inserted in idx(O), with additional information such as remaining subject and predicate value pair.

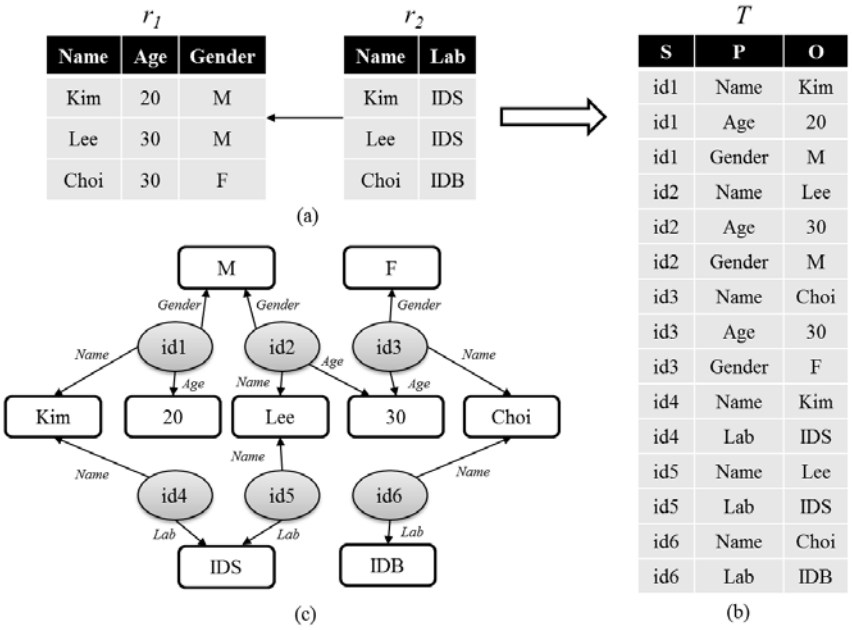


Figure 23: A complete example of Tridex

(a) Relational database (b) Triple database (c) Triple graph

Insertion and deletion of an index key follows the process of a traditional B+-tree. Node split and pointer redistribution occurs when a new triple is inserted or an old triple is deleted. In addition, Tridex has a simple process for updating the payload bucket. When the node is split into two

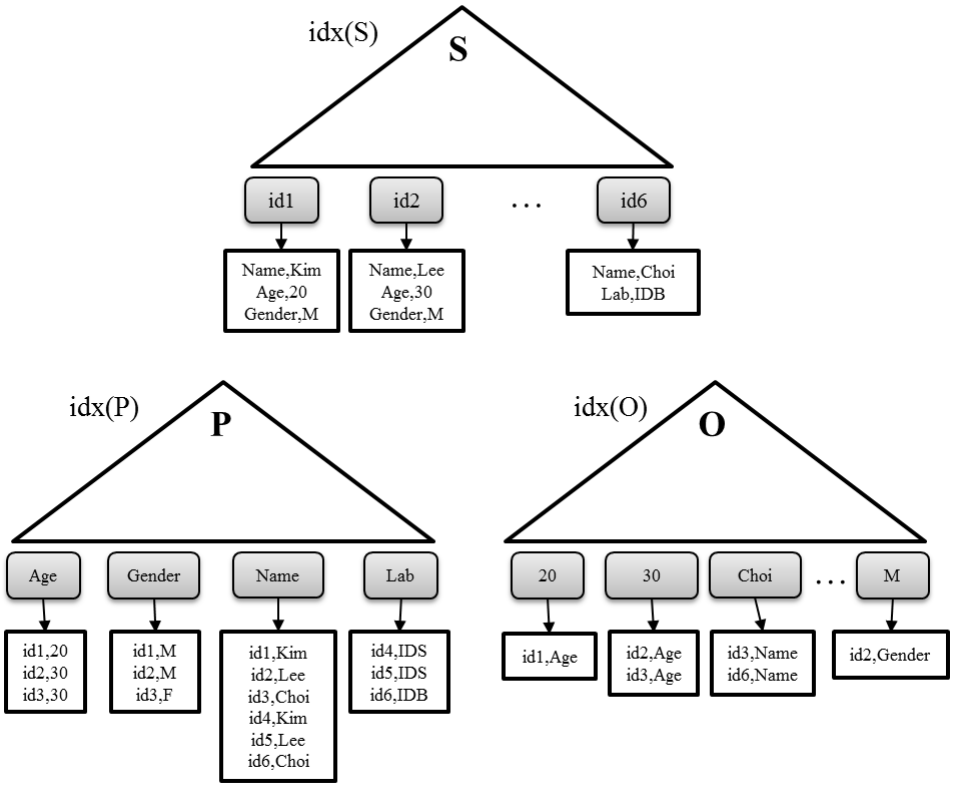


Figure 24: Three index trees with payload buckets

nodes, a new payload bucket for the new node is newly created with the value of the inserted triple, while the unused payload bucket is deleted during node deletion. In the Tridex structure, the subject tree is always updated if there is at least one insertion or deletion of a triple because it uses a distinct subject value, which denotes the unique tuple of a relational database, for the index key. In contrast, the predicate and object trees may not be changed during updates of the triple database. Entire values of the payload bucket are stored in sorted order. The lookup process for the payload bucket finds the location in which the triple value would appear or be discarded gradually. Algorithm 1 and Algorithm 2 show the pseudo algorithm for insertion and deletion of an entry in a Tridex tree.

```
procedure insert(entryType T, value V, pointer P)
    Find the leaf node N that should contain value V in tree type T;
    insert_entry(N,V,P);
    store(N,V);
end procedure

procedure store(node N, value V)
    for every value of payload bucket A_k (1≤k≤n) of N
        V'←V(A_k);
        / V(A_n) is a value of payload bucket A_n /
        if V ≤ V' then
            V(A_k)←V;
            V(A_k+1)←V(A_K);
        end if
    end for
end procedure

procedure insert_entry(node N, value V, pointer P)
    Let T be an entry type of triple;
    if (N has space for (V, P)) then
        insert(T,V,P);
    else split N;
        / Split procedure follows the insertion in B+-tree /
        insert(T,V,P);
    end if
end procedure
```

Algorithm 1: Entry insertion of Tridex

5.2.3 Analysis of Tridex

Space Consumption. Space overhead of a triple index is dominated by the amount of data duplication. We evaluate the space consumption of an index tree and overhead for maintaining the payload of the conventional approach using the following notations. $|T|$ denotes the total number of triples in triple database T. There are three roles in one triple, so that the total

```
procedure delete(nodeType T, value V, pointer P)
    Find the leaf node N that contains value V in tree type T;
    discard(N,V);
    delete_entry(N,V,P);
end procedure

procedure discard(node N, value V)
    for every value of payload bucket A_k (1≤k≤n) of node N
        V'←V(A_k);
        / V(A_n) is a value of payload bucket A_n /
        if V = V' then
            V(A_k)←V(A_{k+1});
            V(A_n)←φ;
        end if
    end for
end procedure

procedure delete_entry(node N, value V, pointer P)
    Let T be an entry type of triple;
    delete(T,V,P);
    if (N is the root and N has only one remaining child) then
        Make the child of N the new root of the tree and delete N;
    else if (N has too few values/pointers) then
        Coalesce or redistribute nodes;
        / Coalesce or redistribution process follows
          the deletion in B+-tree /
        delete(T,V,P);
    end if
end procedure
```

Algorithm 2: Entry deletion of Tridex

number of roles in triple database $|R|$ is $3 \times |T|$. The number of unique predicates in T is $|pred(T)|$ and that of unique objects is $|obj(T)|$. In our definition of triple transformation, one tuple is decomposed into several triples, which have the same subject value. Hence, the number of unique subjects in T $|sub(T)| = |T| / |pred(T)|$. Because every key of the index tree is unique, the number of unique values in T $|uq(T)|$ is obviously different from $|T|$. Finally, f and S denote the fan-out of the tree and average size of all

values in T, respectively.

We computed the estimated space consumption of existing approaches (e.g., MAP, HexTree, TripleT) and our index structure with these factors. For a given database T, MAP generates six different indices, where the height of each tree is given by $|T|$ and f. Thus, the space requirement of MAP, T_{MAP}, is given by $6 \times (log_f|T| \times f) \times (3 \times S)$. There is no payload in MAP, so the space requirement for payload, $P_{MAP} = 0$. HexTree also requires six separate trees for the combinations of roles. However, the average size of an index key is smaller than in MAP and the height of the tree is decreased. In addition, there is some space for payload with HexTree. The size of the payload is computed from the average size of the value and the summation of stored triples. The estimated space consumption of HexTree, T_{HT}, $= 6 \times (log_f|T| \times f) \times (2 \times S) / (|T| / (|sub(T)| + |(pred(T)| + |obj(T)|))$, where payload space, P_{HT}, $= (|sub(T)| + |(pred(T)| + |obj(T)|) \times S$. TripleT has one big tree for keeping the index key, so the tree space for TripleT, T_{TT}, is given by $(log_f|uq(T)| \times f) \times S$, where the payload space P_{TT} is given by $3 \times |uq(T)| \times S$. Finally, the size of Tridex is decided by the total number of roles, so that $T_{TX} = (log_f |sub(T)| + log_f |pred(T)| + log_f|obj(T)|) \times f \times S$, where $P_{TX} = 2 \times |uq(T)| \times S$. Generally, MAP requires much more space for constructing index trees than the other approaches, and T_{HT} uses the minimum storage ($T_{HT} < T_{TX} \fallingdotseq T_{TT} < T_{MAP}$). In contrast, space for payload in TripleT requires the most space among the approaches compared ($P_{MAP} < P_{HT} < P_{TX} < P_{TT}$).

Query Processing. We also estimated the query processing cost for each approach by the given query. For example, in the previous example in Figure 5, consider the given query *find the name of the man who has a*

similar age to the man named Lee. This query is represented by the combination of four triple join patterns <$?s_1$ *name lee* . $?s_1$ *age* $?o_2$. $?s_2$ *age* $?o_2$. $?s_2$ *name* $?o_3$>. To execute the given query, there are four index lookups for MAP and HexTree for each tree. In contrast, five index lookups are needed for TripleT and Tridex, including a payload bucket scan. If one index lookup takes time Z_t and one payload scan takes time Z_p, the query execution time for each index scheme can be summarized as follows: MAP: $4 \times Z_t(MAP)$, HexTree: $4 \times Z_t(HT) + 4 \times Z_p(HT)$, TripleT: $5 \times Z_t(TT) + 5 \times Z_p(TT)$, and Tridex: $5 \times Z_t(TX) + 5 \times Z_p(TX)$. Generally P_{TT} is larger than T_{TX} because the number of index keys is much larger than the number of values in one payload bucket. Thus, we predict that Tridex shows better performance than TripleT with the additional payload search.

Similarly, enforcement of the integrity constraint requires query processing with triple patterns. In the case of a unique constraint, there are two filtering operations and one join for accessing the index tree and payload. With this background, we also estimated the query processing cost for each approach as follows: MAP: $6 \times Z_t(MAP)$, HexTree: $2 \times Z_t(HT) + 2 \times Z_p(HT)$, TripleT: $1 \times Z_t(TT) + 1 \times Z_p(TT)$, Tridex: $1 \times Z_t(TX) + 1 \times Z_p(TX)$.

5.3 Experiments

In our experiments, we show the data and query setup for evaluation and the overall performance of Tridex. Most importantly, we establish baseline methods, such as MAP, HexTree, and TripleT, with the same parameters and environment as with Tridex.

5.3.1 Experimental Setup

Our evaluation plan was divided into two categories: scalability and performance. To measure scalability, we computed the index construction time and space consumption for each triple index scheme. Performance evaluation included query response time and index update time for the given queries. All experiments were executed on a 2.4 GHz quad-core Intel Core2 Q6600 processor with 16 GB RAM running Windows 7 Enterprise edition. We used Oracle Database 11g for the triple database. Although the Oracle database provides semantic features such as RDF store, we ignored system-dependent features of the Oracle database and used only a three-column big table for storing the dataset. This means that our system is independent of the physical data store so that any data-storing technique including a file system could be used for our triple database.

We used two triplized datasets for each experimental index tree: the DBLP bibliography database [41], and a TPC-E database [53] for online transaction processing. DBLP data set contains 1,814,115 triples from 100,000 tuples with 11,582 authors, 6,767 inproceedings, and 134 proceedings. TPC-E data set contains 32,157,167 triples from 5,041,731 tuples in 33 relations and 234 columns which are generated by EGen v1.12.0 [52]. DBLP triple database is much smaller than TPC-E triple database, where the distribution of data is more sparse compared with TPC-E. Both two databases have NULL values within original data set; we replace NULL values with reserved keyword so that performance is not affected seriously by the skewness of data.

Every tuple in each database was transformed into triples using the

definition of our triple database, and we designed a set of various types of queries according to the relation between query set and triple query pattern in Table 1.

As mentioned in Section 5.2.1, every possible pattern of triple queries comes down to a combination of specific triple patterns. Thus, we categorize 14 basic triple patterns, including STP, SGJP, and CGP. Table 9 shows the query set we used for the empirical study. In detail, Q1–Q3 means simple one-dimensional triple pattern-finding queries. An example of Q3 is described by this sentence: *"Find all triples that have predicate Name and object Kim."* In addition, queries Q4–Q6 are examples of two-dimensional query patterns. For example, the request *Find all triples with value Kim* is transformed into query type Q5. Q7–Q12 are examples of simple join operations. Like the SQL join operation in a relational database, *"Find the name of persons of age 20."* can be translated into a combination of triple pattern queries, with Q7 and Q9. More complex queries, such as multiple chain-shaped joins or star-shaped joins, are described by Q13 and Q14. Assigned parameters for every query set (i.e., values of *s* and *p* of query <*s p ?o*>) were randomly chosen during query execution repeatedly from predefined sets of keywords, extracted from the triple database.

Table 9: Query set for triple pattern types

Pattern Types			Coverage
STP	*s p ?o*	Q1	- Simple selection - Range search - Exact match - Not null constraint
	s ?p o	Q2	
	?s p o	Q3	
	?s p ?o	Q4	
	?s ?p o	Q5	
	s ?p ?o	Q6	
SJGP	$?s\ p_1\ o_1\ .\ ?s\ p_2\ o_2$	Q7	- Single join - Primary key constraint
	$s_1\ p_1\ ?x\ .\ s_2\ ?x\ o_2$	Q8	
	$s_1\ p_1\ ?o\ .\ s_2\ p_2\ ?o$	Q9	- Unique constraint
	$s_1\ ?p\ o_1\ .\ s_2\ ?p\ o_2$	Q10	- Foreign key constraint
	$?x\ p_1\ o_1\ .\ s_2\ p_2\ ?x$	Q11	- Single join
	$?x\ p_1\ o_1\ .\ s_2\ ?x\ o_2$	Q12	
CGP	$s_1\ p_1\ ?x\ .\ ?y\ p_2\ ?x\ .$ $?y\ p_3\ ?z\ .\ ?s\ p_4\ ?z$	Q13	- Multiple joins - User-defined domain constraint

5.3.2 Scalability

The major distinguishing factor among matching algorithms is the construction time for tree indices. Both TripleT and Tridex show shorter index construction times whereas MAP and HexTree need additional overhead with respect to construction.

The effects of scalability are shown in Figure 26, in which the graphs on the left correspond to a DBLP data set while the graphs on the right correspond to a TPC-E dataset. Generally, Tridex and TripleT outperform MAP and HexTree in every size factor of generated triples. Generally, the

indexes of TripleT and Tridex are about two times smaller than those of MAP and HexTree. In increasing to 100,000 triples of DBLP and 10,000,000 triples in TPC-E as a maximum, in every case, TripleT and Tridex dominated the triple index timings. This shows the simple and lightweight structure of Tridex, compared to MAP and HexTree, due to reduced data setup time and data insertion time (note that the *y*-axis of the TPC-E graph is presented on a logarithmic scale).

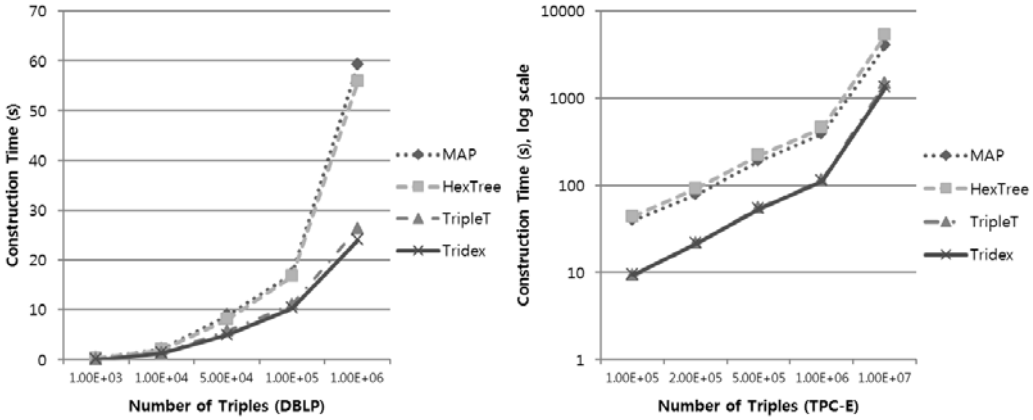


Figure 25: Scalability - index construction time

Next, we consider the memory usage as the number of triples generated is increased. Figure 27 shows the effect of increasing the size of the triple database on memory consumption, using the TPC-E database. Tridex exhibited about 41% better memory consumption versus MAP, but still used nearly twice the memory of HexTree due to the different three index trees and payload buckets. We believe this gap between TripleT and HexTree is affordable for faster query processing.

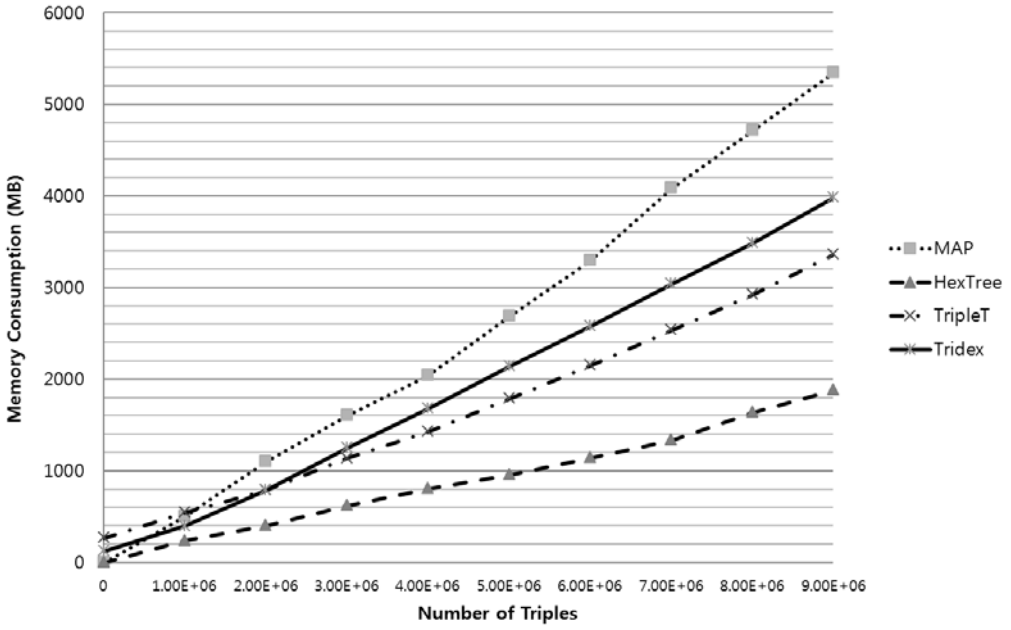
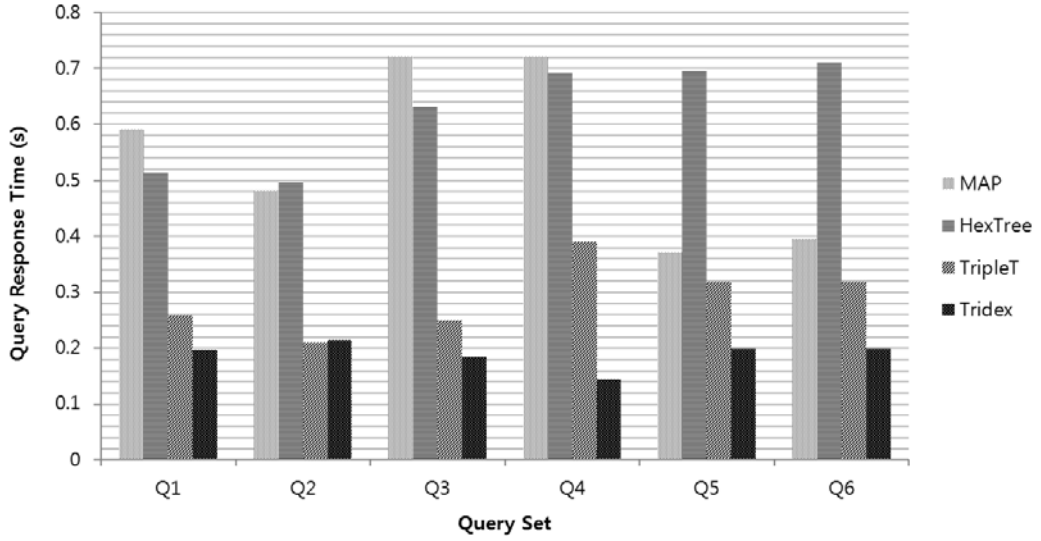


Figure 26: Scalability - index size

5.3.3 Performance

We also tested the overall performance of Tridex, compared to baseline algorithms. Figure 28 compares the query response time of Tridex against MAP, HexTree, and TripleT for query sets Q1 to Q6. Each query requires a different type of simple triple selection to retrieve the set of triples. As a result, Tridex was about average at 166.9% (minimum 59.8% for the next best algorithm, and maximum 257.0% for the worst-case algorithm), among the algorithms compared. In particular, TripleT showed a slightly better performance than Tridex for Q2. This was due to the characteristics of the dataset, in particular, the distribution of the roles of the triples. The number of unique predicates was much smaller than the number of unique objects, such that Tridex needed additional payload scan time for the predicate index tree. Nevertheless, Tridex was more likely to achieve high performance in

most cases.

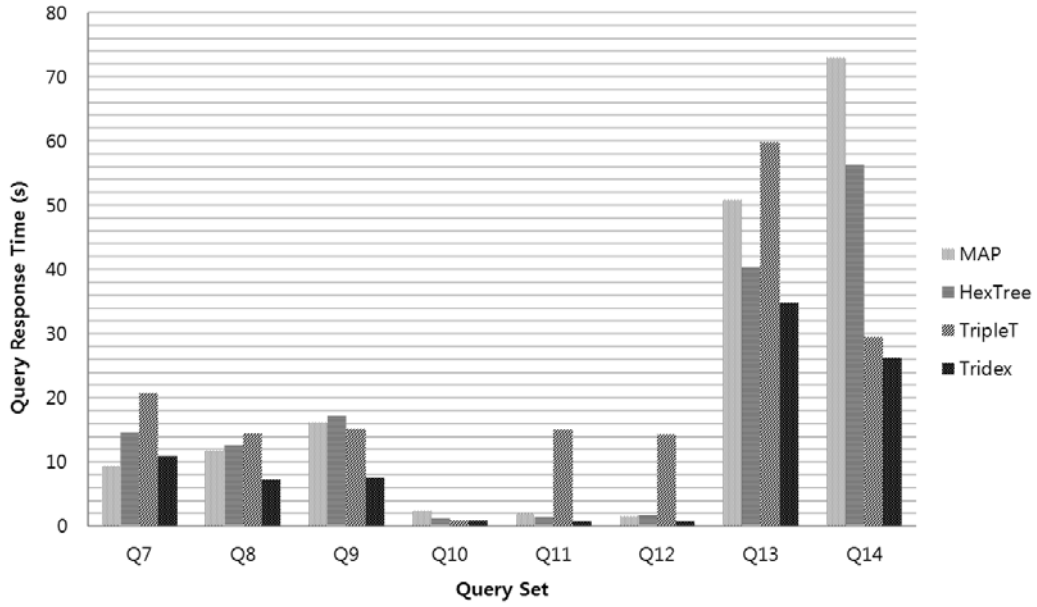


	Q1	Q2	Q3	Q4	Q5	Q6	Avg.
Max. Improved Time (%)	201.0	131.3	295.6	403.5	252.8	258.1	257.1
Min. Improved Time (%)	31.6	-2.3	36.3	17.2	60.9	60.1	59.8
Avg. Improved Time (%)	131.4	84.4	192.9	319.8	133.8	138.9	166.9

Figure 27: Performance - query response time (Q1-Q6)

Figure 29 shows the results of query response times for Q7–Q14. In these tests, we performed more complex triple pattern-finding queries with Tridex and the baseline algorithms. Tridex was far better than the other indexing schemes with respect to most of the query types. The performance of Tridex was improved with a minimum 45.0% for the next best base algorithm to a maximum of 578.1% for the worst-case algorithm, and an average of 237.3%, compared to existing approaches. The advantage of

Tridex was clearly demonstrated via these complex join queries. It can be seen that TripleT showed a significant performance decrease compared to Tridex on Q11 and Q12, which are subject-object join and subject-predicate join queries, due to the distribution of values of each role in the dataset. Figure 30 illustrates the overall results of the performance comparison between Tridex and the other triple index structures with respect to various types of query, using logarithmic axes. Tridex was better able to improve the performance of a triple database, while others had only limited potential.



	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Avg.
Max. Improved Time (%)	92.3	100.4	125.0	185.2	1983.6	1887.5	719.0	179.1	578.1
Min. Improved Time (%)	-13.7	6.3	98.4	1.2	81.9	101.4	15.9	12.6	45.0
Avg. Improved Time (%)	37.4	78.8	111.8	75.3	738.5	709.7	44.7	102.5	237.3

Figure 28: Performance - query response time (Q7-Q14)

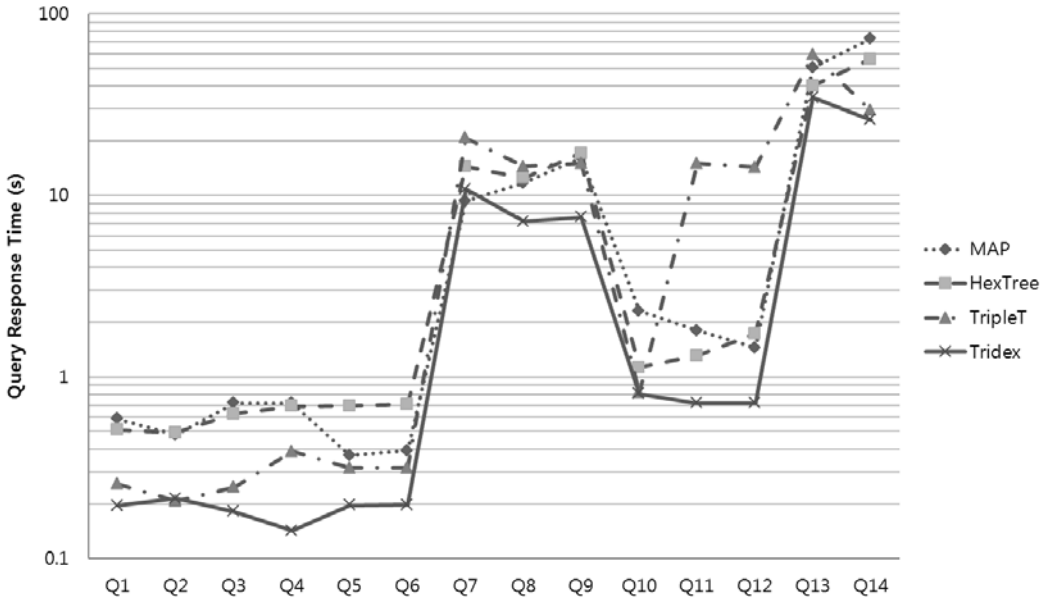


Figure 29: Performance - summary of query response time

In addition, we considered about the effect of limitation of memory space. In this scenario, for searching specified triple patterns by given query set, only a part of leaf nodes and payload buckets of three index trees are stored in main memory. Initial loaded triples are randomly selected, and we apply LRU-K approximation policy [50] for triple replacement. Remaining leaf nodes and payload buckets are stored in disk by file. Figure 31 summarizes partitioned triples within memory and disk.

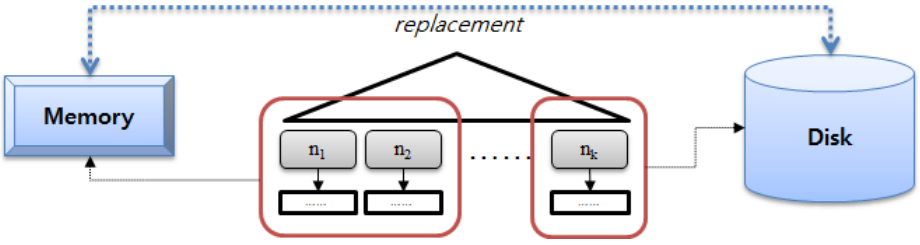


Figure 30: Separated leaf node and payload buckets

We measure the number of disk accesses according to various query patterns. To explore the effect of the limitation of memory space, we set a parameter as memory allocation size and dataset size. For every phase, size of memory allocation is increased by 5% to 100% of total memory space while size of triple data is fixed. Similarly, size of dataset is also increased by 5% step to 100% of total triple dataset while the amount of memory allocation is fixed. We use three query set such as Q5 (simple triple pattern query), Q8 and Q12 (joined triple pattern query) from previous experiment, since the patterns of remaining query set are similar to these three queries.

Figure 32 presents the results for the number of disk accesses by varying memory allocation. MAP and HexTree shows rapid growth of disk accesses than TripleT and Tridex, as the available memory space is decreased. This is because MAP and HexTree use six duplicated tree for indexing all permutations of triple pattern, and it causes the unnecessary tree search for give queries. Tridex shows better performance than TripleT by comparing the performance on 25~70% of available memory space, since the duplication ratio of payload bucket of Tridex is smaller than that of TripleT. Nevertheless, MAP and HexTree are much sensitive to the available memory space than TripleT and Tridex in all sections, irrespective of the query types.

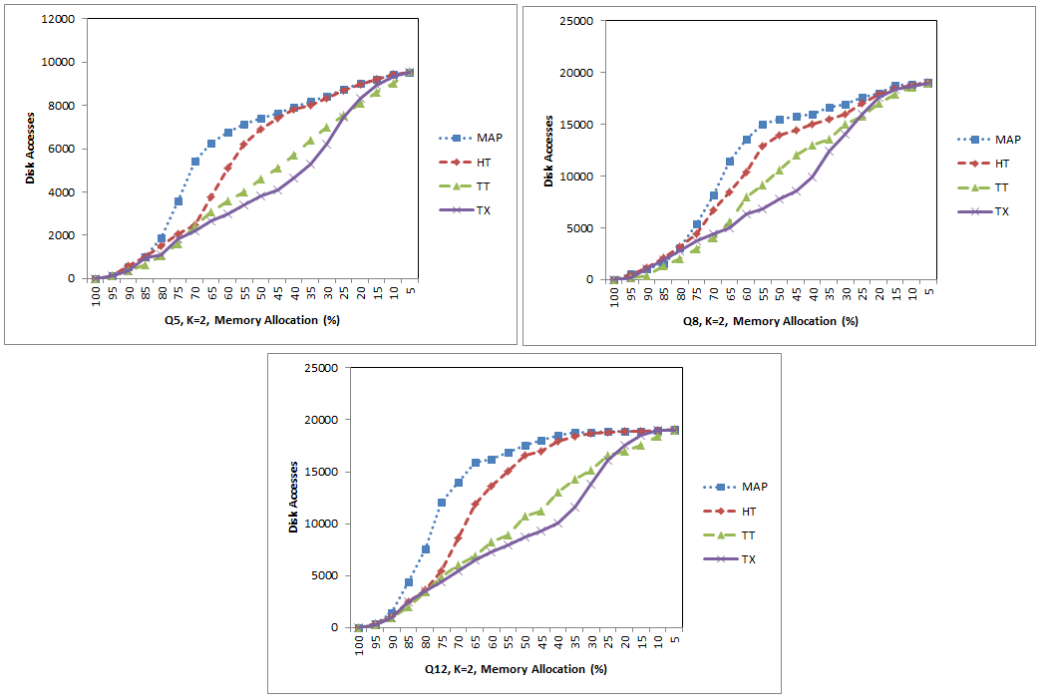


Figure 31: Performance - summary of memory allocation

The next experiment shows the result for disk accesses and dataset size if the size of indexed triples is amortized to min. 10% of original dataset size. We run this experiment using same environment of previous experiment while varying the size of indexed triples. Next figure show the result of the effect of varying dataset size. Initially, four comparative approaches show zero disk access for very small triples, since all indexed triples can be loaded into main memory despite of small memory allocation. However, as the number of triples is increased by 10 million, disk accesses are occurred in MAP firstly when other approaches still does not perform disk access. This is because the size of index tree of MAP is much bigger than other approaches, so that the size of indexed triples exceeds the available memory space. The second thing to note is that HexTree shows the best performance within every section in this experiment, since the natural size of HexTree index

structure is smaller than MAP, TripleT, and Tridex. Similar to previous experiment, TripleT and Tridex show similar performance by increasing the number of indexed triple. However, there is a small gap between TripleT and Tridex. Therefore, the size of index tree and payload bucket not only affects the performance of indexing triples, but also the performance with limited memory space.

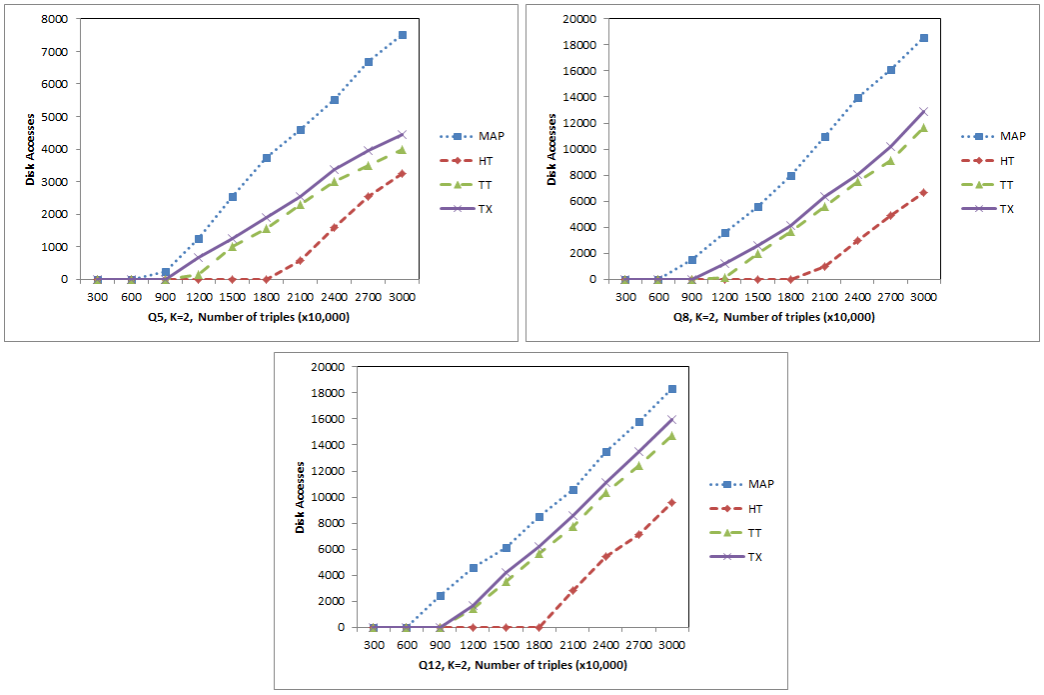


Figure 32: Performance - summary of limited dataset size

Finally, we measured the time necessary for a triple update process. We believe that updating the triple index is important in using a triple database practically; nonetheless, no current approaches provide an analysis of index updating. The complexity of sorting and modifying a triple database according to update operations generally depends on the complexity of the tree data structure, including payloads. If the tree is large, the load for

updating a triple database will be markedly increased. We assumed that updates to 5.51% of a DBLP triple database (=100,000 triples) would be enough to measure the update performance of each triple index. The update experiment was divided into two: *individual update* (update 1,000 triples by 100 times, repeatedly) and *block update* (update 100,000 triples in one process). As shown in Figure 34, Tridex performed slightly better than TripleT, although both MAP and HexTree showed better performance than Tridex for the block update. Obviously, the block update procedure reduces the update time of every index scheme because the I/O cost for accessing the tree and payload is reduced. Hence, we conclude that there is a tradeoff between query execution and index update; nevertheless, the overhead for the index update with Tridex is reasonable.

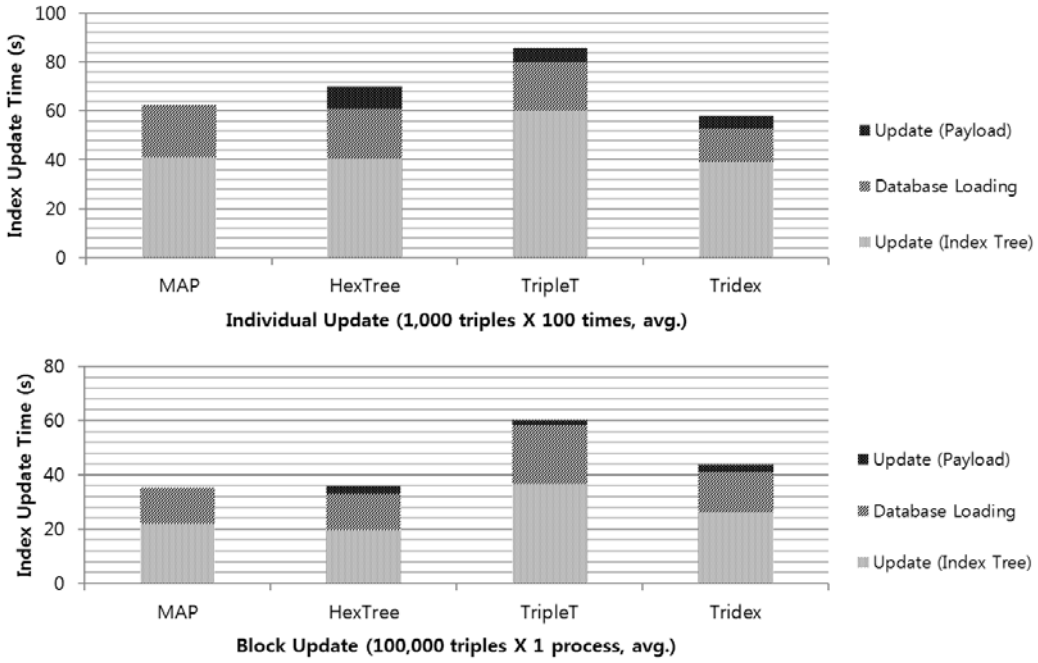


Figure 33: Performance - index update time

In this chapter, we focused on the practical use of a triple index, which can be retrieved from a relational database. We considered that a triple index structure should include the essential features of a relational database, such as integrity constraints and reasonable query processing performance. Tridex is a lightweight triple index structure for an RDB-based triple database. We described the basic structure and logical analysis of Tridex in Section 5.1 and 5.2, and we showed the advantages of our work using an empirical study and comprehensive experimental results in Section 5.3. Further analysis of Tridex should be performed on a full-scale benchmark, such as a triple benchmark system [25, 51]. In addition, in future work, we will consider a heuristic approach to reduce the lookup costs of the triple index for specific types of queries.

For the real-time lookup of an index tree during the query process, the entire index and data nodes as well as the payload bucket should be loaded into the memory. Similar to existing solutions for a triple index, Tridex is also based on a main memory index structure where the entire triple index is maintained in main memory during the query time. Regarding memory usage, we altered our payload structure with a payload buffer by adapting the concept of RDB's index buffer management scheme [4, 20]. The main idea of payload buffer management is to separate the management of active and inactive payload data. The actual data in the payload are kept in physical storage, such as a relational database, while payload buffer has a window of fixed size in memory. The buffer manager selects the payload records according to the policies of payload buffer management. It also decides the set of payload records that will be inserted into the payload buffer. Each data node has a unique payload buffer space of fixed size. The buffer manager

swaps the oldest payload record for a new payload record after a requested query is executed. We believe that an advanced buffer management scheme (e.g., LRU-k approximation) [50] would help to improve the performance of the buffer manager in future work.

Chapter 6. Shortcut Selection

In this chapter, we will discuss another challenge issue about the performance of triple databases, shortcut selection. Despite the well-designed data model and practical index structure, it is difficult to improve the performance of triple database due to the limitation of data management techniques. Most of database systems give attention to assure certain performance with proper index selection, such as materialized view selection in data warehouse system. Actually, it is a critical problem in many applications which are related with several database systems [33]. Many diverse solutions have been proposed and analyzed about the query optimization with proper index. There is no exception in triple database we mentioned in Chapter 3 and 4, that it also should to provide adequate performance for given set of triple queries. We will analyze about the major disadvantage which can occur during the query process of triple database.

The structure of triple database is commonly described as a directed graph, where nodes represent subjects or objects and edges represents predicates. In graph database described by the scheme of triple database, common queries of data retrieving, such as traversing particular paths, can be expensive [21, 37, 59, 40], as well as the queries for the consistency of database mentioned in Chapter 4 and 5. In detail, we describe the problem of query processing in triple database with given example in Figure 35.

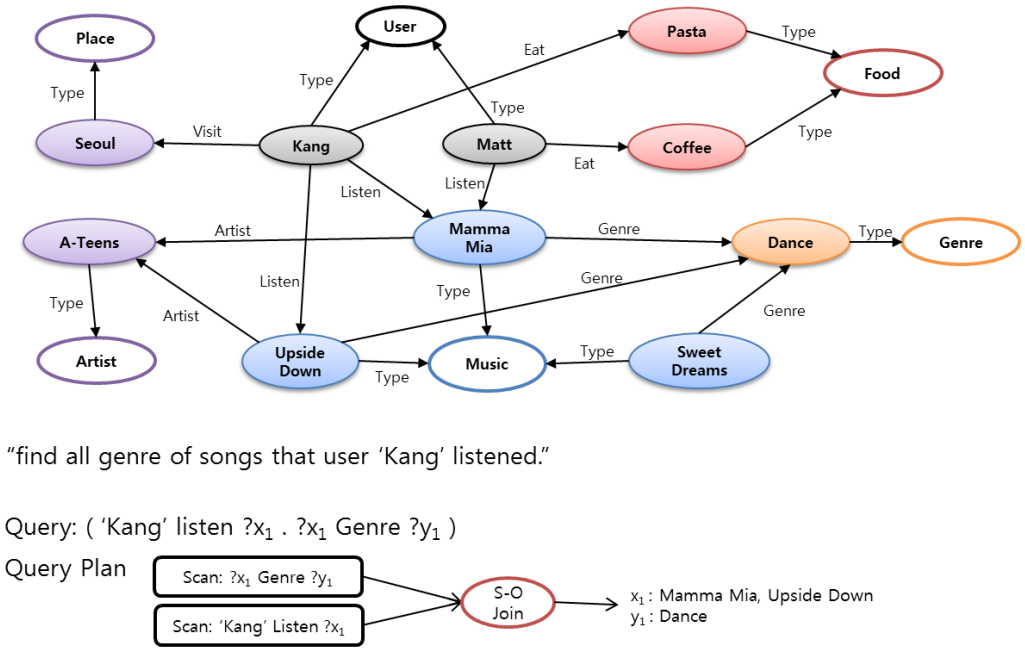


Figure 34: An example graph of triple database

If a user wanted to find all genres of songs that user *'Kang'* listened, this would be require at least one join operation on query from the node *'Kang'* to destination nodes. In SQL style, this data retrieval query can be represented as follows.

```
SELECT B.object
FROM triples A, triples B
WHERE A.predicate = "Listen"
AND A.subject = "Kang"
AND A.object = B.subject
AND B.predicate = "Genre"
```

In other words, triple database with three-column table or property table usually requires *n-1* subject-object joins to connect information for processing path expression queries between the nodes where *n* is the length

between two specific nodes [2]. For speed-up of process of common queries on graph database, large numbers of techniques are introduced until now. We argue that several techniques are useful for improving the performance of triple database, so that we analyze the semantic of triple graph which is retrieved from triple database. By utilizing index structure for integrity constraints and additional features for general triple pattern queries, triple databases get the appropriateness and effectiveness.

It can be challenging issue to reduce self-joins in triple database, since the volume of triple database is extremely large generally. If we assume that there are one million of triples in triple table, self-joins on triple table would cause self-joining triple table *n-1* times repeatedly is significant overhead for database system, since every million triples participate with join process n-1 times repeatedly. As the one of the solutions for reducing self-joins of triple database, triple table is split in several table and stored separate, such as vertical portioned schema. However, there also should be additional joins between separated tables are required for given path expression. Instead of table-partitioned approaches, if we describe the direct path between nodes which are not connected directly and store the result of path expression queries, self-join could be reduced during query execution.

In the thesis, we describes *shortcut*, a direct path between specific nodes. We can get a set of all possible shortcuts by calculating the transitive closure of the given graph [45]. Nevertheless, considering all possible shortcuts in a large graph of triple database is impossible, since the volume of triple database is often too large to enumerate all possible shortcuts. Thus, the optimization of triple database now focuses on the selection of adequate shortcuts. Generally proper shortcuts are selected by the ranking of the

shortcut benefit, which is given by the consideration of profit and cost of generated shortcuts. In other words, shortcut selection problem converges into an optimization problem in which we seek to select shortcuts that maximize the benefit of shortcuts with a given space of time limitation. Figure 36 depicts the result of shortcut creation between node *'Kang'* and the destination node *'Dance'*.

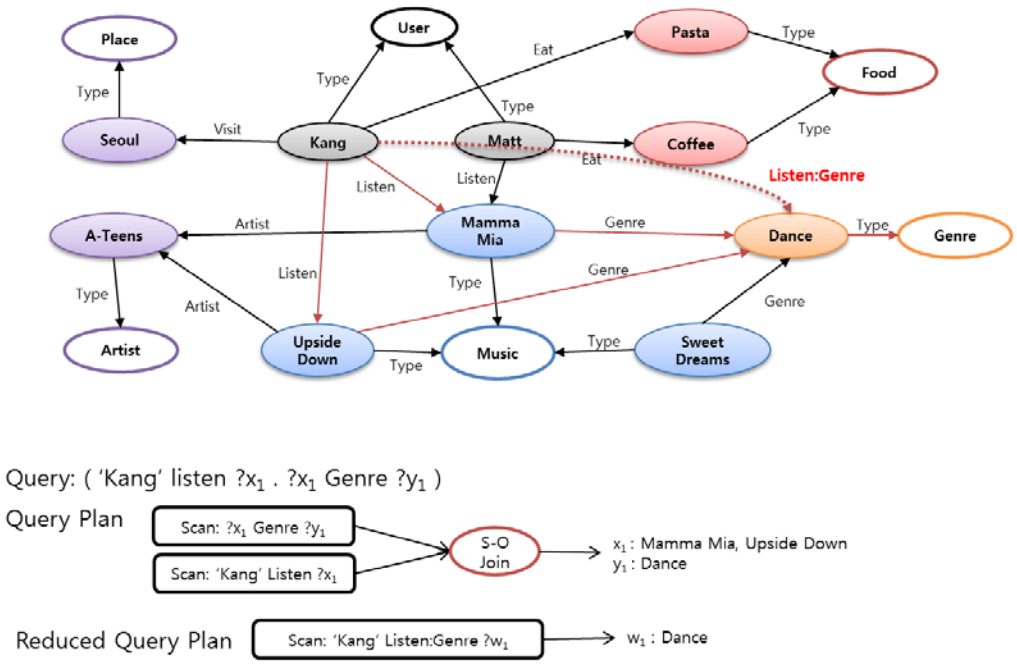


Figure 35: An example of shortcut creation

Until now, state-of-the-art solution for optimizing shortcuts in triple database is *Candidate Shortcut Selection* introduced by V. Dritsou et al [24]. It focuses on the *"interesting"* shortcuts that correspond to frequent accessed paths by augmenting the schema and the data graph of a triple database with additional triples. They define the candidate shortcuts by the relationship between shortcuts and given query workload, and select proper set of

shortcuts that contains “interesting” node which are used for starting or ending node of queries.. However, in most case of practical solutions with relational database, most shortcuts are included in list of candidate shortcuts, so that the number of shortcuts which has to be considered is not decreased particularly. Furthermore, we already assume that triple database can be used for transactional process by considering the consistency of database with integrity constraints. In this situation, we believe that the process of benefit calculation also should reflect the maintenance of database caused by database update. For solving these two problems, we modeled a new model for shortcut selection by adapting *shortcut pruning* process before getting candidate shortcuts, as well as the refined benefit calculation model which contains the measurement of database update. We called our approach *Reduced Candidate Shortcut Selection* (RS).

The rest of this chapter is organized as follows. In Section 6.1, we briefly explain the preliminaries of our approach. We review the drawbacks of current *Candidate Shortcut* Approach in Section 6.2. In Section 6.3, we describe the process of pruning inadequate shortcuts by ranking of candidate shortcuts by using graph ranking techniques. In Section 6.4, we suggest how to model a benefit calculation function of candidate shortcuts using profit and cost parameter. We evaluate our approach and discuss the experimental results in Section 6.5, and conclude our work and discuss the future work in Section 6.6.

6.1 Preliminaries

Note that every triple database can be represented as a graph. We adapt

the definition of triple graph scheme from traditional graph representation scheme [56].

6.1.1 Schema Graph and Instance Graph

Definition 5. ***Triple Graph.*** *A triple graph with triple database T is defined as a directed graph* $G = (V, E)$*, where V is a finite set of nodes, , where V is a finite set of nodes,* $E \subseteq V \times V$ *is a finite multi-set of edges. Each node is correspond to a subject t(s) of object t(o) of given triple* $t \in T$*, and each edge is mapped to a predicate t(p) of t.*

Figure 35 gives an example of a triple graph. The graph describes information of music recommendation service with service logs, and it consists of 20 triples of triple database. Given a node v, $N(v)$ denotes the set of neighbors of v, that is the set of nodes $u \in V$ such that $(u,v) \in E$ or $(v,u) \in E$. Given a set S of nodes, the *neighborhood* of S is the set $S \cup N(u)$ where $u \in S$. Path P from node v_1 to v_k in G denotes the finite sequence (v_1, v_2, …, v_k) of nodes such as (v_i, v_{i+1}) $\in E$, $i=1,..,k-1$ and $v_i \neq v_j$ for each i≠j.

In order to distinguish the schema-level information of a given triple graph from the set triple instances, we define the *schema graph* and *instance graph* as follows.

Definition 6. ***Schema Graph.*** *The schema graph of a given triple graph* $G = (V, E)$ *is a directed graph* $G_s = (V_s, E_s)$*, where* V_s *is a finite set of V and* E_s *is a finite set of E.*

Definition 7. ***Instance Graph.*** *The instance graph of a given schema graph* G_s *is a directed graph* $G_i = (V_i, E_i)$*, where* V_i *is a finite set of instances of nodes in* V_s *and* E_i *is a finite set of instances of edges in* E_s*.*

Schema graph is a schema-level template for given triple database, where instance graph describes the actual contents of triple database. With the definition of schema graph and instance graph, Figure 37 shows an example of the schema graph and instance from triple database.

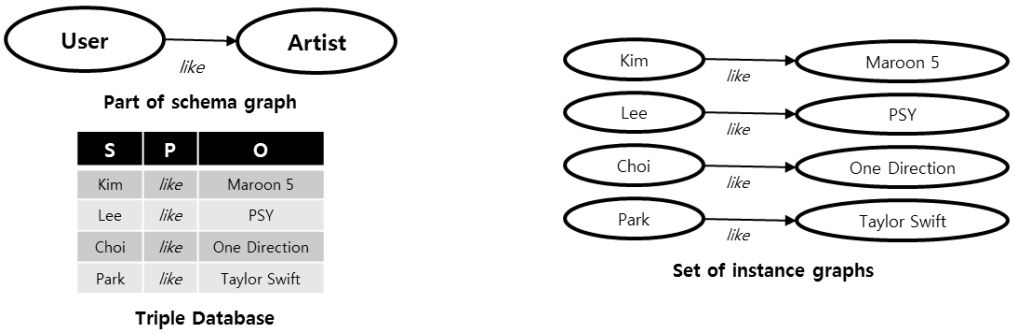


Figure 36: An example of schema and instance graph

6.1.2 Shortcut and Query Workload

From the schema graph, we can define *shortcut* which correspond to direct path between specified pair of nodes (v_i, v_j) where there is no *neighborhood* between v_i and v_j. In detail, a *shortcut collection* ***SC***$=\{sc_1, sc_2, \ldots, sc_n\}$ on a schema graph is defined as a finite set of subpaths which

contain more than one edge. One shortcut on schema graph contains multiple shortcuts on instance graph, where one instance shortcut on instance graph means one additional triple which depicts the shortcut between specific pair of triples. There can be $|V_s| \times |E_s|^2$ shortcuts in schema graph G_s. Similarly, in instance graph G_i, there can be $|V_i| \times |E_i|^2$ shortcuts. Figure 38 depicts a set of all possible shortcuts on a part of schema graph in Figure 35. It reveals the relationship between shortcut sc_1 on schema graph and corresponding shortcuts on instance graph.

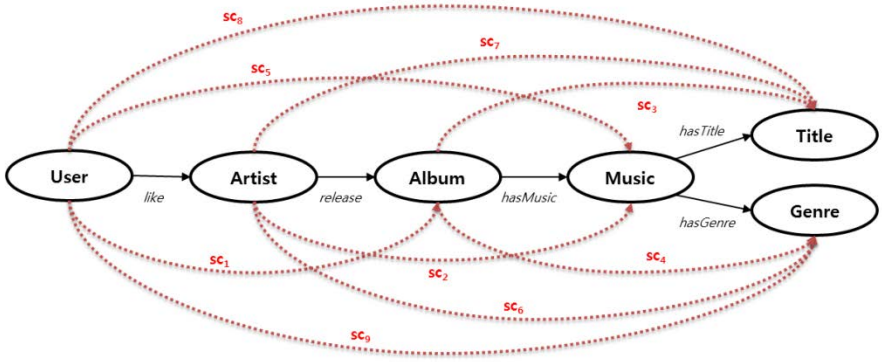


Figure 37: Nine shortcuts on schema graph

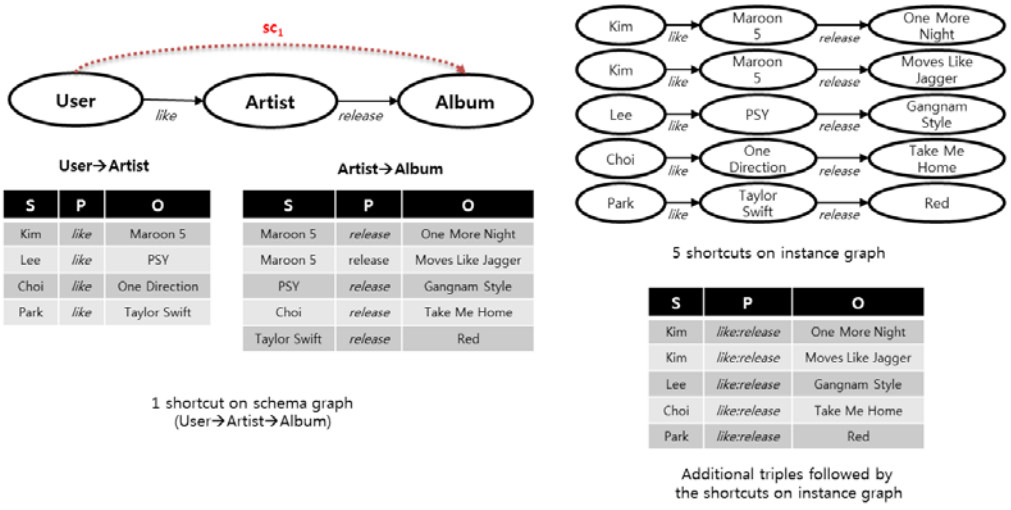


Figure 38: Shortcut on schema graph and instance graph

Another important component of shortcut selection is *a query workload*. A workload, or query workload ***Q*** is a set of queries that is executed on triple database. The definition of query workload is described as follows. Figure 40 depicts an example of query workload with given schema graph in Figure 38. For instance, query q_1 means the request statement *"For each user, find the title of music in certain albums which is released by the artists that user likes."* where query q_3 means *"For each album, find the title of music in that album."*

Definition 8. ***Query Workload.*** *For a given schema graph schema* G_s*, query workload* $Q=\{q_1, q_2, \ldots, q_m\}$*, where each query has non-negative weight* λ_i *for* $1 \leq i \leq m$ *which describes the query frequency. Subquery* Q_s *is a finite set of subpath of given query q containing more than one edge.*

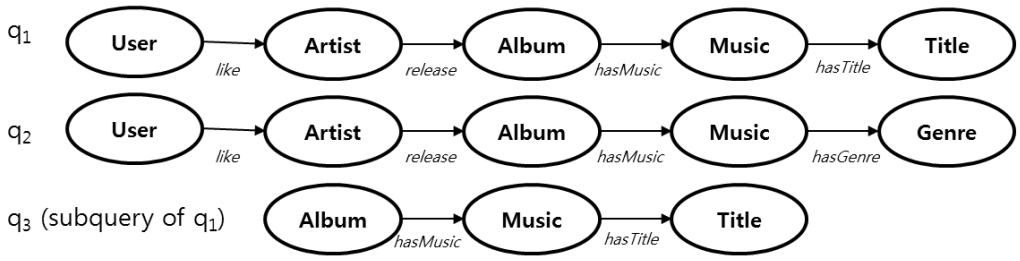


Figure 39: An example of query workload

We also adapt the concept of *related query* and *related node* from previous research on graph database [12]. Related queries $\boldsymbol{RQ}=\{RQ_1, RQ_2, \ldots, RQ_n\}$ is a finite set of queries correspond to shortcut, where at least one of the subqueries is exactly matched with given shortcut collection $SC=\{sc_1, sc_2, \ldots, sc_n\}$. For example, RQ_3 of shortcut $sc_3=\{q_1, q_3\}$ since both queries

have subquery $< ALBUM \xrightarrow[hasMusic]{} MUSIC \xrightarrow[hasTitle]{} TITLE >$. which is exactly matched with sc_3. Similarly, related nodes RN_k^q is a finite set of nodes in given query q_k, where RN_k^{sc} is a finite set of nodes in given shortcut sc_k. For instance, RN_3^{sc} is {*ALBUM*, *MUSIC*, *TITLE*} and RN_1^q is {*USER*, *ARTIST*, *ALBUM*, *MUSIC*, *TITLE*}.

In addition, the shortcut selection problem can be analyzed by graph theory since the shortcut selection in triple database is originally derived from the traditional shortcut problem in graph structure. We survey and adopt the definition of shortcut problem from previous researches [12], and explain the feature of shortcut selection problem by the definition of common graph theory.

According to previous definition, a graph $G = (V, E, len)$ denotes a directed, weighted graph with positive length function $len: E \to \mathbb{R}_{>0}$. Path P from x_1 to x_k in G is a finite sequence $(x_1, x_2, \dots, x_k)$ of nodes such that $(x_i, x_j) \in E$ and $x_i \neq x_j$ for each $i \neq j$. A set of shortcuts for graph G is a set $E' \subseteq (V \times V)$ such that, for any (u, v) in E', it is $dist(u, v) < \infty$. Given a graph G and a set of shortcuts E', the gain $w_G(E')$ of E' is

$$w_G(E') = \sum_{s,t \in V} h_G(s,t) - \sum_{s,t \in V} h_{G[E']}(s,t)$$

where $h_G(s,t)$ is the *hop-distance* from node s to node t. Therefore, if we can get the every query cost of the edges, hop-distance (=sum of length) can be substituted by query cost.

Basically, shortcut problem is to find E' such that the gain $w_G(E')$ of E is maximal. We believe that maximizing sum of shortcut benefit matches this assumption. Besides, shortcut selection (=decision) problem is to decide

if there is a set of shortcuts E' for G such that $w_G(E') > k$ and $|E'| \leq c$. We also map this constraint by the resource constraint of shortcut selection, such as space and time limitation. Since shortcut selection problem is NP-complete [62], if the number of shortcuts we allowed to insert is bounded by a given constant k_{max}, the number of possible solutions of the shortcut problem is at most,

$$\binom{|V|^2}{k_{max}} = \frac{|V|^2!}{(|V|^2 - k_{max})!\, k_{max}!} \leq |V|^{2k_{max}}$$

By the definition of our shortcut selection problem and this conventional definition of graph theory, we find that k_{max} can be given by the related node and related query we explained above for setting affordable shortcut selection problem.

Finally, approximation of the shortcut problem can be solved by a greedy-fashioned approach. For example, given the number of *c* of shortcuts to insert and graph *G*, the greedy approximation scheme consists of iteratively constructing a sequence $\mathrm{G} = G_0, G_1, \ldots G_k$ of graphs where G_{i+1} results from solving the shortcut problem on G_i with only one shortcut allowed to insert. In this manner, we assume that there is one node to add in schema graph G_s. It is likely to get the subgraph $G' = (V_n, E_n, len)$ with new node v_n when we want to build new set of shortcuts due to the node insert. Then we can get the new set of shortcuts $E'' = E' \cup e$ where *e* is the new shortcut between corresponding nodes and new node in *G'*.

6.2 Problem Specification

With the schema graph G_s, instance graph G_i, and the query workload

Q, the main objective of shortcut selection is to derive an appropriate shortcut collection in G_s (shortcuts in G_i is determined by G_s) which can maximize the performance of query execution in Q. In other words, shortcut selection problem is the combination of selecting an appropriate set of shortcut on schema graph and making new paths on instance graph using selected shortcut collection. In detail, there is trade-off between the number of shortcuts and the improvement of query performance. Building all possible shortcuts can guarantee the maximization of improvement of system performance; however it usually requires excessive time and space for generating all possible combinations between nodes. On the other hand, small numbers of shortcut collection gives low overhead for adding new shortcuts on a database, although it rarely contributes to improve the performance of database system.

Actually the shortcut selection has something in common with the view selection issues in relational database [39]. Conceptually, both shortcut selection and materialized view selection are about the additional components structure that can significantly accelerate performance. Shortcuts are constructed between multiple triples, where materialized view may be defined over the multiple tables [3]. And both problems are known to be a NP-complete problem [35]. In fact, many core concepts of shortcut selection in triple database may refer those of materialized view selection. Following the concept the materialized view selection, we also define three parameters of improvement of query performance of triple database as follows:

- *Query Processing Cost*
- *Shortcut Building Cost*

- *Space and Time Limitation*

We rewrite the specification of shortcut selection problem with those three parameters; *Choosing appropriate shortcut selection which minimize query processing cost and shortcut building cost with given space and time limitation*. For the desirable balance among three parameters, *Candidate Shortcut selection* is introduced [24]. It focused on selecting *candidate shortcuts* by query workload, since there are too many possible shortcuts so that it is impossible to build all possible shortcuts in space and time limitation. Candidate shortcuts are a set of "*interesting*" shortcuts which contain the starting or ending node of given queries in query workload. After getting a set of candidate shortcut, system calculates the benefit of shortcuts with heuristic approach such as *bi-criterion optimization and linear relaxation*. The criteria of estimation of shortcut benefit functions are exactly matched with defined parameters, query processing cost and shortcut building time. In the process of *linear relaxation*, they use another heuristics; (i) prohibition of the shortcut with the smallest benefit for the given query, (ii) prohibition of the shortcut with the shortest corresponding query. After all, candidate shortcuts with score are added into instance graph in regular sequence by greedy algorithm. Figure 41 shows an example of candidate shortcuts in schema graph shown in Figure 38 by using query workload in Figure 40.

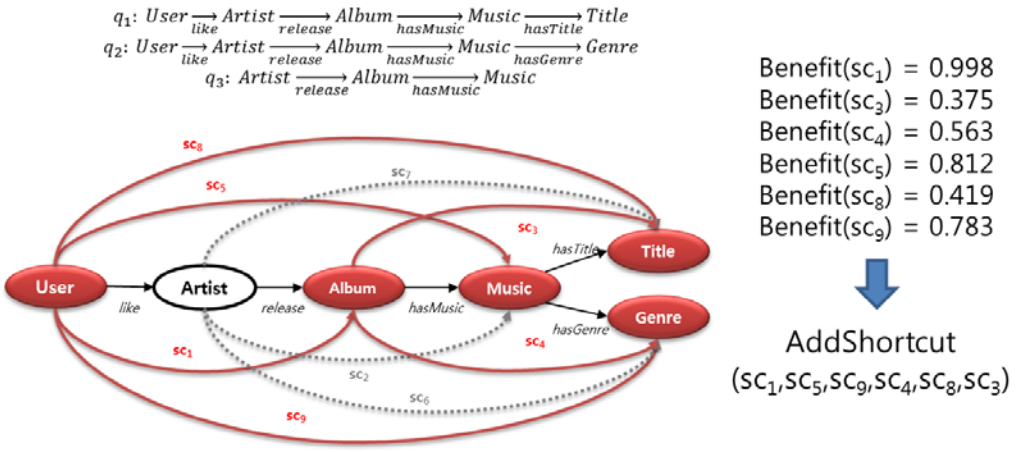


Figure 40: An example of candidate shortcut selection

The main drawback of candidate shortcut selection is that it usually generates too many candidate shortcuts on instance graph, so that the estimation of candidate shortcut is rarely contributed to reduce the shortcut building time. The amount of candidate shortcuts is in proportion as the volume of query workload. In other words, the effectiveness of pruning shortcuts before getting candidate shortcuts is getting much lower when new queries are added into query workload by the requests of various users. For explaining this situation, we observed a relationship between TPC-E triple database and its benchmark queries [53]. Among the 23 benchmark queries, about 87.5% of total attributes in relations participates during the query execution. As a result, if we use TPC-E benchmark queries as the query workload, we can estimate that about 93.7% of total shortcuts are candidate shortcuts of TPC-E triple graph. If we have certain limit for space resource, such as 50% of all possible shortcuts, 43.7% of total shortcut is abandoned despite of their unnecessary calculations before the shortcut benefit calculation process. In other word, candidate shortcut approach is only useful

when the volume of candidate shortcut is equal of smaller than pre-defined given space limitation. Similar to the TPC-E case, we predict that space for candidate shortcut is not much smaller than given shortcut space at most cases.

Another demerit of candidate shortcut approach is the lack of consideration about shortcut maintenance cost. As me mentioned in Chapter 3, 4, and 5, our triple database is ready to accept transactional queries which accompany the changes of database. If there are some changes both schema level and instance level in database, generated shortcut is also recomputed by the database update. Generally, the change of schema graph affects more the performance of query execution with shortcuts than the change of instance graph. However, for assure of the integrity of database, established shortcuts which are related with database update should be recomputed and regenerated every time for change of given parameters. Thus, we believe that the shortcut benefit calculation model has to hold the components that reflect the importance of database update. For example, let there be three nodes *ALBUM*, *TITLE*, *GENRE* in schema graph. Two shortcuts sc_3: *ALBUM→TITLE* and sc_4: *ALBUM→GENRE* is established between each pair of nodes. Every node has its own *update frequency* that represents the probability of update of node value. If the update frequency of *TITLE* is higher than *GENRE* where the volume of sc_3 and sc_4 in instance graph is same, then system gives higher rank to sc_4 than that of sc_3 because the cost for rebuilding shortcut sc_4 may be much lower than the cost of rebuilding shortcut sc_3. Nevertheless, current solution of shortcut selection for triple database does not consider the update of triples. We insist that shortcut benefit calculation model should contain the factor of *shortcut maintenance*

cost at this point of view. Figure 42 describes this example with given schema graph.

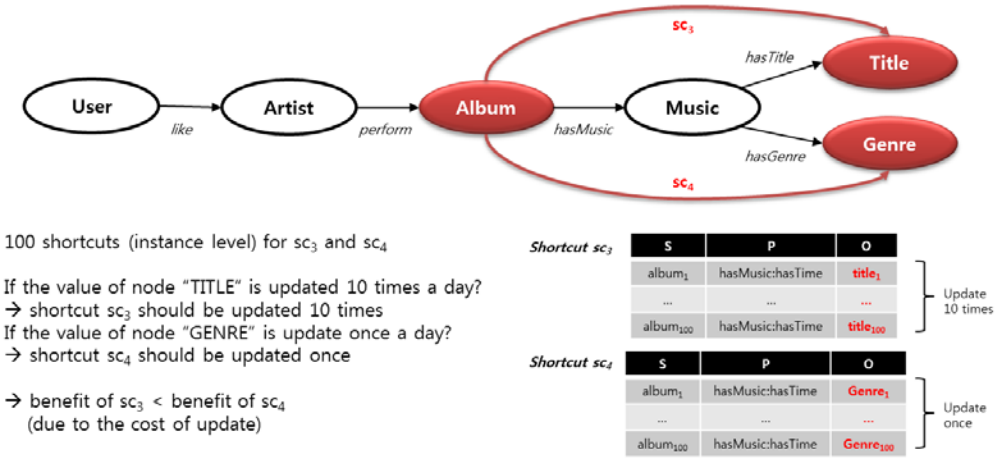


Figure 41: Consideration of shortcut maintenance cost

The goals of this chapter are summarized in two keywords: (i) *reduced candidate shortcut selection* (ii) *shortcut maintenance cost*. For the first goal, we introduce the process of pruning available shortcuts within space limitation by using link analysis algorithm and heuristics before getting a set of candidate shortcuts. And for the remaining goal, we suggest refined shortcut benefit calculation model which consist of profit and cost functions that holds the shortcut maintenance cost factor. Figure 43 depicts the summary of our approach of shortcut selection.

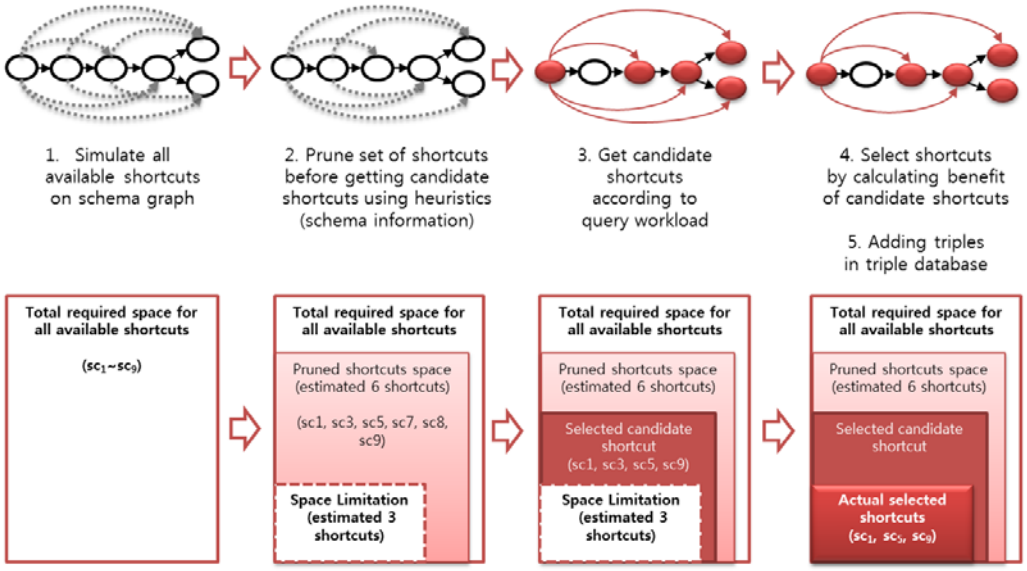


Figure 42: A summary of process for reducing candidate shortcuts

6.3 Reducing Candidate Shortcuts

A shortcut is a virtual path between two nodes in triple graph. The problem of choosing "*important*" shortcut has become the problem of selecting "*important*" pair of nodes from graph. We first obtained some intuitions of heuristics that decide the importance of shortcut collection.

6.3.1 Using Schema Information

As is described in Section 3.2.3, our triple database gathers the schema information of transformed relations and holds the statistical data of original relations, such as the number of tuples in relation, or the number of tuples that reference another tuple in other relations. In the observation of materialized view selection problem, Agrawal et al. insist that there are certain table-subsets such that, even if we were to propose materialized views

on those subset it world only lead to a small reduction in cost for the entire workload [3]. We believe that this observation is also useful for analyze the importance of shortcuts on triple database. In other words, there can be shortcuts with small portion of triples which lead to inexpensive query cost. For example, consider a triple database T_1 with 12,200 triples with query workload *Q*. Schema graph of T has nodes {*USER*, *ARTIST*, *ALBUM*, *NATION*, *CONTINENT*} and the edges {*USER*→*ARTIST*, *ARTIST*→*ALBUM*, *USER*→*NATION*, *NATION*→*CONTINENT*}. Let there be 10,000 triples of $USER \rightarrow ARTIST$, 2,000 triples of $ARTIST \rightarrow ALBUM$, 100 triples of $USER \rightarrow NATION$, and 100 triples of $NATION \rightarrow CONTINENT$. With this example, it is likely that shortcut sc_1 proposed on $USER \rightarrow ARTIST \rightarrow ALBUM$ is much useful than the shortcut sc_2 proposed on $USER \rightarrow NATION \rightarrow CONTINENT$. This is because the set of triples which are affected by sc_1 is larger than the set of triples which are affected by sc_2. Hence, the shortcut benefit of sc_2 is insignificant compared to the shortcut benefit of sc_1, where shortcut benefit is dominated by the *size of triples* if there are no other parameters that affect the shortcut benefit calculation. Since our triple database holds the statistical information of schema information, we apply this intuition for reducing the number of candidate shortcut directly.

In addition, we also use the "*the degree of coupling*" of given two relations for deciding the importance of shortcut. Degree of coupling is defined as the number of tuples in one relation that reference other tuples in another relation. Intuitively, it is likely that most queries are executed by using tightly-coupled relations, than those of loosely-coupled relation. For example, consider a triple database T_2 with 300 triples. Schema graph holds

the nodes {*DEPT*, *STU*, *COURSE*, *NAME*} and edges {*DEPT*→*STU*, *STU*→*COURSE*, *STU*→*NAME*}. Let there be 100 triples of *DEPT*→*STU*, 100 triples of *STU*→*COURSE*, and 100 triples of *STU*→*NAME*. Even though the size of separated set of two triples is same, there can be different numbers of connected triples within shortcut sc1: *DEPT*→*STU*→*COURSE* and sc2: *DEPT*→*STU*→*NAME*. That is, if only 10 of the 100 student take difference 10 course in this semester (other 90 students do not participate any course) where every student has different name, then every triple in subpath <*STU*→*NAME*> has been affected by shortcut sc_2 while only 10% of triples in subpath <*STU*→*COURSE*> have been affected by shortcut sc_1. We called the number of connected triples as the degree of coupling. Actually the degree of coupling is proportional to the number of referencing tuples in relation, which can be counted by the foreign key constraint. We can directly compute the degree of coupling of a set of triples by stored schema information, similarly to the size of triples.

Based on this observation, we approach the heuristic of pruning shortcuts using two factors: (i) From the shortcut collection those size of *RN* is large, to the other shortcut collection whose size of *RN* is small. (ii) From the shortcut collection which affects triples of high degree of coupling, to the other shortcut collection which affects triples of low degree of coupling. Considering schema information of original relational database, each shortcut has the *size of related nodes* (sRN) and *the degree of coupling* (DoC) by following formula:

$$sRN(sc_i) = \sum_{v_s, v_e \in RN_i^{sc}} |E(v_s, v_e)|$$

where $|E(v_s, v_e)|$ is the number of edges on instance graph between a pair of nodes (v_s, v_e) on schema graph.

$$DoC(sc_i) = \sum_{v_s, v_e \in RN_i^{sc}} |fk(E(v_s, v_e))|$$

Where $fk(E(v_s, v_e))$ is the number of edges that denote the foreign key predicate on instance graph between a pair of nodes (v_s, v_e) on schema graph. Final score score(sc_i) is given by $sRN(sc_i) \times DoC(sc_i)$. Since both parameters are given by statistical information, it is easy to get the result of $sRN(sc_i)$ and $DoC(sc_i)$ without spending time before selecting candidate shortcuts. With this pre-computed score of shortcuts, we sort the score of each available shortcut in descending order and prune the shortcuts by the space limitation from the top.

6.3.2 Using *PageRank*

Back to the problem of choosing "*important*" shortcuts, it is likely to get the important pair of nodes from graph for calculating the importance of path between pair nodes. That is, simple link analysis algorithm can be applied for reducing candidate shortcuts. *PageRank* [47] is one of the most popularly used node ranking measure. *PageRank* score of a node is proportional to its parent node's *PageRank* scores, but at the same time, the score inversely proportional to its parent node's out degrees. With the given triple graph, the concept of *PageRank* can be mapped simply into the connections of triples: each triple is a node that has its *PageRank* score, where the score is given by

the number of referencing triples retrieved from the schema information of relational database. Thus, we can apply *PageRank* algorithm to get the importance of shortcut instead of heuristics explained in Section 6.4.1. Figure 44 depicts an example of simplified schema graph we used in this Section.

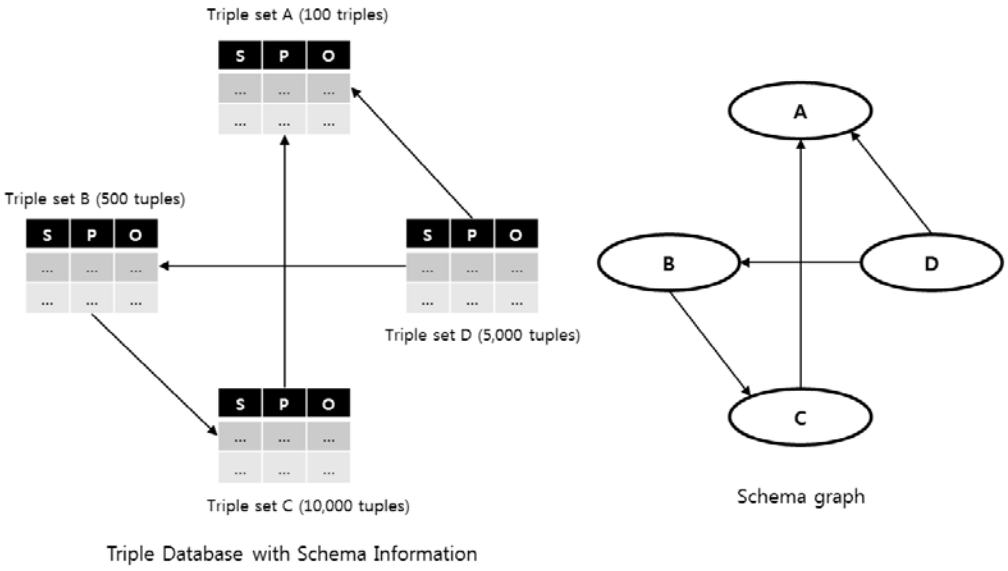


Figure 43: An example of triple database and schema graph

Within the given example, there are four clustered set of triples *A*, *B*, *C*, and *D*. one clustered set is represented as one node on schema graph. Link between the clustered set means a relationship between original relations. Then the score of each node is computed by following formula:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

Where B_u is the set containing all set of triples linking to set *u*. Initial

PageRank score of each node is determined by the policy of the database systems. If system administrator gives importance to the size of the relation, following the intuition from the observation in Section 6.4.1, we can initialize the value of *PageRank* score of each node as the number of triples. In general, each node has same value for initial *PageRank* score so that the sum of initial *PageRank* score is 1. In the example of Figure 44, each clustered set of triples *A, B, C,* and *D* has 0.25 for initial *PageRank* score.

PageRank transferred from a given page to the targets of its outbound links upon the next iteration is divided equally among all outbound links. Outbound links in *PageRank* algorithm can be simulated with sum of the number of edges between nodes in schema graph. Thus, we compute the final *PageRank* score of each node of schema graph as follows:

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

where *N* is the number of nodes in schema graph and *d* is a damping factor, assumed by 0.85 in general. Since there are some nodes without inbound edge (*D*) or outbound edge (*A*), we contain the damping factor with the calculation of *PageRank* score. For example, $PR(A) = 0.496$, $PR(B)$=0.214, $PR(C)$=0.332, and $PR(D)$=0.150 with damping factor d=0.85 after the iteration is completed. Finally, we compute the score of each shortcut with the sum of *PageRank* score of nodes in which shortcut is contained. Following formula summarized the ranking score of shortcut with *PageRank*:

$$score(sc_i) = \frac{\sum_{v_i \in RV_i^{sc}} PR(v_i)}{\left|RV_i^{sc}\right|}$$

$$R(SC) = \frac{\sum_{v_i \in RV_i^{sc}} \left(\frac{1-d}{N} + d \sum_{v \in B_u} \frac{PR(v_i)}{L(v_i)} \right)}{len(RV_i^{sc})}$$

It means that shortcut with more important nodes has ranked higher than the shortcut with less important nodes. For example, there are two candidate shortcuts in Figure 44: $sc_1 = < B \rightarrow C \rightarrow A >$ and $sc_2 = < D \rightarrow B \rightarrow C >$. The final score of sc_1 $score(sc_1) = \frac{PR(B)+PR(C)+PR(A)}{3} = 0.347$ (=0.214 + 0.332 + 0.496) given by the sum of score of participating nodes, where the final score of sc2 $score(sc_2) = \frac{PR(D)+PR(B)+PR(C)}{3} = 0.232$ (=0.150 + 0.214 + 0.332). In conclusion, we assume that shortcut sc_1 is more important than shortcut sc_2. If both sc_1 and sc_2 are candidate shortcut and we has only space for one shortcut by space limitation, we would select sc_1 before getting shortcut benefit calculation of sc_1 and sc_2 since sc_2 is not under consideration of our approach.

In previous example, the final *PageRank* score is converged rapidly because there are few nodes and edges in schema graph. Thus, it takes insignificant time to calculate the entire *PageRank* score of available shortcuts in the example. Even though the size of instance is ultimately large, its schema graph has often much simple structure compared to the vast size of triple instance. However, as the volume of schema graph is getting larger, it may be overhead to calculate the score of each shortcut by *PageRank* algorithm. We applied both heuristics and *PageRank* algorithm within the system implementation, and we will discuss the influence of *PageRank*-

based approach by analyzing the experimental result in Section 6.7.

6.4 Shortcut Benefit Calculation

Another contribution of our approach is to reflect shortcut maintenance cost into the model of shortcut benefit calculation, so that we can get more precise candidate shortcut which is affected by database update. The main objective of shortcut selection problem is the maximization of the benefit which is obtained by the added shortcuts. For structuring the problem of shortcut selection with the benefit factors, we describe a basic representation of shortcut benefit model.

6.4.1 Modeling of Shortcut Benefit Function

Section 6.3.1 describes a part of shortcut selection without the consideration of query workload. After pruning the available shortcuts by schema information or PageRank, we get the appropriate candidate shortcut according to given query workload.

Note that the ultimate goal of shortcut selection is to find a shortcut collection which maximizes the sum of shortcut benefit. Formally, the objective function of shortcut benefit model is defined by a *benefit function* $benefit(sc_i)$ as follows.

$$Shortcuts\ \boldsymbol{SC} = \underset{sc_i \in \boldsymbol{SC}}{argmax} \left(\sum_{i=1}^{m} benefit(sc_i) \right)$$

Basically, the benefit of specific shortcut is acknowledged by the ratio

which defined with the *profit* of specific shortcut and the *cost* of specific shortcut. Thus, the *benefit function* of each shortcut sc_i is defined by *profit function* $profit(sc_i)$ and the *total cost function* $totalCost(sc_i)$.

$$benefit(sc_i) = profit(sc_i) - totalCost(sc_i)$$

6.4.2 Modeling of Shortcut Profit Function

Profit of the shortcut is related with the enhancement of performance when a query is executed in triple database by using given shortcut. System use shortcut if the query contains the path between nodes which can be replaced with that shortcut. Thus, we notice that the profit of shortcut is the difference between the processing cost of query with self-joins and the processing cost of query with join elimination. For example, within an example of schema graph in Figure 45, the profit of sc_8 is given by following formula:

$$queryCost\left(User \xrightarrow[like]{} Artist \xrightarrow[release]{} Album \xrightarrow[hasMusic]{} Music \xrightarrow[hasTitle]{} Title\right)$$
$$-queryCost(User \xrightarrow[like:release:hasMusic:hasTitle]{} Title)$$

Figure 44: Shortcut and query workload

In shortcut selection, the usual criterion for deciding the profit of shortcut is based upon their probability of usage in the future. Since predicting the future frequency of shortcut usage is impossible, we use the given query frequency λ from a log of past query executions as the approximation of frequency of shortcut usage under the assumption that λ has stable value (reference pattern of query is stable). According to this assumption, We can rewrite the profit function *profit()* and query processing cost function *totalQueryCost()* as follows.

$$profit(sc_i) = \sum_{Q \in RQ_i} \begin{pmatrix} totalQueryCost(Q, NULL) \\ -totalQueryCost(Q, sc_i) \end{pmatrix}$$

where RQ_i is a set of related queries that has relationship with sc_i and $QPC(Q, sc_i)$ is the sum of query processing cost corresponding to query Q with given a set of shortcut sc_i. (*NULL* means that there is no shortcut on triple database)

$$totalQueryCost(Q, sc_i) = \sum_{q_k \in Q} \lambda_k \times queryCost(q_k, sc_i)$$

where λ_k is a *query frequency* of the query q_k and $queryCost(q_k, sc_i)$ is the cost of execution of corresponding query q_k with given shortcut sc_i. In general, the cost of query execution is determined by the query response time in system implementation.

6.4.3 Modeling of Shortcut Cost Function

The cost of building shortcut is a pure overhead factor that is requires for attaching additional triples in the database. To achieve the second goal of our approach, the concept of cost of shortcut is extended for practical triple database is extended with the *maintenance cost* of shortcut, as well as the *building cost* of shortcut. Since all the shortcut has to be updated according to the changes of database, we adapt the shortcut maintenance cost function *maintenanceCost(sc_i)* from the concept of view maintenance problem in relational database [43]. As a result, the total cost function *totalCost(sc_i)* is defined with the combination of *shortcut building cost buildCost(sc_i)* and *maintenanceCost(sc_i)* as follows.

$$totalCost(sc_i) = buildCost(sc_i) + maintenanceCost(sc_i)$$

Shortcut building cost is generally derived from the building cost of one shortcut on schema graph and the numbers of shortcuts on instance graph which is retrieved from shortcut on schema graph. For instance, if there is one shortcut from node *A* to node *B* through several nodes, and 100 shortcut instances are generated on the instance where the start node is *A* and end node is *B*, then shortcut building cost is a multiplication of 100 shortcut instances and the building cost of one shortcut of schema graph. Formally, total shortcut building cost is defined as follows.

$$buildCost(sc_i) = \mu(sc_i) \times \delta(sc_i)$$

$$buildCost(\boldsymbol{SC}) = \sum_{sc_i \in \boldsymbol{SC}} \mu(sc_i) \times \delta(sc_i)$$

where $\mu(sc_i)$ is a number of shortcut instances on instance graph which are created by shortcut sc_i on schema graph, and $\delta(sc_i)$ is the building cost of one shortcut sc_i *on schema level.*.

Shortcut maintenance cost is a little different from the calculation of shortcut building cost. We assume that every node in schema graph has *update frequency* φ. φ denotes the probability of changes of database which holds the specific node. For example, if the update frequency of node *A* is 0.1, it means that the value of node A is updated ten times during the 100 query executions. Figure 46 depicts an example of shortcut with update frequency. In this example, node *ALBUM* has zero update frequency, so that it never changes during query execution.

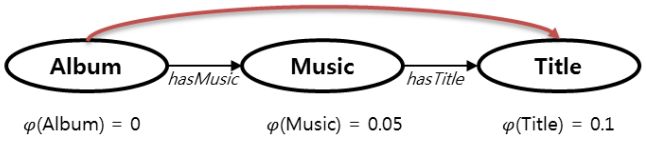


Figure 45: Shortcut with update frequency

Total update frequency of shortcut is obtained by a sum of update frequency of nodes in shortcut. In previous example, shortcut has 0.15 as the update frequency from node *MUSIC* and node *TITLE*. Following formula describes the definition of shortcut maintenance cost function *maintenanceCost(sc$_i$)*.

$$\varphi(sc_i) = \sum_{v_i \in RN_i^{sc}} \varphi(v_i)$$

$$maintenanceCost(sc_i) = \varphi(sc_i) \times \mu(sc_i) \times \rho(sc_i)$$

$$maintenanceCost(\boldsymbol{SC}) = \sum_{sc_i \in \boldsymbol{SC}} \varphi(sc_i) \times \mu(sc_i) \times \rho(sc_i)$$

where $\varphi(v_i)$ and $\varphi(sc_i)$ is the update frequency of a node v_i and a shortcut sc_i respectably, $\mu(sc_i)$ is a number of shortcuts on instance graph, and $\rho(sc_i)$ is a maintenance cost of shortcut sc_i. Without regard to the additional features due to the database update, maintenance cost of one shortcut ρ is basically same as the building cost of one shortcut δ because system rebuilds a set of shortcuts according to the changes of data.

In conclusion, we summarized our shortcut benefit model as follows.

$$Shortcuts\ \boldsymbol{SC} = \{sc_1, sc_2, \ \ldots, sc_m\} = \underset{sc_i \in \boldsymbol{SC}}{argmax} \left(\sum_{i=1}^{m} benefit(sc_i) \right)$$

$$benefit(sc_i) = profit(sc_i) - \ totaclCost(sc_i)$$

$$= \sum_{Q \in RQ_i} (totalQueryCost(Q, \ NULL) - totalQueryCost(Q, \ sc_i))$$

$$- \left(buildCost(sc_i) + maintenanceCost(sc_i) \right)$$

$$= \sum_{Q \in RQ_i} \sum_{q_k \in Q} \lambda_k \left(queryCost(q_k, \ NULL) - queryCost(q_k, \ sc_i) \right)$$

$$- \left(buildCost(sc_i) + maintenanceCost(sc_i) \right)$$

$$= \sum_{Q \in RQ_i} \sum_{q_k \in Q} \lambda_k \big(queryCost(q_k,\ NULL) - queryCost(q_k,\ sc_i) \big)$$

$$- \mu(sc_i) \Big(\delta(sc_i) + \big(\varphi(sc_i) \times \rho(sc_i) \big) \Big)$$

Since $\mu,\ \delta,\ \rho$ is a statistical value from the status of triple database and $\lambda,\ \varphi$ is a pre-computed value for determining query frequency and update frequency, we can get a benefit of one shortcut sc_i by using related queries RQ_i. We compute the benefit of all candidate shortcuts with given query workload Q, and building proper set of shortcuts by rank of sorted candidate shortcut. In conclusion, we can get an appropriate shortcut collection which maximizes the utilization of shortcut construction under the certain query workload.

6.5 Resource Constraints

As we discussed in Section 6.3, the last parameter which determines the improvement of query execution performance is space and time limitation. In particular, there can be some constraint conditions of resource that system is able to use during the shortcut construction. Thus, we refined our shortcut benefit model by attaching the factor of resource constraints, so that the process of shortcut building can be terminated within the given resource constraints. We categorize the three models of resource constraints by the type of resources: *unbounded*, *space constrained*, and *time constrained*.

6.5.1 Unbounded

In the unbounded setting, there is no limit on available resource, especially time and space. In detail, no limit on time resource means that we have enough time to calculate the benefit all candidate shortcuts, where no limit space resource means that we have enough space in storage to store all possible candidate shortcuts from given query workload. Thus, the shortcut selection process only considers the calculation of benefit of generated shortcuts that maximizes the total query processing cost, without regard to the limitation of space and time. Formally the problem in unbounded condition is represented as follows.

$$Shortcuts\ \boldsymbol{SC} = \{sc_1, sc_2, \ \ldots, sc_m\} = \underset{sc_i \in \boldsymbol{SC}}{argmax} \left(\sum_{i=1}^{m} benefit(sc_i) \right)$$

Nevertheless, it is unrealistic scenario at most cases of practical solution. Despite of pruning process of available shortcut, sometimes the volume of considerable candidate shortcuts is too large to fit in the available space. In addition, according to the diversity of give queries, required time for calculating shortcut benefit may offset the performance advantages produced by the shortcut selection.

6.5.2 Space Constrained

In this scenario, space for shortcuts is limited by the threshold of space constraint condition. Usually the total required space for candidate shortcut is dominated by the number of shortcuts on instance graph. In other words,

shortcut benefit calculation model is defined with the reduction in the given space limitation. In our benefit calculation model, factor $\mu(sc_i)$ which denotes the number of shortcuts on instance graph is used for computing the total cost of shortcut building. Thus, the space constrained model of shortcut benefit calculation is refined by the maximization of the sum of shortcut benefits under a certain condition which represents a space constraint, comparing $\mu(sc_i)$ with the given space budget.

$$Shortcuts\ \boldsymbol{SC} = \{sc_1, sc_2, \ \ldots, sc_m\} = \underset{sc_i \in \boldsymbol{SC}}{argmax} \left(\sum_{i=1}^{m} benefit(sc_i) \right)$$

$$\text{under} \sum_{i=1}^{m} \sum_{sc_i \in \boldsymbol{SC}} \mu(sc_i) \leq S \text{ where } S \text{ is the space limitation}$$

6.5.3 Time Constrained

Similar to space constrained model, time constrained model also restrict the use of time resource under certain circumstance. Commonly the amount of query execution time determines the time factor in our shortcut benefit calculation model. In the time constrained model, excessive time for certain query execution may decrease the effectiveness of shortcut selection. Therefore, the query execution time per unit of shortcut dominated the performance of overall system performance. We refine the shortcut benefit calculation model, attaching additional conditions for limited time resource.

$$Shortcuts\ \boldsymbol{SC} = \{sc_1, sc_2, \ \ldots, sc_m\} = \underset{sc_i \in \boldsymbol{SC}}{argmax} \left(\sum_{i=1}^{m} benefit(sc_i) \right)$$

$$\text{under} \sum_{i=1}^{m} \sum_{q_k \in RQ_i} \mu(sc_i) \times \lambda_k \times queryCost(q_k, sc_i) \leq T$$

where T is the time limitation

In order to implement the process of shortcut building process with our shortcut benefit calculation model under resource constrained circumstance, we simply implement a greedy algorithm for shortcut selection in Algorithm 3. The greedy add shortcut algorithm uses a space limitation *S*, time limitation *T*, and the shortcut collections of reduced candidate shortcut *SC* as input. The ranking score of *SC* is computed with the shortcut benefit calculation model with associated parameters, such as the query frequency λ, update frequency φ.

The result of candidate shortcuts is represented as a set of triples. Ranking score and the triple instance are stored together in certain data structure, such as linked list or B+-tree. We can get the score of stored candidate shortcut by iterative manners of algorithm. In detail, certain set of candidate shortcut whose score are computed early are stored in data structure. Our algorithm selects the candidate shortcut repeatedly with the score of next shortcut to be compared, and replace the candidate shortcut with stored shortcuts when the score of new shortcut is larger than the smallest score of stored shortcut. If there is a change of database of parameters, the algorithm needs to update the score of benefit function of stored shortcuts. When the sum of space allocation is exceeding given space limitation of the sum of time duration for selecting candidate shortcut is exceeding given time limitation during the iteration of algorithm, process is stopped and returns stored candidate shortcuts as an output of algorithm.

```
procedure sort(spaceLimit S, timeLimit T, shortcuts SC)
    SC'← ϕ;
    leftSpace←S;
    leftTime←T;
    while (S and T >0) and |SC|>0 do
        for all SC do
            pSC←SC.next();
            // select next candidate shortcut with calculated benefit //
            if pSC.space>leftSpace or pSC.time>leftTime then
                break;
                // cannot store candidate shortcuts anymore //
            else
                leftSpace←leftSpace-pSC.space;
                leftTime←leftTime-pSC.time;
                for all sc_i∈SC' do
                    if pSC.benefit<sc_i.benefit then
                    // compare maximum benefit //
                        sc_i←SC.next();
                    else
                        SC'.add(pSC);
                        break;
                    end if
                end for
                SC.remove(pSC);
                // remove selected candidate shortcut from given set //
            end if
        end for
    end while
    return SC';
end procedure
```

Algorithm 3: Pseudo algorithm for greedy add shortcuts

6.6 Experiments

We design several experiments for evaluating our reduced candidate shortcuts approach and the refined shortcut benefit calculation mode. The first experiment is to evaluate the performance of shortcut building with the variation of space limitation. Time limitation can be also considerable parameters for observing the performance variation. However, we select the

space limitation for the major parameter of resource constraint, due to the convenience of explanation. Actually the pattern of variation of performance according to time limitation is very similar with the pattern of variation of performance according to space limitation. Therefore we use space limitation as the main parameters of the first experiments. The second experiment is designed to observe the effects on performance of query execution. Like the first experiments, we describe the overall performance of baseline algorithm and our approach with the variation of query frequency using two different datasets. In the last experiment, we test about the effectiveness shortcut maintenance cost which is related with triple update. In the experiments, we used two different dataset for testing two different triple databases. The first scenario is the DBLP bibliography database. As we discussed in Chapter 5, our DBLP dataset is smaller and sparser than another dataset. Second dataset is the TPC-E product database. It represents the large size of triple database, which consist of more dense instances. DBLP triple database only contains five nodes in schema graph with simple connections between nodes. In contrast, a schema graph of TPC-E triple database contains 33 nodes, and nodes are connected tighter and closer than those of DBLP database. Figure 47 shows a schema graph of DBLP triple database we used for experiment. Detailed experimental settings are explained in the following Section.

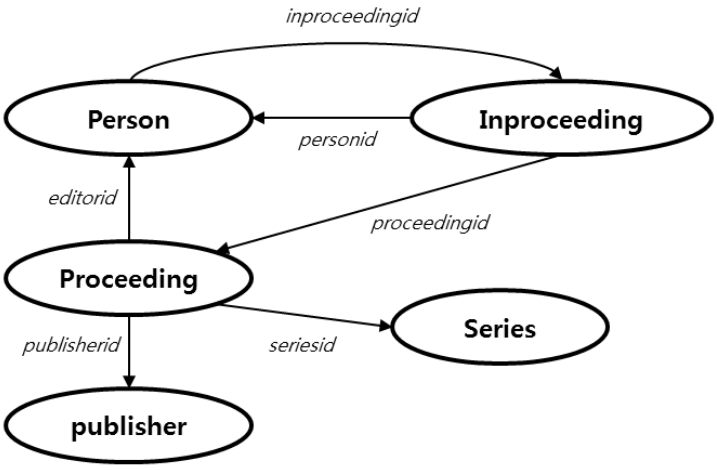


Figure 46: Schema graph of DBLP triple database

6.6.1 Experimental Setup and Query Set

DBLP bibliography database consist of five nodes *PERSON*, *PROCEEDING*, *INPROCEEDING*, *PUBLISHER*, *SERIES* with six edges in schema graph as follows.

- $PERSON \xrightarrow[inproceedingid]{} INPROCEEDING$
- $INPROCEEDING \xrightarrow[personid]{} PERSON$
- $PROCEEDING \xrightarrow[editorid]{} PERSON$
- $INPROCEEDING \xrightarrow[proceedingid]{} PROCEEDING$
- $PROCEEDING \xrightarrow[publisherid]{} PUBLISHER$
- $PROCEEDING \xrightarrow[seriesid]{} SERIES$

For the instance graph, it contains 1,814,115 triples from 100,000 tuples. We estimate 10 different set of available shortcuts on schema graph of DBLP triple database, except the cycle of the path expression. If the system uses the space limitation as the half of the space for available shortcuts, system only

concern five shortcuts for selecting candidate shortcut before the query workload is considered. Other dataset, triplized TPC-E product database, consists of typical product and transaction information generated by *EGen* [52]. It contains 32,157,167 triples from 5,041,731 tuples in original relational database. For both dataset, triple database is generated by the definition we explained in Chapter 3 with the three-column table scheme with the metadata of original relations, such as the statistical information of the relations.

For the evaluation of trustworthy result of query execution performance, we surveyed various types of triple queries and categorized them into several patterns. By the result of analysis of real-world triple pattern query [6], most common type of SPARQL query is SELECT query. During the logs from USEWED 2001 Challenge, 96.9% of queries via DBPedia and 99.7% of queries via of SWDF (Semantic Web Dog Food) is about SELECT query. Only 1.5% of DBPedia and 0.01% from SWDF are other types of SPARQL queries, such as CONSTRUCT. In addition, according to the analysis of frequency of appearance of the different SPARQL features, FILTER (16.26% of DBPedia and 5.21% of SWDF) and DISTINCT (49.19% of DBPedia and 47.28% of SWDF) operation are the most used SPARQL optional keyword. It means that experimental query set should contain proper condition clauses which reduce the amount of the resulted triples.

In the viewpoint of the usage of single triple pattern, we gathered some specified graph pattern with constant (C) and variable (V) in triple pattern queries. For example, pattern <C C V> means *"finding the value of object with given subject and predicate"*. By the result of gathered logs of triple pattern queries, <C C V> query pattern is the most used for deriving the

content of triple. In detail, 66.35% of single triple pattern queries on DBPedia and 47.79% of single triple pattern queries on SWDF belong to <C C V> pattern query. <C V V> and <V C C> pattern queries are also very common for triple database. It means that test query set should consist of common patterns such as <s p ?o>, <?s p o>, and <s ?p ?o> principally. We also adapted the result of distribution of the number of triple patterns in the query, by the explanation of percentage of triple patterns, most of triple queries is just contain single triple pattern (66.41% of DBPedia and 97.25% of SWDF).

Finally, there are remarkable results of triple pattern queries which contain join operation. Join operation is the conjunction of two single triple patterns, where both have at least one variable in common without regarding to the role of variable. So there can be six types of joins, *Subject-Subject, Predicate-Predicate, Object-Object, Subject-Predicate, Subject-Object, Predicate-Object.* According to the previous researches, 4.25% of total queries have at least single join. In detail, most common types of join operation is *Subject-Subject*(S-S), *Subject-Object*(S-O), and *Object-Object*(O-O) join. Only below 3% of join queries is related with predicate. In addition, there is another research that explains the example of experimental setup of triple query set [33]. It contains 14 queries over standard triple store representation of three-column single table. They only consider two types of join operation; *Subject-Subject* and *Subject-Object* join.

Based on these previous researches of triple pattern queries, we generate set of queries for testing our reduced shortcut selection. Query set consist of 20 different queries for two dataset, ten queries for each triple database. Since shortcut selection is about to reduce the self-join on triple database, we

only consider join queries with min. 1 to max. 6 degrees of join combinations. For example, third query (q_3) is defined by the direct path between *INPROCEEDING* and *PROCEEDING*, where two entities are connected by the attribute *proceedingid*. Thereby q_3 can be represented as the simple *Subject-Object* join as follows.

$$q_3 : INPROCEEDING \xrightarrow[proceedingid]{} PROCEEDING$$

For another example, we assume that query q_5 is a variation of q_3 with additional *Subject-Subject* join. As a result, we can represent the example of q_5 as follows.

$$q_5 : INPROCEEDING \xrightarrow[proceedingid]{} PROCEEDING ,$$

$$INPROCEEDING \xrightarrow[personid]{} PERSONID$$

In this manner, ten different set of queries are used for evaluating the effect of reduced shortcut selection on DBLP triple database, as well as another ten different set of query are used for TPC-E triple database. Table 10 summarizes the description of query set.

Table 10: A Summary of query set

	Domain	# of joins	S-S	S-O	O-O	Query Frequency	SPARQL Description
q_1	DBLP	1	1	0	0	0.10 (5^{th})	<?x p_1 o_1 . ?x p_2 o_2>
q_2		1	0	1	0	0.08 (6^{th})	<s_1 p_1 ?x . ?x p_2 o_2>
q_3		1	0	0	1	0.07 (7^{th})	<s_1 p_1 ?x . s_2 p_2 ?x>
q_4		2	1	1	0	0.18 (2^{nd})	<?x p_1 o_1 . ?x p_2 o_2 . s_3 p_3 ?x>
q_5		2	1	0	1	0.12 (3^{rd})	<?x p_1 ?y . ?x p_2 o_2 . s_3 p_3 ?y>
q_6		3	2	1	0	0.23 (1^{st})	<?x p_1 ?y . ?x p_2 o_2 . ?x p_3 o_3 . s_4 p_4 ?y>
q_7		3	1	1	1	0.12 (3^{rd})	<?x p_1 ?y . ?x p_2 o_2 . s_3 p_3 ?x . s_4 p_4 ?y>
q_8		4	2	1	1	0.04 (8^{th})	<?x p_1 ?y . ?x p_2 o_2 . ?x p_3 o_3 . s_4 p_4 ?x . s_5 p_5 ?y >
q_9		5	2	2	1	0.03 (9^{th})	<?x p_1 ?y . ?x p_2 o_2 . ?x p_3 o_3 . s_4 p_4 ?x . s_5 p_5 ?x . s_6 p_6 ?y >
q_{10}		6	3	2	1	0.03 (9^{th})	<?x p_1 ?y . ?x p_2 o_2 . ?x p_3 o_3 . s_4 p_4 ?x . s_5 p_5 ?x . s_6 p_6 ?x . s_7 p_7 ?y>
q_{11}	TPC-E	1	0	1	0	0.10 (5^{th})	<?x p_1 ?y . ?y p_2 ?z>
q_{12}		1	1	0	0	0.08 (6^{th})	<?x p_1 ?y . ?x p_2 ?z>
q_{13}		2	0	2	0	0.12 (3^{rd})	<?x p_1 ?y . ?y p_2 ?z . ?z p_3 ?a>
q_{14}		2	1	1	0	0.12 (3^{rd})	<?x p_1 ?y . ?y p_2 ?z . ?x p_3 ?a>
q_{15}		3	0	3	0	0.18 (2^{nd})	<?x p_1 ?y . ?y p_2 ?z . ?z p_3 ?a . ?a p_4 ?b>
q_{16}		3	1	2	0	0.23 (1^{st})	<?x p_1 ?y . ?y p_2 ?z . ?z p_3 ?a . ?x p_4 ?b>
q_{17}		4	1	3	0	0.04 (8^{th})	<?x p_1 ?y . ?y p_2 ?z . ?z p_3 ?a . ?a p_4 ?b . ?y p_5 ?c>
q_{18}		4	1	2	1	0.07 (7^{th})	<?x p_1 ?y . ?y p_2 ?z . ?z p_3 ?a . ?x p_4 ?b . ?c p_5 ?y>
q_{19}		5	1	3	1	0.03 (9^{th})	<?x p_1 ?y . ?y p_2 ?z . ?z p_3 ?a . ?a p_4 ?b . ?x p_5 ?c . ?d p_6 ?y>
q_{20}		6	2	3	1	0.03 (9^{th})	<?x p_1 ?y . ?y p_2 ?z . ?z p_3 ?a . ?a p_4 ?b . ?x p_5 ?c . ?y p_6 ?d . ?e p_6 ?z>

6.6.2 Baseline Algorithms

We compared our reduced shortcut selection method with state-of-the-art shortcut selection within triple database: *Candidate Shortcut Selection* (CS) method. We implement baseline methods from the previous research of candidate shortcut method. Two different reduced candidate shortcut selections are used for comparing the performance of query response time

and shortcut maintenance cost: *Reduced Candidate Shortcut with Heuristics* (RSh) and *Reduced Candidate Shortcut with PageRank* (RSp). *RSh* uses heuristics with schema information we explained in Section 6.4.1 for pruning available shortcuts before getting candidate shortcuts, where *RSp* uses naïve *PageRank* algorithm explained in Section 6.4.2 instead of heuristics.

We compare additional algorithm about join optimization of triple pattern queries, which is not using shortcut selection method. There are some researches about optimizing self-joins on relational database. *SCALE* (Sort for Clustered Access with Lazy Evaluation) [32] is one of the state-of-the-art research about the efficient self-join algorithms, which takes advantages of the fact that both inputs of a self-join operation are instances of the same relation. *SCALE* first sorts the relation on one join attribute, say R.A. In this way, for every value of the other join attribute, say R.B, its matching R.A tuples are essentially clustered. As *SCALE* scans the sorted relation, each tuple is joined with its matching tuples co-existing in memory. For tuples where full-range clustered accesses to their matching tuples are not possible, they are buffered and the unfinished part of join processing deferred. Such lazy evaluation minimizes the need for “random” access to the matching tuples. Since *SCALE* is originally developed for relational database, we assumed that single three-column table of triple database is an example of big single instance of the relation. We analyzed the fundamental aspects of *SCALE* and mapped into triple database. As a result, we simply implemented the process of *SCALE* for our DBLP and TPC-E dataset except the memory optimization.

6.6.3 Performance of Shortcut Building Time

In this section, we test the performance of shortcut building time based on the candidate shortcut selection. We carry out this comparison for two dataset with two query workloads: DBLP and TPC-E triple database. For the parameters for variation, we used space limitation (% of estimated total space for available storage) of *S* is from 20 to 100. Figure 48 shows that across three approaches, *RSh* achieves the highest space efficiency for given space limitation. Furthermore, *RSp* also shows better performance than *CS* until the available space for candidate shortcut is limited with about 80% of total space. Since the overhead of *PageRank* computation is increased according to the increase of considerable shortcuts, the performance of shortcut building time of *RSp* is slightly over *CS*. In TPC-E triple database, the gap between *CS* and both *RSp* and *RSh* is increased when the available space is limited severely. It can be explained that the process of pruning shortcut makes the amount of considerable shortcut drastically before the selection of candidate shortcuts. However, similar to DBLP database, the performance of shortcut building of *RSp* is decreased as the available space for shortcut is extending. It is reasonable that PageRank computation requires additional overhead before pruning shortcuts. As a conclusion, *RSh* can be considered as the solution for less shortcut building time when the system shows a stable performance for producing certain amount of shortcuts when system has permitted only limited space for shortcuts.

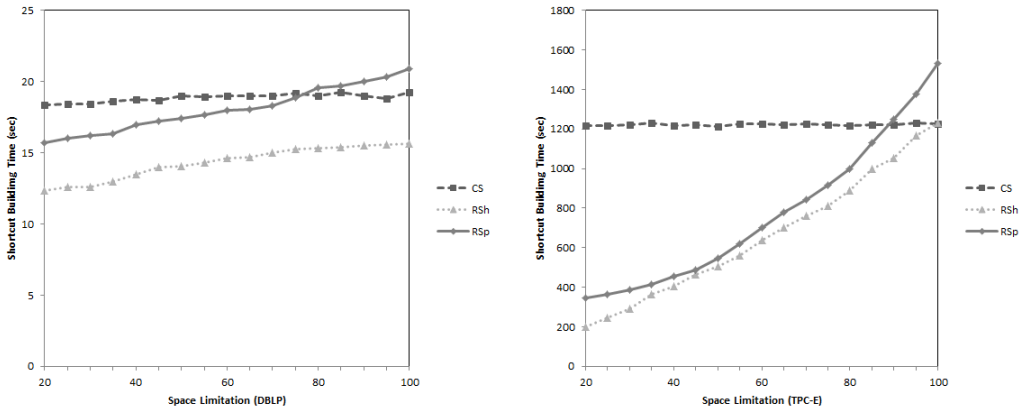


Figure 47: Performance of shortcut building time

6.6.4 Performance of Query Response Time

Next, we illustrate the evaluation of performance of query response time using query workload q_1-q_{10} and q_{11}-q_{20} in Section 6.6.1. We compare four versions of shortcut selection approach – *NS* (No shortcut), *CS*, *SCALE* with our shortcut selection method *RSh*, *RSp*. No shortcut means that there is no shortcuts on triple database, so that the result of given query is retrieved by repetition of self-joins. With the *CS*, *RSh* and *RSp*, system uses shortcut for related queries, if a set of proper shortcuts are established according to the generation process of candidate shortcut selection. If there is no shortcut which correspond to specific query, system just follows traditional process of query execution with self-joins. Parameter variations on this experiment is query frequency λ. Each query has its own score for query frequency, which is determined by the previous researches of triple pattern query survey we described in Section 6.6.1. We predict that the result of query response can be affected not only by the choose of shortcut selection method, but also by the type of query that holds the high query frequency, since the

characteristics of each query in query workload is slightly different from the others.

As a result in Figure 49, *SCALE* and *CS* shows better performance of query response time than *RSh* and *RSp* for query q_1~q_6, which denote the short join query, where *CS* when query q_3, q_4, and q_5 hold the high query frequency. In detail, *CS*, *RSh* and RSp show better performance than NS with similar query response time. The fact is that shortcut selection affects the join query execution, regardless of the decrease of the number of candidate shortcut. of For short join query, *SCALE* shows about 85.91% of performance enhancement on q_1, 67.19% on q_2, and 92.61% on q_4, comparing with worst case. Besides, SCALE does not shows good performance on q_3 and q_5, where query contains O-O join. *SCALE* should cluster values of same relations and there are much more distinct values in object value in triple database than subject and predicate, *SCALE* does not have advantages for q_3 and q_5 than *RSp*, and *RSh*. As a result, we can say that there is little difference between the set of constructed shortcuts in *RSh*, *RSp*, and *CS*, while *RSp* and *RSh* have an advantage of shortcut building time for limited space boundary.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	SUM	Overall
NS	3.24	3.99	3.91	10.68	17.84	20.05	24.88	40.16	57.96	78.42	261.13	18.275
SCALE	1.42	1.28	3.84	3.52	9.96	7.48	12.84	20.38	27.24	43.8	131.76	8.5496
CS	1.78	1.34	2.16	3.87	8.16	6.81	9.88	16.13	25.09	23.04	98.26	6.9532
RSh	1.78	1.5	2.18	4	8.55	6.9	10.01	16.44	25.01	24.07	100.44	7.1148
RSp	1.82	1.44	2.34	3.98	8.16	6.79	10.56	16.99	26.48	24.01	102.57	7.1798

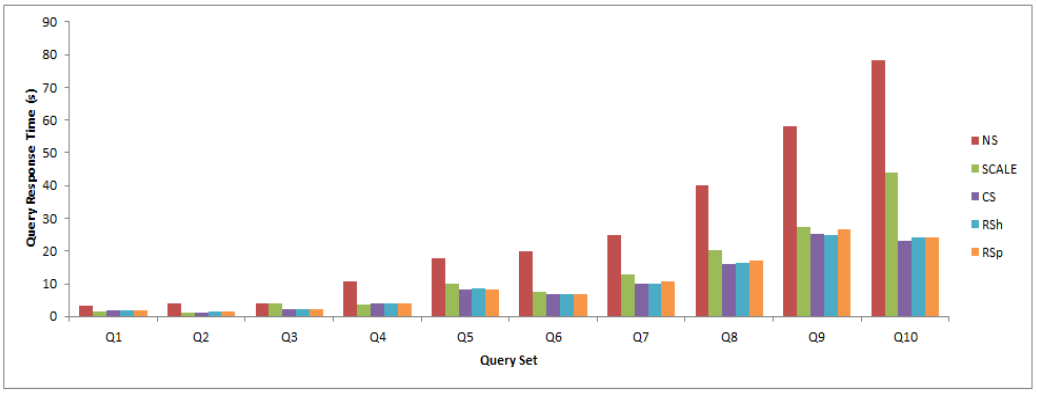


Figure 48: A Summary of query response time (DBLP)

Figure 50 depicts another experiments on TPC-E triple database with a given query workload q_{11}-q_{20}. Similar to the previous experiments, we observed the performance of query response time of query response in terms of the variation of query frequency. The performance evaluation of *SCALE* is much lower than other approaches on scalable triple database, so *SCALE* marks worst query response time with eight queries of TPC-E. In detail, *CS* shows the best performance on q_1 and q_2, which stand for simple single join pattern query. We believe that *CS* can generate more instance-level shortcuts by single schema-level shortcut on TPC-E database, since there are much more instance triples on original TPC-E triple database than DBLP.

	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	SUM	Overall
NS	4.18	4.54	6.78	6.12	15.86	15.26	45.95	55.01	80.96	199.92	434.58	22.8089
SCALE	10.48	11.04	28.29	29.94	36.41	39	60.91	74.42	100.41	128.93	519.83	38.9686
CS	3.56	3.78	5.16	5.59	12.54	13.88	30.11	38.83	66.89	80.24	260.58	15.7344
RSh	3.64	4.01	4.99	5.31	13.92	13.99	32.03	40.11	67.01	85.48	270.49	16.3077
RSp	3.61	3.99	5.83	5.7	14.01	14.02	34.01	43.18	63.2	86.01	273.56	16.6695

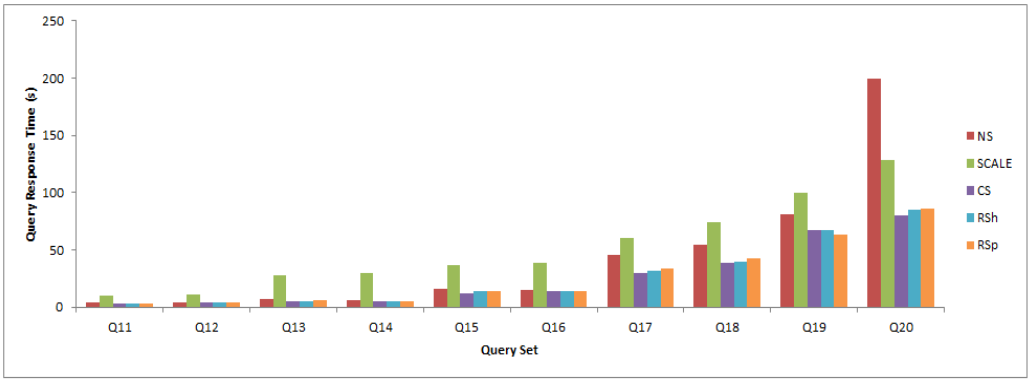


Figure 49: A Summary of query response time (TPC-E)

As a summary, *RSp* and *RSh* shows better performance than other approaches in one of the ten experiments in DBLP, and two of the ten experiments in TPC-E. Although *RSp* and *RSh* do not overwhelm baseline algorithms in the viewpoint of performance of query execution, our approach still has merit if we consider the balance between query processing cost and shortcut building cost.

6.6.5 Space Limitation and Shortcut Maintenance Cost

The last experiment of this chapter is the impact of space limitation and shortcut maintenance cost. As we discussed in Section 6.5 and 6.6, parameters that concern about the selection of candidate shortcut is resource limitation and update frequency φ. Update frequency denotes *"how often the given database has been updated actually?"*. Since our shortcut benefit calculation model reflects the active change of triple database, we predict

that *RSh* and *RSp* will show better result than *CS* when the average value of δ is high, In contrast, there will be little difference between *CS* and both *RSh* and *RSp* when the update weight parameter is given with low value. In fact, *RSh* and *RSp* shows exactly same performance with *CS* when φ=0 (the contents of database never changes).

This section explains the changes of query response time for given space limitation. We set the limitation of generated shortcuts by the scale of estimated size of possible entire shortcuts. In other words, this set of experiments examines the effect of reduced candidate shortcuts to the self-join reducing. We increased the amount of usable space by the 10% of the entire during the join query execution. Note that *SCALE* does not use the scheme of shortcut selection. So we assume that a criterion of vertical partitioning is the amount of participated triples during the process of table partitioning. We assume that the space limitation of *SCALE* is same as the portion of input triples of clustering function. For instance, if we set the space limitation of *SCALE* as 100%, all triples are used for clustering optimization and joined by definition of *SCALE* process. In contrast, only 10% of triples are clustered during the process of clustering of *SCALE* if we set the criterion of space limitation of *SCALE* as 10%. Actually *SCALE* is for the relational database with carefully designed clustering function and access process that use the characteristics of relational database. So there can be a possibility that *SCALE* does not show best performance on the fixed data structure such as triple table. However, we believe that this assumption is fair and square for all compared approach at least checking the effect of resource allocation.

Figure 51 and Figure 52 show the impact of space limitation, denoted

by the query response time of query q_6 *(short join query)* and $q10$ (long join query) on DBLP. We choose four queries for this experiment by the variance of length and database domain. q_6 is the most popular queries on *DBLP* and q_{20} is most complex queries on TPC-E. With these queries, we can obtain the changes of aspect of the influence of space limitation. We can that *RSh*, *RSp* outperforms other approaches with every step of space allocation increase. Besides, we observe that a gap between *CS* and *RSp*, *RSh* on small space limitation is much bigger than large space limitation. Actually there is little difference of query response time between *CS* and *RSp*, *RSh* after the space limitation is increased to 60% or upper.

	100	90	80	70	60	50	40	30	20	10
NS	20.05	20.5	20.61	20.54	21	20.89	20.54	20.51	20.35	20.78
SCALE	7.48	7.91	8.88	10.02	11.04	12.91	14.35	16.54	18.21	20.12
CS	6.81	7.2	7.84	8.43	9.84	11.5	12.81	14.35	15.9	18.79
RSh	6.9	7.21	7.68	8.54	9.64	10.68	11.33	11.88	13.01	15.24
RSp	6.79	7.1	7.7	8.55	9.66	10.24	10.61	11.28	12.05	13.99
OPTIMAL	6.95	7.01	7.15	7.14	7.14	7.8	9.26	11.42	12.58	13

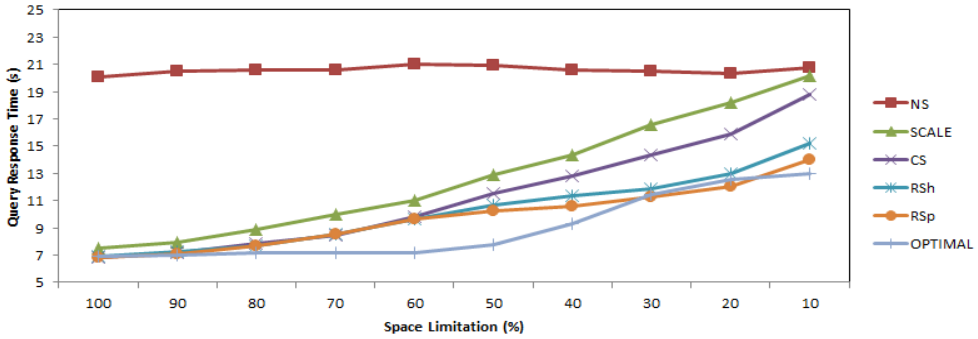


Figure 50: A summary of effect of space limitation (DBLP) (q_6)

	100	90	80	70	60	50	40	30	20	10
NS	78.42	78.01	78.92	79	79.96	78.24	78.13	78.98	79.1	77.62
SCALE	43.8	44.91	45.1	48.12	50.04	57.28	64.11	65.89	72.17	78
CS	23.04	24.01	25.2	30.12	42.35	53.38	59.12	61.78	65.25	67.08
RSh	24.07	24.01	26.46	25.14	29.11	33.98	42.68	52.12	55.26	60.21
RSp	24.01	23.99	26.1	25.1	27.53	34.01	41.86	53	55.27	57.28
OPTIMAL	24.05	24.01	24.78	24.42	25.24	27.01	28.64	35.45	42.18	43.26

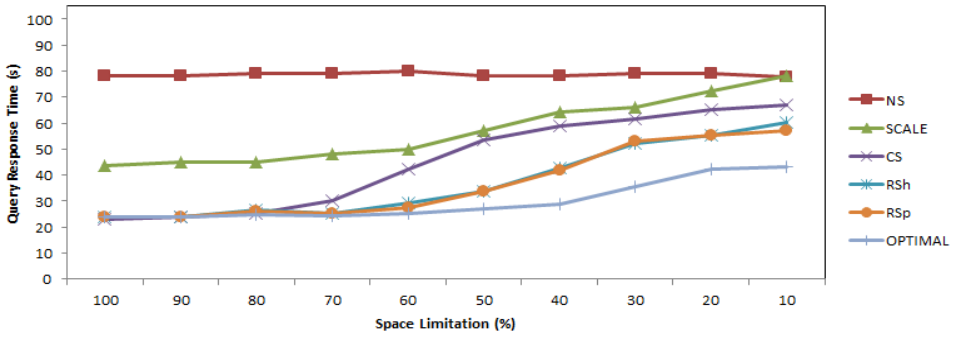


Figure 51: A summary of effect of space limitation (DBLP) (q_{10})

Similarly, Figure 53 and Figure 54 show the performance result of space limitation of our approach and *SCALE*. We can see that the average query response time of *RSh*, *RSp* are much smaller than that of *SCALE*. For lower space limitation, our approach has great advantages than *CS* and *SCALE*. As the possible space allocation is increased, gap between our approach and existing solution is reduced respectably. As a result, our reduced candidate shortcut selection approach not only takes advantages of the candidate shortcut selection on limited resource circumstance, but also be competitive with state-of-the-art join optimization algorithms of relational database.

	100	90	80	70	60	50	40	30	20	10
NS	10.12	10.33	10.06	10.06	10.99	10.31	10.58	10.01	10.59	10.34
SCALE	29.94	29.98	30.77	32.01	33.61	35	35.98	37.01	38.76	39.32
CS	5.59	5.9	6.22	6.72	7.54	8.3	9.2	9.3	9.78	9.98
RSh	5.5	5.4	5.59	5.54	5.67	6.34	7.4	8.57	9.27	9.5
RSp	5.7	5.55	5.6	5.54	5.71	6.41	7.5	8.68	9.35	9.48
OPTIMAL	5.31	5.28	5.35	5.3	5.55	5.6	5.9	6.54	6.95	7.34

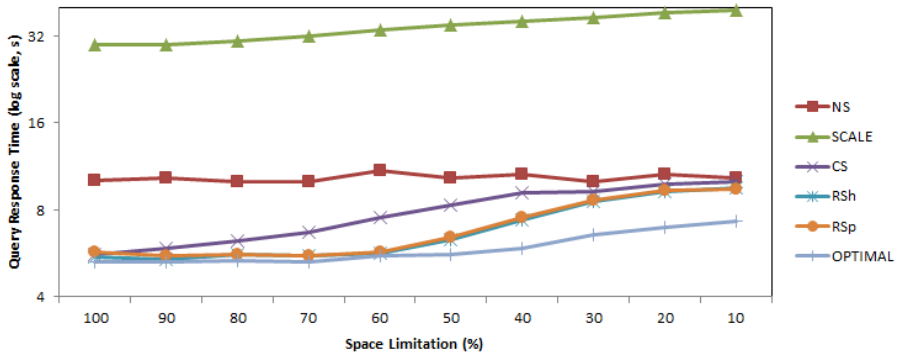


Figure 52: A summary of effect of space limitation (TPC-E) (q_{14})

	100	90	80	70	60	50	40	30	20	10
NS	199.92	199.9	203.17	205.12	200.08	198.4	197.19	201.48	204.13	200.01
SCALE	128.93	135.44	142.22	153.88	158.66	170.06	181.01	190.65	200.12	218.44
CS	83.24	83.11	87.66	90.44	96.24	105.91	115.11	124.69	142.11	184.91
RSh	85.48	86.12	87.88	90.77	92.5	93.01	96.88	103.45	115.12	133.12
RSp	86.01	86	87.02	91.12	92	92.88	95.11	102.55	113.9	128.84
OPTIMAL	84	84.5	84.01	84.82	86.01	85.88	86.5	92.6	104.89	128.33

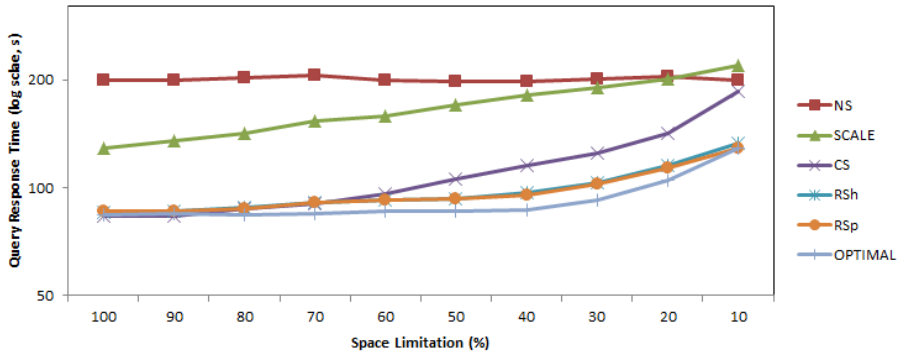


Figure 53: A summary of effect of space limitation (TPC-E) (q_{20})

Figure 55 shows the impact of shortcut maintenance cost. We perform experiments on DBLP dataset with 100% space limitation and randomly distributed update frequency $\varphi \leq 0.03$, since over 97% of triple queries are

SELECT queries without any change of database, as we mentioned ago. When we set the update frequency φ to 0.03, every node that constructs schema graph has 3% of node value update probability during one query execution. Then the benefit of every shortcut that contains updated nodes should be re-calculated by the shortcut benefit function and the shortcut instances should be rebuilt after query execution. So we extend the criteria of update frequency to 5% of total graph node for further changes of database.

It is likely that update is rarely occurred within certain triple database with small schema graph, such as DBLP. More the volume of database is getting larger; more the overhead for database update has been increased extremely since every update of database can cause the recalculation of shortcut benefit including existing shortcuts. In conclusion, *RSp* shows the best result for overhead of database update when the system use high update frequency, because *RSp* gives lower shortcut benefit score to the shortcuts with high update frequency, so that high ranked constructed shortcut by *RSp* are less influenced than *CS* by the database update.

	0	0.3	0.6	0.9	1.2	1.5	1.8	2.1	2.4
CS	0	5.86	20.78	48.72	74.89	96.23	120.3	145.71	177.67
RSp	0	5.9	17.33	35.22	40	56	69.11	83.81	100.01
	2.7	3	3.3	3.6	3.9	4.2	4.5	4.8	5
CS	238.12	306.12	395.91	495.01	622.33	759	899.43	1052.4	1245.2
RSp	114.87	129.04	150.05	180.12	210.9	270.41	335.12	420.91	538.75

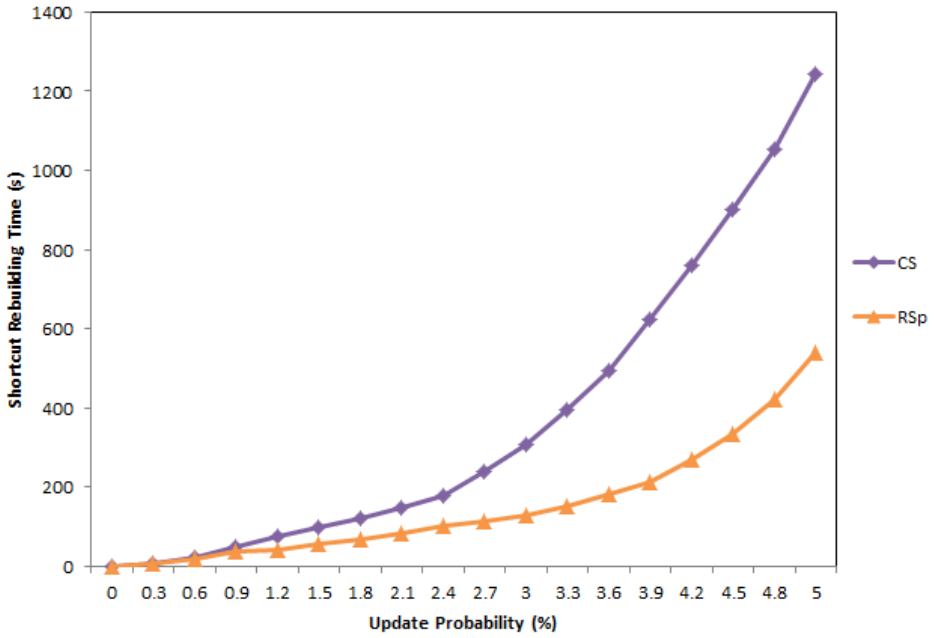


Figure 54: Effect of shortcut maintenance cost

For the practical use of triple database, query optimization is needed to exploit a reasonable performance of triple pattern finding queries, including the process of the enforcement of integrity constraints. Self-joins in triple database is the most challenging issue that decreased the capabilities of triple database. Shortcut selection between distinct nodes is the solution which can eliminate the self-join problem in query execution. In order to reduce the amount of calculation of shortcut benefit, candidate shortcut selection method focus on the "*interesting*" shortcuts that are related with given queries. However, candidate shortcut selection is not perfectly suitable to be used due to the large volume of candidate shortcuts in practical use of triple

database, as well as the lack of consideration of database update. To overcome the limitation, we present a novel technique from candidate shortcut selection based on the utilization of schema information. In our approach, the total amount of considerable shortcuts is reduced enormously by the well-designed pruning process with heuristics and link analysis algorithm such as *PageRank*. In addition, we refined the model of shortcut benefit calculation with shortcut maintenance cost which reflects the impact of database update. Our experimental result shows that reduced candidate shortcut selection provides appropriate set of shortcuts that is useful for eliminating self-joins in triple database. It is expected that our approach is much useful than current approach when the changes of database is occurred actively.

Chapter 7. Conclusion

Triple database in Semantic Web is promising data structure for the integration and sharing of data across different applications. This thesis introduces the fundamental aspects of practical triple database that supports the transformation from relational database to triple data format. Every tuple in relational database is split and reassembled with a set of triple without the loss of information. Not only guaranteeing some level of reliability of database, but also providing adequate features that help to enhance the performance of database is needed for current triple database. The thesis bridges the gap between relational database and triple database by three additional features of triple database. First, we analyze the enforcement of integrity constraints on triple database. The process of enforcement provides certain reliability of triple database. Second, Tridex, a lightweight triple index structure for triple database and integrity constraints was proposed to index triples with less data redundancy. Our triple index structure promises better performance on the data manipulation queries, such as the process of checking integrity constraints due to the lightweight index tree, as well as conventional data retrieval queries. Last, we present a novel technique to select shortcuts for reducing self-joins of triple database. Shortcuts on triple database is pruned carefully and ranked with shortcut benefit calculation model which considers about the update of triple database. In conclusion, three components of the thesis focus on the practical use of triple database commonly. The experiments show the competitive performance in terms of scalability and performance.

Bibliography

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *The VLDB Journal*, 18(2):385-406, 2009.

[2] 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)*, pages 411-422. VLDB Endowment, 2007.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*, pages 496-505. Morgan Kaufmann Publishers Inc., 2000.

[4] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data (SIGMOD '93)*, pages 207-216. ACM, 1993.

[5] M. Arenas, E. Prud'hommeaux, and J. Sequeda. A direct mapping of relational data to RDF. `http://www.w3.org/TR/2010/WD-rdb-direct-mapping-20101118/`. Nov 18, 2010.

[6] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente, An Empirical Study of Real-World SPARQL Queries. In proceedings of the 1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) in the 20th International World Wide Web Conference (WWW2011). 2011.

[7] I. Astrova , N. Korda, and A.Kalja. Rule-based transformation of SQL relational databases to OWL ontologies," In *Proceedings of the 2nd International Conference on Metadata & Semantics Research*. 2007.

[8] I. Astrova, N. Korda, and A. Kalja. Storing OWL ontologies in SQL relational databases. In *proceeding of World Academy of Science, Engineering and Technology*, pages 167-172. Citeseer, 2007

[9] I. Astrova. Rules for mapping SQL relational databases to OWL ontologies. *Metadata and Semantics*, pp. 415-424, 2009.

[10] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller. Triplify: light-weight linked data publication from relational databases. In *Proceedings of the 18th international conference on World wide web (WWW '09)*, pages 621-630. ACM, 2009.

[11] J. Barrasa, Ó. Corcho, and A. Gómez-pérez. R2O, an extensible and semantically based database-to-ontology mapping language. In *Proceedings of the 2nd Workshop on Semantic Web and Databases (SWDB2004)*, pages 1069-1070. Springer, 2004.

[12] R. Bauer, G. de Angelo, D. Delling, D. Wagner. The shortcut problem – complexity and approximation. In *Proceedings of SOFSEM 2009: Theory and Practice of Computer Science, Lecture Notes in Computer Science*, 5404:105-116, 2009.

[13] P. Bera, A. Krasnoperova, and Y. Wand. Using ontology languages for conceptual modeling. *Journal of Database Management*. 21(1):1-28, 2005.

[14] A. Bertails and E. G. Prud'hommeaux. Interpreting relational databases in the RDF domain. In *Proceedings of the 6th international conference on Knowledge capture (K-CAP '11)*, pages 129-136. ACM, 2011.

[15] C. Bizer and A. Seaborne. D2RQ - Treating Non-RDF databases as virtual RDF graphs. In *proceedings of the 3rd International Semantic Web Conference (ISWC2004)*, Poster. 2004.

[16] J. Broekstra, A. Kampman, and F. Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *Proceedings of the 1st International Semantic Web Conference (ISWC 2002)*, pages 54-68. 2002.

[17] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW Alt. '04)*, pages 74-83. ACM, 2004.

[18] F. Cerbah. Learning highly structured semantic repositories from relational databases: the RDBToOnto tool. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications (ESWC'08)*, pages 777-781. Springer-Verlag, 2008.

[19] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *Proceedings of the 31st international conference on Very large data bases (VLDB '05)*, pages 1216-1227. . VLDB Endowment, 2005.

[20] H. -T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th international conference on Very Large Data Bases - Volume 11 (VLDB '85)*, pages 127-141. VLDB Endowment, 1985.

[21] P. Constantopoulos, V. Dritsou, and E. Foustoucos. Developing query patterns. In *Proceedings of the 13th European conference on Research and advanced technology for digital libraries (ECDL'09)*, pages 119-124. Springer-Verlag, 2009.

[22] S. Das, S. Sundara, and R. Cyganiak. R2RML: RDB to RDF mapping language. `http://www.w3.org/TR/r2rml/`. Sep 27, 2012.

[23] V. Dritsou, P. Constantopoulos, A. Deligiannakis, and Y. Kotidis. Optimizing query shortcuts in RDF databases. In *Proceedings of the 8th extended semantic web conference on The semantic web: research and applications - Volume Part II (ESWC'11)*, pages 77-92. Springer-Verlag, 2011.

[24] V. Dritsou, P. Constantopoulos, A. Deligiannakis, and Y. Kotidis. Shortcut selection in RDF databases. In *Proceedings of the 27th International Conference on Data Engineering Workshops(ICDEW)*, pages 194-199. IEEE, 2011.

[25] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11)*, pages 145-156. ACM, 2011.

[26] G. M. Dupont, G. Chalendar, K. Khelif, D. Voitsekhovitch, G. Canet, and S. Brunessaux. Evaluation with the VIRTUOSO platform: an open source platform for information extraction and retrieval evaluation. In *Proceedings of the 2011 workshop on Data infrastructures for supporting information retrie- val evaluation (DESIRE '11)*, pages 13-18. ACM, 2011.

[27] O. Erling and I. Mikhailov. F support in the Virtuoso DBMS. *Studies in Computational Intelligence*, 221:7-24. Springer, 2009.

[28] G. H. L. Fletcher and P. W. Beck. 2009. Scalable indexing of RDF graphs for efficient join processing. In *Proceedings of the 18th ACM conference on Information and knowledge management (CIKM '09)*, pages 1513-1516. ACM, 2009.

[29] A. Harth and S. Decker. Optimized index structures for querying RDF from the Web. In *Proceedings of the Third Latin American Web Congress (LA-WEB '05)*, page 71. IEEE, 2005.

[30] M. Hert, G. Reif, and H. C. Gall. A comparison of RDB-to-RDF mapping languages. In *Proceedings of the 7th International Conference on Semantic Systems (I-Semantics'11)*, pages 24-32. ACM, 2011.

[31] M. Hert, G. Reif, and H. C. Gall. Updating relational data via SPARQL / update. In *Proceedings of the 2010 EDBT/ICDT Workshops (EDBT '10)*, Article 24, 8pages. ACM, 2010.

[32] J. Huang, D. J. Abadi, and K. Ren, Scalable SPARQL Querying of Large RDF Graphs. In Proceedings of the VLDB Endowment, 4(11):1123-1134. 2011.

[33] R. M. Ismail, Maintenance of materialized views over peer-to-peer data ware-house architecture. In Proceedings of International Conference on Computer Engineering & Systems (ICCES), pages 312-318, IEEE, 2011.

[34] V. Jalali, M. Zhou, and Y. Wu. A study of RDB-based RDF data management techniques. In Proceedings of the 12th international conference on Web-age information management (WAIM'11), pages 366-378. Springer-Verlag, 2011.

[35] H. Karloff and M. Mihail. On the complexity of the view-selection problem. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposi-um on Principles of database systems (PODS '99)*, pages 167-173. ACM, 1999.

[36] D. Kim, S-g. Lee, J. Shim, J, Chun, Z. Lee, and H. Park. Practical ontology systems for enterprise application. In *Proceedings of the 10th Asian Computing Science conference on Advances in computer science: data management on the web (ASIAN'05)*, pages 79-89. ACM, 2005

[37] Y. Kotidis. Extending the data warehouse for service provisioning data. *Data and Knowledge Engineering*, 59(3):700-724. 2006.

[38] C. P. de Laborda and S. Conrad. Relational.OWL: a data and schema representation format based on OWL. In *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling (APCCM '05)*, pages 89-96. 2005.

[39] A. Labrinidis, Q. Luo, J. Xu, and W. Xue. Caching and materialization for Web databases. *Journal of Foundations and Trends in Databases*, 2(3):169-266. 2010.

[40] P. -A. Larson and V. Deshpande. A file structure supporting traversal recursion. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data (SIGMOD '89)*, pages 243-252. ACM, 1989.

[41] M. Ley. The DBLP computer science bibliography. `http://www.informatik.uni-trier.de/~ley/db/`. Nov 15, 2012.

[42] L. Ma, C. Wang, J. Lu, F. Cao, Y. Pan, and Y. Yu. Effective and efficient semantic web data management over DB2. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*, pages 1183-1194. ACM, 2008.

[43] I. Mami and Z. Bellahsene. A survey of view selection methods. *ACM SIG MOD Record*, 41(1):20-29. 2012.

[44] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB End*owment*, 1(1):647-659. 2008.

[45] E. Nuutila, Efficient transitive closure computation in large digraphs. *Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series*, 74:124. Finnish Academy of Technology, 1995

[46] Openlink Software. Mapping relational data to RDF with Virtuoso's RDF views. `http://virtuoso.openlinksw.com/whitepapers/relational%20rdf%20views%20mapping.html`. Nov 15, 2012.

[47] L. Page , S. Brin , R. Motwani , T. Winograd. The PageRank citation ranking: bringing order to the Web. In *Proceedings of the 7th International World Wide Web Conference*, pages 161-172. 1998.

[48] J. B. Rodriguez and A. G-Pérez. Upgrading relational legacy data to the semantic web. In *Proceedings of the 15th international conference on World Wide Web (WWW '06)*, pages 1069-1070. ACM, 2006.

[49] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal recursion: a practical approach to supporting recursive applications. *ACM SIGMOD Record*, 15(2):166-176. 1986.

[50] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A data warehouse intelligent cache manager. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96)*, pages 51-62. Morgan Kaufmann Publishers Inc., 1996.

[51] M. Schmidt, T. Hornung, M. Meier, C. Pinkel, and G. Lausent. SP2Bench: a SPARQL performance benchmark. *Semantic Web Information Management*, pages 371-393. Springer, 2010.

[52] Transactional Processing Performance Council. EGen - software package designed for TPC-E. `http://www.tpc.org/tpce/egen-download-request.asp`. Dec 01, 2012.

[53] Transactional Processing Performance Council. TPC Benchmark E (TPC-E). `http://www.tpc.org/tpce/default.asp`. Dec 01, 2012.

[54] W3C, W3C RDF Working Group. `http://www.w3.org/2011/rdf-wg/wiki/Main_Page`. Nov 15, 2012.

[55] W3C. W3C Resource Description Framework. `http://www.w3.org/RDF /`. Nov 15, 2012.

[56] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. In *Proceedings of the VLDB Endowment*, 1(1):1008-1019. 2008.

[57] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proceedings of the first international workshop on Semantic Web and databases,* pages 131-150. 2003

[58] K. Wilkinson. Jena property table implementation. In *Proceedings of the 2nd International Workshop on Scalable Semantic Web Knowledge,* 2006.

[59] L. Xu, S. Lee, and S. Kim. E-R model based RDF data storage in RDB. In *Proceedings of the 3rd IEEE international Conrefence on Computer Science and Information Technology (ICCSIT),* pages 258-262. IEEE, 2010

[60] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficiently querying rdf data in triple stores. In *Proceedings of the 17th international conference on World Wide Web (WWW '08),* pages 1053-1054. ACM, 2008.

국문초록

시맨틱 웹에서의 데이터 양이 증가함에 따라, 대용량의 데이터를 유연한 형식으로 저장하고 시스템 간의 정보 공유를 하는 것은 필수적인 항목이 되었다. 트리플(*Triple*)은 시맨틱 웹에서 사용되는 유연한 데이터 표현 방식의 표준으로, 만약 기업이 보유하고 있는 관계 데이터베이스 기반의 컨텐츠 데이터를 트리플 형식으로 표현할 경우, 그 유연성과 활용성 때문에 다양한 목적으로 시맨틱 웹에서 데이터를 활용할 수 있다. 본 논문에서는 트리플을 기반으로 한 트리플 데이터베이스의 신뢰도를 확보하기 위하여, 트리플 데이터베이스를 기업에서 사용하기 위한 필수적인 요소들을 구현하였다. 첫째, 트리플 데이터베이스를 위한 무결성 제약 조건을 제안한다. 무결성 제약 조건은 관계 데이터베이스로부터 추출된 것으로, 정확하게 동일한 의미를 가지고 트리플 데이터베이스에 적용되도록 해석되었다. 또한 정보의 손실 없이 데이터를 트리플로 바꾸어 저장하는 것뿐만 아니라, 저장된 트리플을 빠른 질의 처리 속도와 더불어 유용하게 사용하는 것도 실용성의 측면에서 중요하다. 그러나 현재까지의 트리플 기반 인덱스 연구들은 트리플이 중복되어 색인되거나, 하나의 트리 안에 너무 많은 색인 키를 저장하는 문제를 가지고 있다. 이러한 문제를 해결하기 위하여 둘째로 트리플 데이터베이스를 위한 새로운 인덱스 구조를 제안한다. 새로운 인덱스 구조는 트리플의 중복을 최소화하는 구조로 설계되었으며, 트리플 구성 요소에 기반하여 인덱스 트리를 분리함으로써 보다 빠르고 가벼운 색인 키 검색을 가능하게 한다. 셋째, 트리플 데이터베이스를 위한 새로운 단축 경로 선택(*Shortcut Selection*) 기법을 제안한다. 단축 경로 선택

기법은 트리플 데이터베이스에서 질의를 수행할 때 가장 많이 발생하는 성능 저하 요인인 자기 조인(*Self-Join*)을 해결하기 위한 방법이다. 일반적으로 한 번의 질의를 위해 트리플 테이블 전체가 조인에 참가할 경우 막대한 질의 비용이 발생하게 된다. 제안하는 새로운 단축 경로 선택 기법은 조인이 발생하는 질의에 대해 미리 시작점으로부터 끝점까지 이어지는 단축 경로에 해당하는 트리플을 우선적으로 추가하여 조인을 사전에 차단하는 기법으로써, 기존 연구에서 고려하고 있지 않은 트리플 그래프 특성에 기반한 단축 경로 우선 차단과 데이터베이스 갱신을 고려한 갱신 빈도(*Update Frequency*) 기반의 이득 계산(*Benefit Calculation*) 모델을 새롭게 설계하였다. 다양한 분야의 데이터를 이용한 질의 시간 측정 등의 실험을 통하여, 본 연구에서 제시한 기법들이 트리플 데이터베이스를 효율적으로 사용하는 데 최신 연구 대비 향상된 성능을 보인다는 것을 검증하였다.

주요어 : 트리플, 데이터베이스, 트리플 데이터베이스, 시맨틱 웹, 무결성 제약, 인덱스 구조, 단축 경로 선택 기법, 질의 최적화

학번 : 2005-21319

감사의 글

돌이켜 보면 지난 8 년간의 시간이 얼마나 빠른 속도로 흘러갔는지 모르겠습니다. 아무 것도 알지 못하던 석사 신입생으로 연구실에 입학하여 선배들과 함께 밤을 새면서 일하던 것이 어제의 일 같은데, 어느새 시간이 흘러 학위 논문을 마무리하는 글을 쓰고 있는 것이 아직도 실감이 나지 않습니다. 가장 먼저, 연구실에 있는 동안 성심으로 보살펴 주시고 아낌없는 가르침을 베풀어 주신 지도교수님 이상구 교수님께 깊은 감사를 드립니다. 석사 입학 때부터 지금까지 바쁘신 가운데서도 지속적으로 연구에 대한 관심을 보여주시고, 갈 길을 잃고 헤매고 있을 때 나아가야 할 방향을 제시해 주신 혜안에 대해 존경을 담아 감사의 인사를 올려 드립니다. 또한 여러 가지 일로 바쁘심에도 불구하고 학위 논문을 위한 연구 기간 내내 귀중한 조언과 지도를 해주신 김형주, 심규석, 김선, 심준호 교수님께도 큰 감사를 드립니다.

긴 시간 동안 함께 웃고 함께 울었던 연구실 식구들에 대한 감사의 말씀을 빼놓을 수 없을 것입니다. 대학원 입학 때부터 든든하게 도움을 주신 익훈 형, 태희 형, 정옥란 박사님께 감사 드립니다. 학부부터 대학원까지 함께 했던 귀중한 동기 진규, 상희, 유천, 종원 형에게도 감사의 인사를 전하고 싶습니다. 연구실 생활의 처음부터 끝까지 동고동락하며 많은 추억을 만들었던 정연 형, 동주형, 재원 형, 상근에게 특별한 감사의 말을 전합니다. 이제 연구실의 최선참 기현, 선배 같이 든든한 후배 재석, 친구 같은 착한 동생 종흠, 착하고 열심히 하는 영기, 아래 생활을 즐겁게 해준 연찬, 그리고 짧게나마 즐거운 인연을 함께 나누었던 동진, 현준, 한빛, 일성, 유현, 태연에게도 감사합니다. 먼저 졸업해서

다방면에서 활약하고 있는 경종 형, 성환 형, 영곤, 주호 형, 철기, 동민, 광현씨, Babar, 인범, 민석, 성은, 상일, 용진, 청림, 병주, 그리고 이름을 미처 말하지 못한 많은 선후배님들께 감사합니다. 아울러 동기를 제외하고 연구실에서 가장 많은 시간을 함께 한 것 같은 명예박사 미연씨, 혜숙씨에게도 고마움을 전합니다.

30 년이 넘는 시간 동안 한결같이 제 편에서 저를 응원해 주신 아버님, 어머님께 진심으로 감사합니다. 아들을 위해 많은 것을 희생하신 두 분이 아니었다면 지금 감히 제가 이 자리에 서 있지도 못할 것입니다. 멀리 미국에서 일하고 있는 승현이에게도 힘들 때마다 옆에서 격려해 주고 도움이 되어 줘서 고맙다는 말을 전합니다. 할아버님과 할머님께서는 제가 힘들 때 기도로 버틸 수 있게 해주신 귀한 버팀목입니다. 지금보다 훨씬 건강히 오래 계셨으면 더 바랄게 없습니다. 이름을 다 말하지 못한 친척분들과 주위 분들께도 감사의 인사를 올립니다.

끝으로, 오늘에 이르기까지 부족함 하나 없이 항상 지켜주시고 가득 채워 주시기만 했던 하나님께 감사합니다. 하나님의 보호 아래 모든 분들의 응원에 힘입어 제가 이 자리까지 올 수 있었습니다. 8 년 전 연구실의 문을 처음 두드릴 때의 두근거리는 마음을 잊지 않고 살아가겠습니다. 모든 분들께 다시 한번 감사 드리며, 저의 소중한 부모님과 가족에게 이 부족한 논문을 바칩니다.