TOWARDS AN INTEGRATED SEMANTIC WEB: INTEROPERABILITY BETWEEN DATA MODELS

by

Sunitha Ramanujam

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Latifur Khan, Co-Chair

Dr. Kevin Hamlen, Co-Chair

Dr. B. Prabhakaran

Dr. Bhavani Thuraisingham

Dedicated to my family.

TOWARDS AN INTEGRATED SEMANTIC WEB:
INTEROPERABILITY BETWEEN DATA MODELS

by

SUNITHA RAMANUJAM, B.E., M.E.Sc, M.S.

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2011

PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the Guide for the Preparation of Masters Theses and Doctoral Dissertations at The University of Texas at Dallas. It must include a comprehensive abstract, a full introduction and literature review and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin and legibility requirements. In such cases, connecting texts which provide logical bridges between difference manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.

ACKNOWLEDGMENTS

Thanks are due to my advisors Dr. Latifur Khan and Dr. Kevin Hamlen for their guidance and support throughout the course of my degree and to Dr. Bhavani for providing academic and financial assistance whenever it was required. I am profoundly grateful and indebted to Dr. Kevin Hamlen for steering me through the theoretical aspects of my work that seemed overwhelming and insurmountable till he stepped in. I would also like to thank Mr. Steven Seida for introducing me to, and assisting me with, my area of research.

The material in this dissertation is based in part upon work supported by the Air Force Office of Scientific Research under Award No. FA-9550-08-1-0260 and we thank Dr. Robert Herklotz for his support. This work was also supported in part by the National Science Foundation under award NSF-0959096, and by Raytheon Company, and we are grateful for the same.

I would like to express my gratitude to Dr. Gupta for bringing to light an important theoretical angle that was crucial to my work, and to my friend and colleague Meera Sridhar for first introducing me, a novice, to programming language concepts and LaTeX, both of which were vital to my work. Thanks are also due to my committee member, Dr. Prabhakaran, for his encouragement.

My heartfelt gratitude goes to my primary pillars of support during my long academic journey - my family. I am grateful to my parents who, despite their age, made numerous exhausting trips to the U.S. in order to help make my job of juggling family, home, and school a little easier; my husband Sriram, who did a marvelous job of keeping up the word he gave me 16 years ago when he promised to support any educational endeavors that I may embark on; our daughter, Deeksha, for playing the roles of baby-sitter and general helper umpteen times without complaint; our son and resident comedian, Vishnu, for making me crack up even during my most stressful moments; and our toddler, Vineeth, for simply being his endearing self and bringing so much joy into my life.

Lastly, I would like to thank Rhonda Walls for saving me many a trip with all her help, and my friends, particularly Khairunnisa Hassanali, for being there for me in every way.

November 2011

TOWARDS AN INTEGRATED SEMANTIC WEB:

INTEROPERABILITYBETWEEN DATA MODELS

Publication No. ____________________

Sunitha Ramanujam, Ph.D.
The University of Texas at Dallas, 2011

Supervising Professors: Dr. Latifur Khan, Co-Chair
Dr. Kevin Hamlen, Co-Chair

The astronomical growth of the World Wide Web resulted in the emergence of data representation methodologies and standards such as the Resource Description Framework (RDF) that aim to enable rapid and automated access to data. The widespread deployment of RDF resulted in the emergence of a new data model paradigm, the RDF Graph Model. This, in turn, spawned an associated demand for RDF graph data modeling and visualization tools that ease the burden of data management off the administrators. However, while there is a large selection of such tools available for more established data models such as the relational data model, the assortment of tools for RDF stores are fewer in comparison as the RDF paradigm is a more recent development. This dissertation presents R2D (RDF-to-Database), a relational wrapper for RDF Data Stores, which aims to transform, at run-time, semi-structured RDF data into an equivalent, domain-specific, virtual relational schema, thereby bridging the gap between RDF and RDBMS concepts and making existing relational tools available to RDF Stores. R2D's transformation process involves two components - a schema mapping component that uses a declarative mapping language to achieve its objective, and a query transformation component that translates SQL queries into equivalent SPARQL queries. A semantics-preserving transformation methodology for R2D's components is presented with proofs that establish the fact that an SQL query, $\hat{sql}$, run over the translated relational schema, $\mathbb{R}$, obtained from an RDF Graph, $\mathbb{G}$, through R2D's schema mapping process, returns the same result that an equivalent SPARQL query, $\dot{spq}$, obtained by translating $\hat{sql}$ using R2D's query translation process, would when run on the original RDF graph, $\mathbb{G}$. Additionally the dissertation also presents D2RQ++, an enhancement over an existing read-only relational schema-to-RDF mapping tool, D2RQ, that presents legacy data stored in relational databases as virtual RDF graphs. D2RQ++ enables bi-directional data flow by providing data manipulation facilities that permit triples to be inserted, updated, and/or deleted into appropriate tables in the underlying relational database. The primary R2D and D2RQ++ functionalities, high-level system architectures, performance graphs, and screen-shots that serve as evidence of the feasibility of our work are presented along with a semantic-preserving version of R2D's components and the relevant proofs of semantics-preservation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1
INTRODUCTION

The unleashing of the Internet has resulted in a plethora of information sources becoming available (Imai and Yukita 2003), making today's world increasingly networked and progressively more reliant on electronic sources of data. The need to augment human reasoning and decision making abilities has resulted in the emergence of an evolutionary stage of the World Wide Web, namely, the Semantic Web.

1.1 Current Trend in the Internet Community The Semantic Web and RDF

The Semantic Web is envisioned to facilitate the automated storage, exchange, and usage of machine-readable information interspersed throughout the web (Singh, Iyer, and Salam 2005). Conceived by Tim Berners-Lee, the Semantic Web is essentially a Web of Data that extends existing Web documents by adding meta-data to them, thereby presenting web data in a manner that is understood by computers. The goal of the Semantic Web initiative is to address the deficiencies of HTML documents which are unable to separate the presentation details from the information contained in the HTML pages, thus making it impossible for software agents to identify, isolate, extract, and process relevant information. The Semantic Web initiative attempts to eliminate these deficiencies by including machine-interpretable descriptions of the data and concepts interspersed throughout the web thereby enabling software agents to establish links between data and integrate data from disparate sources. In other words, Semantic Web technologies are to data what HTML is to documents; i.e., just as HTML enables online documents to generate the illusion of one voluminous book, Semantic Web technologies help create the illusion of one huge database from all the disparate data sources in the world (Berners-Lee 1999). The core components of the Semantic Web are:

- A data model - Resource Description Framework (Manola and Miller 2004; Klyne and Carroll 2004; Hayes 2004; Grant Clark 2005)
- Data interchange formats - RDF/XML, N-Triples, N3 (Beckett 2004)
- Notations for formal description of ontologies or vocabularies - RDF Schema (Brickley and Guha 2004), Web Ontology Language (McGuinness and Harmelen 2004)

Semantic Web technologies would be extremely useful in areas where concepts have multiple names in different countries. One example of such an area is the life sciences where medicines and illnesses are known by various names depending on the geographical location where they are found. Semantic web technologies could see past the different nomenclatures and aggregate relevant information appropriately without ambiguities due to naming differences.
In order to realize the semantic web initiative various standards, specifications, and notations are being developed to provide a formal description of concepts, terms, and relationships within a given knowledge domain. Some of these include the Resource Description Framework (Manola and Miller 2004), the RDF Vocabulary Description Language (Brickley and Guha 2004), and the Web Ontology Language (Lacy 2005; McGuinness and Harmelen 2004) notations. RDF, which is the current buzzword in the Semantic Web Community, is the foundation for the Semantic Web and the focus of the research presented in this dissertation. The RDF standard is proposed by the World Wide Web consortium for encoding knowledge with the express purpose of changing the web from being a platform for distributed presentations to one for distributed knowledge (Tauberer 2006). There are two types of entities in RDF - resources and literals. RDF provides the ability to attach properties to a resource and associate values (which can, in turn, be resources or literals themselves) with these properties. RDF does not limit its applicability to web resources alone; it can also be used to encode information and relations between real-world entities/resources such as people, places of interest, abstract concepts, etc. RDF records information in the form of

triples, each consisting of a subject, a predicate, and an object. The subject is the resource that is of interest. The object is a resource or a literal whose interpretation depends on the predicate. The predicate is typically a verb and denotes the relationship that exists between the subject and the object. RDF's suitability to unstructured and semi-structured data that is typically available on the web, and the simplicity and flexibility offered by RDF data models have resulted in increasing demand for data stores that use the RDF Graph model and offer the ability to store and query RDF data (Muys 2007).

1.2 Challenges due to the Current Trend

1.2.1 Challenge 1

The Semantic Web initiative and its associated technologies brought to the fore-front two distinct challenges. On the one hand, the growing number of RDF stores have, as with any data store with massive amounts of information, spawned an associated requirement for tools and technologies for the management and visualization of this data. However, most of the current data modeling, data visualization, data management, and business intelligence tools that are available in the market today are still based on the more mature and efficient models such as relational and tabular models (Teswanich and Chittayasothorn 2007). The tools available for RDF data are fewer and less mature than the selection for relational database management systems (RDBMS). While efforts are ongoing to develop new tools for this purpose, alternate research efforts are underway that focus on integrating benefits and features available in existing methodologies with the advantages offered by newer technologies. These alternative efforts, which involve establishing a mapping from RDF data into equivalent relational schemas, will be particularly beneficial to small and medium-sized organizations that are typically resource constrained and that may not have the ability or inclination to take risks associated with investing in fledgling technologies such as RDF and the tools for the same (Hendler 2006). Further, relational databases have been around for several decades more than semantic web technologies, giving them the advantage of time to improve their efficiency, reliability, and performance (Borodaenko 2009), to refine their tools and methodologies, and of being widely accepted and supported by a variety of applications (Korotkiy and Top 2004; Krishna 2006). For the same reasons, skilled personnel experienced in relational methodologies are available in greater numbers than RDF experts. In order to avoid the learning curves associated with new tools and continue to leverage the advantages offered by the traditionally-oriented tools without losing out on the benefits offered by the newer web technologies and standards, the gap between the two needs to be bridged by establishing a means to represent RDF data as relational schema.

1.2.2 Challenge 2

On the other hand, as a direct consequence of relational technologies being around for decades and being the primary backend support system of Information Technology (Borodaenko 2009; Choi, Moon, Baik, Wie, and Park 2010), enormous amounts of enterprise data that are used in all walks of life exist in relational database management systems (Lv and Ma 2008; Zhou, Chen, Zhang, and Zhou 2008; Zhou 2010). However, despite the extensive adoption of relational databases, there do exist contexts where the relational model is not as good a fit as other models. For example, to derive customer-centered marketing strategies aimed at increasing profit margins and improving market shares out of relational data, extensive data mining operations involving large amounts of resources are required. Having additional data analysis abilities such as inferencing would help increase the value of enterprise data. At the same time, in order to make the vision of a ubiquitous Semantic Web a reality, and for the Semantic Web Initiative to be truly successful and widely adopted, there has to be a way for Semantic Web technologies to access the vast amounts of relational data (Zhou 2009). These requirements bring us to the second challenge - representing relational database content as equivalent RDF graphs. There are several research efforts currently in existence that do achieve this translation from relational database schemas to equivalent RDF graphs; however, almost all efforts are read-only in nature. In other words, while the data from underlying relational databases can be queried from the corresponding RDF interfaces, other Data Manipulation Language (DML) operations such as inserts, updates, and deletes cannot be performed on the relational databases through the RDF interfaces. In order for the relational-to-RDF translation to be truly effective it is important for the data flow to be bi-directional, i.e., to have data not only leave the relational database in the form of SELECT queries, but to also have data enter the relational database in the form of DML operations. The current body of work in the bi-directional arena is minimal and hence needs to be addressed to maximize the usefulness of the translation and make the time, effort, and resources expended on the translation process worthwhile.

1.3 Addressing the Challenges

1.3.1 Addressing Challenge 1

The motivation behind our research, with regards to the first challenge, is to arrive at a solution to the bridging problem without the need to create an actual physical relational schema and duplicate data and we propose one such solution. Our approach, called R2D (RDF-to-Database) (Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009e; Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009c; Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009a; Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009b), is a bridge that hopes to enable existing traditional tools to work seamlessly with RDF Stores without having to make extensive modifications or waste valuable resources by replicating data unnecessarily. Two approaches are adopted in the realization of R2D's objectives. The first approach is a formal, semantics-preserving approach to R2D's translation mechanism that includes two components as listed below.

1. Schema Mapping: RDF-to-Relational Schema mapping that preserves the meaning of schemas and produces an equivalent, domain-specific relational schema corresponding to a given RDF store.

2. Query Translation: SQL-to-SPARQL query translation that preserves the meaning of queries and produces an equivalent SPARQL query for every input SQL query.

R2D's semantics-preserving translation ensures that an SQL query, $\hat{sql}$, run over the translated relational schema, $\mathbb{R}$, obtained from an RDF Graph, $\mathbb{G}$, through R2D's schema mapping process, returns the same result that an equivalent SPARQL query, $\dot{spq}$, obtained by translating $\hat{sql}$ using R2D's query translation process, would when run on the original RDF graph, $\mathbb{G}$.

The second approach to realizing R2D's objective is a practical, implementation oriented approach where R2D is designed as a JDBC wrapper around RDF stores that provides a relational interface to data stored in the form of RDF triples. In this approach, the RDF Store is explored and mapped to a relational schema at run-time and end-users of visualization tools are presented with the normalized relational version of the store on which they can perform query operations as they would on an actual physical relational database schema. In a nutshell, the implementation of R2D consists of three modules the details of which are summarized below.

1. RDFMapFileGenerator: Automatic RDF-to-Relational Schema mapping file generator utility.

2. DBSchemaGenerator: Parser that takes the above map file as input and generates a domain-specific, virtual relational schema for the corresponding RDF store.

3. SQL-to-SPARQL Translation: Utility that takes an SQL statement as input, parses and converts it to a corresponding SPARQL statement, executes the same, and returns the results in a tabular format.

R2D also includes support for the relationalization of provenance information stored in RDF stores using the concept of reification (Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009d; Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2010). Reification is an important RDF concept that provides the ability to make assertions about statements represented by RDF triples. With the increasing number of online resources and sources of information that become available each day, the need to authenticate the available sources becomes essential in order to be able to judge the validity, reliability, and trustworthiness of the information (Almendra and Schwabe 2006). This authentication is facilitated by augmenting the sources with provenance information, i.e., information describing the origin, derivation, history, custody, or context of a physical or electronic object (Buneman, Chapman, and Cheney 2006). RDF Reification, a means of validating a statement/triple based on the trust level of another statement (Powers 2003), is the solution offered by the WWW consortium for users of RDF stores to record provenance information. Thus, RDF reification is a key component of any application requiring stringent records of the basis/foundation behind every piece of information in the data store. In particular, reification plays a critical role in security-intensive applications where it is imperative to maintain the privacy and ownership of sensitive data. The provenance information captured using reification can be used, in such applications, to monitor and control data access.

1.3.2 Addressing Challenge 2

As a means to address the second challenge, we present D2RQ++ (Ramanujam, Khadilkar, Khan, Seida, Kantarcioglu, and Thuraisingham 2010; Ramanujam, Khadilkar, Khan, Kantarcioglu, Thuraisingham, and Seida 2010) , an enhancement to an existing, extensively adopted relational-to-RDF read-only translation tool called D2RQ. D2RQ++ includes the ability to propagate data changes specified in the form of RDF triples back to the underlying relational database.

When triples cannot explicitly be translated into equivalent concepts within the underlying relational database schema, D2RQ++ continues to adhere to the Open-World Assumption by permitting those triples to be housed in a separate native RDF store. When information on a particular entity is requested, the output returned is a union of the data pertaining to the entity from the relational database as well as any triples that have the entity as the subject and that may exist in the native RDF store. Thus, RDF triples submitted for insertion/update/ deletion are never rejected due to mismatches with the underlying relational schema, thereby maintaining the Open-World Assumption of the Semantic Web world while still being able to work with technologies such as RDBMSs which are based on the Closed-World Assumption.

1.4 Research Contributions

1.4.1 Contributions towards Challenge 1

The objectives and contributions of the portion of our research that addresses the first challenge described in Section 1.2 are as follows.

- We present a semantics-preserving translation mechanism for R2D that preserves the meanings of schemas and queries and rigorously proves the validity of R2D's schema mapping and query transformation process through appropriate theorems and lemmas.

- We propose a mapping scheme for the translation of RDF Graph structures to an equivalent normalized relational schema. The proposed mapping schema includes several constructs and rules to handle a variety of blank nodes (Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009a; Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009b) as well as constructs to handle provenance data stored using the RDF concept of reification (Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009d; Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2010).

- Based on the RDF-to-RDBMS map file created, we propose a transformation process that presents a non-generic, domain-specific, virtual relational schema view of the given RDF store and any reification information included in the same.

- We propose a mechanism to transform any relational SQL queries issued against the virtual relational schema into the SPARQL equivalent, and return the triples data to the end-user in a relational format. The proposed mechanism includes string matching procedures and aggregation facilities.

- The proposed framework imposes no restrictions on the nature of RDF triples or their storage mechanisms as it is a purely virtual layer that does not involve duplication of the RDF data. Hence, data updates are immediately visible through R2D without explicit synchronization activities.

- We provide all of the above in the form of a JDBC interface that can be plugged into existing visualization tools and we present the feasibility of our algorithms and processes through experiments conducted using the LUBM Benchmark (Guo, Pan, and Heflin 2005) data set as well as a real-life crime dataset, and an open source visualization tool, RDF store, and relational database.

1.4.2 Contributions towards Challenge 2

The contributions of the part of our research that addresses, and proposes a solution to, the second challenge discussed in Section 1.2 are as follows:

- We present algorithms to translate RDF update triples into equivalent relational attributes/tuples thereby enabling DML operations on the underlying relational database schema.

- We propose extensions to the existing D2RQ Mapping Language in order to support translation of blank node and reification node structures into equivalent relational tuples.

- With every DML operation we ensure preservation of the Open-World Assumption by maintaining a separate native RDF store to house triples that violate integrity constraints or that are mismatched with the underlying relational database schema.

- We incorporate the above algorithms and extensions into D2RQ++, an enhanced version of the highly popular D2RQ open-source relational-to-RDF mapping tool, thereby enabling bi-directional data flow between an RDF interface and its underlying relational database.

- We also provide D2R++-Server, an enhanced version of the Graphical User Interface D2R-Server, through which end-users can now specify DML requests that need to be propagated back into the underlying relational database.

1.5 Organization of the Dissertation

The organization of the dissertation is as follows. A brief overview of other related research efforts currently underway in the data interoperability arena, in either direction (RDF-to-Relational-Database and Relational-Database-to-RDF), is provided in the following chapter while R2D mapping preliminaries in terms of the modus operandi, mapping constructs, and types of relationships handled are given in chapter 3. Chapter 4 presents detailed descriptions of the various algorithms involved in the mapping process for regular RDF triples and is supported by chapter 5 which details the implementation specifics of the proposed system with sample visualization screenshots and performance graphs for the map file generation process with and without various sampling methods and for a diverse range of queries on databases of various sizes. Chapters 6 and 7 discuss the algorithmic extensions and implementation specifics, respectively, that facilitate the relationalization of provenance data, i.e., RDF reification data. A semantics-preserving approach to R2D's translation mechanism is discussed in Chapter 8 with the associated mathematical definitions and proofs included in Appendices A, B, and C. Bi-directional data transfer between relational databases and RDF stores is presented in the form of D2RQ++ and its associated GUI tool, D2R++-Server, in Chapter 9. Lastly, chapter 10 discusses the future directions for our research and concludes the dissertation.

CHAPTER 2
LITERATURE REVIEW

With RDF being the current buzzword in the Semantic Web community, research efforts are underway in various aspects of RDF such as RDF-ising relational and legacy database systems, and spreadsheet data, transforming traditional SQL queries into RDF query languages such as RDQL (Seaborne 2004), SPARQL (Prud'hommeaux and Seaborne 2008; Harris and Seaborne 2011), and SPARQ2L (Anyanwu, Maduko, and Sheth 2007), and optimizing performance of queries issued against RDF data sources. However, the overall concept and objectives of R2D are unique since almost all research efforts attempt to integrate relational database concepts and Semantic Web concepts from a perspective that is opposite to that considered in our work. Relevant research efforts from both perspectives (Relational-to-RDF and vice versa) are presented in the next few subsections.

2.1 RDF-to-Relational Translation

The RDF vocabulary description language (Brickley and Guha 2004) is often compared to the object oriented system as both paradigms include concepts of classes and properties or attributes. However, while OO systems define classes in terms of the attributes that instances of the classes may have, the RDF vocabulary is property-centric and defines properties in terms of the resource classes to which they apply. Using this property-centric approach additional properties can be defined for existing classes without having to redefine the class definitions (Brickley and Guha 2004). Due to the similarities between the RDF and OO paradigms it is natural to assume that the techniques used to tranform OO models into relational database schema elements (Blaha, Premerlani, and Shen 1994; Orenstein and D.N.Kamber 1995; Narasimhan, Navathe, and Jayaraman 1994) can be applied to the problem of transforming RDF graphs into relational database schemas. However, this is not the case in reality because the RDF graph model does not comprise only of classes and properties like its OO counterpart. RDF graphs also contain more complex structures such as blank nodes and reification node structures for which an appropriate relational transformation process cannot be adapted from existing OO-to-RDF mapping efforts since such a process does not exist. Object Oriented technology is not constrained by normalization concepts such as multi-valued attributes or many-to-many relationships (Narasimhan, Navathe, and Jayaraman 1994) and, hence, transformation of OO models into relational schemas does not have to consider these concepts. However, RDF graphs, on the other had, can, and in most scenarios, do contain complex relationships that have to be tranformed into appropriate normalized tables in an underlying relational database schema. For all these reasons, adapting existing OO-to-relational schema translation techniques for RDF-to-relational-schema transformation is not sufficient.

While the inadequacy of applying OO-to relational-schema transformation techniques resulted in new research efforts aimed at studying and achieving transformations from RDF graphs to relational database schemas and vice versa, the amount of work in this (RDF-to-Relational) direction (Xu, Lee, and Kim 2010; Teswanich and Chittayasothorn 2007; Pan and Heflin 2003; Borodaenko 2009) is far more limited when compared to the other (Relational-to-RDF) direction. In (Xu, Lee, and Kim 2010) the authors extract and transform the schema elements of an RDF graph into an equivalent Entity-Relationship diagram from which an actual physical relational database schema is then derived. The motivation behind the work in (Xu, Lee, and Kim 2010) is to provide a domain-specific RDFS storage strategy and the reason for the authors' choice of relational databases to achieve this goal is to capitalize on the sophisticated query processing capabilities, optimization techniques, and other facilities such as concurrency control and data recovery options that are offered by RDBMSs. By adopting this method, the authors in (Xu, Lee, and Kim 2010) are able to by-pass the need to query RDF data using the less mature SPARQL query language. Another research whose objectives are very closely aligned with R2D is the RDF2RDB project

(Teswanich and Chittayasothorn 2007). Like in R2D, the authors in (Xu, Lee, and Kim 2010; Teswanich and Chittayasothorn 2007) attempt to arrive at a domain-specific, meaningful relational schema equivalent for an RDF store but the similarity ends there. RDF2RDB, like (Xu, Lee, and Kim 2010) and most of the other transformation efforts described below, involves data replication with the triples data being dumped into a relational schema, and therefore is subject to synchronization and space issues (Jiang, Ju, and Xu 2009). Moreover, in order to successfully map the RDF data into an equivalent relational schema, (Xu, Lee, and Kim 2010) and RDF2RDB requires the presence of ontological information in the form of schema definitions such as rdfs:class and rdf:property. R2D, on the other hand, can arrive at mapping information with or without explicit ontology information. In the absence of RDF Schema definitions, R2D discovers the mapping through extensive examination of the triple patterns and the relationships between resources.

Furthermore, the relational mapping in (Teswanich and Chittayasothorn 2007) involves the creation of a table for each property in the RDF graph regardless of the cardinality of the relationship represented by the property. As a result, the resulting schema may not be truly normalized and may contain more tables than necessary due to the presence of properties representing 1:N or N:1 types of relationships. R2D avoids these unnecessary tables by taking such conditions into consideration. The authors in (Teswanich and Chittayasothorn 2007; Xu, Lee, and Kim 2010) also do not discuss the details of how blank nodes are handled by their research, if at all, while R2D is capable of wading through a variety of blank nodes and arriving at meaningful transformations of the same.

The Hybrid model presented in (Pan and Heflin 2003) is another mapping methodology that is similar to (Teswanich and Chittayasothorn 2007) in terms of relational schema generation. This methodology generates a table for every property in the ontology, and, hence, results in unnecessary tables in the case of 1:N relationships between subject and object resources. This is avoided in R2D where 1:N relationships are handled through the addition of a foreign key column to the table on the N-side of the relationship. The hybrid model also fails on RDF graphs which do not include schema/structure information while R2D is able to glean structural information even in the absence of these ontological constructs (by examination of instance data).

The Samizdat RDF Store (Borodaenko 2009) is yet another effort that attempts to translate RDF data into an equivalent domain-specific relational schema. For those RDF triples that are not translatable into an equivalent relational entity, Samizdat provides the commonly adopted Triples table to house such triples. Samizdat uses database triggers to reduce the impact of RDFS/OWL inference on query performance and access to the RDF data in the relational schema is provided in the form of a user interface through which SQUISH (Miller, Seaborne, and Reggiori 2002) queries can be issued. Samizdat involves actual physical data loading into the relational database and, hence, is also plagued by data duplication and synchronizations issues discussed above. Further, Samizdat does not provide any support for the SPARQL query language that is now considered at the de-facto language for Semantic Web applications.

2.2 Relational-to-RDF Translation

The D2RQ project (Bizer, Cyganiak, Garbers, and Maresch ; Bizer and Cyganiak 2007; Bizer and Seaborne 2004; Bizer 2003), an extensively adopted open source project, and one that our work in addressing the first challenge described in Section 1.2 is very closely modeled on in terms of mapping language constructs, is another significant player in the RDBMS-RDF mapping arena. D2RQ facilitates the querying of a non-RDF database using SPARQL, enables access of information in a non-RDF database using Jena (McBride 2002) or Sesame (Broekstra, Kampman, and Harmelen 2002) APIs, and enables non-RDF database content access as Linked Data (Berners-Lee 2006; Heath and Bizer 2011) over the Web. Three main components comprise the D2RQ platform the D2RQ mapping language, the D2RQ Engine, and the D2R Server (Bizer and Cyganiak 2006). The D2RQ mapping language is a declarative language that expresses mappings between an RDF schemata and a relational schema. The D2RQ Engine is the component that uses the mappings, generated using the

D2RQ mapping language, to translate semantic web toolkits API calls to SQL queries issued against the underlying relational database and return the obtained results to the higher layers in the D2RQ architecture. The D2R Server is an HTTP server that publishes the contents of a relational database as linked data on the Web. The goals of D2RQ are the exact reverse of the goals of our research pertaining to challenge 1 in Section 1.2. While they attempt to create a mapping from a relational database to an RDF Graph, and transform RDF queries into corresponding SQL queries, thereby making relational data accessible through RDF applications, our Challenge 1 goal is to enable RDF triples to be accessed through relational applications. Hence, despite the concept of mapping files and query conversions being common between D2RQ and R2D, each of the two researches addresses very different needs.

The work in (Erling and Mikhailov 2009; Blakeley 2007), Virtuoso RDF Views, is yet another effort that, like D2RQ, also uses a declarative meta schema consisting of quad map patterns that define the mapping of SQL data to RDF ontologies. Like D2RQ, the objective of the Virtuoso RDF Views project is to present existing relational data, without any actual physical data duplication, as virtual RDF graphs that can be queried directly using RDF query languages such as SPARQL. Two key technologies comprise the heart of Virtuoso RDF Views - RDF Meta-Schema and a declarative Meta-Schema Language for mapping SQL data into RDF concepts. The most significant components of the RDF Meta-Schema are quad map patterns, IRI classes, and literal classes. Quad map patterns are used to define the transformation of relational columns into triples in a SPARQL graph pattern and each quad map pattern consists of four quad map values (corresponding to graph, subject, predicate, object). IRI classes are used to construct subject IRIs for each primary key column value (and, indirectly, for each foreign key column value as well) in the relational database - they define how key values are combined into an IRI string and vice versa. Literal classes define how one or more non-primary-key-and-non-foreign-key columns get converted into a literal object. RDF views are defined by combining the above elements to declare a collection of quad patterns through which the relational database is mapped.

RDF123 (Han, Finin, Parr, Sachs, and Joshi 2008), an open source translation tool, also uses a mapping concept, however its domain is spreadsheet data and it attempts to overcome the limitations of current spreadsheet-to-RDF mapping tools that typically map every spreadsheet row to an instance and every column to a property thereby leading to a star-shaped RDF graph. RDF123 aims to achieve richer spreadsheet-to-RDF translation by allowing the users to define mappings between the spreadsheet semantics and RDF graphs. The RDF123 Architecture consists of two key components - the RDF123 Application and the RDF123 Web Service. The RDF123 Application provides the users with an interactive graphical interface for creating a map graph that expresses the relationships between the rows and fields in their excel worksheets and for storing the created map graph in an RDF syntax. The RDF123 Web Service translates online spreadsheets to RDF and hosts the URIs of the RDF documents that arise from these online spreadsheets.

Light-weight efforts at publishing RDF triples as linked data from relational databases are discussed in (Auer, Dietzold, Lehmann, Hellmann, and Aumueller 2009; Chen and Yao 2010). Triplify (Auer, Dietzold, Lehmann, Hellmann, and Aumueller 2009) achieves this by extending SQL and using the extended version as a mapping language. The objective behind the Triplify initiative is to eliminate the high entrance barrier for publishing database content as RDF by neither defining nor requiring the usage of new extensive mapping languages. Triplify's primary domain of applicability is web applications that are built using the scripting language PHP, and MySQL and it is based on mapping HTTP-URI requests onto relational database queries. The data resulting from the relational queries are transformed into RDF statements by Triplify and the transformed data is published on the Web as Linked Data (or using other RDF serializations such as N3, N-Triples, etc.). The View-Based Triplify method in (Chen and Yao 2010) achieves the RDBMS-to-RDF transformation using a set of simple mapping rules based on which traditional relational views, each of which describes a distinct RDF class (and includes all the corresponding properties), are created. Users can then issue SPARQL queries against the resulting linked data. These SPARQL queries are converted into equivalent SQL queries using a translation algorithm proposed by the authors and the appropriate data is returned to the users in linked data format. (Ling and Zhou 2010) is another mapping effort that uses a set of well-defined rules, like (Chen and Yao 2010), to map relational schema metadata into an equivalent RDFS ontology which is then written into an RDF/XML file. Instance data is generated on demand from the underlying relational database based on the mapping correspondences established.

Several endeavors (Ismail, Yaacob, and Kareem 2008; Wang, Miao, Zhang, and Zhou 2009; Cheong, Chatwin, and Young 2009; Myroshnichenko and Murphy 2009; Wang, Lu, Zhang, Miao, and Zhou 2009) are underway in the data integration arena as well that aim to present a semantically unified RDF model derived from multiple underlying heterogeneous databases. (Ismail, Yaacob, and Kareem 2008) is a research effort that uses the mapping technique to integrate heterogeneous databases with the objective of providing a homogeneous read-only view of data in the underlying databases. A data-source describing method using SPARQL graph patterns is proposed by the authors in (Wang, Miao, Zhang, and Zhou 2009; Wang, Lu, Zhang, Miao, and Zhou 2009) to specify the semantic mapping between the required RDF ontology and the underlying relational schema. They also address the problem of query reformulation from the domain ontology to the underlying relational databases by including a query rewriting algorithm that generates semantically correct SQL query execution plans corresponding to the issued SPARQL queries. In (Cheong, Chatwin, and Young 2009) the authors present an RDF-based Semantic Schema Transformation System (RSSMTS) where they adopt a Localized Data Integration approach to combining information from multiple heterogeneous databases. The RSSMTS framework consists of five components RDF Mapping Files, which map the database schema metadata to corresponding concepts in WordNet; WordNet database, which is used to identify semantic relations such as hypernyms, synonyms, etc. of a concept; Translator Module, which, by using the mapping files and the WordNet database, is responsible for identifying semantically related information from the various underlying databases; Query Engine, which takes care of query reformulation and execution; and Staging Area, where the results from the various underlying databases are stored, consolidated, and returned to the end-user. In (Myroshnichenko and Murphy 2009), the authors present a set of mapping rules that enable a well-formed Entity-Relationship schema to be semantically mapped to an equivalent OWL-Lite schema. The reason for the authors choice of OWL-Lite over OWL-DL and OWL-Full is the fact that OWL-Lite is computationally guaranteed. Through their mapping methodology, the authors in (Myroshnichenko and Murphy 2009) hope to simplify and speed up the data integration process in heterogeneous databases.

Other mapping efforts in the reverse direction include the work presented in (de Laborda and Conrad 2006; An, Borgida, and Mylopoulos 2004; An, Borgida, and Mylopoulos 2006). In (de Laborda and Conrad 2006) the authors use relational.OWL to extract the semantics of a relational database, automatically transform them into a machine-readable and understandable RDF or OWL ontology, and use RDQuery (de Laborda, M.Zloch, and Conrad 2006) to translate SPARQL queries to SQL. The authors in (An, Borgida, and Mylopoulos 2004; An, Borgida, and Mylopoulos 2006) also essentially perform a relational-to-ontology mapping but here, they expect to be given some target ontology and some simple correspondences between the atomic relational schema elements and the concepts in the ontology to begin the mapping process with. 3Store (Harris 2005) is an implementation where the model includes non-application-specific tables such as triples, symbols, datatypes, etc. Using this model, it would be impossible for the user to determine the problem domain addressed by the model or to infer the schema by identifying the entities, the attributes, and any relationships that exists between any of them. R2D offers the users the ability to do just this and enables them to actually arrive at a complete Entity-Relationship Diagram using the RDF-to-Relational Schema transformation process.

2.3 Bi-directional Relational-to-RDF Translation

Each of the efforts presented in the previous sub-section is uni-directional as all of them just allow a read-only view of the relational database with nothing coming back into the same. ONTOACCESS (Hert, Reif, and Gall 2010) is the only effort we have been able to identify that attempts bi-directionality. In ONTOACCESS, the authors define a new mapping language called R3M which is very similar to the D2RQ mapping language (Bizer 2003), and they include support for the SPARQL/Update language (Seaborne et al. 2008) for data manipulation. D2RQ++ (Ramanujam, Khadilkar, Khan, Seida, Kantarcioglu, and Thuraisingham 2010; Ramanujam, Khadilkar, Khan, Kantarcioglu, Thuraisingham, and Seida 2010), on the other hand, avoids learning curves associated with new languages by reusing, and extending when required, D2RQ's mapping language. By reusing D2RQ's mapping language, D2RQ++ also eliminates the effort and resources associated with creation of new languages. Another primary difference between ONTOACCESS and our approach, i.e., D2RQ++, is support for the Open-World Assumption. While ONTOACCESS accepts only those updates/inserts that have an equivalent relational concept in the underlying database, D2RQ++ can work with mismatched data as well (as described in the previous section), which is a key requirement of RDF's Open-World Assumption, thus proving itself to be an authentic Semantic Web application.

Yet another difference between ONTOACCESS and D2RQ++ is the ability to accommodate updates/deletes of blank node structures. Blank Nodes are used to represent complex relationships between entities and are an integral component of the RDF specification. ONTOACCESS makes no mention of how incoming blank node structures are handled while D2RQ++ is capable, as can be seen from Chapter 9, of translating a variety of blank nodes, into equivalent relational structures thereby enabling the blank node contents to be transmitted to the underlying relational schema.

As can be seen from the discussions, none of the existing research efforts, except one, address the issue of enabling bi-directional data transfer between relational and RDF applications. ONTOACCESS is the only research that comes close to the objectives of D2RQ++ but it, too, has certain drawbacks as described above.

2.4 SPARQL-to-SQL Translation

The query processing component of R2D which comprises the SQL-to-SPARQL transformation process, once again, has no comparable counterpart while many efforts are underway in the other direction. In (Chebotko, Lu, Jamil, and Fotouhi 2006), the authors propose an algorithm to translate SPARQL queries with arbitrary complex optional patterns to an equivalent SQL statement to be fired against a single relational table called Triples(subject, predicate, object) that stores the RDF triples. The authors achieve their objective using two algorithms. The first algorithm is BGPtoSQL that translates a basic graph pattern into an equivalent SQL pattern in a way that ensures that the SQL query retrieves, from the triples store, the RDF sub-graph matching the basic graph pattern. The second algorithm is SPARQLtoSQL which uses BGPtoSQL to translate each basic graph pattern in the query to an equivalent SQL query and join the resulting relations from each SQL query into one relation under SPARQL semantics.

The authors in (Chen, Wu, Wang, and Mao 2006) discuss a methodology that supports integration of heterogeneous relational databases using the RDF model. Given a set of semantic mappings between relational schemas and RDF ontology, the goal in (Chen, Wu, Wang, and Mao 2006) is to effectively answer RDF queries by rewriting them into a set of equivalent source SQL queries. They include this idea in their extended work, DartGrid (Wu, Chen, Wang, Wang, Mao, Tang, and Zhou 2006), which uses a visual mapping tool to manually align an existing relational database to an existing ontology. In (Yan, Wang, Zhou, Qian, Ma, and Pan 2008), the authors partition the RDF graph data by adding an extra column to the triples table to store sub-graph information with the objective of reducing join costs and improving query performance. An SQL-based RDF Querying Scheme is presented in (Chong, Das, Eadon, and Srinivasan 2005) where the RDF querying capability is made a part of the SQL, however, the RDF data is, once again, stored as a collection of triples in a single database table. The motivations behind the research in (Chong, Das, Eadon, and Srinivasan 2005) were to eliminate the shortcomings in current approaches for efficient and scalable querying of RDF data which include inefficiency and difficulty in integrating with SQL queries used in database applications. The authors attempt to avoid these problems through the introduction of RDF_MATCH table function which provides the ability to search for an arbitrary graph pattern against the RDF data, the ability to perform inferencing based on RDFS rules, and the ability to include a collection of user-defined rules as an optional data source. The RDF_MATCH function takes four arguments - the graph pattern to be matched, specified by a collection of one or more triple patterns; a list of RDF models; rulebases, if any (optional); and namespace aliases. The result returned by RDF_MATCH is a table of rows which contain values for the variables used in the graph patterns.

As can be seen from the discussions above, none of the research efforts address the issue of enabling relational applications to access RDF data without data replication and, hence, to the best of our knowledge, R2D is the first endeavor to address this issue.

CHAPTER 3
R2D ARCHITECTURE and MAPPING PRELIMINARIES

In this chapter, we describe the system architecture and mapping language comprising the R2D framework. This work was published in ***International Journal of Semantic Computing*** 2009 and ***Electronic Commerce Research Journal*** 2010, both of which were co-authored by Anubha Gupta, Latifur Khan, and Bhavani Thuraisingham from University of Texas at Dallas (UTD) and Steven Seida from Raytheon Company. As stated earlier, the principal goal of this research is to ensure seamless availability of RDF data to existing tools, in particular, data visualization tools, that are equipped to work with relational or tabular data. The architecture of the proposed system and the deployment sequence are illustrated in Figure 3.1.

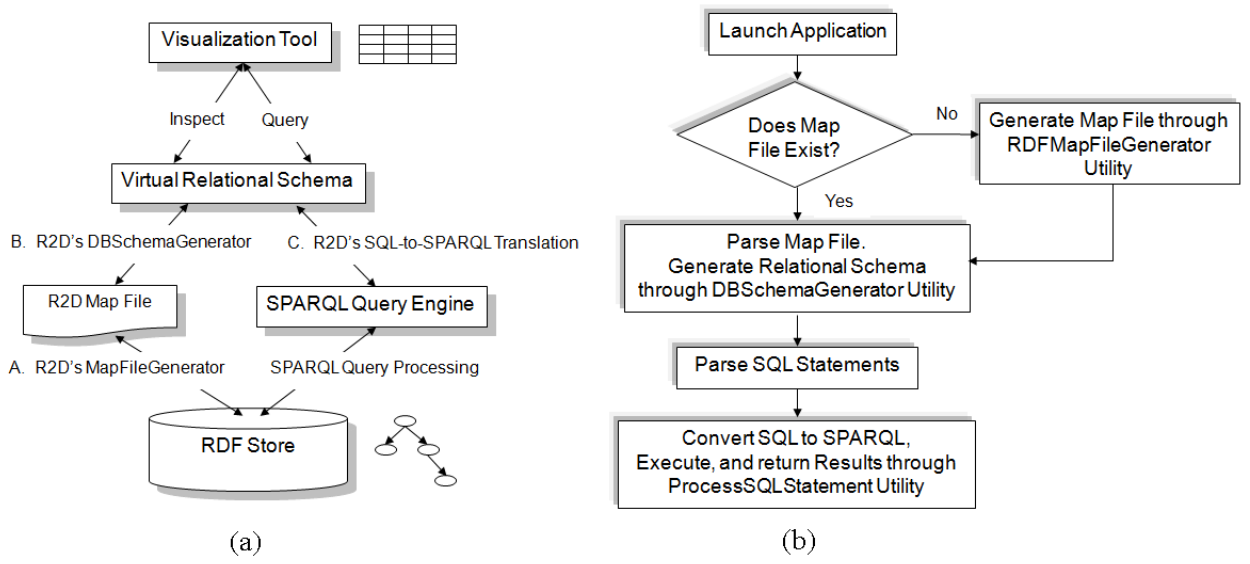

Visualization Tool
Inspect
Query
Virtual Relational Schema
B. R2D's DBSchemaGenerator
C. R2D's SQL-to-SPARQL Translation
R2D Map File
SPARQL Query Engine
A. R2D's MapFileGenerator
SPARQL Query Processing
RDF Store
(a)
Launch Application
Does Map File Exist?
No
Generate Map File through RDFMapFileGenerator Utility
Yes
Parse Map File. Generate Relational Schema through DBSchemaGenerator Utility
Parse SQL Statements
Convert SQL to SPARQL, Execute, and return Results through ProcessSQLStatement Utility
(b)

Figure 3.1: (a) R2D System Architecture; and (b) Deployment Sequence

The RDF Store at the bottom of Figure 3.1(a) is examined by the RDFMapFileGenerator Algorithm (Item A in Figure 3.1(a)) and an RDF-to-RelationalSchema mapping file is generated, if it does not already exist, by the algorithm using the constructs discussed in Section 3.1. The DBSchemaGenerator Algorithm (Item B in Figure 3.1(a)) takes this mapping file as input and presents to the relational visualization tool a domain-specific, virtual relational schema corresponding to the RDF store. Users of the visualization tool can choose to issue SQL queries against the virtual relational schema to access the RDF data. At this point R2D's SQL-to-SPARQL Translation Algorithm (Item C in Figure 3.1(a)) performs the necessary query translations, invokes the SPARQL query engine, and returns the results to the visualization tool in a tabular format.
At the heart of the transformation of RDF Graphs to virtual relational database schemas is the R2D mapping language which is a declarative language that expresses the mappings between RDF constructs and relational database schema constructs. The constructs of the R2D mapping language used for the relationalization of regular RDF data and RDF reification data are presented in the next two subsections.

3.1 Mapping Constructs used for Regular RDF Triples

In order to better explain the constructs comprising the R2D mapping language, examples from the sample scenario in Figure 3.2, based on the LUBM dataset (Guo, Pan, and Heflin 2005), are included where applicable.

r2d:TableMap: The r2d:TableMap construct refers to a table in a relational database. In most cases, each rdfs:class object will map to a distinct r2d:TableMap, and, in the absence of rdfs:class objects, the r2d:TableMaps are inferred from the instance data in the RDF Store. ***Example:*** *The RDF Graph in Figure 3.2 results in the creation of a TableMap called "Student.*

The mapping constructs specific to an r2d:TableMap are as follows.

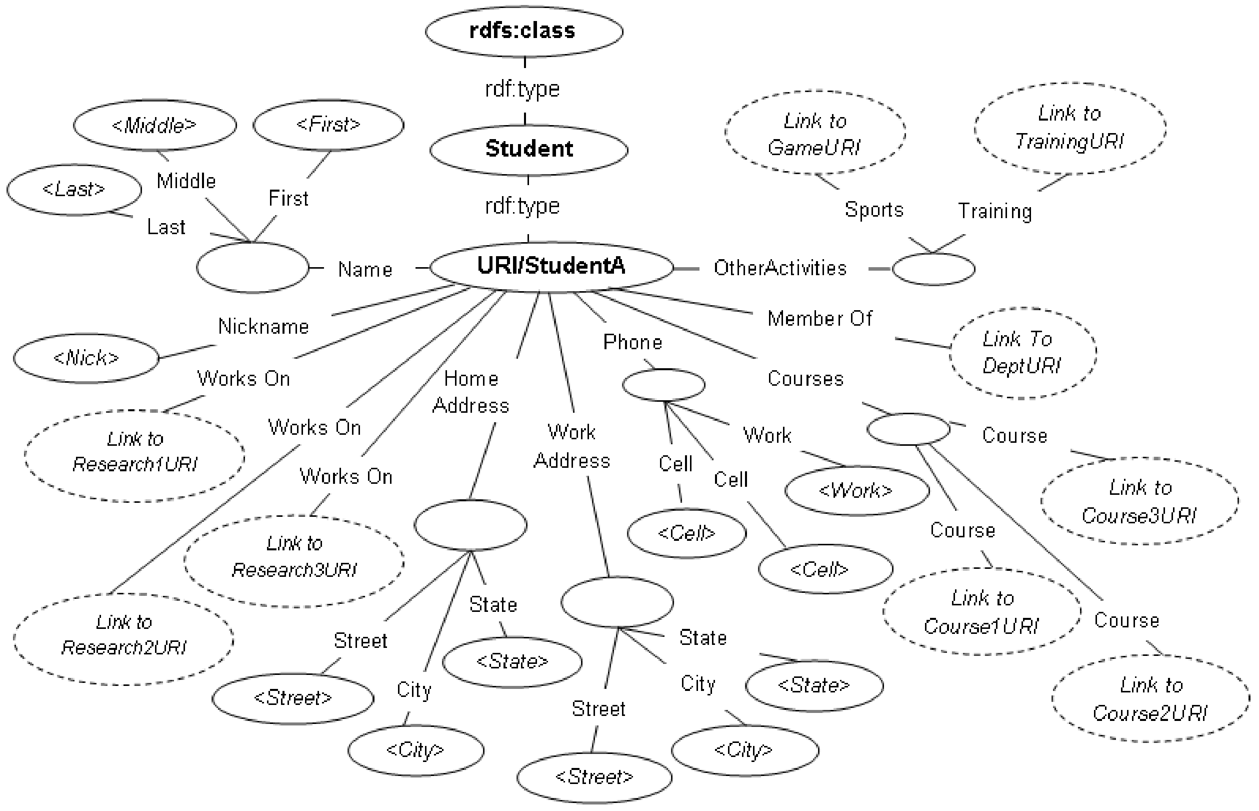


Figure 3.2: Sample Scenario Based on LUBM Schema

r2d:keyField: The r2d:keyField construct specifies the primary key attribute for the r2d:TableMap to which the field is attached. The data value associated with the field specified by r2d:keyField is the object of the"rdf:type" predicate belonging to the rdfs:class subject corresponding to its r2d:TableMap. ***Example:*** *An r2d:keyField (primary key) called "Student_PK" field is attached to the "Student" TableMap and one of its values, corresponding to the sample scenario in Figure 3.2, is "URI/StudentA".*

r2d:ColumnBridge: r2d:ColumnBridges relate single-valued RDF Graph predicates/properties to relational database columns. Each rdf:Property object maps to a distinct column attached to the table specified in the rdfs:domain predicate. In the absence of rdf:property/domain information, they are discovered by exploration of the RDF Store data. ***Example:*** *The "Nickname" and "Member Of" predicates in Figure 3.2 become r2d:ColumnBridges belonging to the "Student" r2d:TableMap.*

r2d:MultiValuedColumnBridge(MVCB): Those RDF Graph predicates that have multiple object values for the same subject are mapped using the MVCB construct. MVCBs typically correspond to RDF constructs such as RDF containers (rdf:Bag, rdf:Alt, rdf:Seq) and RDF collections and are used to indicate 1:N/N:1 and N:M relationships between the virtual relational schema tables. ***Example:*** *The "Works On" predicate in Figure 3.2 is an example of an MVCB mapping.*

r2d:SimpleLiteralBlankNode (SLBN): SLBNs help relate RDF Graph blank nodes that consist purely of distinct simple literal objects to relational database columns. ***Example:*** *The object of the "Name" predicate in Figure 3.2 is an example of an SLBN which has distinct literal predicates of "First", "Middle", and "Last", which are, in turn, translated into columns of the same names in the "Student" r2d:TableMap.*

r2d:MultiValuedSimpleLiteralBlankNode (MVSLBN): This construct maps duplicate SLBNs and, while the processing of the predicates is identical to the (SingleValued) SLBN, this construct results in the generation of a separate r2d:TableMap with a foreign key relationships to the table representing the subject resource of the blank node. In the event the predicates leading to the blank nodes are distinct, an r2d:MultiValuedPredicate (MVP) is created and a "TYPE" column corresponding to the MVP is included in the r2d:TableMap. ***Example:*** *The objects of the "HomeAddress" and the "WorkAddress" predicates in Figure 3.2 together form a MVSLBN.*

r2d:ComplexLiteralBlankNode (CLBN): This construct refers to blank nodes in the RDF Graph that have multiple literal object values for the same subject and the predicate concept associated with the blank node. An r2d:ComplexLiteralBlankNode typically results in the generation of a separate r2d:TableMap with a foreign key relationship to the table representing the subject resource of the blank node. ***Example:*** *The object of the "Phone" predicate in Figure 3.2 is an example of a CLBN that has multiple object (⟨Cell⟩) values for the subject (URI/StudentA) and a predicate (Cell) concept associated with the blank node. The relational transformation for "Phone" involves the generation of an r2d:TableMap of the same name. This "Phone" r2d:TableMap includes as columns a*

"Type" field that holds the values of the predicates off of the MVBN (in our sample scenario, the "Type" field will hold a value of "Cell" and "Work"), and a "Value" field that holds the object values of the predicates off of the MVBN. Additionally, the r2d:TableMap also includes, as foreign key, the "Student_PK" column which references the primary key of the "Student" r2d:TableMap.

r2d:MultiValuedComplexLiteralBlankNode (MVCLBN): This construct maps duplicate complex literal blank nodes and the processing of the predicates is identical to the (SingleValued) CLBN case except in the event the predicates leading to the blank nodes are distinct, in which case an r2d:MultiValuedPredicate (MVP) is created and a "TYPE" column corresponding to the MVP is included in the r2d:TableMap. ***Example:*** *Consider a scenario where the "Phone" predicate in Figure 3.2 is replaced with two similar predicates, "PastPhNums" and "CurrentPhNums", each of which are CLBNs. The objects of these two predicates together form an MVCLBN.*

r2d:SimpleResourceBlankNode (SRBN): This construct helps map blank nodes that have multiple predicates leading to resource objects belonging to the same object class. SRBNs typically identify 1:N/N:1 or N:M relationships between the subject resource and the object resource classes. RDF containers that represent collections of similar resource objects are represented using the SRBN construct. ***Example:*** *The object of the "Courses" predicate in Figure 3.2 is an example of a SRBN that has multiple resource objects that are instances of the"Course" class/r2d:TableMap.*

r2d:ComplexResourceBlankNode (CRBN): CRBNs represent blank nodes that have distinct or non-distinct predicates leading to objects belonging to different object classes. This construct also identifies 1:N/N:1 or N:M relationships between the subject resource class and each of the object classes and typically result in the creation of as many join tables as the number of distinct object classes leading off of the CRBN. RDF containers that represent collections of different types of object resources are represented using CRBNs. ***Example:*** *The object of the "OtherActivities" predicate is an example of a CRBN that has* *multiple resource objects each of which is an instance of a different (one "Sports" and one "Training") class.*

r2d:MultiValuedSimple/ComplexResourceBlankNode (MVSRBN and MVCRBN): Duplicate simple/complex resource blank nodes are represented using the MVSRBN and MVCRBN constructs respectively. Like other MultiValued constructs, the processing for these is also identical to their SingleValued counterparts except in the event the predicates leading to the blank nodes are distinct, in which case an r2d:MultiValued-Predicate (MVP) is created and a "TYPE" column corresponding to the MVP is included in the r2d:TableMap. ***Example:*** *Consider a scenario where the "Courses" predicate in Figure 3.2 is replaced with multiple predicates each representing the courses taken in a particular year, such as "2007Courses", "2008Courses", and "2009Courses", each of which are SRBNs. The objects of these predicates together form an MVSRBN.*

r2d:MixedBlankNode: Blank Nodes consisting of a mixture of literal, resource, and other blank node objects are mapped using the r2d:MixedBlankNode construct. This construct results in the creation of a r2d:TableMap which contains as fields every literal or resource leaf node object that is an element of the tree rooted at the r2d:MixedBlankNode.

The mapping constructs specific to single-valued and multi-valued column bridges and blank nodes are described below.

r2d:belongsToTableMap(BTTM): This construct connects a r2d:ColumnBridge or MVCB to an r2d:TableMap. Every r2d:ColumnBridge must specify a value for either this construct or the r2d:belongsToBlankNode construct. ***Example:*** *The "Nickname" predicate in Figure 3.2 is associated with the resource "URI/StudentA", an instance of the "Student" r2d:TableMap. Hence, the BTTM construct corresponding to "Nickname" r2d:ColumnBridge is set to a value of "Student", thereby connecting the ColumnBridge to a table.*

r2d:belongsToBlankNode (BTBN): This construct ties a r2d:ColumnBridge or MVCB to a blank node. ***Example:*** *The "FirstName" r2d:ColumnBridge corresponding to the "First" predicate in Figure 3.2 is associated with the "Name" SVBN. Hence, for the*

"FirstName" r2d:ColumnBridge the BTBN construct is used to associate it to the "Name" blank node.

r2d:refersToTableMap (RTTM): This construct is optional for column bridges and is only used for those triples that contain a resource object for a predicate. This construct is used to generate primary key-foreign key relationships within the virtual relational schema. ***Example:*** *The object of the "Member Of" predicate in Figure 3.2 is a resource that translates to another r2d:TableMap called "Department". Hence the "MemberOf" r2d:ColumnBridge includes the RTTM construct with a value of "Department".*

r2d:predicate: The r2d:predicate construct is used to store the fully qualified property name of the predicate which corresponds to the column bridge. This information is used during the SQL-to-SPARQL translation to generate the SPARQL WHERE clauses required to obtain the value of the r2d:ColumnBridge

r2d:MultiValuedPredicate (MVP): The MVP construct is used in scenarios where there are multiple predicate names that refer to the same overall object type despite each individual object having a different value. r2d:MultiValuedPredicates are also used to keep track of the predicates associated with RDF containers and RDF collections. MVPs typically result in the creation of a "TYPE" column in the r2d:TableMap corresponding to the resource associated with the MVP. ***Example:*** *The predicates off of the "Phone" CLBN in Figure 3.2 are examples of a MVP called "Phone_Type" that represents the fact that multiple predicates (⟨Cell⟩, ⟨Work⟩) refer to the same overall object type (i.e., a string representing phone number).*

r2d:datatype: This construct specifies the datatype of its column bridge and is derived from the rdfs:range predicate or, in its absence, by examination of the object values of the predicate.

The virtual relational schema generated by R2D for the sample scenario in Figure 3.2 is illustrated in Figure 3.3 and the schema generation details (for regular RDF triples) are elaborated on in Chapter 4.

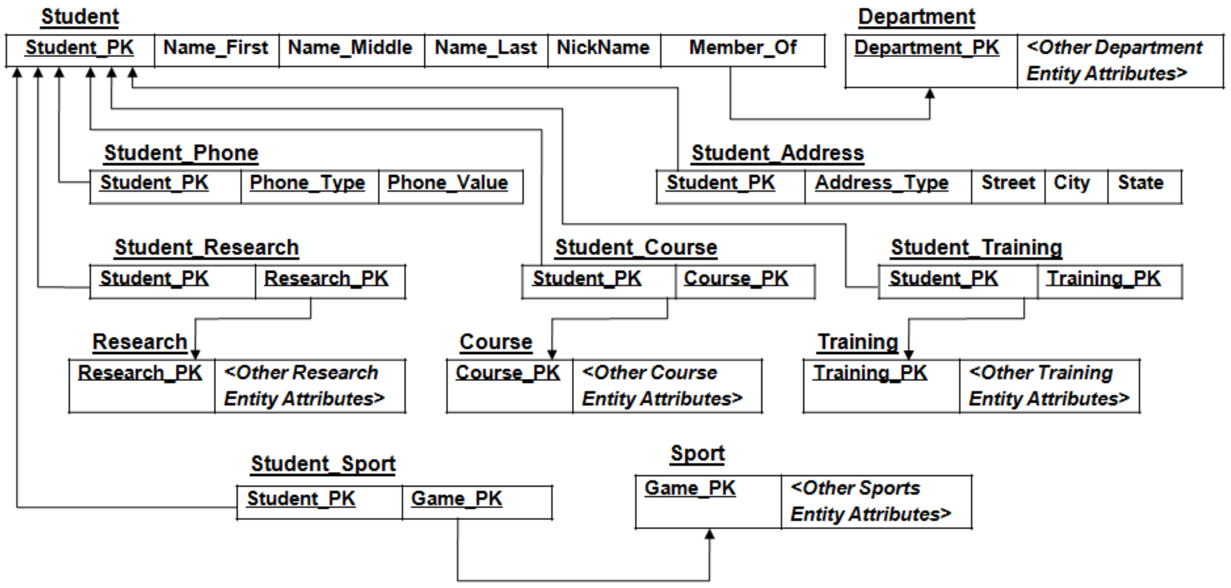


Figure 3.3: Equivalent Virtual Relational Schema generated by R2D for Figure 3.2

3.2 Mapping Constructs used for RDF Reification Data

In order to better understand the constructs comprising the reification component of the R2D mapping language, let us consider the sample scenario illustrated in Figure 3.4. In Figure 3.4's graph, every solid node with outgoing edges, such as *OffenceURI*, represents a subject/resource. Edges, such as *Address*, *Description*, and *Victim*, represent predicates and the solid nodes at the end of the edges, such as ⟨*Street*⟩, ⟨*Description*⟩, and ⟨*Victim*⟩, represent objects. Empty solid nodes, such as the nodes at which the *Address* and *ReportingOfficer* predicates terminate represent blank nodes. The nodes in dashed lines with the "s", "p", "o", and "t" predicates, amongst others, represent reified nodes. "s", "p", "o", and "t" represent the "rdf:subject", "rdf:predicate", "rdf:object", and the "rdf:type" predicates of the reification quad.

Other predicates of the reification nodes (other than "s", "p", "o", and "t" predicates) represent non-quad predicates (NQPs). The rest of the empty nodes in dashed lines (other than the reified nodes) that are the objects of non-quad reification predicates, such as the objects of the *Phone*, *Dependents*, and *OfficerAddress* predicates, represent reification blank

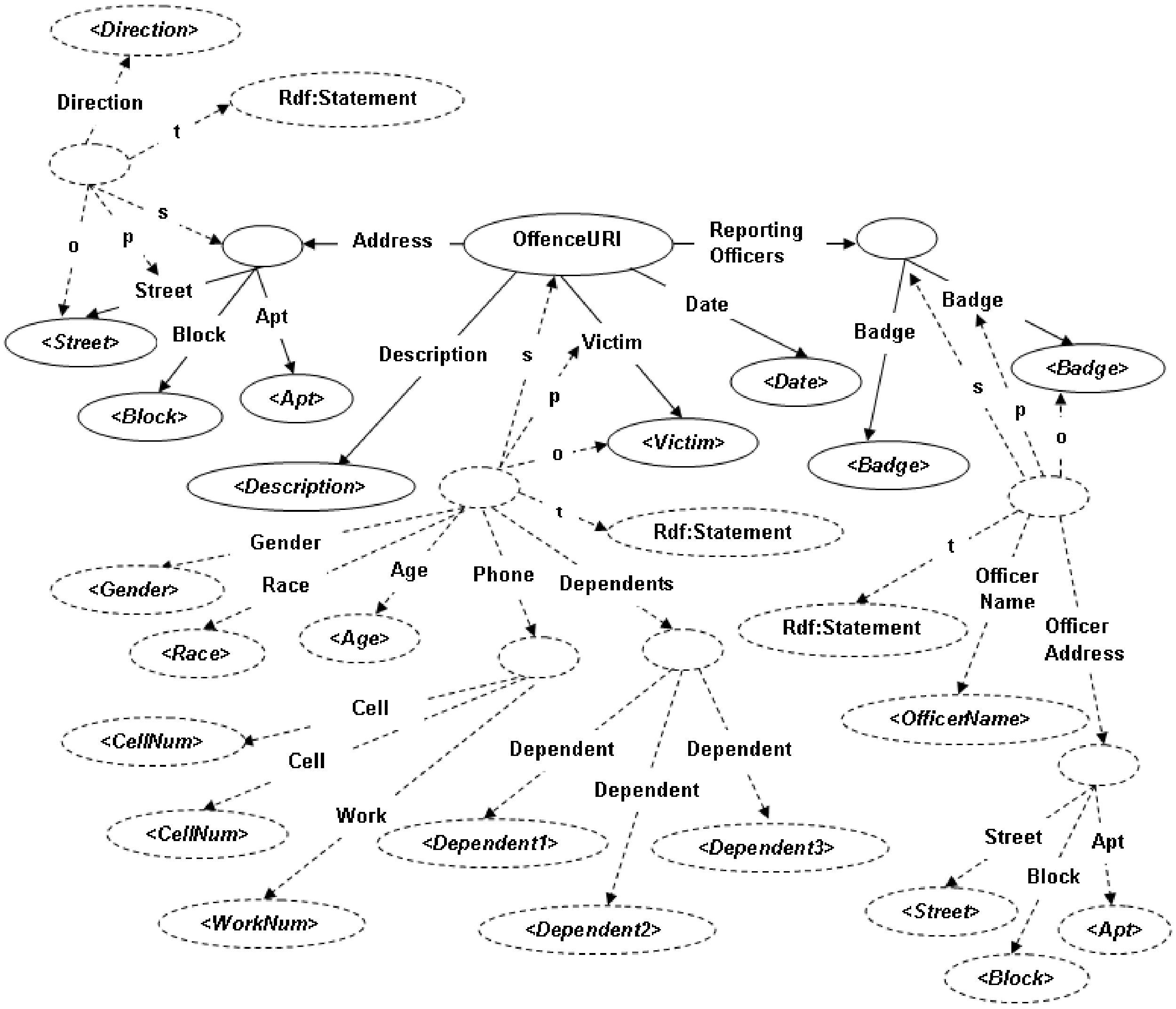


Figure 3.4: Sample Scenario involving Crime Data

nodes. The non-quad reification properties chosen in this example may not represent actual provenance information. They were primarily chosen to illustrate proof of concept. Elements of Figure 3.4 are used, wherever applicable, to facilitate better comprehension of the reification-specific mapping constructs which are discussed in the remainder of the section.

r2d:ReificationNode: The r2d:ReificationNode construct is used to map blank nodes associated with "reification quads". Under certain scenarios an r2d:ReificationNode results in the generation of a new "reification" r2d:TableMap. These scenarios are discussed in detail in Chapter 6. ***Example:*** The non-solid nodes corresponding to the *Address-Street* predicate, the *Victim* predicate, and the *ReportingOfficers-Badge* predicate in Figure 3.4 are examples of r2d:ReificationNodes named *Address_Street_Reif*, *Victim_Reif*, and *ReportingOfficers_Badge_Reif* respectively.

The mapping constructs specific to r2d:ReificationNodes are discussed next.

r2d:BelongsToTableMap: This constructs connects an r2d:ReificationNode to the r2d:TableMap corresponding to the resource associated with "rdf:subject" of the r2d:-ReificationNode. This information is recorded in the R2D Map File for use during the SQL-to-SPARQL translation. ***Example:*** *OffenceURI* is the value of the *rdf:subject* predicate of the *Victim_Reif* r2d:ReificationNode. The r2d:TableMap corresponding to *OffenceURI* is *Offence*. Hence, the r2d:BelongsToTableMap construct corresponding to *Victim_Reif* is set to a value of *Offence*, thereby connecting the reification node to a relational table.

r2d:BelongsToBlankNode: This construct connects an r2d:ReificationNode to the r2d:[Simple/Complex][Literal/Resource]BlankNode corresponding to the blank node associated with the "rdf:subject" of the r2d:ReificationNode. ***Example:*** The *rdf:subject* of the *Address_Street_Reif* reification node in Figure 3.4 consists of a blank node resource called *Address*, which is an r2d:SimpleLiteralBlankNode. Hence, for this reification node the r2d:BelongsToBlankNode construct is used to associate *Address_Street_Reif* to the *Address* blank node.

NOTE: Since the *rdf:subject* of a reification node can either refer to a proper resource or a blank node, the r2d:BelongsToTableMap and r2d:BelongsToBlankNode constructs are mutually exclusive. These are primarily required to enable the generation of appropriate SPARQL WHERE clauses during SQL-to-SPARQL translation.

r2d:ReifiedPredicate: This construct is used to record the predicate corresponding to the *"rdf:predicate"* property of the reification quad mapped by the r2d:ReificationNode construct. This information is, again, required for appropriate SPARQL query generation. ***Example:*** The complete URI of the *Victim* predicate of *OffenceURI* is recorded under the *Victim_Reif* reification node using the r2d:ReifiedPredicate construct.

Predicates of r2d:ReificationNodes are mapped using the r2d:ColumnBridge construct described earlier in this section. Some of the important mapping constructs specific to r2d:ColumnBridges include:

r2d:BelongsToReificationNode: This construct connects an r2d:ColumnBridge to an r2d:ReificationNode entity and is a mandatory component of r2d:ColumnBridges belonging to reification nodes. ***Example:*** The r2d:BelongsToReificationNode associated with the *Victim_Phone* r2d:ComplexLiteralBlankNode is assigned a value of *Victim_Reif*, thereby linking the *Victim_Phone* column with its reification node.

r2d:DataType: This construct specifies the datatype of the r2d:ColumnBridge to which it is associated and comes into play when the structure of the virtual relational database schema objects is examined. ***Example:*** The *Address_Block* column bridge may have an r2d:DataType of *Integer* while the *Victim_Gender* column bridge has an r2d:DataType of *String*.

r2d:Predicate: This construct is used to store the fully qualified property name of the predicate which is associated with the reification r2d:ColumnBridge. This information is used during the SQL-to-SPARQL translation to generate the SPARQL WHERE clauses required to obtain the value of the r2d:ColumnBridge. ***Example:*** The complete URI of

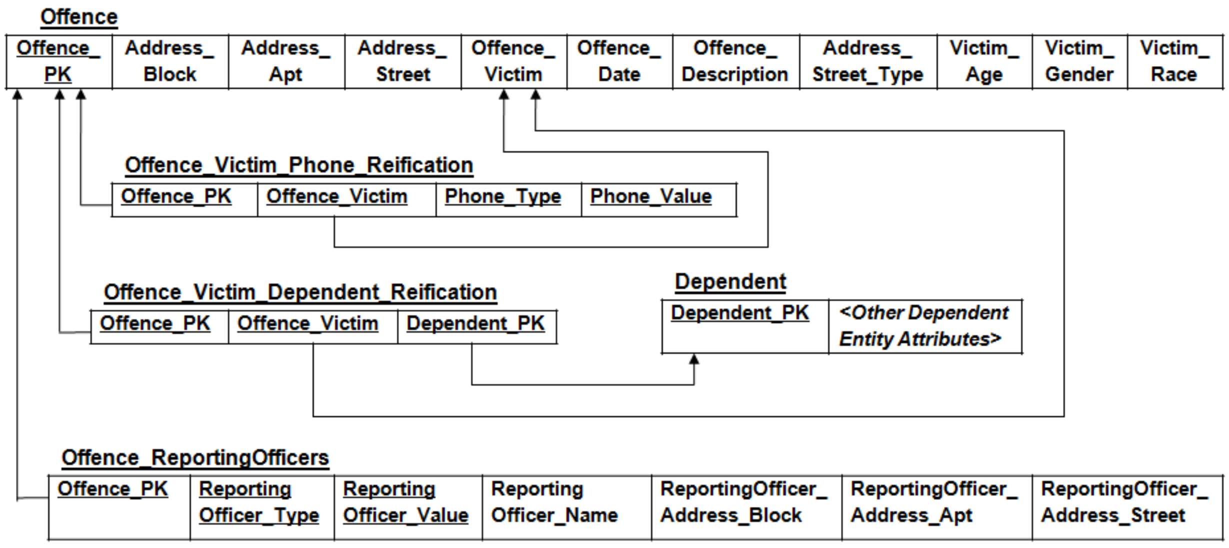


Figure 3.5: Equivalent Relational Schema corresponding to Figure 3.4's Sample Scenario

the *Victim_Gender* predicate of the *Victim_Reif* reification node is recorded using the r2d: Predicate construct.

Figure 3.5 illustrates the equivalent relational schema that corresponds to the Offence sample scenario depicted in Figure 3.4. The exact modus operandi of the map-file-to-relational-schema translation process is detailed in Chapter 6.

3.3 Types of Relationships addressed in R2D

The predicates and various types of blank nodes in Figure 3.2 and the relationships they represent in the corresponding virtual relational schemata are discussed below. The simple predicates typically map to a column in a relational schema. Blank nodes with multiple distinct or non-distinct predicates such as *"Courses"*, *"Phone"*, and *"Other Activities"* typically highlight 1:N/N:1 or N:M relationships, while blank nodes such as *"Name"*, with literal predicates, are typically equivalent to columns.

(a) *r2d:ColumnBridge Relationships (1:1 Relationships without Blank Nodes)*

In this kind of a relationship, most often one side of the relationship translates into a column/attribute in the table represented by the other side of the relationship. An example of a 1:1 relationship without blank nodes in Figure 3.2 is the triple (⟨*URI/StudentA*⟩ ⟨*Nick*⟩ ⟨*Nickname*⟩) referring to the relationship between an instance (⟨*URI/StudentA*⟩) of the Student class and his/her Nickname.

(b) *r2d:SimpleLiteralBlankNode Relationships (1:1 Relationships with Blank Nodes)*
These kinds of relationships are processed, for the purposes of transformation into a relational schema equivalent, by ignoring the blank node and treating the predicates of the blank nodes as multiple 1:1 relationships-without-blank-nodes to the subject of the blank node. Each predicate of the blank node essentially becomes an attribute of the table representing the subject instance. An example of a 1:1 relationship with blank nodes in Figure 3.2 is the triple (⟨*URI/StudentA*⟩ ⟨*Name*⟩ ⟨*blankNode*⟩).

(c) *r2d:ColumnBridge Relationships with r2d:refersToTableMap construct (N:1 Relationships without Blank Nodes)*
In N:1 relationships without Blank Nodes, the primary key of the table representing the instance on the "1" side of the relationship is included as a foreign key in the table representing the instance on the "N" side of the relationship. An example of a N:1 relationship without blank nodes in Figure 3.2 is the triple (⟨*URI/StudentA*⟩ ⟨*MemberOf*⟩ ⟨*Link to DepartmentID*⟩) referring to the relationship between an instance (⟨*URI/StudentA*⟩) of the Student class and an instance of the Department class.

(d) *r2d:MultiValuedColumnBridge (MVCB) Relationships with/without r2d:refersToTableMap construct (N:1 or N:M Relationships without Blank Nodes)*
r2d:MultiValuedColumnBridges with literal objects (i.e., without r2d:refersToTableMap construct) are equivalent to multi-valued attributes in relational terminology and, hence, result are considered to represent 1:N relationship between the subject and the object of the predicate corresponding to the MVCB. Thus, for MVCBs, a new table is generated with a foreign key that references the table corresponding to the class to which the subject belongs. MVCBs with resource objects (i.e. with r2d:referstoTableMap construct)

typically represent N:M relationships and hence, the processing of such MVCBs is similar to the processing discussed in category (f) below. An example of an MVCB with resource objects in Figure 3.2 is the triple (⟨*URI/StudentA*⟩ ⟨*WorksOn*⟩ ⟨*Link to Research*⟩) referring to the relationship between an instance (⟨*URI/StudentA*⟩) of the Student class and instances of the Research class.

(e) *r2d:ComplexLiteralBlankNodes and r2d:MultiValuedSimpleLiteralBlankNodes (N:1 Relationships with Blank Nodes)*
These relationships typically result in the generation of a separate table with a foreign key that references the table corresponding to the class to which the subject of this blank node object belongs. An example of a r2d:ComplexLiteralBlankNode in Figure 3.2 is (⟨*URI/StudentA*⟩ ⟨*Phone*⟩ ⟨*blankNode*⟩) and an example of an r2d:MultiValuedSimpleLiteralBlankNode is (⟨*URI/StudentA*⟩ ⟨*Home/Work Address*⟩ ⟨*blankNode*⟩). Both these examples result in the generation of r2d:MultiValuedPredicates due to the presence of distinct predicates for the phone number and address nodes.

(f) *r2d:SimpleResourceBlankNodes and r2d:ComplexResourceBlankNodes (N:M Relationships with Blank Nodes)*
N:M relationships with or without blank nodes result in the generation of a new join table that has, as foreign keys, the primary keys of the tables corresponding to the classes to which the subject and the resource object belong. An example of an N:M relationship with a blank node leading to similar object resources (i.e., a blank node of type r2d:SimpleResourceBlankNode) in the scenario in Figure 3.2 is the triple (⟨*URI/StudentA*⟩ ⟨*Projects*⟩⟨*blankNode*⟩) and an example of one with different object resources (r2d:ComplexResourceBlankNode) is the triple (⟨*URI/StudentA*⟩ ⟨*OtherActivities*⟩ ⟨*blankNode*⟩).

This background on R2D fundamentals provides the foundation behind R2D functionalities, the details of which, along with the details of the algorithms that comprise the R2D framework, are presented in a comprehensive manner in the next chapter.

CHAPTER 4
R2D SYSTEM DESIGN FOR REGULAR RDF TRIPLES

In keeping with the objectives of this research, several RDF-to-RDBMS bridging algorithms were designed and developed in addition to the design of the RDF-to-Relational mapping language discussed in the previous chapter. These include A) an algorithm that would, given an RDF Data Store, derive the mapping file automatically, B) an algorithm to parse the generated mapping file and generate, for the RDF Store, a list of relational tables, columns, and the relationships between them, and C) an algorithm to transparently transform any SQL statements issued against the virtual relational schema into its SPARQL equivalent, and return the results from the RDF Store in a relational/tabular format. The various modules highlighted in Figure 3.1 (a) and the corresponding algorithms are described at length in the following subsections.

4.1 RDFMapFileGenerator

The first step in the R2D Framework is the map file generator process accomplished using the RDFMapFileGenerator algorithm that takes an RDF store as input and automatically generates an RDF-to-Relational mapping file as output. Notional mappings between some key OWL/RDFS Ontology terminologies and R2D constructs to relational concepts can be found in (Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009a; Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009b).
However, the transformation process is not always as straightforward and as well-defined as the notional mappings suggest. As mentioned earlier, there are currently many RDF Graphs in existence that either do not have any, or have incomplete structural information included

along with the data. RDFMapFileGenerator works on RDF Stores with or without such structural information and the details are listed below.

The RDFMapFileGenerator algorithm generates mappings for RDF Stores with and without ontological information in the form of RDF Schema definitions such as rdfs:class, rdf:property, etc. This algorithm arrives at an RDF-to-Relational mapping file through extensive exploration of the triples data in the RDF Store and, consequently, is a bottleneck in the transformation process in terms of the response times. As a result, a number of sampling methods have been incorporated in the algorithm as can be seen in line 1 of the RDFMapFileGenerator Algorithm.

For RDF Stores without ontological information, two types of data sampling have been implemented, namely, Convenience/Haphazard Sampling, and Systematic sampling. In the case of stores containing ontological information, two variations of Stratified Sampling have been implemented; one where the sample size for each class is proportional to the class size, and the other where the sample size is independent of the class size, i.e., the sample size is the same for each class. These sampling methods have resulted in large reductions in response times as can be seen in Chapter 5.

The data structure discovery process as illustrated in Algorithm 4.1 is as follows. When structural information about the triples database is present, lines 2-11 of RDFMapFileGenerator discover the schema definitions and create appropriate Table and Column mappings based on the schema information.

Lines 12-35 process instance data to identify and account for those predicates that may not have been defined through explicit rdf:property definitions. This is done using three procedures, ProcessLiteralPredicate (Line 16), ProcessResourcePredicate (Line 22), and ProcessBlankNodePredicate (Line 19). The ***ProcessLiteralPredicate*** procedure, as the name suggests, is used to process predicates that have literal objects (such as *Nickname* predicate). For every literal predicate that does not have a column corresponding to itself, a new column is added to the TableMap corresponding to the resource to which the predicate belongs. If the resource contains more than one such predicate (i.e. the resource contains multiple literal

Algorithm 4.1 RDFMapFileGenerator

Input: RDF: The RDF Store of Interest

Output: RDF-to-Relational Schema Mapping File

```
1: Get sampling type, get/calculate sample size, calculate sample period (if systematic
   sampling)
2: if exists(RDFSchema Information) then
3:   for every resource that is an instance of rdfs:class do
4:     $TableMaps \longleftarrow resource_{n}ame$ {add}
5:   end for
6:   for every resource that is an instance of rdf:property do
7:     Get/Create TableMap, tblMap, corresponding to rdf:domain value of resource
8:     $tblMap.ColumnBridges \longleftarrow PropertyResource\_name$ {add}
9:     $tblMap.ColumnBridges.datatype \longleftarrow PropertyResourcesrdf : rangevalue$
10:  end for
11: end if
12: for every unprocessed (data) resource in the RDF store do
13:   Create a TableMap, tblMap, for this resource
14:   for every predicate of the resource do
15:     if object of predicate is literal then
16:       literalColumns += ProcessLiteralPredicate(resource, tblMap, predicate)
17:     end if
18:     if Object of predicate is a blank node then
19:       Call ProcessBlankNodePredicate(resource, tblMap, predicate)
20:     end if
21:     if Object of predicate is a resource then
22:       resourceColumns += ProcessResourcePredicate(resource, tblMap, predicate)
23:       ConsolidateResourcePredicates(resourceColumns)
24:       tblMap.setColumns(literalColumns); tblMap.setColumns(resourceColumns)
25:     end if
26:   end for
27:   if NOT(similarTableExists(tblMap)) then
28:     TableMaps += tblMap
29:   else
30:     discard tblMap
31:   end if
32:   if sampleSize reached then
33:     return
34:   end if
35: end for
```

object values for the same predicate), then the column type of the corresponding column is set to r2d:MultiValuedColumnBridge, otherwise it is a simple r2d:ColumnBridge.

The ***ProcessResourcePredicate*** procedure handles predicates that have resource objects. A new potential column is added for every resource predicate that belongs to the subject resource. After all resource predicates are processed the duplicate predicates (i.e., predicates that have objects belonging to the same object class) are examined and eliminated and this is done through the ***ConsolidateResourcePredicates*** procedure (Line 23). During the consolidation process, any (duplicate) potential columns that refer to the same object resource class (such as the *WorksOn* predicate) are combined and set to r2d:MultiValuedColumnBridges while columns referring to distinct object resource classes are set to r2d:ColumnBridge. This consolidation is mandatory in order to arrive at a normalized and logically sound relational schema. In cases where the objects belong to the same object class but the predicates have distinct values (such as the predicates off the *Phone* blank node), a MultiValuedPredicate object is created which reflects this fact. These MultiValuedPredicates typically become “TYPE” fields in the corresponding relational schema.

Predicates leading to blank nodes are handled through the ***ProcessBlankNode*** procedure. In this procedure, for every blank node encountered an object of type BlankNode is created. If every predicate off of the blank node contains a literal object (such as the *Name* and *Phone* blank nodes) then, for each predicate off of the blank Node, the ProcessLiteralPredicate procedure is called which works as described above. If every column generated through the ProcessLiteralPredicate procedure is a simple r2d: ColumnBridge (such as the *Name* blank node) then the BlankNode is set to r2d:SimpleLiteralBlankNode. If any of the columns are r2d:MultiValuedColumnBridges (such as the *Phone* blank node) then the BlankNode is set to r2d:ComplexLiteralBlankNode. If no such blank node has been previously encountered, this blank node is added to the set of blank nodes. If a similar blank node is already an element of the set of blank nodes, the blank node type is set to r2d:MultiValuedSimpleLiteralBlankNode (such as the blank nodes corresponding to the

HomeAddress and *WorkAddress* predicates) or r2d:MultiValuedComplexLiteralBlankNode respectively.

In case of blank nodes that contain only resource objects, every predicate off of such blank nodes is processed using the ProcessResourcePredicate procedure, also discussed above. As before, the consolidation process is carried out after all predicates off of the blank nodes are processed. If the number of consolidated columns is equal to 1 (such as in the case of the Courses blank node), the blank node type is set to r2d:SimpleResourceBlankNode, otherwise (as in the case of the OtherActivities blank node) it is set to r2d:ComplexResourceBlankNode. As in the previous case, if a similar blank node exists, the node type is set to r2d:MultiValued-SimpleResourceBlanknode or r2d:MultiValuedComplexResourceBlankNode respectively; otherwise, the blank node is added to the set.

Blank nodes that contain a mixture of literal objects, resource objects, and other blank nodes, are considered to be of type r2d:MixedBlankNodes and they are processed using the Depth-First-Search tree algorithm. The topmost blank node is considered the root of the tree and the procedure is as follows. For every literal or resource predicate off of a blank node, a column is created and added to the blank node entity. Additionally, for every blank node predicate off of a blank node, a new Blank Node entity is created and added to the set of blank nodes and is also added as a column to the original blank node. This way, the hierarchy of the tree rooted at the topmost blank node is maintained. This hierarchy is required during the SQL-to-SPARQL conversion to retrieve data associated with blank nodes appropriately.

Further, every resource object encountered and processed is stored in memory in order to avoid duplicate processing of the same in the event of multiple triples containing the same resource object. This information serves to improve the performance of the algorithm. When these "similar" resources are encountered during instance data processing, the algorithm skips the potential TableMap creation process and the time-consuming duplicate-TableMap detection process, thereby resulting in better efficiency.

Handling of Predicates with Object Resources belonging to multiple r2d:TableMaps *(i.e., a Foreign Key that has multiple tables that it needs to reference):*

RDF Graphs consists of many examples where the relational transformation creates a situation where an attribute A_{FK} in Entity E_{FK} could hold values corresponding to multiple entities, say E_1 to E_N (Let the set of attributes of each of these E_is be $A_1, A_2, \ldots A_N$). This situation is handled as follows.

The attribute list of E_{FK}, A_{EFK} is modified to include fields that reference the key field attributes of each of the entities, E_1 to E_N, which A_{FK} references. Thus,

$A_{EFK} = A_{EFK} \cup A_{ReferencingE1PK} \cup A_{ReferencingE2PK} \ldots \cup A_{ReferencingENPK} A_{FK}$

Lastly, each attribute $A_{ReferencingEiPK}$ in Entity E_{FK} is set to reference the key attribute of E_i (E_{iPK}). For every row in E_{FK}, one or more of the attributes $\{A_{ReferencingE1PK}, A_{ReferencingE2PK}, \ldots, A_{ReferencingENPK}\}$ will have a value while the others will be null. Since the relational schema corresponding to the given RDF graph generated by R2D is virtual involving no physical space/resource utilization, having multiple columns, many of which could be null, to represent the above scenario (foreign key referencing multiple tables) does not result in any resource wastage and is a simple solution to this requirement. An example of such a scenario would be the triple

⟨*StudentURI (Subject), Advisor(Predicate), AdvisorURI (Object)*⟩.

The Advisor object of a student resource could contain an instance from any one of the classes in the set {Full Professor, Associate Professor, Assistant Professor}. Again, the relational transformation of the above scenario would consist of four tables, namely, *Student*, *FullProfessor* (FP), *AssistantProfessor* (ASP), and *AssociateProfessor* (ACP). The *Student* table contains an *Advisor* column which is a foreign key. This foreign key needs to reference all three professor tables. As described above, this situation is handled by adding three separate columns to the *Student* table, *Advisor1* referencing the primary key of *FP*, namely FP_PK, *Advisor2* referencing *ASP*, and *Advisor3* referencing *ACP*. Further, for every row in the *Student* table only one of the three *Advisor* columns contains a value while the other two are null.

As another example, let us consider the following triple

⟨*PublicationURI (Subject) Author (Predicate) AuthorURI(Object)*⟩,

The Author object of a publication resource could contain instances from any of the classes in the set Full Professor (FP), Associate Professor (ACP), Assistant Professor (ASP), Graduate Student (GS), Undergraduate Student (UGS). Applying the consolidate method described above (in the Advisor example) results in the addition of five separate columns to the *Publication* table, *Author1* through *Author5*, referencing the primary keys of *FP*, *ACP*, *ASP*, *GS*, and *UGS* tables respectively. In this example, the 5 fields are not mutually exclusive, unlike in the Student-Advisor scenario, and, thus, any or all of the 5 *Author* fields could contain values for each publication record.

4.2 DBSchemaGenerator

The map file generation process is followed by the actual relational schema generation process which is the next stage in the R2D process and is achieved using the DBSchemaGenerator algorithm. This algorithm takes the RDF-to-Relational Schema mapping file generated by the RDFMapFileGenerator algorithm in Chapter 4, section 4.1 and returns a virtual, appropriately normalized relational database schema consisting of entities/tables and the relationships between them.

The DBSchemaGenerator Algorithm modus operandi is as follows. For each entry of type r2d:TableMap in the map file, a relational table, RelTable, is created in the virtual relational database schema. Entries of type r2d:ColumnBridge and r2d:MultiValuedColumnBridge whose r2d:belongsToTableMap value corresponds to the Table-Map, RelTable, are processed as follows. Every entry of r2d:ColumnBridge simply becomes a column in RelTable. If the r2d:ColumnBridge refers to another resource (as indicated by the r2d:refersToTableMap construct), a foreign key relationship is established between RelTable and the referred-to table. For every entry of type r2d:MultiValuedColumnBridge, which is comparable to multi-valued attributes in relational database terminology, a new table, NormTable, is created and the r2d:MultiValuedColumnBridge as well as the primary key of RelTable are added as columns to NormTable. Further, if the predicate corresponding to the r2d:MultiValuedColumnBridge is a r2d:MultiValuedPredicate, an additional "TYPE" column is created and added to NormTable. If the r2d:MultiValuedColumnBridge is a literal the NormTable type is set to "LiteralMVCBTable"; otherwise it is set to "ResourceMVCBTable".

Non-nested blank nodes of various kinds are handled as follows. For r2d:SimpleLiteral-BlankNodes (such as the blank node object of the *Name* predicate) of the kind illustrated in Chapter 3, Figure 3.2 every r2d:ColumnBridge entry that belongs to the blank node (as indicated by the r2d:belongsToBlankNode construct) is simply added as a column to the Table to which the r2d:SimpleLiteralBlankNode belongs (as indicated by the r2d:belongsToTableMap construct for the blank node). The processing of r2d:ComplexLiteralBlankNodes (such as the object of the *Phone* predicate) is very similar to the processing of r2d:MultiValued-ColumnBridges described above with the difference being the table type of the created table, which is set to "CLBNTable". Entries of type r2d:SimpleResourceBlankNode (object of the *Courses* predicate) and r2d:ComplexResourceBlankNodes (object of the *OtherActivities* predicate) result in creation of join tables, with the primary keys of tables corresponding to the subject resource and the object resource included as fields in the join table. Further, if the predicates corresponding to the column bridges belonging to these blank nodes are MultiValued, an additional "TYPE" column is created and added to the join table.

The processing of r2d:MultiValuedSimpleLiteralBlankNode, results in the creation of a new table, contrary to the r2d:SimpleResourceBlankNode scenario. This table has as columns the primary key of the table corresponding to the blank nodes r2d:belongsTo-TableMap value, and all the r2d:ColumnBridges that belong to the r2d:MultiValuedSimple-LiteralBlanknode. The processing of r2d:MultiValuedComplexLiteralBlanknode, r2d:Multi-ValuedSimpleResourceBlanknode, and r2d:MultiValuedComplexResourceBlankNode is very similar to their SingleValued counterparts with the only difference being the inclusion of an additional field in the event the predicate corresponding to the blank node is an "MVP". The table type values are set according to the type of blank nodes encountered. The reason for having table types and blank nodes is to maintain knowledge of the RDF graph structure in order to accurately translate SQL Statements issued against the relational schema into its appropriate SPARQL equivalent for precise data retrieval.

The final type of blank nodes processed by DBSchemaGenerator is mixed/nested blank nodes where the predicates off of the blank nodes are any combination of literals, resources, and other blank nodes. Due to the limitless kinds of such structured combinations that are possible, it would be impossible to even attempt to arrive at a corresponding normalized representation of the same. Hence, mixed/nested blank nodes of type r2d:MixedBlankNode are handled by creating a table, NormTable, which has as columns the primary key column of the table corresponding to the blank nodes r2d:belongsToTableMap construct, and the literal and resource objects that are at the leaf nodes of the tree rooted at the topmost mixed/nested blank node. This is achieved through a recursive procedure that explores the predicates in a depth-first manner.

4.3 SQL-to-SPARQL Translation

The SQL-to-SPARQL Translation procedure, the last procedure in the deployment sequence illustrated in Figure 3.1(b), corresponds to the final phase of the R2D transformation process where SQL statements issued against the virtual relational schema are parsed, translated into equivalent SPARQL (Prud'hommeaux and Seaborne 2008) queries that are executed against the RDF Store, and the results are returned in relational format. The SQL-to-SPARQL Translation algorithm, which takes an SQL Statement as input and returns an appropriate SPARQL equivalent as output,includes the ability to process queries involving underlying blank nodes, and to provide pattern matching and data aggregation abilities. The details of the algorithm are listed below.

The SQL-to-SPARQL Translation algorithm transforms single or multiple table queries with or without multiple where clauses (connected by AND, OR, or NOT operators) and Group By clauses. Within each individual where clause, the algorithm handles operators in

Algorithm 4.2 SQL-to-SPARQL Translation

Input: SQL Query

Output: Tabular results from execution of equivalent SPARQL Query

```
 1: Parse the input SQL query
 2: listOfFields ⟵ Array containing fields in the SELECT clause
 3: listOfTables ⟵ Array containing tables in the FROM clause
 4: whereClause ⟵ portion of the SQL Query after the WHERE keyword
 5: **if** exists(GROUP BY clause) **then**
 6:    groupByField ⟵ Array containing aggregated fields in SELECT clause
 7:    groupByFunction ⟵ Array containing aggregation functions on fields in SELECT
       clause
 8: **end if**
 9: SPARQLQuery ⟵ ProcessQuery
10: Execute SPARQLQuery
11: **for** every row in the result set **do**
12:    **for** every field in the SPARQL SELECT list **do**
13:       **if** isFieldPK(field) **then**
14:          Replace field with the fields tables ?subject variable
15:       **end if**
16:       $ResultRow \longleftarrow ResultRow \cup fieldValue\ of\ field\ from\ line\ 19's\ result\ set$
17:    **end for**
18:    **if** GROUP BY Fields present **then**
19:       **for** every groupByField and groupByFunction in list **do**
20:          Get the groupByField Value from line 19s result set
21:          **if** (current ResultRow == previous ResultRow) **then**
22:             Perform aggregation per groupByFunction for the groupByField
23:          **else**
24:             $ResultRow \longleftarrow ResultRow \cup groupByFieldvalue$
25:          **end if**
26:       **end for**
27:    **end if**
28:    $QueryResults \longleftarrow QueryResults \cup ResultRow$
29: **end for**
```

Table 4.1: SQL-to-SPARQL Translation Algorithm - Supporting Procedures

Procedure, its Input, and its Output	**Short Description**
ProcessQuery **Input:** List of Fields, Tables, and Where Clause **Output:** Equivalent SPARQL Query.	This procedure takes the list of fields and tables, and the where clause in the original SQL query as input and generates a SPARQL equivalent of the same as output. The SPARQL SELECT list is generated within this procedure and the SPARQL WHERE and FILTER clauses are generated using the ProcessWhereClause and ProcessPredicatesForTables procedures called within this procedure.
ProcessWhereClause **Input:** SQL WHERE clause **Output:** SPARQL FILTER clause constructs.	This procedure examines the SQL WHERE clause to identify those fields that have been used in the WHERE clause but are not a part of the SELECT list. Resolution/conversion of the LIKE SQL construct into an equivalent REGEX construct in SPARQL is also performed here.
ProcessPredicatesForTables **Input:** List of Tables and Fields in the SQL statement **Output:** SPARQL WHERE and FILTER clause constructs.	Generation of the SPARQL WHERE clause and additions to the SPARQL FILTER clause are performed by this procedure. This is where most of the complexity of the SQL-to-SPARQL Translation algorithm lies as predicates corresponding to every table/column/blank node type are processed and transformed here.

the following set $\{>, <, =, <=, >=, !=, \text{LIKE}\}$. Lines 1-8 of the algorithm essentially perform parsing of the input query to identify the tables, fields, the where clause, and Group By clause, if present. The ProcessQuery procedure, called in line 9, transforms the SQL Query into its SPARQL form while lines 10-29 execute the generated SPARQL query, process the results, and present the same in a tabular format. Lines 18-27 perform data aggregation as per the Group By functions specified in the SQL Statement. Aggregation is achieved by appending an ORDER BY clause to the transformed SPARQL query and the actual group functions are calculated on the data obtained through the execution of the appended SPARQL query. In order to better understand the transformation procedure let us consider the following query based on the sample scenario illustrated in Figure 3.2 in Chapter 3, Section 3.1.

SELECT Name_First, Name_Last, Phone_Value, Department_Name FROM employee, employee_Phone, department WHERE employee.employee_PK = employee_Phone.employee_PK and employee_Phone.Phone_Type = '<http://Phone/Cell>' and employee.department_id = department.department_id AND (name_First LIKE 'ABC%' OR employee_pk = '<http://empl/123>');

The SPARQL SELECT is generated by adding a variable for every field (including aggregated fields, and fields in the SQL WHERE clauses) in the SQL SELECT list. After this step the SPARQL SELECT list for our example is as follows:

SparqlSELECT = SELECT ?Name_First ?Name_Last, ?Phone_Value ?Department_Name

The SQL WHERE clauses are added, with minor modifications, to the FILTER clause of the SPARQL statement. If the field in the SQL WHERE clause is a primary key, the field name is replaced with the ?subject<*Index*> variable where *Index* corresponds to the table, or the parent table in the case of derived tables (corresponding to blank nodes) to which the field belongs. WHERE clauses involving non-primary key fields are added directly to the SPARQL FILTER clause. In the case of the LIKE operator, the value on the right-hand-side is converted to an equivalent regular expression construct (by appropriately using the "ˆ", "$", ".", and ".*" special characters in place of the "%" and "?" characters used in the LIKE expression) and the regex function is used on this converted expression in the FILTER clause. Upon completion of the SQL WHERE clause processing, the FILTER clause for our example is:

SparqlFILTER = FILTER (?Phone_Type = <http://Phone/Cell> && *employee_department_id = subject1* && *(regex(?Name_First, "ˆABC") || ?subject0 = http://empl/123)* }

The WHERE clause corresponding to *employee.employee_PK = employee_Phone.employee _PK* is eliminated in the SPARQL equivalent since *employee_Phone* is a derived table corresponding to the *employee* resource itself. Further, since the primary key field refers to the subject resource, the primary key fields associated with the employee and department tables are replaced with the corresponding ?subject$i$ variables where i is the unique *tableIndex* associated with the tables to which the primary keys belong.
The SPARQL WHERE clause is generated as follows. For non-derived tables and derived tables corresponding to multi-valued attributes clauses of the form *?subject<tableIndex> <Field.Predicate> ?<Field.Name>* are added for every field in the table. For derived tables corresponding to blank nodes and for fields belonging indirectly to non-derived tables (i.e. SimpleLiteralBlankNode fields), clauses of the form
?subject<tableIndex> <BlankNode.Predicate> ?<BlankNode.Name> and
? *<BlankNode.Name> Field.Predicate <Field.Name>*
are added to the SPARQL WHERE clause. The SPARQL WHERE clause after the processing of predicates associated with the non-derived table, *Department*, and the processing of fields belonging indirectly to the non-derived table, *Employee* (caused by the SimpleLiteralBlankNode corresponding to the multi-valued attribute, *Name*), is as follows:
SparqlWHERE = WHERE {
?subject0 <http://empl/Name> ?employee_name .
?employee_Name <http://Name/First> ?Name_First .

?employee_Name <http://Name/Last> ? Name_Last .

?subject0 <http://empl/deptId> ?employee_department_id .

?subject1 <http://dept/dept_name> ?Department_Name .}

Since a field cannot be specified in the FILTER clause without being a part of the SPARQL WHERE clause, the field *employee_department_id* is added to the SPARQL WHERE clause above despite not being a part of the SPARQL or SQL SELECT list.

For derived tables corresponding to multi-valued attributes or non-mixed blank nodes that contain multi-valued predicates, such as *EmployeePhone*, SPARQL where clauses of the form *?subject<tableIndex> ?<MVPColumn.Name> ?<NonMVPColumn.Name>* and *?<BlankNode.Name> ?<MVPColumn.Name> ?<NonMVPColumn.Name>*
are added, respectively. Further, for every predicate belonging to the multi-valued predicate field, a clause of the form *?MVPColumn.Name = <PredicateName>* is added to the SPARQL FILTER clause. The processing of predicates associated with the derived table, *Employee_Phone*, containing a multi-valued predicate column called *Phone_Type* results in the following additions to the SPARQL WHERE clause:

SparqlWHERE = SparqlWHERE +

?subject0 <http://empl/Phone> ?employee_Phone .

?employee_Phone ?Phone_Type ?Phone_Value .

Lastly, in the case of mixed blank nodes, for each field belonging to the mixed blank node table, the sequence of predicates leading from the topmost subject (of the mixed blank node) to the field are obtained by traversing the tree structure stored during the MapFileGeneration process and a Where clause is added to the SPARQL WHERE for each of the predicates in the sequence.

The SPARQL WHERE and FILTER clauses are added to the SPARQL Query and the final query is:

SparqlQUERY = SparqlSELECT + SparqlWHERE + SparqlFILTER

This transformed query is executed by the SQL-to-SPARQL-Translation Algorithm using the SPARQL Query Engine and the retrieved data is returned in relational format seamlessly.

CHAPTER 5
R2D IMPLEMENTATION DETAILS FOR REGULAR RDF TRIPLES

The hardware used for the Map File creation process and the LUBM queries were executed was a personal computer running the Windows Vista operating system with 4GB RAM and 2 GHz Intel Dual Core processor. The software platforms and tools used include Jena 2.5.6 (McBride 2002) to store the RDF triples data, MySQL 5.0 (MySQL) to house the RDF triples data persistently, Java 1.5 for development of the algorithms and procedures detailed in Chapter 4, DataVision v1.2.0 (DataVision) to visualize and generate reports based on the RDF data, and GRUFF v1.0.19 (Gruff) to compare the performance of R2D queries against.

5.1 Experimental Dataset

The LUBM dataset (Guo, Pan, and Heflin 2005), which consists of a university domain ontology comprising resources such as Universities, Departments, Professors, Students, Courses, etc., was used in the experimentation process. In order to illustrate relationalization of blank nodes, we made certain modifications that involve additions of blank nodes to the LUBM Schema. These modifications include the addition of an *EmailAddress* r2d:SimpleLiteralBlank-Node that involved altering the original simple literal *EmailAddress* property of resources into an SLBN consisting of two simple literal predicates, *PrimaryEmail* and *SecondaryE-mail*. The second type of blank node added was an r2d:ComplexLiteralBlankNode called *ContactNo* which was created by modifying the original simple literal *Phone* property belonging to all *Professor* (and its subclasses) resources into a blank node with multiple simple literal *CellPhone* predicates and one simple literal *HomePhone* predicate. Query numbers 1, 4, and 8 in the LUBM test queries include selection of fields corresponding predicates belonging to the SLBN and CLBN and query performance of the same is illustrated in Figure 5.6.

5.2 Experimental Results

The relational equivalent of the RDF Graph in Figure 3.2 was generated using the RDFMapFileGenerator and DBSchemaGenerator Algorithms detailed in Chapter 3 and the open source visualization tool DataVision, which expects a relational schema as input, was used to view the virtual relational schema generated, query the data using SQL statements, and generate reports off of the data.

The time taken by the map file generation process without any data sampling incorporated for RDF stores of various sizes, with and without ontological information, was compared with time taken for the same process when several sampling methods are applied and the results are illustrated in Figure 5.1. The process is especially time-intensive for large databases without structural information but this is only to be expected since the RDFMapFileGenerator has to explore every resource to ensure that no property is left unprocessed.

The sampling techniques applied improved the performance of the algorithm by a large factor, as can be seen in Figure 5.1. The processing times resulting from Convenience Sampling with sample sizes consisting of a fixed number of records are independent of the size of the data store and are almost constant since this technique only processes the first n rows regardless of the size of the database. Systematic sampling, on the other hand, does not yield as flat a line as Convenience sampling in the graphs above as it involves selecting samples periodically from the entire data store and, hence, is not as independent of the size of the data store as the former. For a similar reason, the Stratified Sampling scenario where the sample size is equally divided between the number of classes (Type B), regardless of the number of resources in each class, yields an almost constant response time contrary to its counterpart where the sample size for each class is proportional to the number of resources in each class (Type A).

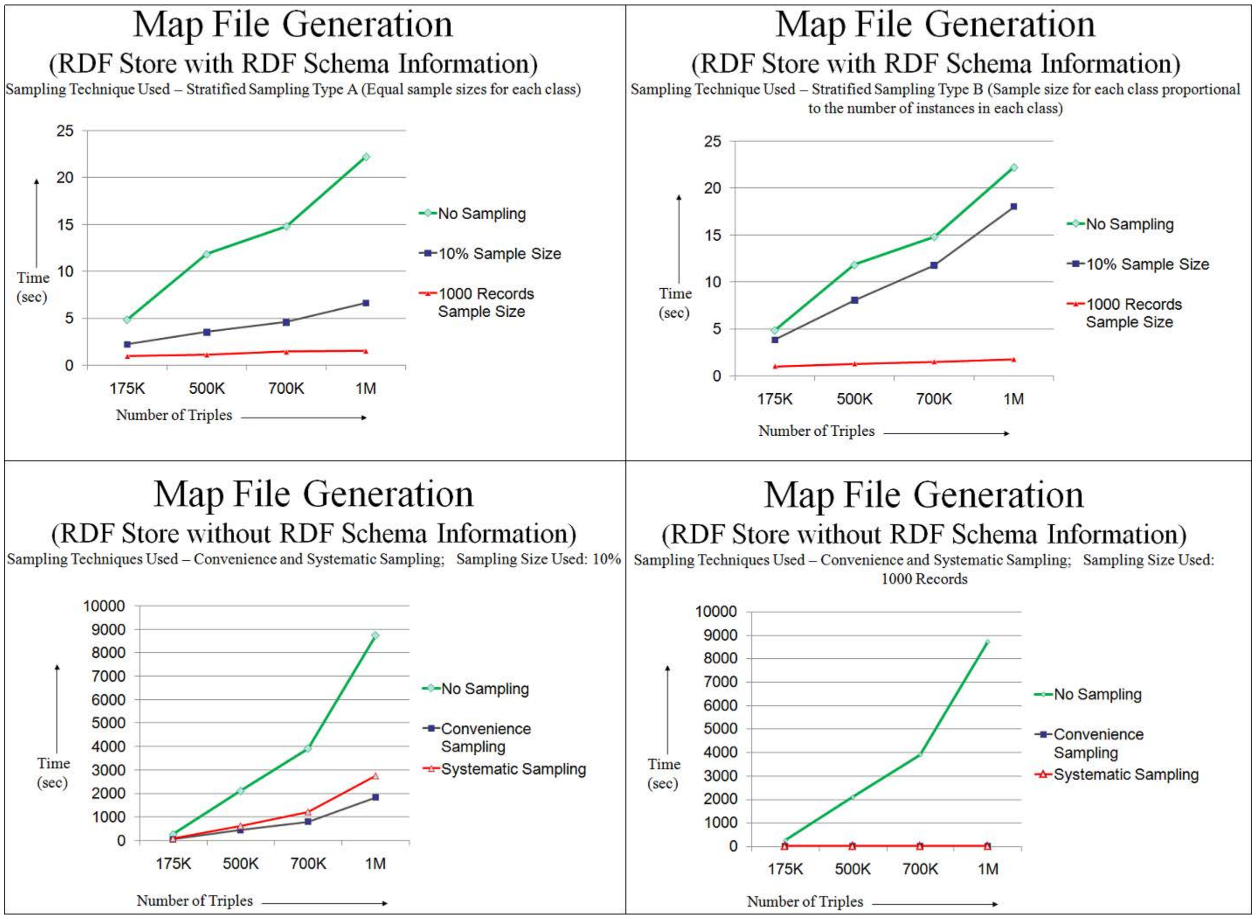
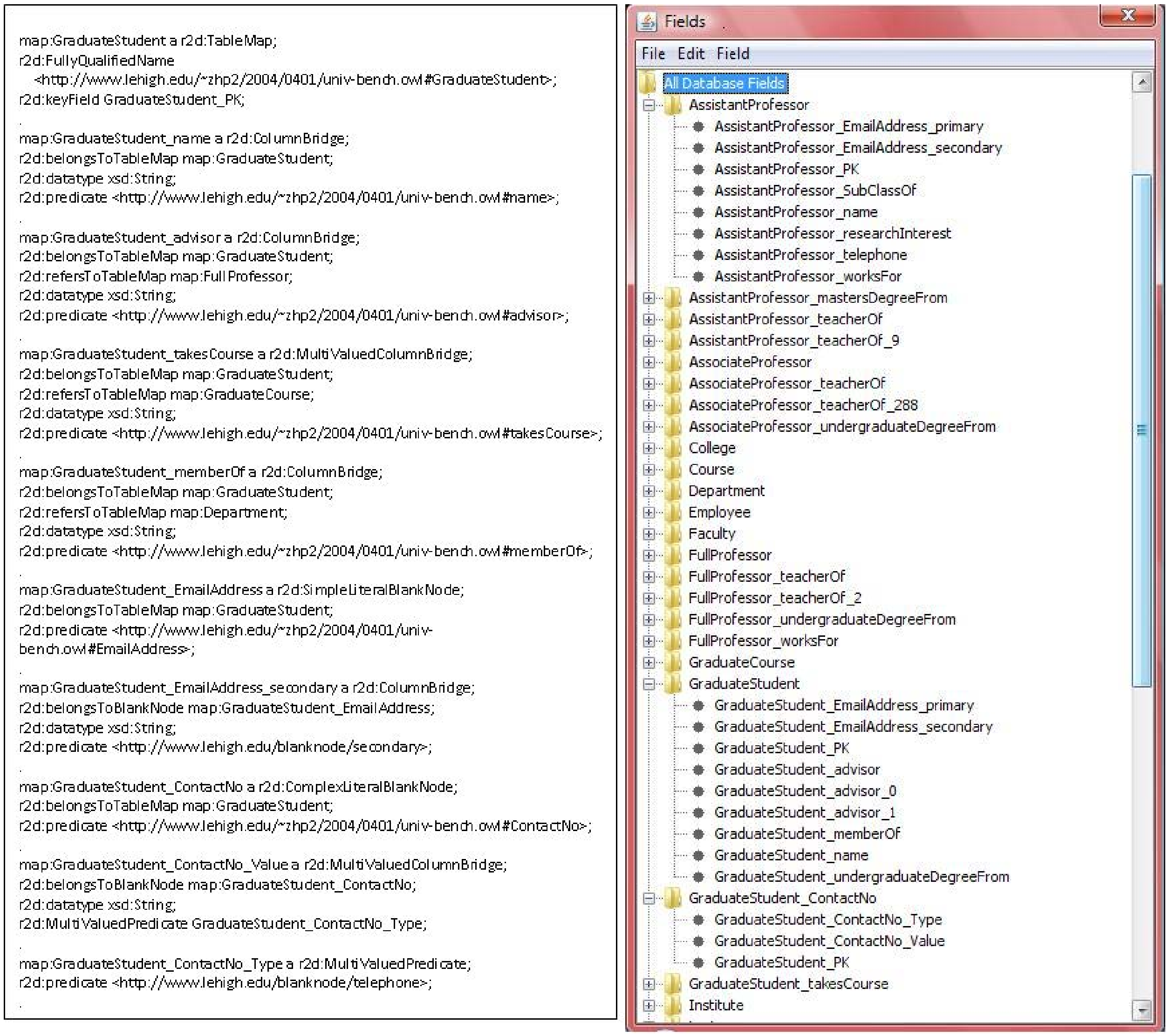


Figure 5.1: Map File Generation Times with and without Sampling

Sampling techniques are especially useful in scenarios where the structure of similar resources are quite well defined with only minor variations as, in such situations, the sampling methods do not run the risk of overlooking structural information that is not evident in the chosen sample data subset. Further, if a domain expert with knowledge of the structural information of the RDF store is available, the automatic map file generation process becomes optional. This step can be bypassed, and the time saved, by providing the map file manually. Figure 5.2 includes an excerpt from the map file generated by the RDFMapFileGenerator algorithm along with an inset of a part of the database schema as seen by DataVision. This schema is populated through the GetDatabaseMetaData Interface in the Connection class of the JDBC API within which the two algorithms, RDFMapFileGenerator and DBSchemaGenerator, are triggered. As can be seen, the various blank nodes that are part of the dataset are appropriately resolved and normalized into 1:N or N:M tables, or columns in existing tables using the algorithm described in Chapter 4, Section 4.2. The r2d:SimpleLiteralBlankNode associated with *Professor/Student-EmailAddress* is resolved into columns belonging to the *Professor/Student* tables and the r2d:ComplexLiteralBlankNode associated with *Graduate-Student-ContactNo* is resolved into a 1:N table of the same name.

Note that there are several tables in the virtual relational schema that seem like duplicates (such as *AssistantProfessor_TeacherOf* and *AssistantProfessor_TeacherOf_9, FullProfessor_TeacherOf* and *FullProfessor_TeacherOf_2*). These tables are not actually duplicates. The first table in the pair is a join table for the N:M relationship that exists between *<Assistant/Full>Professor* and *Course* classes while the second table in the pair is the join table for the N:M relationship that exists between the *<Assistant/Full>Professor* and *GraduateCourse* classes. The join table names in R2D's virtual relational schema are derived from the relevant predicate names. Since the predicate names of the *Professor-Course* triples and the *Professor-GraduateCourse* triples are identical in the LUBM dataset, the RDFMapFileGenerator algorithm appended a unique identifier (the numbers at the end of the table names) to the second join table in order to avoid duplicate table names in the virtual relational schema.



```
map:GraduateStudent a r2d:TableMap;
r2d:FullyQualifiedName
   <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateStudent>;
r2d:keyField GraduateStudent_PK;
.
map:GraduateStudent_name a r2d:ColumnBridge;
r2d:belongsToTableMap map:GraduateStudent;
r2d:datatype xsd:String;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#name>;
.
map:GraduateStudent_advisor a r2d:ColumnBridge;
r2d:belongsToTableMap map:GraduateStudent;
r2d:refersToTableMap map:FullProfessor;
r2d:datatype xsd:String;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#advisor>;
.
map:GraduateStudent_takesCourse a r2d:MultiValuedColumnBridge;
r2d:belongsToTableMap map:GraduateStudent;
r2d:refersToTableMap map:GraduateCourse;
r2d:datatype xsd:String;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#takesCourse>;
.
map:GraduateStudent_memberOf a r2d:ColumnBridge;
r2d:belongsToTableMap map:GraduateStudent;
r2d:refersToTableMap map:Department;
r2d:datatype xsd:String;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf>;
.
map:GraduateStudent_EmailAddress a r2d:SimpleLiteralBlankNode;
r2d:belongsToTableMap map:GraduateStudent;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#EmailAddress>;
.
map:GraduateStudent_EmailAddress_secondary a r2d:ColumnBridge;
r2d:belongsToBlankNode map:GraduateStudent_EmailAddress;
r2d:datatype xsd:String;
r2d:predicate <http://www.lehigh.edu/blanknode/secondary>;
.
map:GraduateStudent_ContactNo a r2d:ComplexLiteralBlankNode;
r2d:belongsToTableMap map:GraduateStudent;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#ContactNo>;
.
map:GraduateStudent_ContactNo_Value a r2d:MultiValuedColumnBridge;
r2d:belongsToBlankNode map:GraduateStudent_ContactNo;
r2d:datatype xsd:String;
r2d:MultiValuedPredicate GraduateStudent_ContactNo_Type;
.
map:GraduateStudent_ContactNo_Type a r2d:MultiValuedPredicate;
r2d:predicate <http://www.lehigh.edu/blanknode/telephone>;
.
```

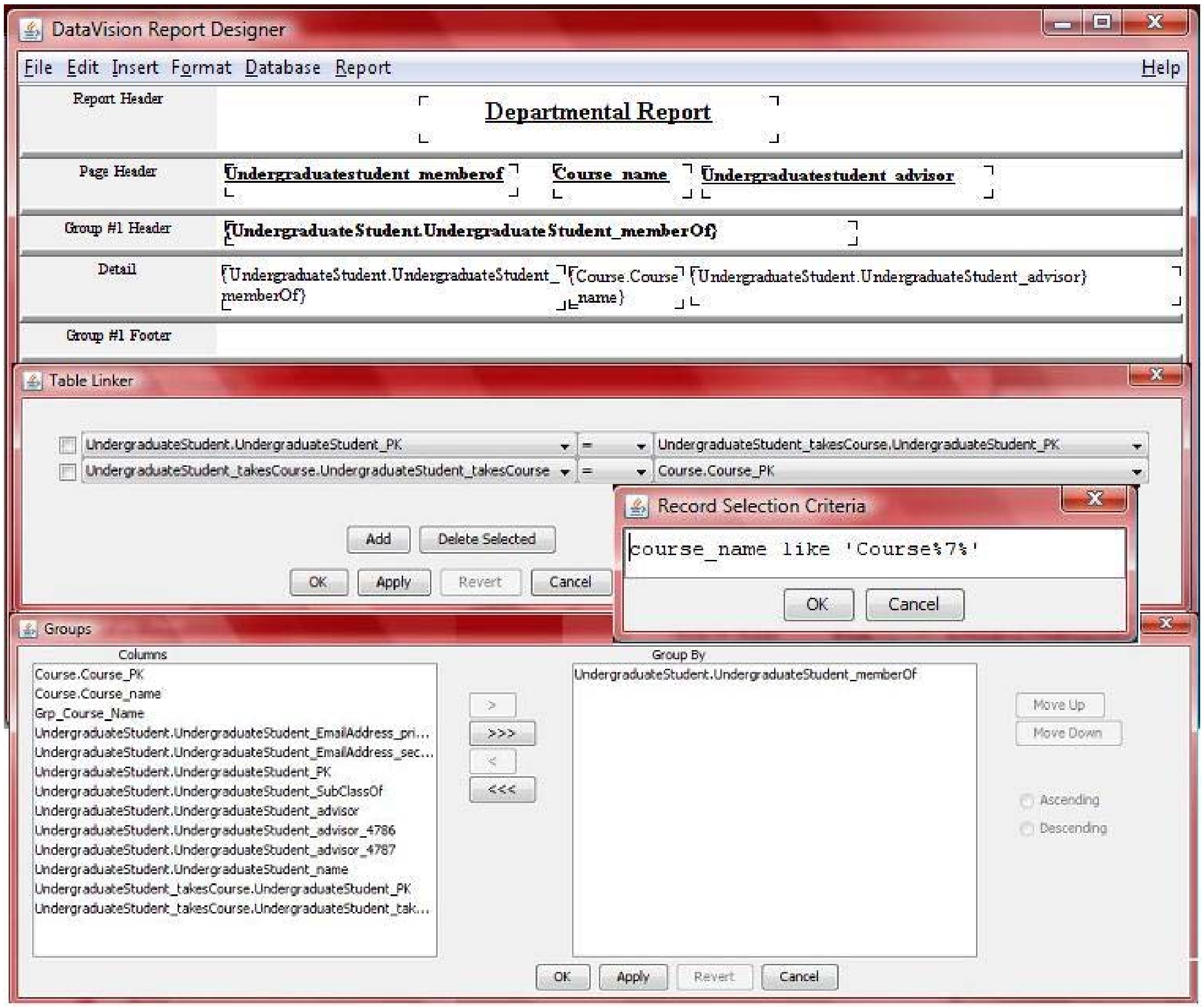
```
All Database Fields
  AssistantProfessor
    AssistantProfessor_EmailAddress_primary
    AssistantProfessor_EmailAddress_secondary
    AssistantProfessor_PK
    AssistantProfessor_SubClassOf
    AssistantProfessor_name
    AssistantProfessor_researchInterest
    AssistantProfessor_telephone
    AssistantProfessor_worksFor
  AssistantProfessor_mastersDegreeFrom
  AssistantProfessor_teacherOf
  AssistantProfessor_teacherOf_9
  AssociateProfessor
  AssociateProfessor_teacherOf
  AssociateProfessor_teacherOf_288
  AssociateProfessor_undergraduateDegreeFrom
  College
  Course
  Department
  Employee
  Faculty
  FullProfessor
  FullProfessor_teacherOf
  FullProfessor_teacherOf_2
  FullProfessor_undergraduateDegreeFrom
  FullProfessor_worksFor
  GraduateCourse
  GraduateStudent
    GraduateStudent_EmailAddress_primary
    GraduateStudent_EmailAddress_secondary
    GraduateStudent_PK
    GraduateStudent_advisor
    GraduateStudent_advisor_0
    GraduateStudent_advisor_1
    GraduateStudent_memberOf
    GraduateStudent_name
    GraduateStudent_undergraduateDegreeFrom
  GraduateStudent_ContactNo
    GraduateStudent_ContactNo_Type
    GraduateStudent_ContactNo_Value
    GraduateStudent_PK
  GraduateStudent_takesCourse
  Institute
```

Figure 5.2: Map File Excerpt and a Portion of the Equivalent Relational Schema as seen by Datavision

Figure 5.3 is a screenshot of DataVision's Report Designer which illustrates DataVision's query building process for a sample query involving the SQL LIKE operator and a GROUP BY clause. Based on the fields chosen (in the "Report Designer" window), the table linkages (i.e., joins, illustrated in the "Table Linker" inset) specified, and additional record selection and grouping criteria specified (illustrated in the "Record Selection Criteria" and "Groups" insets respectively), DataVision generates an appropriate SQL query, as shown in the "SQL Query" inset in Figure 5.4, to extract the required data. At this juncture, the Statement Interface, the Prepared Statement Interface, and the ResultSet Interface that are part of the JDBC interface are invoked. These interfaces trigger the SQL-to-SPARQL Translation algorithm, which generates a SPARQL equivalent of the given SQL statement as illustrated in Figure 5.4, and return the obtained results to DataVision in the expected tabular format, as illustrated in Figure 5.5.

While DataVision, like any other relational reporting/visualization tool, has options to specify aggregation and grouping conditions and functions, the DataVision support group has, for various reasons that are not applicable to our academic test environment, disabled the GROUP BY facility. For the purposes of our research, we have enabled the functionality and the results are as displayed in Figure 5.5 below.

In order to compare the performance of queries executed through the virtual relational schema offered by R2D against the query performance achieved through RDF visualization tools, XML files corresponding to the LUBM dataset were generated for RDF stores of various sizes and a selection of four queries were run using R2D and Allegrograph's Gruff. These queries were selected at random from the set of LUBM Benchmark SPARQL queries and their equivalent SQL versions were executed using R2D. Figure 5.6 displays the response times of each of the queries as the sizes of the databases vary. As can be observed, R2D's performance is far superior to the existing direct RDF visualization. This could be because Gruff persists data on the hard disk in a proprietary manner, requiring additional time/resources for disk I/O, while R2D utilizes Jena's in-memory store to house the RDF data. The time taken for the SQL-to-SPARQL conversion (SQL-to-SPARQL Translation Algorithm) is negligible and



Departmental Report

Undergraduatestudent_memberof | Course_name | Undergraduatestudent_advisor

{UndergraduateStudent.UndergraduateStudent_memberOf}

{UndergraduateStudent.UndergraduateStudent_memberOf} {Course.Course_name} {UndergraduateStudent.UndergraduateStudent_advisor}

UndergraduateStudent.UndergraduateStudent_PK = UndergraduateStudent_takesCourse.UndergraduateStudent_PK

UndergraduateStudent_takesCourse.UndergraduateStudent_takesCourse = Course.Course_PK

```
course_name like 'Course%7%'
```

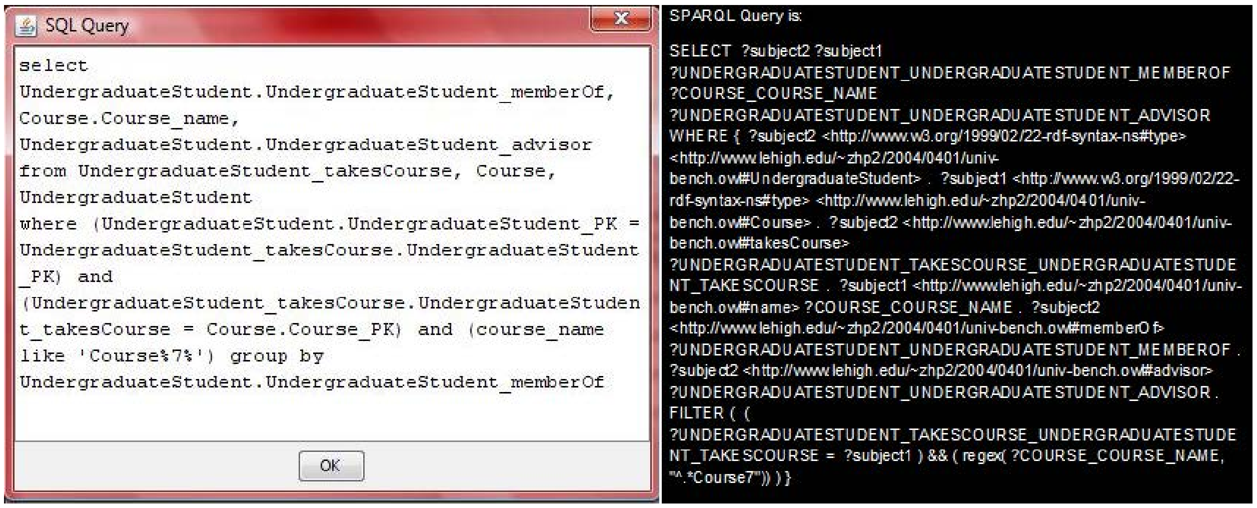
Columns:

- Course.Course_PK
- Course.Course_name
- Grp_Course_Name
- UndergraduateStudent.UndergraduateStudent_EmailAddress_pri...
- UndergraduateStudent.UndergraduateStudent_EmailAddress_sec...
- UndergraduateStudent.UndergraduateStudent_PK
- UndergraduateStudent.UndergraduateStudent_SubClassOf
- UndergraduateStudent.UndergraduateStudent_advisor
- UndergraduateStudent.UndergraduateStudent_advisor_4786
- UndergraduateStudent.UndergraduateStudent_advisor_4787
- UndergraduateStudent.UndergraduateStudent_name
- UndergraduateStudent_takesCourse.UndergraduateStudent_PK
- UndergraduateStudent_takesCourse.UndergraduateStudent_tak...

Group By:

- UndergraduateStudent.UndergraduateStudent_memberOf

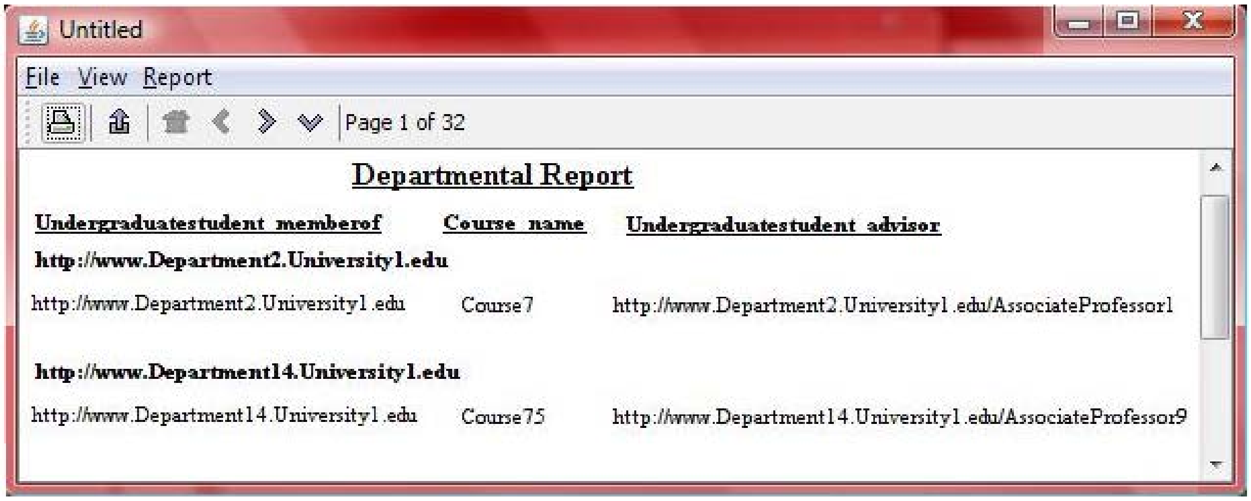
Figure 5.3: DataVision Query Processing



```
select
UndergraduateStudent.UndergraduateStudent_memberOf,
Course.Course_name,
UndergraduateStudent.UndergraduateStudent_advisor
from UndergraduateStudent_takesCourse, Course,
UndergraduateStudent
where (UndergraduateStudent.UndergraduateStudent_PK =
UndergraduateStudent_takesCourse.UndergraduateStudent
_PK) and
(UndergraduateStudent_takesCourse.UndergraduateStuden
t_takesCourse = Course.Course_PK) and (course_name
like 'Course%7%') group by
UndergraduateStudent.UndergraduateStudent_memberOf
```

```
SPARQL Query is:

SELECT  ?subject2 ?subject1
?UNDERGRADUATESTUDENT_UNDERGRADUATESTUDENT_MEMBEROF
?COURSE_COURSE_NAME
?UNDERGRADUATESTUDENT_UNDERGRADUATESTUDENT_ADVISOR
WHERE {  ?subject2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#UndergraduateStudent> .  ?subject1 <http://www.w3.org/1999/02/22-
rdf-syntax-ns#type> <http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#Course> .  ?subject2 <http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#takesCourse>
?UNDERGRADUATESTUDENT_TAKESCOURSE_UNDERGRADUATESTUDE
NT_TAKESCOURSE .  ?subject1 <http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#name> ?COURSE_COURSE_NAME .  ?subject2
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf>
?UNDERGRADUATESTUDENT_UNDERGRADUATESTUDENT_MEMBEROF .
?subject2 <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#advisor>
?UNDERGRADUATESTUDENT_UNDERGRADUATESTUDENT_ADVISOR .
FILTER ( (
?UNDERGRADUATESTUDENT_TAKESCOURSE_UNDERGRADUATESTUDE
NT_TAKESCOURSE =  ?subject1 ) && ( regex( ?COURSE_COURSE_NAME,
"^.*Course7")) ) }
```

Figure 5.4: SQL-to-SPARQL Conversion



Departmental Report

```
Undergraduatestudent_memberof       Course_name   Undergraduatestudent_advisor
http://www.Department2.University1.edu
http://www.Department2.University1.edu    Course7    http://www.Department2.University1.edu/AssociateProfessor1

http://www.Department14.University1.edu
http://www.Department14.University1.edu   Course75   http://www.Department14.University1.edu/AssociateProfessor9
```

Figure 5.5: Tabular Results as seen through DataVision

nearly constant. Thus, R2D does not add any overheads to the SPARQL query performance. SQL queries issued against relational databases created by physically duplicating RDF data may possibly exhibit superior performance than their SPARQL equivalents since refined performance optimization options (such as indexes, mature query optimizers, etc.) have been at the disposal of relational databases for many decades now.

Further, for each row of the RDBMS with n columns, there are n triple tuples in the corresponding RDF Store. Thus, the RDBMS equivalent of the RDF Stores generally has a fraction of the data in the RDF Stores which could be yet another contributor to better RDBMS response times than the RDF data store. However, this improved performance comes at the expense of additional disk space that is required due to duplication of data into the RDBMS, and additional system resources and human effort required to ensure that the duplicated data is kept synchronized with the original RDF store. On the other hand, for possibly a small price in terms of response time, R2D offers an avenue for users to continue to take advantage of the vast assortment of visualization tools that are readily available without having to "reinvent the wheel" for RDF stores or duplicate/synchronize RDF data. With skilled database administrators becoming rarer and more expensive, the importance of applications such as R2D becomes more pronounced as they offer a means to bypass the requirement of databases and their management.

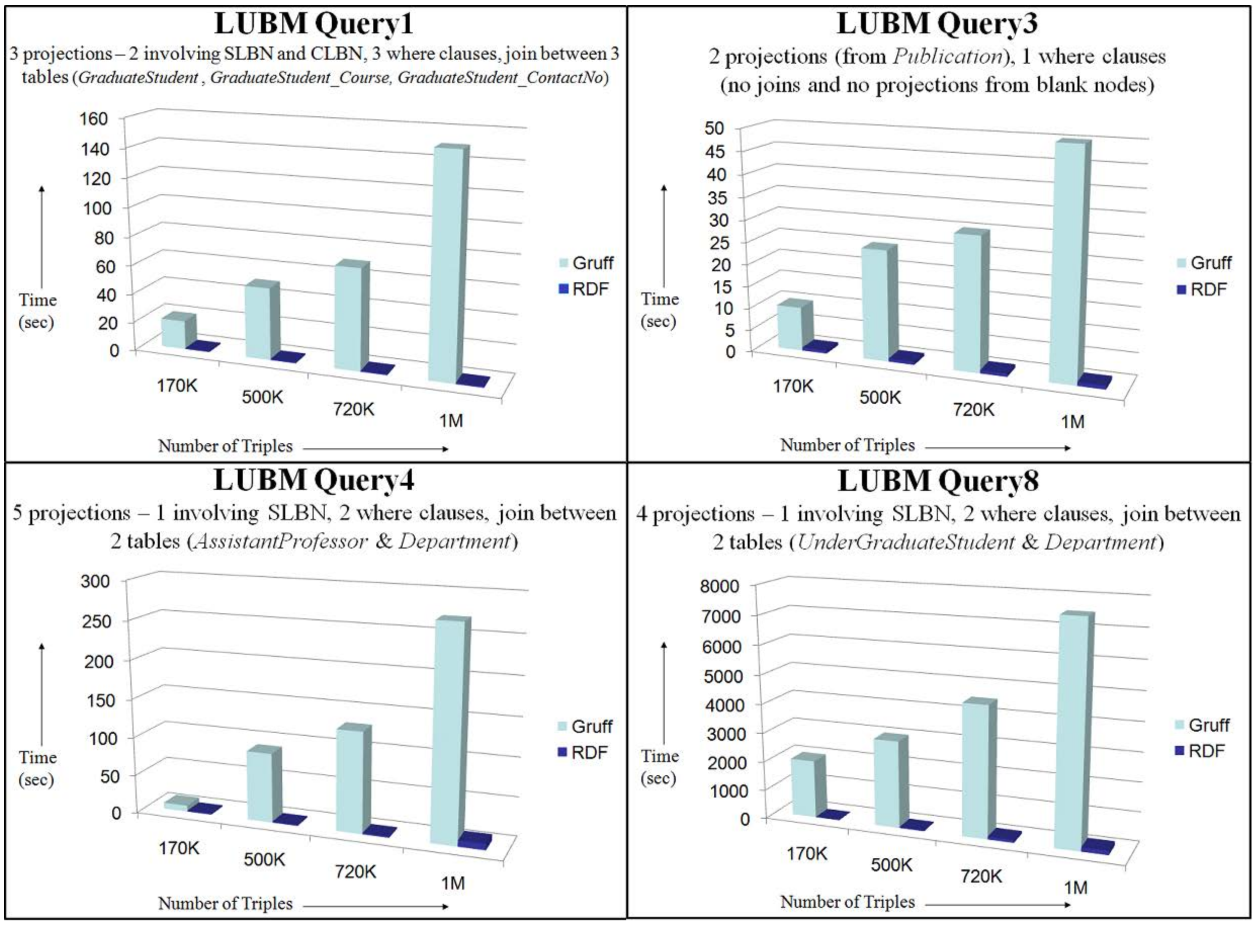


Figure 5.6: Response Times for Selected LUBM Queries

CHAPTER 6
R2D SYSTEM DESIGN FOR RDF REIFICATION DATA

The algorithm aspects associated with translation of simple reification nodes that do not include blank nodes and reification nodes that contain non-quad predicates leading to blank nodes are presented in the following subsections.

6.1 Mapping Reification Nodes – RDFMapFileGenerator

The first stage in the R2D transformation framework involves generation of a map file containing the correlations between meta-data obtained from the input RDF store and its relational schema equivalent. The algorithm that carries out this function is the RDFMapFileGenerator and the details specific to relationalization of RDF reification are listed in Algorithm 6.1.

The reification data processing methodology within RDFMapFileGenerator is quite similar to that used to map regular resources and blank nodes. Every blank node corresponding to a reification quad is mapped using the r2d:ReificationNode construct. If the "rdf:subject" property of the "reification quad" mapped by the r2d:Reification construct is a resource, the r2d:BelongsToTableMap construct is used to associate the "reification quad" with the r2d:TableMap corresponding to the resource. If the "rdf:subject" property is a blank node, the r2d:BelongsToBlankNode construct is used to associate the "reification quad" to the r2d:[Simple/Complex][Literal/Resource]BlankNode associated with the "rdf:subject" blank node. Further, if the rdf:object property of the "reification quad" refers to another resource, then r2d:RefersToTableMap construct is used to store this relationship. This information is used in the case of 1:N relationships between two TableMap entities during the SQL-to-

Algorithm 6.1 RDFMapFileGenerator

Input: RDF Store: The RDF Store of Interest

Output: RDF-to-Relational Schema Mapping File

```
1: for every reification quad in the RDF Store do
2:   ReificationNodes += Node for reification quad
3:   if rdf:subject of reification quad is a resource then
4:     ReificationNode.BelongsToTableMap = TableMap corresponding to rdf:subject resource
5:   else
6:     {/* rdf:subject is a blank node*/}
7:     ReificationNode.BelongsToBlankNode = [Simple/Complex][Resource/Literal] Blank Node corresponding to rdf:Subject blank node
8:   end if
9:   if rdf:object of reification node is a resource then
10:    ReificationNode.RefersToTableMap = TableMap corresponding to the rdf:object resource
11:  end if
12:  ReificationNode.ReifiedPredicate = rdf:Predicate of reification quad
13:  for every non-quad predicate (NQP) of reification quad do
14:    if Object of reification NQP is a blank node = blankNode then
15:      ProcessBlankNodeNQP(Triples of tree rooted at blankNode)
16:    else
17:      ColumnBridge.Name = reification non-quad predicate Name
18:      ColumnBridge.Predicate = URI of reification non-quad predicate
19:      ColumnBridge.BelongsToReificationNode = ReificationNode
20:      ReificationNode.ColumnBridges += ColumnBridge
21:    end if
22:  end for
23: end for
```

Algorithm 6.2 ProcessBlankNodeNQP

Input: RDF Triples in tree rooted at blankNode

Output: Processed blankNode NQP and associated r2d:ColumnBridges

```
 1: if all objects of blankNode are simple literals then
 2:    if repeating predicates exist then
 3:       blankNode.Type = ComplexLiteralBlankNode
 4:    else
 5:       blankNode.Type = SimpleLiteralBlankNode
 6:    end if
 7: else if all objects of blank node are resources then
 8:    if all resources belong to same overall object class then
 9:       blankNode.Type = SimpleResourceBlankNode
10:    else
11:       blankNode.Type = ComplexResourceBlankNode
12:    end if
13: else
14:    blankNode.Type = MixedBlankNode
15: end if
16: blankNode.BelongsToReificationNode = ReificationNode
17: blankNodes += blankNode
18: for every predicate of reification blankNode NQP do
19:    if object of the predicate is a simple literal or resource then
20:       ColumnBridge.Name = predicate Name
21:       ColumnBridge.Predicate = URI of predicate
22:       ColumnBridge.BelongsToBlankNode = blankNode
23:       reification blankNode.ColumnBridges += reification blankNode.ColumnBridges +=
          ColumnBridge
24:    else
25:       {/* object is a blank node */}
26:       Repeat lines 1-26
27:    end if
28: end for
```

SPARQL transformation.

Every simple (i.e., non-blank-node) non-quad predicate of the reification blank node is mapped using the r2d:ColumnBridge construct and is associated with its reification node using the r2d:BelongsToReificationNode construct as shown in of Algorithm 6.1. Furthermore, the datatype of the object corresponding to the non-quad predicate is mapped using the r2d:Datatype construct and the URI of the non-quad predicate itself is recorded using the r2d:Predicate construct, for use during the SQL-to-SPARQL transformation. Mapping file excerpts corresponding to reification nodes and simple non-quad predicates of reification nodes can be found in (Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2009d; Ramanujam, Gupta, Khan, Seida, and Thuraisingham 2010).

Reification nodes that contain blank nodes are mapped using one of the blank node constructs discussed in Chapter 3, Section 3.2. As can be seen from Algorithm 6.1, when a non-quad reification predicate with an object that is a blank node is encountered, Algorithm 6.2 is called where the type of blank node is assessed by examining the objects of the blank node and the blank node is mapped using the r2d:[MultiValued][Simple/Complex][Literal/Resource] BlankNode construct as applicable. Further, the reification blank node NQP is associated with its parent reification node using the r2d:BelongsToReificationNode construct as before. Predicates belonging to the reification blank node NQP are mapped in the same manner as simple non-quad reification predicates, i.e., using the r2d:ColumnBridge construct. These predicates are linked to their parent blank node NQP using the r2d:BelongsToBlankNode construct thereby keeping the graph structure intact. Tables 6.1 and 6.2 lists map file excerpts corresponding to the *Victim_Phone*, *Victim_Dependents*, and *ReportingOfficer_Officer-Address* blank nodes in Figure 3.4 and the predicates belonging to the above blank nodes.

Complex reification blank nodes, such as ones that contain one or more nested blank node predicates, are processed using the Depth-First-Search tree algorithm (similar to mixed blank nodes processing in Chapter 4). Every complex reification blank node is mapped using the r2d:MixedBlankNode construct and is connected to the parent r2d:ReificationNode using the r2d:BelongsToReificationNode construct.

Table 6.1: Excerpts from the R2D Map File depicting mapping entries corresponding to reification Blank Nodes

Map File Excerpt for Reification Blank Nodes
map:Offence_Victim_Reif a r2d:ReificationNode;
r2d:belongsToTableMap map:Offence;
r2d:datatype xsd:String;
r2d:reifiedPredicate <http://offence/Victim>;
.
map:Offence_Victim_Phone a r2d:ComplexLiteralBlankNode;
r2d:belongsToReificationNode map:Offence_Victim_Reif;
r2d:predicate <http://offence_victim_reif/Phone>;
.
map:Offence_Victim_VictimDependent a r2d:SimpleResourceBlankNode;
r2d:belongsToReificationNode map:Offence_Victim_Reif;
r2d:refersToTableMap map:Dependent;
r2d:predicate <http://offence_victim_reif/VictimDependent>;
.
map:Offence_ReportingOfficers_Officer_Reif a r2d:ReificationNode;
r2d:belongsToBlankNode map:Offence_ReportingOfficers;
r2d:datatype xsd:String;
r2d:reifiedPredicate <http://offence/ReportingOfficers/Officer>;
.
map:Offence_ReportingOfficers_Officer_Address a r2d:SimpleLiteralBlankNode;
r2d:belongsToReificationNode map:Offence_ReportingOfficers_Officer_Reif;
r2d:predicate <http://offence/ReportingOfficers/Officer/Address>;
.

Table 6.2: Excerpts from the R2D Map File depicting mapping entries corresponding to reification Blank Node predicates

Map File Excerpt for Predicates of Reification Blank Nodes

map:Phone_Value a r2d:MultiValuedColumnBridge;
r2d:belongsToBlankNode map:Offence_Victim_Phone;
r2d:datatype xsd:String;
r2d:MultiValuedPredicate Phone_Type;
.
map:Phone_Type a r2d:MultiValuedPredicate;
r2d:predicate <http://offence_victim_reif/VictimCellPhone>;
r2d:predicate <http://offence_victim_reif/VictimWorkPhone>;
.
map:Offence_Victim_VictimDependent_Dependent a r2d:MultiValuedColumnBridge;
r2d:belongsToBlankNode map:Offence_Victim_VictimDependent;
r2d:refersToTableMap map:Dependent;
r2d:datatype xsd:String;
r2d:predicate <http://offence_victim_reif/Dependent>;
.
map:Offence_ReportingOfficers_Officer_Address_Block a r2d:ColumnBridge;
r2d:belongsToBlankNode map:Offence_ReportingOfficers_Officer_Address;
r2d:datatype xsd:String;
r2d:predicate <http://offence/ReportingOfficers/Officer/Address/Block>;
.

6.2 Relationalizing Reification Data – DBSchemaGenerator

The second stage of the R2D transformation framework involves the actual virtual, normalized, relational schema generation for the input RDF store based on information in the map file created in stage one. The algorithm that provides this functionality is called the DBSchemaGenerator and the details of the same pertaining to the relational transformation of reification data are listed in Algorithm 6.3.

The circumstances and conditions under which a new r2d:TableMap is created to house the reification data in the virtual relational schema corresponding to the given input RDF store are listed in Table 6.3.

Relational transformation of simple non-quad reification predicates and non-quad reification blank nodes is discussed below.

I. **Relationalization of Simple Literal/Resource NQPs and Reification SLBN NQPs:**

(a) *Simple/SLBN NQP reifies a predicate or blank node belonging to an r2d:Tablemap or a predicate belonging to an SLBN of the r2d:TableMap:* If the reified predicate is a literal MVCB or a resource MVCB representing an N:M relationship between the {subject, object} resource pair, the reification predicate is added as a column into the normalized table that results from the relational transformation of the MVCB. In the case of reification SLBN NQPs, the predicates belonging to the reification SLBN are each added as columns to the normalized table. If the reified predicate is a resource MVCB representing a 1:N relationship between the {subject, object} pair, the simple NQP, or, in the case of reification SLBN NQPs, every predicate belonging to the reification SLBN, is added as a column to the TableMap representing the N-side of the relationship. If the reified predicate has a blank node object or if the reified predicate belongs to an SLBN of an r2d:TableMap, the NQP (or every predicate belonging to the SLBN in the case

Algorithm 6.3 DBSchemaGenerator

Input: MapFile: RDF-to-Relational Schema Mapping File
Output: A Normalized Relational Schema

```
1: for every entry of type r2d:ReificationNode do
2:   ParentTable = ReificationNode.BelongsToTableMap OR ReificationNode.BelongsToBlankNode
3:   if ParentTable.Type = "Table" or ParentTable.Type = SLBN then
4:     if ReificationNode.ReifiedPredicate refers to MultiValuedColumnBridge (MVCB) then
5:       ReificationTable = GetReifTblForMVCB(ParentTable, ReificationNode, ReifiedPredicate, MapFile)
6:     else
7:       ReificationTable = GetReifTblForNonMVCB(ParentTable, ReificationNode, MapFile)
8:     end if
9:   else if ParentTable.Type = CLBN  then
10:    ReificationTable = TableMap corresponding to CLBN
11:  else if ParentTable.Type = S/CRBN  then
12:    ReificationTable = GetReifTblForRBN(ParentTable, ReificationNode, MapFile)
13:  else
14:    { /* ParentTable.Type = MultiValuedS/CL/RBNs */ }
15:    ReificationTable = TableMap corresponding to ParentTable
16:  end if
17:  for every entry of type r2d:ColumnBridge with r2d:BelongsToReificationNode = ReificationNode do
18:    if column does not exist in ReificationTable then
19:      ReificationTable.columns += column
20:    end if
21:  end for
22:  for every reification blankNode (except of type r2d:MixedBlankNode) with r2d:BelongsToReificationNode= ReificationNode do
23:    ReificationTable.columns += blankNode.columns (derived as discussed in Chapter 4)
24:  end for
25:  for every entry of type r2d:MixedBlankNode (containing nested blank nodes) do
26:    for every r2d:ColumnBridge belonging to any blank node in any path of the tree rooted at the r2d:MixedBlankNode do
27:      ReificationTable.columns += column
28:    end for
29:  end for
30: end for
```

Algorithm 6.4 GetReifTblForMVCB
Input: ParentTable, ReificationNode, MVCB, MapFile
Output: Reification Table

```
1: if MVCB is Literal MVCB or resource MVCB having N:M relationship with another
   Table then
2:   ReificationTable = Table corresponding to MVCB
3: else
4:   if all non-quad predicates (NQPs) of ReificationNode are either ColumnBridges (CB)
     or SLBNs or MixedBlankNodes (MBNs) then
5:     ReificationTable = Table on N-side of the relationship
6:   else
7:     { /* NQPs include blank nodes of type [MV]CLBN or [MV]C/SRBN */ }
8:     ReificationTable = CreateNewReificationTable() /* if one does not already exist */
9:     ReificationTable.Columns += ParentTable.PKField, += ParentTable.ReifiedField
10:    Tables += ReificationTable
11:  end if
12: end if
```

Algorithm 6.5 GetReifTblForNonMVCB
Input: ParentTable, ReificationNode, MapFile
Output: Reification Table

```
1: if all non-quad predicates (NQPs) of ReificationNode are either ColumnBridges (CB) or
   SLBNs or MixedBlankNodes (MBNs) then
2:   if ParentTable.Type = "Table" then
3:     ReificationTable = ParentTable
4:   else
5:     { /* ParentTable.Type = SLBN */ }
6:     ReificationTable = Parent Table to which SLBN belongs
7:   end if
8: else
9:   { /* NQPs include one or more blank nodes of type [MV]CLBN or [MV]C/SRBN */}
10:  ReificationTable = CreateNewReificationTable() /* if one does not already exist */
11:  ReificationTable.Columns += ParentTable.PKField, += ParentTable.ReifiedField
12:  Tables += ReificationTable
13: end if
```

Table 6.3: Conditions under which a new r2d:TableMap is created for reification data

Reified Entity/ Attribute	**Type of Relationship Represented (if any)**	**Non-Quad Predicate (NQP) Types in Reification Node**	**Is a New Reification Table Created?**
r2d:ColumnBridge or Predicate leading to a blank node in r2d:TableMap		ColumnBridges, SLBNs, and MixedBlankNodes	No
		[MV]CLBNs, [MV]{S/C}RBNs	Yes
r2d:MVCB in r2d:TableMap	MVCB is a resource MVCB representing 1:N relationship between subject, object pair	ColumnBridges, SLBNs, and MixedBlankNodes	No
		[MV]CLBNs, [MV]{S/C}RBNs	Yes
	MVCB is a resource MVCB representing N:M relationship between subject, object pair or MVCB is a Literal MVCB	All	No
Predicate belonging to an SLBN of a TableMap		ColumnBridges, SLBNs, and MixedBlankNodes	No
		[MV]CLBNs, [MV]{S/C}RBNs	Yes
Predicate belonging to a CLBN of a TableMap		All	No
Predicate belonging to an SRBN and CRBN of a TableMap	S/CRBN represents 1:N relationship between subject, object pair	ColumnBridges, SLBNs, and MixedBlankNodes	No
		[MV]CLBNs, [MV]{S/C}RBNs	Yes
	S/CRBN represents N:M relationship between subject, object pair	All	No
Predicate belonging to an MultiValuedS/CL/RBN		All	No

Algorithm 6.6 GETREIFTBLFORRBN

Input: ParentTable, ReificationNode, MapFile

Output: Reification Table

```
1: if S/CRBN represents 1:N relationship between the subject and object resources then
2:   if all non-quad predicates (NQPs) of ReificationNode are either ColumnBridges (CB)
     or SLBNs or MixedBlankNodes (MBNs) s then
3:     ReificationTable = TableMap on the N-side of the relationship
4:   else
5:     {/* NQPs include one or more blank nodes of type [MV]CLBN or [MV]C/SRBN */
       }
6:     ReificationTable = CreateNewReificationTable() /* if one does not already exist */
7:     ReificationTable.Columns += ParentTable.PKField, += ParentTable.ReifiedField
8:     Tables += ReificationTable
9:   end if
10: else
11:   {/* S/CRBN represents 1:N relationship between the subject, object pair */ }
12:   ReificationTable = TableMap corresponding to the S/CRBN
13: end if
```

of reification SLBN NQPs) is added as a column to the TableMap to which the blank node or SLBN belongs.

(b) *Simple/SLBN NQP reifies a predicate belonging to an CLBN or MultiValued{S/C}{L/R}BN of an r2d:TableMap:* In this case, the simple NQP or, in the case of reification SLBN NQPs, each predicate belonging to the reification SLBN, is simply added as a column to the table that results from the relational transformation of the CLBN or MultiValued{S/C}{L/R}BN. Details on the tables that result from the relational transformation of these *(CLBN, MultiValued{S/C}{L/R}BN)* blank nodes can be found in Chapter 4.

II. **Relationalization of Reification CLBN, SRBN, and CRBN NQPs (and their MultiValued counterparts):**

(a) *Reification CLBN/C/SRBN NQP reifies a simple predicate or blank node belonging to an r2d:TableMap:* In this case, reification CLBN and {C/S}RBN NQPs always results in the creation of a new reification table with foreign keys fields that reference the reified predicate as well as the primary key field of the r2d:TableMap to which the reified predicate belongs. The Phone predicate in Figure 3.4 is one such reification CLBN NQP that reifies a simple predicate belonging to the Offence resource and the relationalization of the same results in the creation of another tablemap, Offence_Victim_Phone_Reification, into which the columns corresponding to the reification CLBN (derived as detailed in Chapter 4) are added. Similarly, the Dependents predicate in Figure 3.4 is an example of a SRBN reification blank node NQP and its relationalization results in the creation of a new reification table called Offence_Victim_Dependents_Reification, into which the columns corresponding to the reification SRBN are added.

(b) *CLBN/C/SRBN reifies a predicate belonging to a blank node or a predicate associated with an MVCB representing an N:M relationship:* If the reified predicate belongs to an SLBN or a {S/C}RBN that represents 1:N relationship between the {subject, object} resource pair then the procedure detailed in (II) (a) above is followed. If the reified predicate belongs to any other type of blank node, the predicates belonging to the reification CLBN/{C/S}RBN NQPs are simply added as columns (columns for these reification blank node NQPs are derived in the same manner as the columns for non-reification blank nodes and the details of the same can be found in Chapter 4) to the table that results from the relational transformation of the blank node to which the reified predicate belongs.

III. **Relationalization of Reification MixedBlankNodes:**

Reification blank nodes of type r2d:MixedBlankNodes, where the non-quad predicates include one or more (nested) blank nodes, are handled by flattening their relational equivalents as, due to the numerous types of such mixed combinations that are possible, it would be nearly impossible to arrive at an accurate normalized representation of the same. As a result, predicates of literal and resource objects that are at the leaf nodes of the tree rooted at the r2d:MixedBlankNode are simply translated into columns in the virtual table that results from the relational transformation of the tablemap or the blank node to which the parent reification node (of which this mixed blank node is an NQP) belongs as listed in Algorithm 6.3.

6.3 Querying Reification Data – SQL-to-SPARQL Translation

The final stage of the R2D transformation framework involves the translation of SQL statements issued against the virtual relational schema generated by stage 2 into equivalent SPARQL queries that are executed against the actual RDF store. This is achieved through the SQL-to-SPARQL translation algorithm, which in addition to performing the SQL-to-SPARQL translation, also ensures that the triples retrieved from the RDF store are returned to the relational visualization tool in the expected tabular format. The translation algorithm presented in this chapter extends the earlier version presented in Chapter 4 by including the ability to translate queries issued against the virtual tables housing reification data corresponding to reification blank nodes.

The SQL-to-SPARQL translation process transforms single or multiple table queries with or without multiple where clauses (connected by AND, OR, or NOT operators) and Group By clauses. Within each individual where clause, the algorithm handles operators in the following set $>, <, =, <=, >=, ! =$, LIKE. Due to the length and complexity of the algorithm, only a high level description of the algorithm is discussed below along with examples to illustrate the translation process.

In order to understand the intricacies of the translation algorithm, let us consider the following SQL query based on the relational schema scenario depicted in Figure 3.5.

SELECT offence_victim, victim_gender, victim_phone_type, victim_phone_value, dependent_ fullname, reportingOfficers_badge, reportingOfficers_badge_officerName, reportingOfficerAddress_Street

FROM Offence, Offence_Victim_Phone_Reification, Offence_Victim_Dependent_Reification, Dependent, Offence_ReportingOfficers

WHERE Offence.Offence_pk = Offence_Victim_Phone_Reification.Offence_pk

AND Offence.Offence_victim = Offence_Victim_Phone_Reification.Offence_victim

AND Offence.Offence_pk = Offence_Victim_Dependent_Reification.Offence_pk

AND Offence.Offence_victim = Offence_Victim_Dependent_Reification.Offence_victim

AND Offence_Victim_Dependent_Reification.Dependent_Pk = Dependent.Dependent_Pk

AND Offence.Offence_Pk = Offence_ReportingOfficers.Offence_pk

AND Offence.Offence_Victim = 'DummyLName, ABC';

The first step in the translation process involves the generation of the SPARQL SELECT clause. For every field in the original SQL SELECT list, a variable is added to the SPARQL SELECT list. The SPARQL SELECT list after fields processing is as follows: *SPARQLSelect = SELECT ?offence_victim, ?victim_gender, ?victim_phone_type, ?victim_phone_value, ?dependent_fullname, ?reportingOfficers_badge, ?reportingOfficers_badge_officerName, ?reportingOfficerAddress_street*

The processing of regular columns for generation of SPARQL WHERE and FILTER clauses is described in Chapter 4. The resulting SPARQL WHERE clause after processing of regular, non-reification columns as detailed in Chapter 4 is as follows:

SPARQLWhere = WHERE {

?Offence <http://Offence/Victim> ?offence_victim .

?Offence <http://Offence/ReportingOfficers> ?Offence_ReportingOfficers .

?Offence_ReportingOfficers <http://Offence/ReportingOfficers/Badge> ?reportingOfficers_-badge .

?Dependent <http://Dependent/FullName> ?dependent_fullname .

SPARQLFilter = FILTER (?offence_victim = "DummyLName, ABC") }

Processing of fields corresponding to simple reification predicates and predicates belonging to reification blank nodes is as follows.

(a) **Processing of Simple Reification NQPs when Reification Quad reifies a Resource:** If the reification quad to which the field corresponding to the simple NQP belongs reifies a resource, clauses of the form

[OPTIONAL] { ?reificationQuad <rdf:subject> ?resourceTableMap .
?reificationQuad <rdf:predicate> ?reificationQuad.r2d:ReifiedPredicate .
?reificationQuad <rdf:object> ?reifiedObjectField .
?reificationQuad <non-quadPredicate> ?reificationColumn .}

are added to the SPARQL WHERE clause. The reification quad corresponding to the *victim_gender* column is one such reification. The OPTIONAL keyword is optional and is only required for queries involving outer joins. Also, if the field corresponding to the object being reified is not part of the SPARQL WHERE clause, an appropriate selection clause is added to the same. The SPARQL WHERE clauses resulting from the processing of the *victim_gender* column are:

REIFClause1 = ?Offence <http://Offence/Victim> ?offence_victim .
?Victim_Reif <rdf:subject> ?Offence .
?Victim_Reif <rdf:Predicate> <http://Offence/Victim> .
?Victim_Reif <rdf:Object> ?offence_victim .
?Victim_Reif <http://Offence/Victim/Gender> ?victim_gender.

Processing of reification columns belonging to {Literal/Resource}MultiValuedColumnBridge ({L/R}MVCB) tables is similar to the above case with an additional step to identify the parent table from which the {L/R}MVCB table is derived through normalization.

In the case of RMVCB tables where the rdf:object of the reification quad is a resource that maps to another r2d:TableMap (through the r2d:refersToTableMap construct), an additional clause of the form

?subjectResourceTableMap <reificationQuad.r2d:ReifiedPredicate> ?objectResourceTableMap .

is added to the SPARQL WHERE clause.

(b) **Processing of Simple Reification NQPs when Reification Quad reifies a Blank Node:** If the reification quad to which the field corresponding to the simple NQP belongs reifies a blank node, clauses of the form given below are added to the SPARQL WHERE clause. Further, if the rdf:object of the reification quad is a resource mapping to another r2d:TableMap then the following additional clause of the form *?BlankNode <reificationQuad.r2d:ReifiedPredicate> ?objectResourceTableMap .* is appended to the SPARQL WHERE Clause.

?ParentTableofBlankNode <BlankNodePredicate> ?BlankNode .
[OPTIONAL] {*?reificationQuad <rdf:subject> ?BlankNode .*
?reificationQuad <rdf:predicate> ?reificationQuad.r2d:ReifiedPredicate .
?reificationQuad <rdf:object> ?reifiedObjectField .
?reificationQuad <non-quadPredicate> ?reificationColumn }

The *reportingOfficers_officerName* reification column belonging to the reification node that reifies the *"ReportingOfficers"* CLBN in Figure 3.4 is an example of such a reification and the addition to the SPARQL WHERE clause after processing of the same is as given below.

REIFClause2 = Offence <http://Offence/ReportingOfficers> ? Offence_ReportingOfficers .
?RepOfficers_Officer_Reif <rdf:subject> ?Offence_ReportingOfficers .
?RepOfficers_Officer_Reif <rdf:Predicate> <http://Offence/ReportingOfficers/Badge> .
?RepOfficers_Officer_Reif <rdf:Object> ?reportingOfficers_badge .
?RepOfficers_Officer_Reif <http://Offence/ReportingOfficers/Badge/OfficerName> ?reportingOfficers_badge_officerName .

(c) **Processing of Reification Blank Node NQPs when Reification Quad reifies a Resource:** If the reification quad to which the reification blank node belongs reifies a resource (and not a blank node), clauses of the form

[OPTIONAL] { ?reificationQuad <rdf:subject> ?resourceTableMap .
?reificationQuad <rdf:predicate> ?reificationQuad.r2d:ReifiedPredicate .
?reificationQuad <rdf:object> ?reifiedObjectField .
?reificationQuad <reificationBlankNode predicate> ?reificationBlankNode .
?reificationBlankNode <ReificationColumn predicate> ?reificationColumn .}

are added to the SPARQL WHERE clause.
The reification quad that contains the *Phone* reification CLBN NQP and the *Dependents* reification SRBN NQP in Figure 3.4 is an example of the scenario described above and the corresponding SPARQL WHERE clauses generated to facilitate the selection of the *victim_phone_type*, *victim_phone_value*, and *dependent_pk* (and, through appropriate joins, *dependent_fullname*) fields are as follows.

REIFClause3 = ?Victim_Reif <rdf:subject> ? Offence .
?Victim_Reif <rdf:predicate> <http://Offfence/Victim> .
?Victim_Reif <rdf:object> ?offence_victim .

?Victim_Reif <http://Offence/Victim/Phone> ?Victim_Phone .

?Victim_Phone ?victim_phone_type ?victim_phone_value .

REIFClause4 = ?Victim_Reif <rdf:subject> ?Offence .

?Victim_Reif <rdf:predicate> <http://Offfence/Victim> .

?Victim_Reif <rdf:object> ?offence_victim .

?Victim_Reif <http://Offence/Victim/Dependents> ?Victim_Dependents .

?Victim_Dependents <http://Offence/Victim/Dependents/Dependent> ?victim_dependent_pk .

The join between the virtual relational tables *Dependent* and *Offence_Victim_Dependent_Reification* (required to obtain the *dependent_fullname* field value) is specified as an additional condition in the SPARQL FILTER clause which now becomes

SPARQLFilter = FILTER (?victim_dependent_pk = ?Dependent && ?offence_victim = "DummyLName, ABC")

(d) **Processing of Reification Blank Node NQPs when Reification Quad reifies a Blank Node:** If the reification quad to which the reification blank node belongs reifies a blank node, clauses of the form given below are added to the SPARQL WHERE clause. Further if the rdf:object of the reification quad to which the reification blank node belongs is a resource mapping to another r2d:TableMap then the following additional clause of the form *?BlankNode <reificationQuad.r2d:ReifiedPredicate> ?objectResourceTableMap .* is appended to the SPARQL WHERE Clause.

?ParentTableofReifiedBlankNode <ReifiedBlankNodePredicate> ?BlankNode .

[OPTIONAL] {?reificationQuad <rdf:subject> ?BlankNode .

?reificationQuad <rdf:predicate> ?reificationQuad.r2d:ReifiedPredicate .

?reificationQuad <rdf:object> ?reifiedObjectField .

?reificationQuad <reificationBlankNode predicate> ?reificationBlankNode .
?reificationBlankNode <reificationColumn predicate> ?reificationColumn .}
The reification quad to which the *OfficerAddress* reification SLBN belongs in Figure 3.4 is an example of one such reification. The SPARQL WHERE clauses resulting from the processing of the *reportingOfficerAddress_Street* field belonging to the *OfficerAddress* reification SLBN are as given below.

REIFClause5 = ?Offence <http://Offence/ReportingOfficers> ? Offence_ReportingOfficers .
?RepOfficers_Officer_Reif <rdf:subject> ?Offence_ReportingOfficers .
? RepOfficers_Officer_Reif <rdf:predicate> <http://Offence/ReportingOfficers/Badge> .
? RepOfficers_Officer_Reif <rdf:Object> ?reportingOfficers_badge .
? RepOfficers_Officer_Reif <http://Offence/ReportingOfficers/Address> ?reportingOfficer_Address .
?reportingOfficer_Address <http:/Offence/ReportingOfficers/Address/Street> ?reportingOfficerAddress_Street .

(e) **Translation of complex reification blank nodes:** : For fields belonging to complex reification blank nodes (i.e., mixed blank nodes), the sequence of predicates leading from the complex reification blank node to the (leaf) field are obtained by traversing the tree structure stored during the map file generation process. A WHERE clause is added to the SPARQL WHERE for each of the predicates in sequence.
After the translation procedures described above are applied to the given example SQL statement, the final transformed SPARQL Query is:

SPARQL Statement = SPARQLSelect + SPARQLWhere + REIFClause1 = REIFClause2 + REIFClause3 + REIFClause4 + REIFClause5 + SPARQLFilter

The transformed SPARQL Query is executed and the retrieved data is returned in relational format seamlessly.

CHAPTER 7
R2D EXPERIMENTAL RESULTS FOR RDF REIFICATION DATA

The hardware used for our simulation exercises was a Windows machine with 4GB RAM and 2 GHz Intel Dual Core processor. The software platforms and tools used include Jena 2.5.6 (McBride 2002) to manipulate the RDF triples data, MySQL 5.0 (MySQL) to house the RDF data in a persistent manner, and DataVision v1.2.0 (DataVision), an open source relational tool, to visualize, query, and generate reports based on the RDF data. Lastly, BEA Workshop Studio 1.1 Development Environment along with Java 1.5 was used for the development of the algorithms and procedures detailed in Chapter 6. The implementation of R2D and the performance experiments conducted using the same are described below.

7.1 Experimental Datasets

The dataset is used in the experiments below is an augmented version of a subset of crime data downloaded from a police department website. The data has triples pertaining to cities and zip codes where crimes were committed, and details of committed crimes as illustrated in Figure 3.4. The DataVision screenshots include actual, valid crime data in all cases except for the data corresponding to the reification blank nodes. This reification data is synthetic and was introduced primarily to illustrate the relationalization process for reification blank nodes. Further, the voluminous datasets used in the query performance evaluations were also artificially generated through a data loading program and the structure of this dataset is a sub-set of the structure illustrated in Figure 3.4. The structure of the simulated data was kept identical to that of the actual original crime dataset from the police department in order to facilitate the extrapolation of the obtained results to actual crime data of those volumes. For query performance experiments, Jena's in-memory model was used to load and query the data.

7.2 Simulation Results

The relational equivalent of the crime data was generated using the RDFMapFileGenerator and DBSchemaGenerator Algorithms detailed in Chapter 6. The time taken by the map file generation process without any data sampling incorporated for RDF stores of various sizes, with and without reification information, was compared with time taken for the same process when several sampling methods (Convenience sampling, Systematic Sampling) were applied and the results are illustrated in Figure 7.1. Reified versions of the crime dataset were created by adding reification information to the *Address (Address_Type)* and *Victim (Gender, Race, Age)* objects in Figure 3.4. This reification information was created for 50% of the offence data in the data stores.

The process is especially time-intensive for large databases without structural information (which is the case with our experimental data set) but this is only to be expected since the RDFMapFileGenerator has to explore every resource to ensure that no property is left unprocessed. Furthermore, since even adding reification information for only 50% of the triples in the RDF store resulted in a 25% increase in the size of the data store, the increase in map file generation time for databases with reification information is also predictable. However, the sampling techniques applied improved the performance of the algorithm by a large factor.

Figure 7.2 is a screenshot of the database schema as seen by DataVision. The r2d:Simple-LiteralBlankNode associated with *Offence-Address* is resolved into columns belonging to the Offence table, and the r2d:ComplexLiteralBlankNode associated with *Offence-ReportingOfficers* is resolved into a 1:N table of the same name. Further, the simple reification NQPs *(Victim_Gender, Victim_Race, Victim_Age, OffenceAddress_AddressType)* are resolved into columns in the parent table *(Offence)* and the SimpleLiteralBlankNode reification NQP, *Offence_ReportingOfficers_OfficerAddress*, is resolved into columns in the *Offence_ReportingOff-*

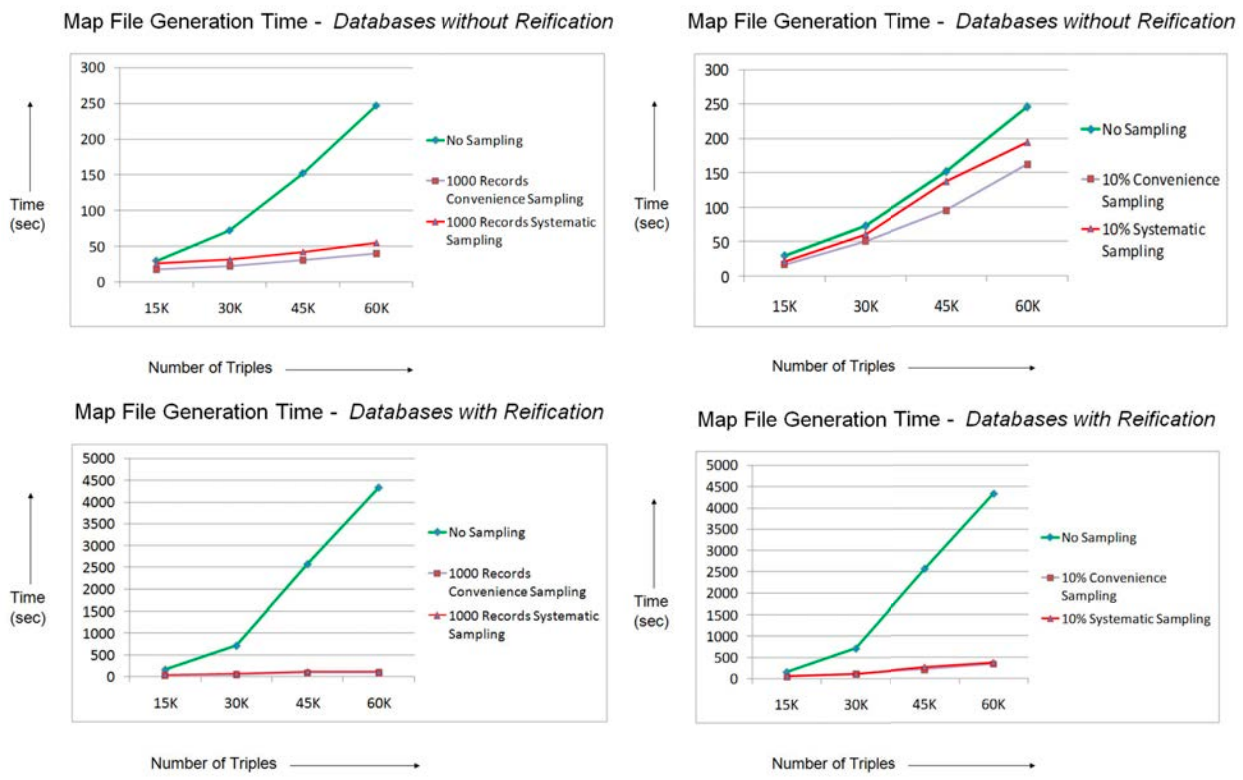


Figure 7.1: Map File Generation Times with/without Sampling for Reified/Un-reified Data

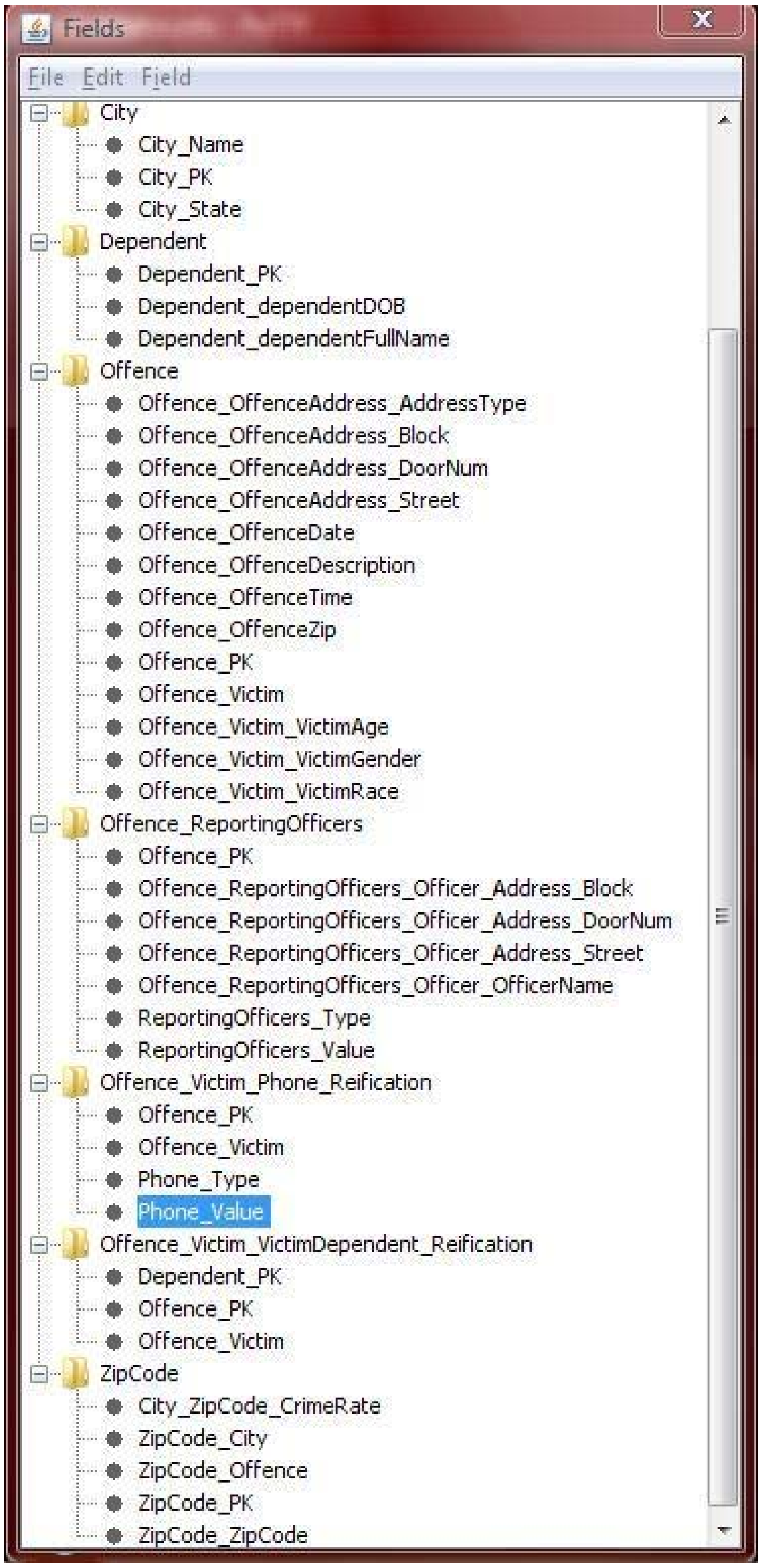
icers CLBN table as discussed in Chapter 6, Section 6.2. The reification ComplexLiteral and SimpleResource blank nodes corresponding to *Offence_Victim_Phone* and *Offence_Victim_-Dependents* predicates are resolved into columns in 1:N *(Offence_Victim_Phone_Reification)* and N:M *(Offence_Victim_Dependents_Reification)* tables respectively.

This schema is populated through the GetDatabaseMetaData Interface in the Connection class of the JDBC API within which the two algorithms, RDFMapFileGenerator and DBSchemaGenerator, are triggered.
Figure 7.3 illustrates DataVision's Report Designer Window and query building process for a sample query involving the reification columns of *offence_victim_gender* and *offence_victim_-dependent_pk* (used to derive the *Dependent_FullName* value).

Based on the fields chosen (in the "Report Designer" window), the table linkages (i.e., joins, illustrated in the "Table Linker" inset) specified, and additional record selection criteria specified (illustrated in the "Record Selection Criteria" inset respectively), DataVision generates an appropriate SQL query, as shown in the "SQL Query" inset in Figure 7.4, to extract the required data. At this juncture, the Statement, the Prepared Statement, and the ResultSet JDBC Interfaces are invoked. These, in turn, trigger the SQL-to-SPARQL Translation algorithm, which generates a SPARQL equivalent of the given SQL statement as also illustrated in Figure 7.4, and return the obtained results to DataVision in the expected tabular format, as illustrated in Figure 7.5.

Figures 7.6 and 7.7 include screenshots of SQL-to-SPARQL translations and query results as seen by DataVision resulting from two more SQL queries involving reification CLBN and SLBN NQPs respectively, executed, through DataVision, on the virtual relational schema illustrated in Figure 7.2.

In order to study the performance impact incurred by reification, a selection of 4 queries, each of which consisted of two versions, were executed on simulated crime datasets of various sizes. The second version of each was created by including one or more reification fields to the first version. Figure 7.8 displays the response times of each of the queries as the sizes of the databases vary. As was anticipated, reification adds overheads to query processing times



City
- City_Name
- City_PK
- City_State

Dependent
- Dependent_PK
- Dependent_dependentDOB
- Dependent_dependentFullName

Offence
- Offence_OffenceAddress_AddressType
- Offence_OffenceAddress_Block
- Offence_OffenceAddress_DoorNum
- Offence_OffenceAddress_Street
- Offence_OffenceDate
- Offence_OffenceDescription
- Offence_OffenceTime
- Offence_OffenceZip
- Offence_PK
- Offence_Victim
- Offence_Victim_VictimAge
- Offence_Victim_VictimGender
- Offence_Victim_VictimRace

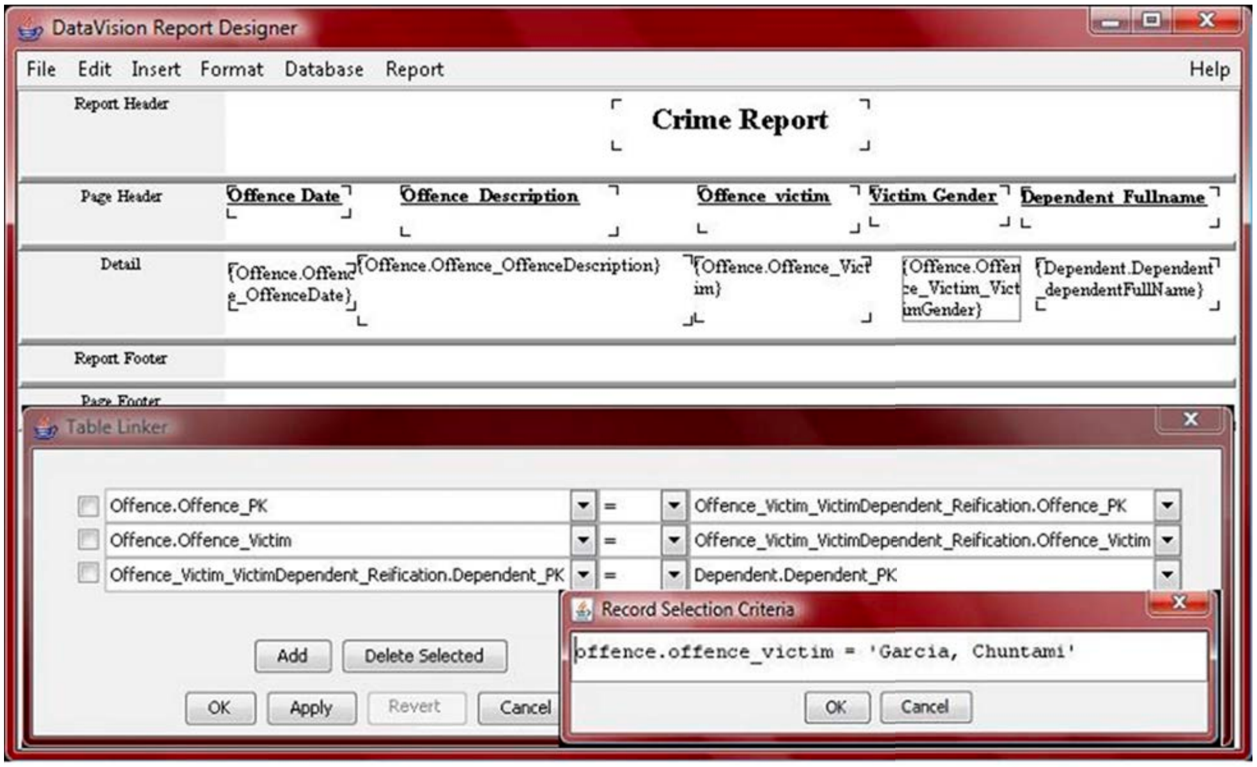
Offence_ReportingOfficers
- Offence_PK
- Offence_ReportingOfficers_Officer_Address_Block
- Offence_ReportingOfficers_Officer_Address_DoorNum
- Offence_ReportingOfficers_Officer_Address_Street
- Offence_ReportingOfficers_Officer_OfficerName
- ReportingOfficers_Type
- ReportingOfficers_Value

Offence_Victim_Phone_Reification
- Offence_PK
- Offence_Victim
- Phone_Type
- Phone_Value

Offence_Victim_VictimDependent_Reification
- Dependent_PK
- Offence_PK
- Offence_Victim

ZipCode
- City_ZipCode_CrimeRate
- ZipCode_City
- ZipCode_Offence
- ZipCode_PK
- ZipCode_ZipCode

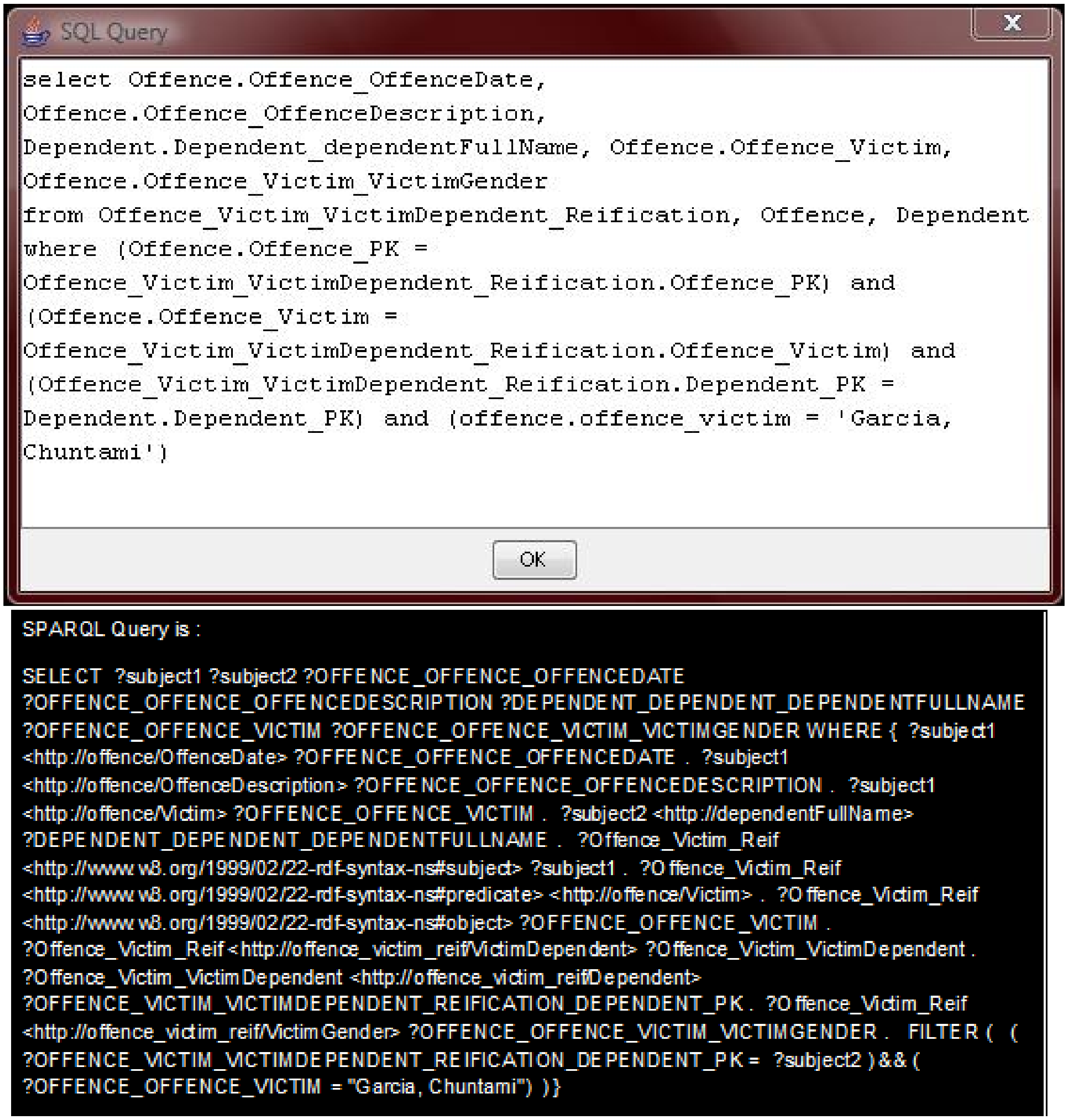
Figure 7.2: Equivalent Relational Schema as seen through DataVision


DataVision Report Designer
File Edit Insert Format Database Report Help
Report Header
Crime Report
Page Header
Offence Date
Offence Description
Offence victim
Victim Gender
Dependent Fullname
Detail
{Offence.Offence_OffenceDate}
{Offence.Offence_OffenceDescription}
{Offence.Offence_Victim}
{Offence.Offence_Victim_VictimGender}
{Dependent.Dependent_dependentFullName}
Report Footer
Page Footer
Table Linker
Add
Delete Selected
OK
Apply
Revert
Cancel
Record Selection Criteria
OK
Cancel

☐	Offence.Offence_PK	=	Offence_Victim_VictimDependent_Reification.Offence_PK
☐	Offence.Offence_Victim	=	Offence_Victim_VictimDependent_Reification.Offence_Victim
☐	Offence_Victim_VictimDependent_Reification.Dependent_PK	=	Dependent.Dependent_PK

```
offence.offence_victim = 'Garcia, Chuntami'
```

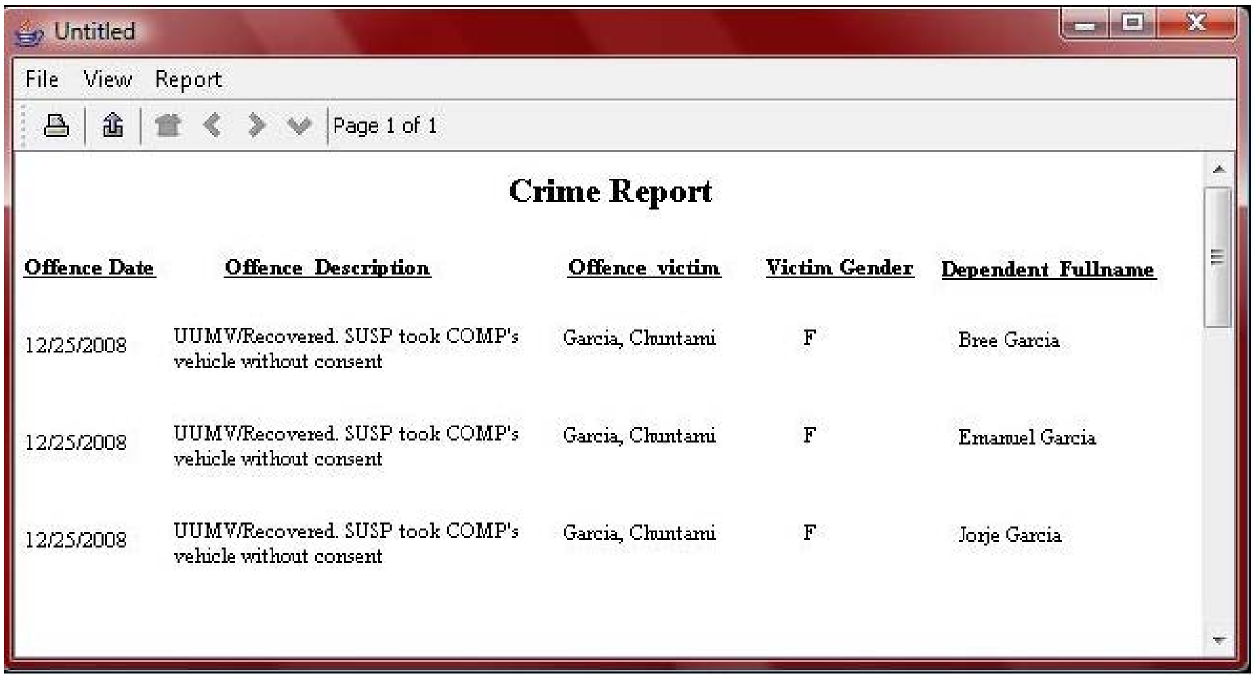
Figure 7.3: DataVision's Report Designer and Query Processing



```
select Offence.Offence_OffenceDate,
Offence.Offence_OffenceDescription,
Dependent.Dependent_dependentFullName, Offence.Offence_Victim,
Offence.Offence_Victim_VictimGender
from Offence_Victim_VictimDependent_Reification, Offence, Dependent
where (Offence.Offence_PK =
Offence_Victim_VictimDependent_Reification.Offence_PK) and
(Offence.Offence_Victim =
Offence_Victim_VictimDependent_Reification.Offence_Victim) and
(Offence_Victim_VictimDependent_Reification.Dependent_PK =
Dependent.Dependent_PK) and (offence.offence_victim = 'Garcia,
Chuntami')
```

```
SPARQL Query is :

SELECT  ?subject1 ?subject2 ?OFFENCE_OFFENCE_OFFENCEDATE
?OFFENCE_OFFENCE_OFFENCEDESCRIPTION ?DEPENDENT_DEPENDENT_DEPENDENTFULLNAME
?OFFENCE_OFFENCE_VICTIM  ?OFFENCE_OFFENCE_VICTIM_VICTIMGENDER WHERE {  ?subject1
<http://offence/OffenceDate> ?OFFENCE_OFFENCE_OFFENCEDATE .  ?subject1
<http://offence/OffenceDescription> ?OFFENCE_OFFENCE_OFFENCEDESCRIPTION .  ?subject1
<http://offence/Victim> ?OFFENCE_OFFENCE_VICTIM .  ?subject2 <http://dependentFullName>
?DEPENDENT_DEPENDENT_DEPENDENTFULLNAME .  ?Offence_Victim_Reif
<http://www.w3.org/1999/02/22-rdf-syntax-ns#subject> ?subject1 .  ?Offence_Victim_Reif
<http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate> <http://offence/Victim> .  ?Offence_Victim_Reif
<http://www.w3.org/1999/02/22-rdf-syntax-ns#object> ?OFFENCE_OFFENCE_VICTIM .
?Offence_Victim_Reif <http://offence_victim_reif/VictimDependent> ?Offence_Victim_VictimDependent .
?Offence_Victim_VictimDependent <http://offence_victim_reif/Dependent>
?OFFENCE_VICTIM_VICTIMDEPENDENT_REIFICATION_DEPENDENT_PK .  ?Offence_Victim_Reif
<http://offence_victim_reif/VictimGender> ?OFFENCE_OFFENCE_VICTIM_VICTIMGENDER .   FILTER (  (
?OFFENCE_VICTIM_VICTIMDEPENDENT_REIFICATION_DEPENDENT_PK =  ?subject2 ) && (
?OFFENCE_OFFENCE_VICTIM  = "Garcia, Chuntami")  ) }
```

Figure 7.4: SQL Query generated by DataVision and its equivalent SPARQL query as generated by R2D's SQL-to-SPARQL Translation Module



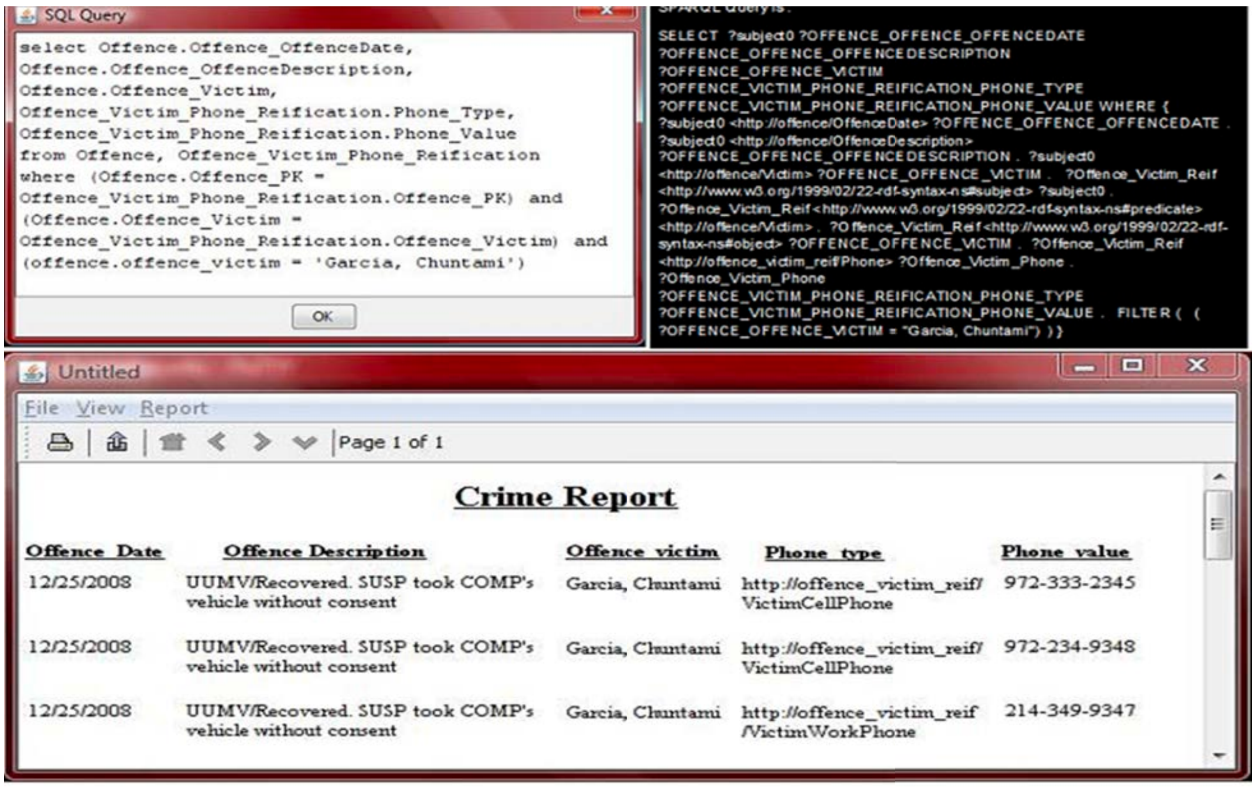
Crime Report

Offence Date	Offence Description	Offence victim	Victim Gender	Dependent Fullname
12/25/2008	UUMV/Recovered. SUSP took COMP's vehicle without consent	Garcia, Chuntami	F	Bree Garcia
12/25/2008	UUMV/Recovered. SUSP took COMP's vehicle without consent	Garcia, Chuntami	F	Emanuel Garcia
12/25/2008	UUMV/Recovered. SUSP took COMP's vehicle without consent	Garcia, Chuntami	F	Jorje Garcia

Figure 7.5: RDF Triples presented to DataVision in a Relational Tabular Format

as adding a reification quad for a triple results in the addition of a minimum of 4 to 5 extra triples to the data store. However, since the time taken for the SQL-to-SPARQL translation is negligible and nearly constant, R2D does not add any overheads due to reification to the SPARQL query performance.

As stated earlier, SQL queries issued against relational databases created by physically duplicating RDF data may possibly exhibit even better performance, however, this improved performance comes at the expense of additional disk space due to duplication of data, and additional system resources and human effort required to synchronize the data. R2D, on the other hand, circumvents duplication and synchronization issues and huge learning curves associated with mastering newer technologies by leveraging existing tools, knowledge, and expertise and skillfully integrating the old and the new in a cost-effective manner and with minimum overheads.



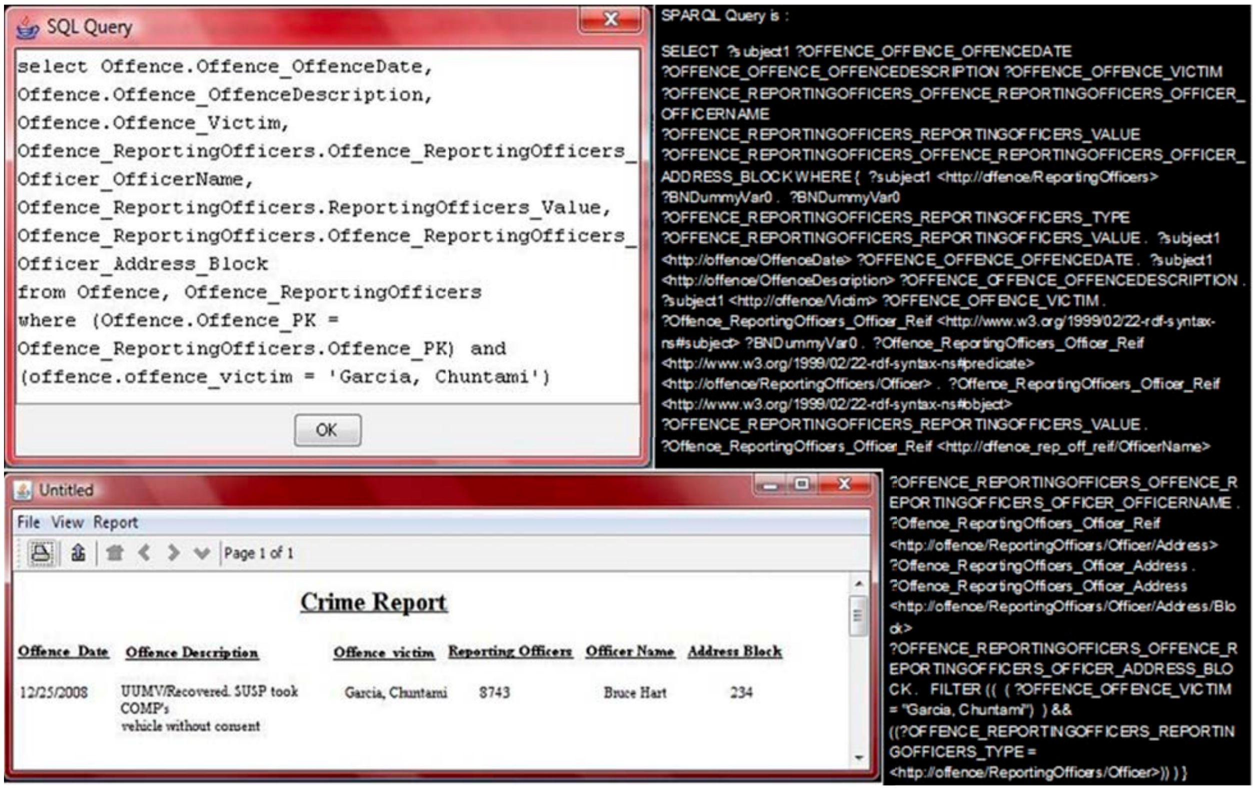
```
select Offence.Offence_OffenceDate,
Offence.Offence_OffenceDescription,
Offence.Offence_Victim,
Offence_Victim_Phone_Reification.Phone_Type,
Offence_Victim_Phone_Reification.Phone_Value
from Offence, Offence_Victim_Phone_Reification
where (Offence.Offence_PK =
Offence_Victim_Phone_Reification.Offence_PK) and
(Offence.Offence_Victim =
Offence_Victim_Phone_Reification.Offence_Victim) and
(offence.offence_victim = 'Garcia, Chuntami')
```

```
SELECT  ?subject0 ?OFFENCE_OFFENCE_OFFENCEDATE
?OFFENCE_OFFENCE_OFFENCEDESCRIPTION
?OFFENCE_OFFENCE_VICTIM
?OFFENCE_VICTIM_PHONE_REIFICATION_PHONE_TYPE
?OFFENCE_VICTIM_PHONE_REIFICATION_PHONE_VALUE WHERE {
?subject0 <http://offence/OffenceDate> ?OFFENCE_OFFENCE_OFFENCEDATE .
?subject0 <http://offence/OffenceDescription>
?OFFENCE_OFFENCE_OFFENCEDESCRIPTION .  ?subject0
<http://offence/Victim> ?OFFENCE_OFFENCE_VICTIM .   ?Offence_Victim_Reif
<http://www.w3.org/1999/02/22-rdf-syntax-ns#subject> ?subject0 .
?Offence_Victim_Reif <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
<http://offence/Victim> .  ?Offence_Victim_Reif <http://www.w3.org/1999/02/22-rdf-
syntax-ns#object> ?OFFENCE_OFFENCE_VICTIM .  ?Offence_Victim_Reif
<http://offence_victim_reif/Phone> ?Offence_Victim_Phone .
?Offence_Victim_Phone
?OFFENCE_VICTIM_PHONE_REIFICATION_PHONE_TYPE
?OFFENCE_VICTIM_PHONE_REIFICATION_PHONE_VALUE .  FILTER (  (
?OFFENCE_OFFENCE_VICTIM = "Garcia, Chuntami") )}
```

Crime Report

Offence Date	Offence Description	Offence victim	Phone type	Phone value
12/25/2008	UUMV/Recovered. SUSP took COMP's vehicle without consent	Garcia, Chuntami	http://offence_victim_reif/VictimCellPhone	972-333-2345
12/25/2008	UUMV/Recovered. SUSP took COMP's vehicle without consent	Garcia, Chuntami	http://offence_victim_reif/VictimCellPhone	972-234-9348
12/25/2008	UUMV/Recovered. SUSP took COMP's vehicle without consent	Garcia, Chuntami	http://offence_victim_reif/VictimWorkPhone	214-349-9347

Figure 7.6: SQL-to-SPARQL Translation and Tabular Results for Query involving Reification CLBN NQP (*Victim_Phone*)



```
select Offence.Offence_OffenceDate,
Offence.Offence_OffenceDescription,
Offence.Offence_Victim,
Offence_ReportingOfficers.Offence_ReportingOfficers_
Officer_OfficerName,
Offence_ReportingOfficers.ReportingOfficers_Value,
Offence_ReportingOfficers.Offence_ReportingOfficers_
Officer_Address_Block
from Offence, Offence_ReportingOfficers
where (Offence.Offence_PK =
Offence_ReportingOfficers.Offence_PK) and
(offence.offence_victim = 'Garcia, Chuntami')
```

```
SPARQL Query is :

SELECT ?subject1 ?OFFENCE_OFFENCE_OFFENCEDATE
?OFFENCE_OFFENCE_OFFENCEDESCRIPTION ?OFFENCE_OFFENCE_VICTIM
?OFFENCE_REPORTINGOFFICERS_OFFENCE_REPORTINGOFFICERS_OFFICER_
OFFICERNAME
?OFFENCE_REPORTINGOFFICERS_REPORTINGOFFICERS_VALUE
?OFFENCE_REPORTINGOFFICERS_OFFENCE_REPORTINGOFFICERS_OFFICER_
ADDRESS_BLOCK WHERE { ?subject1 <http://offence/ReportingOfficers>
?BNDummyVar0 . ?BNDummyVar0
?OFFENCE_REPORTINGOFFICERS_REPORTINGOFFICERS_TYPE
?OFFENCE_REPORTINGOFFICERS_REPORTINGOFFICERS_VALUE . ?subject1
<http://offence/OffenceDate> ?OFFENCE_OFFENCE_OFFENCEDATE . ?subject1
<http://offence/OffenceDescription> ?OFFENCE_OFFENCE_OFFENCEDESCRIPTION .
?subject1 <http://offence/Victim> ?OFFENCE_OFFENCE_VICTIM .
?Offence_ReportingOfficers_Officer_Reif <http://www.w3.org/1999/02/22-rdf-syntax-
ns#subject> ?BNDummyVar0 . ?Offence_ReportingOfficers_Officer_Reif
<http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
<http://offence/ReportingOfficers/Officer> . ?Offence_ReportingOfficers_Officer_Reif
<http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
?OFFENCE_REPORTINGOFFICERS_REPORTINGOFFICERS_VALUE .
?Offence_ReportingOfficers_Officer_Reif <http://offence_rep_off_reif/OfficerName>
?OFFENCE_REPORTINGOFFICERS_OFFENCE_R
EPORTINGOFFICERS_OFFICER_OFFICERNAME .
?Offence_ReportingOfficers_Officer_Reif
<http://offence/ReportingOfficers/Officer/Address>
?Offence_ReportingOfficers_Officer_Address .
?Offence_ReportingOfficers_Officer_Address
<http://offence/ReportingOfficers/Officer/Address/Blo
ck>
?OFFENCE_REPORTINGOFFICERS_OFFENCE_R
EPORTINGOFFICERS_OFFICER_ADDRESS_BLO
CK . FILTER (( ( ?OFFENCE_OFFENCE_VICTIM
= "Garcia, Chuntami") ) &&
((?OFFENCE_REPORTINGOFFICERS_REPORTIN
GOFFICERS_TYPE =
<http://offence/ReportingOfficers/Officer>)) ) }
```

Crime Report

Offence Date	Offence Description	Offence victim	Reporting Officers	Officer Name	Address Block
12/25/2008	UUMV/Recovered. SUSP took COMP's vehicle without consent	Garcia, Chuntami	8743	Bruce Hart	234

Figure 7.7: SQL-to-SPARQL Translation and Tabular Results for Query involving a Simple Reification NQP (*Officer_Name*) and a Reification SLBN NQP (*Officer_Address*)

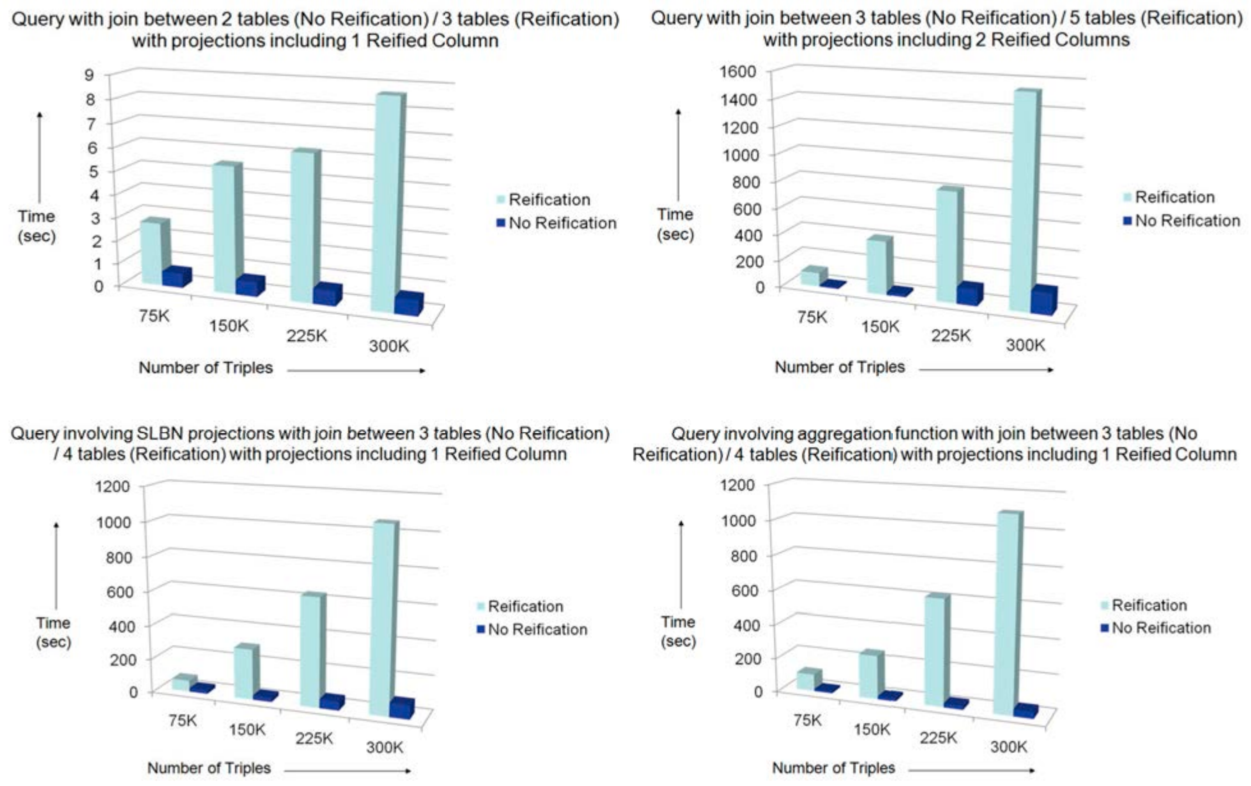


Figure 7.8: Response times for the chosen Queries

CHAPTER 8
SEMANTICS-PRESERVING TRANSLATION

As stated in earlier chapters, our approach, called R2D (RDF-to-Database), is a bridge that, without creating an explicit relational schema and duplicating data, enables existing traditional tools to work seamlessly with RDF Stores without necessitating extensive modifications or wasting valuable resources by replicating data unnecessarily. R2D is a JDBC wrapper around RDF stores that provides a relational interface to data stored in the form of RDF triples. The RDF Store is explored and mapped to a relational schema at run-time, and end-users of visualization tools are presented with the normalized relational version of the store on which they can perform operations as they would on an actual physical relational database schema. In a nutshell, R2D, whose commutative diagram is illustrated in Figure 8.1, consists of two major functionalities/modules the details of which are summarized below.

1. Schema Mapping: Automatic RDF-to-Relational Schema mapping utility that generates a domain-specific, normalized, virtual relational schema corresponding to a given RDF store.

2. Query Translation: An SQL-to-SPARQL query translation utility that takes an SQL statement as input, parses and converts it to a corresponding SPARQL statement, executes the same, and returns the results in a tabular format.

The focus of this chapter is to establish the accuracy of the R2D framework by proving the semantics-preserving nature of its translation modules. The importance of data integrity and accuracy and the impact of a lack of the same has been expounded in numerous publications (Redman 1998; Wang and Strong 1996; Lucas 2010; Cong, Fan, Geerts, Jia, and Ma 2007; Strong, Lee, and Wang 1997; Even and Shankaranarayanan 2007; Miettinen and Korhonen

$$\begin{tikzcd}
\text{RDF Graph, } G \arrow[r, "f(G)"] & \text{Relational Schema, } R \arrow[d] \\
\text{SPARQL Query, } q \arrow[u] & \text{SQL Query, } s \arrow[l, "h(s)"]
\end{tikzcd}$$

Figure 8.1: R2D Commutative Diagram

2008; Duarte, Salazar, Quintas, Santos, Neves, Abelha, and Machado 2010; Welzer, Brumen, Golob, and Druovec 2002). In general, the mildest effects of the absence of accurate data are increased operational costs and low customer satisfaction (Redman 1998; Wang and Strong 1996; Lucas 2010). Erroneous data, such as wrong pricing information in retail databases, cost consumers billions of dollars annually (Cong, Fan, Geerts, Jia, and Ma 2007; Strong, Lee, and Wang 1997). Further, they can also result in operational inefficiency and capital losses for organizations (Even and Shankaranarayanan 2007).

The more devastating effects of poor-quality/incorrect data are particularly observed in medical applications (Miettinen and Korhonen 2008) where impeccably accurate data is mandatory to ensure timely and effective treatment of patients (Duarte, Salazar, Quintas, Santos, Neves, Abelha, and Machado 2010). In fact, data accuracy is so vital in healthcare environments that having no data is preferrable to having inaccurate data (Welzer, Brumen, Golob, and Druovec 2002). Likewise, other mission critical applications such as Air Traffic Control, Banking, and Electronic Trading Systems, to name a few, can be catastrophically affected by inaccurate data resulting in heavy financial losses, loss of privacy, operational chaos, and, in extreme cases, even in loss of life. Under such scenarios, it becomes imperative to ensure that there is no loss of data integrity or accuracy during R2D's translation processes.

Because it is crucial that queries are answered correctly, it is important to have a formal guarantee that R2D's transformation process is actually semantics-preserving. In other words, we are required to prove that an SQL query, $\hat{sql}$, run over the translated relational schema, $\mathbb{R}$, obtained from an RDF Graph, $\mathbb{G}$, through R2D's schema mapping process, returns the same result that an equivalent SPARQL query, $\dot{spq}$, obtained by translating $\hat{sql}$ using R2D's query translation process, would when run on the original RDF graph, $\mathbb{G}$. Formally proving that R2D's query rewriting process is semantics-preserving is achieved in two steps, and correspondingly, using the following two theorems.

Theorem 1 $\mathcal{E}[\![f(\phi)]\!] = \phi$

Theorem 2 $\mathcal{S}[\![\hat{sql}]\!]\phi = \mathcal{Q}[\![h(\hat{sql})]\!]\phi$

where ϕ is the canonical form of an RDF graph, $\mathbb{G}$. This is the mathematical equivalent of Algorithm 4.1 in Chapter 4. $\mathcal{E}[\![\mathbb{R}]\!]$ represents the denotation of a relational schema, $\mathbb{R}$, and $f(\phi)$ is R2D's translation from the canonical form ϕ, of $\mathbb{G}$, to $\mathbb{R}$. Amongst the components comprising Theorem 2, $\mathcal{S}[\![\hat{sql}]\!]$ represents the denotation of an SQL query, $\hat{sql}$, $\mathcal{Q}[\![\dot{spq}]\!]$ represents the denotation of a SPARQL query, $\dot{spq}$, and $h(\hat{sql})$ is R2D's translation from $\hat{sql}$ to $\dot{spq}$.

From Theorems 1 and 2 it can be inferred that $\mathcal{S}[\![\hat{sql}]\!]\mathcal{E}[\![f(\phi)]\!] = \mathcal{Q}[\![h(\hat{sql})]\!]\phi$. In subsequent sections we present the various components comprising the two theorems and prove the theorems for a representative subset of RDF graphs and SQL language thereby establishing the soundness and completeness of both modules comprising R2D and, consequently, ensuring data integrity maintenance throughout the translation process.

8.1 Core Languages

In order to accomplish the task of proving that R2D's translation is semantics-preserving while, at the same time, keeping the proof reasonably small and readable, we restrict our attention to the canonical form of RDF graphs, detailed in section 8.1.1, and to a subset of the SQL-2003 query language (SQL2003), called the SQL core language, discussed in section 8.1.2, that is powerful enough to to express most queries in the language.

$$\mathbb{G} = E_{\mathbb{G}} \qquad \text{(RDF Graph)}$$

$$E_{\mathbb{G}} = V_{\mathbb{G}_{\mathcal{R}}} \cup V_{\mathbb{G}_{\mathcal{B}}} \times L_{\mathbb{G}} \times V_{\mathbb{G}}$$

$$V_{\mathbb{G}_{\mathcal{R}}} \subseteq \mathcal{I} - \mathcal{C},\ V_{\mathbb{G}_{\mathcal{B}}} \subseteq \mathcal{B},\ V_{\mathbb{G}_{\mathcal{L}}} \subseteq \mathcal{L},\ V_{\mathbb{G}_{\mathcal{C}}} \subseteq \mathcal{C}$$
$$V_{\mathbb{G}} \quad \subseteq \quad V_{\mathbb{G}_{\mathcal{R}}} \cup V_{\mathbb{G}_{\mathcal{B}}} \cup V_{\mathbb{G}_{\mathcal{L}}} \cup V_{\mathbb{G}_{\mathcal{C}}} \qquad \text{(Vertices in } \mathbb{G}\text{)}$$
$$L_{\mathbb{G}} = \mathcal{P} \cup \{\texttt{type}\} \qquad \text{(Edge Labels in } \mathbb{G}\text{)}$$

$$E_{\mathbb{G}} \subseteq \quad E_{\mathbb{G}_c} \cup E_{\mathbb{G}_a}$$
$$e_c \in E_{\mathbb{G}_c} = V_{\mathbb{G}_{\mathcal{R}}} \times \{\texttt{type}\} \times V_{\mathbb{G}_{\mathcal{C}}} \qquad \text{(Class Definition Edges)}$$
$$e_a \in E_{\mathbb{G}_a} = V_{\mathbb{G}_{\mathcal{R}}} \cup V_{\mathbb{G}_{\mathcal{B}}} \times \mathcal{P} - \{\texttt{type}\} \times V_{\mathbb{G}_{\mathcal{R}}} \cup V_{\mathbb{G}_{\mathcal{B}}} \cup V_{\mathbb{G}_{\mathcal{L}}} \qquad \text{(Property Edges)}$$

$$\mathcal{I} \qquad \text{(Internationalized Resource Identifiers)}$$
$$\mathcal{B} \qquad \text{(Blank Nodes)}$$
$$\mathcal{L} \qquad \text{(Literals)}$$
$$\mathcal{P} \qquad \text{(Non-RDFS Predicates)}$$
$$\mathcal{C} \subseteq \mathcal{I} \qquad \text{(RDFS Classes)}$$

Figure 8.2: Mathematical Model of an RDF Graph

8.1.1 RDF Graphs

Mathematical Model of RDF Graphs

We define an RDF Graph, $\mathbb{G}$, as an unordered set of edges of the form (v_1, l, v_2) where $v_1, v_2 \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{C}$ represent vertices comprising the resource, literal, and blank nodes in the graph and $l \in \mathcal{I}$ is the predicate of the triple associated with the edge. We also restrict our attention to graphs which do not contain reification information and where each resource in the graph is an instance of exactly one class. The complete mathematical model of an RDF Graph is as detailed in Figure 8.2.

Canonical Form of RDF Graphs

Without loss of generality, for simplicity, an arbitrary total ordering on classes in $\mathbb{G}$ is imposed in order to avoid arbitrary traveral of classes during the canonical form rewriting procedure and all edges going backwards between classes are reversed. Further, since RDF Graphs contain sub-trees that can be generalized or transformed into other sub-trees, to keep the proof tractable, without loss of generality, RDF graphs are restricted to a canonical form, called ϕ, which is arrived at by applying the following transformations to the RDF Graph, $\mathbb{G}$.

1. **Blank Nodes:** Edges that originate in a blank node are collapsed repeatedly by removing the edge and appending the edge label of the removed edge to the edge label of the edge that connects the blank node to its parent node.

2. **1:N Relationship Edges:** Edge labels of resource-to-resource edges (i.e., edges of the form (v_1, l, v_2) where both v_1 and v_2 are Internationalized Resource Identifiers or IRIs) that represent One-to-Many relationships between classes are prefixed with an indicator in order to ensure that the columns corresponding to the 1:N edges are added to the tables on the 1-side of the relationship (i.e., to tables corresponding to the classes to which the second vertices (v_2) belong).

3. **Multi-Valued Attribute (MVA) Edges:** Multi-Valued Attribute Edges (resource-to-literal edges) that have multiple literal object values for the same resource-predicate pair, i.e., edges that have multiple v_2 values for the same $v_1 - l$ pair, are removed from the RDF Graph. For every class-predicate pair ($c-p$ pair), where c is the class of which v_1 in a MVA edge, (v_1, l, v_2), is an instance, a new class, c', that is a concatenation of c and p is added to the canonical form, ϕ. Further for every MVA edge, (v_1, l, v_2), a new resource, r', is added to ϕ that is a concatenation of v_1 and v_2. Lastly, three new edges are added to ϕ in lieu the MVA edge, the first of which associates the new resource, r' to the new class, c' (as an instance), and the other two edges add the two

participating entities (v_1 and v_2) of the original MVA edge as attributes of the new resource r'.

4. **N:M Relationship Edges:** Transformation of resource-to-resource edges that represent Many-to-Many relationships between the classes to which v_1 and v_2 belong is very similar to the transformation of MVA edges with the principle difference being that the new class, c', created for each class-class pair (c_1, c_2 pair), where c_1 and c_2 are the classes of which v_1 and v_2 in a N:M edge, (v_1, l, v_2), are instances, is a concatenation of c_1 and c_2.

Complete details of the RDF Graph canonical form rewriting procedure are enumerated in Table 8.1.

Table 8.1. Canonical Form of RDF Graphs

Feature	**Transformation**
`Blank Nodes`	$\texttt{RemoveBNodes}(\mathbb{G}) = \mathbb{G} - \{(x, l_1, y), (y, l_2, z) \mid y \in \mathcal{B}\}$ $\cup\{(x, l_1\_l_2, z) \mid (x, l_1, y), (y, l_2, z) \in \mathbb{G}, y \in \mathcal{B}\}$ $\texttt{CollapseBNEdges} = \mathit{fix}(\texttt{RemoveBNodes})$

Continued on Next Page…

Table 8.1. – Continued

Feature	Transformation
1:N Relationship Edges	$\texttt{ToMany}(\mathbb{G}) = \{(c_1, c_2, l) \mid (x, l, y_1), (x, l, y_2), (x, \texttt{type}, c_1), (y1, \texttt{type}, c_2), (y_2, \texttt{type}, c_2) \in \mathbb{G}, y_1 \neq y_2\}$ $\texttt{ManyTo}(\mathbb{G}) = \{(c_1, c_2, l) \mid (x_1, l, y), (x_2, l, y), (x_1, \texttt{type}, c_1), (x_2, \texttt{type}, c_1), (y, \texttt{type}, c_2) \in \mathbb{G}, x_1 \neq x_2\}$ $\texttt{OnlyToMany}(\mathbb{G}) = \texttt{ToMany}(\mathbb{G}) - \texttt{ManyTo}(\mathbb{G})$ $\texttt{OnlyToManyEdges}(\mathbb{G}) = \{(x, l, y) \in \mathbb{G} \mid (c_1, c_2, l) \in \texttt{OnlyToMany}(\mathbb{G}), (x, \texttt{type}, c_1), (y, \texttt{type}, c_2) \in \mathbb{G}\}$ $\texttt{PrefixToManyEdges}(\mathbb{G}) = \mathbb{G} - \texttt{OnlyToManyEdges}(\mathbb{G}) \cup \{(x, 1N\_l, y) \mid (x, l, y) \in \texttt{OnlyToManyEdges}(\mathbb{G}), (x, \texttt{type}, c_1), (y, \texttt{type}, c_2) \in \mathbb{G}\}$
Multi-Valued Attribute Edges	$\texttt{MVAEdgeLabels}(\mathbb{G}) = \{(c, l) \mid (x, l, y_1), (x, l, y_2), (x, \texttt{type}, c) \in \mathbb{G}, y_1, y_2 \in \mathcal{L}, y_1 \neq y_2\}$ $\texttt{MVAEdges}(\mathbb{G}) = \{(x, l, y) \in \mathbb{G} \mid (c, l) \in \texttt{MVAEdgeLabels}(\mathbb{G}), (x, \texttt{type}, c) \in \mathbb{G}, y \in \mathcal{L}\}$ $\texttt{FixMVAEdges}(\mathbb{G}) = \mathbb{G} - \texttt{MVAEdges}(\mathbb{G}) \cup \{(x\_y, \texttt{type}, c\_l), (x\_y, c\_\texttt{type}, x), (x\_y, l, y) \mid (x, l, y) \in \texttt{MVAEdges}(\mathbb{G}), (x, \texttt{type}, c) \in \mathbb{G}\}$

Continued on Next Page…

Table 8.1. – Continued

Feature	Transformation
N:M Relationship Edges	$\texttt{MToM}(\mathbb{G}) = \texttt{ManyTo}(\mathbb{G}) \cap \texttt{ToMany}(\mathbb{G})$ $\texttt{MToMEdges}(\mathbb{G}) = \{(x, l, y) \in \mathbb{G} \mid (c_1, c_2, l) \in \texttt{MToM}(\mathbb{G}), (x, \texttt{type}, c_1), (y, \texttt{type}, c_2) \in \mathbb{G}\}$ $\texttt{FixManyToManyEdges}(\mathbb{G}) = \mathbb{G} - \texttt{MToMEdges}(\mathbb{G}) \cup \{(x\_y, \texttt{type}, c_1\_c_2), (x\_y, c_1\_\texttt{type}, x), (x\_y, l, y) \mid (x, l, y) \in \texttt{MToMEdges}(\mathbb{G}), (x, \texttt{type}, c_1), (y, \texttt{type}, c_2) \in \mathbb{G}\}$

8.1.2 SQL Core Language

The SQL core language consists of two clauses, the SELECT clause and the FROM clause, and the syntax, in the form of a BNF Grammar, and a mathematical model of the core language are illustrated in Figures 8.3 and 8.4. Restricting our attention to this core language incorporates the interesting parts of the proof for the whole language.

WHERE clauses are not expressed in this core language as the translation of the components of WHERE clauses that have syntactic equivalents in the SPARQL language to equivalent FILTER clauses is a straightforward syntactic transformation. Further, there are several features, listed in table 8.2, in the SQL language that are not supported in R2D as there is no equivalent yet in SPARQL. There may be ways of encoding these features, however, since it is expected that there might be future revisions to SPARQL that will allow easy incorporation of these additional unsupported features at some point in the future, preemptive encoding of these features is deemed unwarranted.

Table 8.2. Unsupported SQL Features

Feature
Sub-queries in Comparison Predicate
IN Operator
Sub-queries in IN Predicate
CASE Expressions
Sub-queries in SELECT clause
Window Functions such as RANK, DENSE_RANK, ROW_NUMBER
Spatial Data Functions such as TREAT
Multi-set Operators
Sequence Expression such as NEXTVAL

BNF Grammar of SQL Core Language

Mathematical Model of SQL Core Language

```
              ⟨query specification⟩ ::= SELECT⟨select list⟩⟨table expression⟩ AS ⟨table alias⟩

                     ⟨select list⟩⟩ ::= ⟨asterisk⟩|⟨select sublist⟩[{⟨comma⟩⟨select sublist⟩}...]
                   ⟨select sublist⟩ ::= ⟨numeric value expression⟩ AS ⟨identifier⟩
         ⟨numeric value expression⟩ ::= ⟨term⟩|⟨numeric value expression⟩⟨plus sign⟩⟨term⟩
                             ⟨term⟩ ::= ⟨numeric primary⟩|⟨term⟩⟨asterisk⟩⟨numeric primary⟩
                  ⟨numeric primary⟩ ::= ⟨parenthesized value expression⟩
                                       |⟨nonparenthesized value expression primary⟩

   ⟨parenthesized value expression⟩ ::= ⟨left paren⟩⟨numeric value expression⟩⟨right paren⟩
             ⟨nonparenthesized value
                 expression primary⟩ ::= ⟨unsigned integer⟩|⟨column reference⟩
                                       |⟨quote⟩⟨nonquote character string⟩⟨quote⟩

                 ⟨column reference⟩ ::= ⟨identifier⟩|⟨table name⟩⟨dot⟩⟨identifier⟩

                 ⟨table expression⟩ ::= ⟨from clause⟩
                      ⟨from clause⟩ ::= FROM⟨table reference list⟩
             ⟨table reference list⟩ ::= ⟨table reference⟩[{⟨comma⟩⟨table reference⟩}...]
                  ⟨table reference⟩ ::= ⟨table name⟩|⟨derived table⟩
                       ⟨table name⟩ ::= ⟨identifier⟩


                    ⟨derived table⟩ ::= ⟨query specification⟩
                      ⟨table alias⟩ ::= ⟨identifier⟩
```

Figure 8.3: SQL Core Language Grammar

$\widehat{sql} \in \mathbb{SQL} = (\mathbb{SL} \times \mathbb{TR}, \hat{t}_{al})$	(SQL Statement)
$\hat{s} \in \mathbb{SL} = * \mid \mathbb{SSL}$	(Select List)
$\hat{sl} \in \mathbb{SSL} = \widehat{nve}$ `AS` $\hat{cl}_{al}$	(Select Sublist)
$\widehat{nve} \in \mathbb{NVE} = \hat{e} \mid \widehat{nve} + \hat{e}$	(Numeric Value Expression)
$\hat{e} \in \mathbb{TERM} = \hat{np} \mid \hat{e} * \hat{np}$	(Term)
$\hat{np} \in \mathbb{NP} = \hat{n} \mid \hat{p}$	(Numeric Primary)
$\hat{p} \in \mathbb{PVE} = (\widehat{nve})$	(Paranthesized Value Expression)
$\hat{n} \in \mathbb{NPVE} = \hat{c} \mid \hat{i} \mid \hat{t}.\hat{cl}$	(Non-paranthesized Value Expression Primary)
$\rho_{\mathbb{TR}} = \hat{tr} :: . \mid \hat{tr} :: \rho_{\mathbb{TR}}$	(Table Reference List)
$\hat{tr} \in \mathbb{TR} = \hat{t} \mid \hat{dt}$	(Table Reference Element)
$\hat{dt} \in \mathbb{DT} = \widehat{sql}$	(Derived Table Reference)
$\hat{c} \in \mathbb{CHAR}$	(Character Element)
$\hat{i} \in \mathbb{INT}$	(Integer Element)
$\hat{cl},\ \hat{cl}_{al} \in \mathbb{COLUMN}$	(Column Identifier)
$\hat{t},\ \hat{t}_{al} \in \mathbb{TABLE}$	(Table Identifier)

Figure 8.4: Mathematical Model of SQL Core Language

Canonical Form of SQL Core Language

There are some syntaxes in the SQL language that are generalizations of other syntaxes. For example, a SELECT List item with column alias ($n\hat{v}e$ `AS` $\hat{cl}_{al}$) is a generalization of the scenario where a SELECT List item exists without column alias($n\hat{v}e$). Similarly, a sub-query with query alias ($\hat{sql} = (\mathbb{SL} \times \mathbb{TR}, \hat{t}_{al})$) generalizes the case where a query is specified without a query alias ($\hat{sql} = (\mathbb{SL} \times \mathbb{TR})$). Therefore, without loss of generality we assume that SQL queries are limited to those that incorporate generalized syntaxes wherever applicable and we assume that a rewriting procedure exists that simply assigns unique alias names to queries and SELECT list items that do not have an alias name.

8.2 Denotational Semantics

In this section, we describe the denotational semantics of the applicable components comprising Theorems 1 and 2.

8.2.1 Relational Schema Semantics

We define a relational schema as a partial function from table cells to data. A table cell is a triple of the form (τ, κ, w), where τ is a tuple consisting of a τ_{id}, representing a table identifier (or table name), and a τ_t, representing the table type (Regular, Multi-Valued Attribute, Many-to-Many Relationship table). κ is a tuple representing a column and it comprises a κ_{id}, representing a column identifier (or column name), and a κ_t, representing the type of column (Regular/Non-Key, Primary Key, Foreign Key). Lastly, w represents the row number of the cell. A complete mathematical model of a relational schema is given in Figure 8.5.

$$\mathbb{R} : \tau \times \kappa \times \omega \rightharpoonup \texttt{DATA}$$

$$\tau = \tau_{id} \times \tau_t$$

$$\tau_{id} = \texttt{Table Identifier}$$

$$\tau_t \in \{\tau_r, \tau_{mva}, \tau_{nm}, \tau_{tmp}, \tau_{tmp_{mva}}, \tau_{tmp_{nm}}\} \quad (\textit{Table Type})$$

$$\tau_r = \texttt{Regular Table}$$

$$\tau_{mva} = \texttt{MVA Table}$$

$$\tau_{nm} = \texttt{Join Table}$$

$$\tau_{tmp} = \texttt{Temporary Table}$$

$$\tau_{tmp_{mva}} = \texttt{Temporary MVA Table}$$

$$\tau_{tmp_{nm}} = \texttt{Temporary N:M Table}$$

$$\kappa = \kappa_{id} \times \kappa_t$$

$$\kappa_{id} = \texttt{Column Identifier}$$

$$\kappa_t \in \{\kappa_{pk}, \kappa_{fk}, \kappa_r\} \quad (\textit{Column Type})$$

$$\kappa_{pk} = \texttt{Primary Key Column}$$

$$\kappa_{fk} = \texttt{Foreign Key Column}$$

$$\kappa_r = \texttt{Non-key Column}$$

$$\omega = \texttt{Row Identifier}$$

Figure 8.5: Mathematical Model of a Relational Schema

Mathematical Model of a Relational Schema

Denotational Semantics of a Relational Schema

Our relational schema model contains tables that can be classified into two broad categories: regular tables and temporary tables. Denotational semantics of temporary table cells, identified by (τ, κ, w) and containing data d, are straightforward with each Non-1N cell resulting in an edge of type (t_{id}, k_{id}, d) and each 1N cell resulting in an edge of type (d, k_{id}, t_{id}), where t_{id}, k_{id}, and d represent the table identifier, the column identifier, and the data element in the cell at row w, column k_{id} in table t_{id}. For MVA and NM Tables, each of which is comprised of two columns, a type column $(t'_{id}\texttt{\_type}, \kappa_{pk})$ and a non-type column (k'_{id}, κ_{pk}), the Non-Type cells generate no additional edges while the Type cells generate two edges of the form $\{(v_1, l, v_2), (v_1', l', v_2')\}$, where vertices v_1 and v_1' of each edge are a concatenation of the data, d and d', in the two columns in row w of the MVA/NM table, the edge labels, l, of the two edges are the type $(t'_{id}\texttt{\_type})$ and the non-type (k'_{id}) column identifiers of the two columns comprising the MVA/NM table, and vertices v_2 and v_2' of the two edges are the data elements, d and d', in the two table cells in row w. In the case of regular tables, Key Column cells result in an edge of type $(d, \texttt{type}, t_{id})$, where `type` represents the primary key column, Non-Key Non-1N cells result in edges of type (d', k_{id}, d), and Non-Key 1N cells result in edges of type (d, k_{id}, d'), where d' is the value in the Key Column (primary key) cell of the table identified by t_{id}. The complete set of denotational semantics for a relational schema is listed in Figures B.1 and B.2.

8.2.2 SQL Semantics

We define our SQL Query as a function from graphs to graphs. Our SQL query model consists of two sets, the SELECT items (Columns) set and the FROM items (Tables) set. The Columns set contains character and integer constants, columns from tables or sub-queries, and arithmetic expressions. The Tables set contains simple tables and derived tables or sub-queries. SQL queries are modeled as a tuple, one element of which a set of tuples that

$$\frac{}{\mathcal{E}[\![\emptyset]\!] = \emptyset}(\textsc{EmptySchema})$$

$$\frac{\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_r) \quad k = (k_{id}, \kappa_{pk})}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d, \texttt{type}, t_{id})\} \cup \phi)}(\text{KCRT})$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_r) \quad k = (k_{id}, k_t) \quad k_t \neq \kappa_{pk} \\ \texttt{1NPrefixExists}(k_{id}) = \texttt{false} \quad \mathbb{R}(t, (t_{id}\_\texttt{type}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d', k_{id}, d)\} \cup \phi}(\text{NKCN1NRT})$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_r) \quad k = (k_{id}, k_t) \quad k_t \neq \kappa_{pk} \\ \texttt{1NPrefixExists}(k_{id}) = \texttt{true} \quad \mathbb{R}(t, (t_{id}\_\texttt{type}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d, k_{id}, d')\} \cup \phi}(\text{NKC1NRT})$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{mva}) \quad t_{id} = t'_{id}\_p \quad k = (k_{id}, \kappa_{pk}) \\ k_{id} = t'_{id}\_\texttt{type} \quad \mathbb{R}(t, (k'_{id}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d\_d', \texttt{type}, t_{id}), (d\_d', k_{id}, d)\} \cup \phi}(\text{MVATTC})$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{mva}) \quad t_{id} = t'_{id}\_p \\ k = (k_{id}, \kappa_{pk}) \quad k_{id} \neq t'_{id}\_\texttt{type} \quad \mathbb{R}(t, (t'_{id}\_\texttt{type}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d'\_d, k_{id}, d)\} \cup \phi}(\text{MVATNTC})$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{nm}) \quad t_{id} = t_{id1}\_t_{id2} \quad k = (k_{id}, \kappa_{pk}) \\ k_{id} = t_{id1}\_\texttt{type} \quad \mathbb{R}(t, (k'_{id}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d\_d', \texttt{type}, t_{id}), (d\_d', k_{id}, d)\} \cup \phi}(\text{NMTTC})$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{nm}) \quad t_{id} = t_{id1}\_t_{id2} \\ k = (k_{id}, \kappa_{pk}) \quad k_{id} \neq t_{id1}\_\texttt{type} \quad \mathbb{R}(t, (t_{id1}\_\texttt{type}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d'\_d, k_{id}, d)\} \cup \phi}(\text{NMTNTC})$$

Figure 8.6: Relational Schema Denotational Semantics - Non-Temporary Tables

$$\frac{\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{tmp}) \quad k = (k_{id}, k_t) \quad \texttt{1NPrefixExists}(k_{id}) = \texttt{false}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(t_{id}, k_{id}, d)\} \cup \phi}\text{(TmpTN1N)}$$

$$\frac{\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{tmp}) \quad k = (k_{id}, k_t) \quad \texttt{1NPrefixExists}(k_{id}) = \texttt{true}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d, k_{id}, t_{id})\} \cup \phi}\text{(TmpT1N)}$$

$$\frac{\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{tmp_{mva}}) \quad k = (k_{id}, k_t)}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(t_{id}, k_{id}, d)\} \cup \phi}\text{(MVATmpT)}$$

$$\frac{\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{tmp_{nm}}) \quad k = (k_{id}, k_t)}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(t_{id}, k_{id}, d)\} \cup \phi}\text{(NMTmpT)}$$

Figure 8.7: Relational Schema Denotational Semantics - Temporary Tables

represent the crossproduct of the elements of the SELECT and FROM sets and the other is the alias name of the query. Although there were other ways, such as structures, to model SQL queries, we chose to represent SQL queries as crossproducts of SELECT and FROM set elements because modeling SQL queries as a structure unnecessarily clutters the proof with mathematical machinery that only serves to unpack the structure. Thus, modelling the SQL queries as crossproducts of SELECT and FROM sets alleviates the burden of processing list structures in the semantics.

Denotational Semantics of an SQL Query

The denotation of a simple table, t_{id}, in an SQL FROM set, on an input graph ϕ, is a forest where every tree in the forest has a height of 1. There is one tree in the forest for every primary key column value in $\hat{t}$ in ϕ, i.e., for every v_1 of (v_1, l, v_2) where $v_2 = \hat{t}$ and $l = \texttt{type}$, and the root of each tree, which serves as a row identifier, is set to the value of the primary key, i.e., $r_{id} = v_1$. The edges of each tree in the forest are attributes of the primary key column value that forms the root of the tree. In other words every edge, (v_1', l', v_2'), where $v_1' = r_{id}$ and $l' \neq \texttt{type}$, is added, with the label, l', prefixed with the table name, to the tree rooted at r_{id}. Lastly, in the case of regular (non-MVA and non-N:M tables) the primary key column value is also added as an edge to its corresponding tree. Thus, each tree in the forest represents a row in the table identified by t_{id}.

The denotation of a table's column ($\hat{t}.\hat{cl}$ `AS` $\hat{cl}_{al}$) in an SQL SELECT set is the edges in the forest comprising the denotation of table $\hat{t}$ that are labeled with the tag $\hat{t}\_\hat{cl}$, the edge labels of which are subsequently renamed with the column alias name, $\hat{cl}_{al}$. The denotation of a constant (character or integer) SELECT element ($\hat{c}$ `AS` $\hat{cl}_{al}$) from a table in the FROM set is simply the addition of n edges of the form (v_1, l, v_2) where n is the number of trees in the forest comprising the denotation of the table from which the constant is to be selected. Thus, a constant SELECT element results in the addition of one edge, (v_1, l, v_2), per tree where v_1, the row identifier, is the root of the tree, i.e., $v_1 = r_{id}$, and $l = \hat{cl}_{al}$, and $v_2 = \hat{c}/\hat{\imath}$.

$$\mathbb{T} \subseteq \mathbb{TR}$$
$$i : \mathbb{T} \to \mathbb{N}$$
$$\hat{sql} = \{\hat{s}_1 + \hat{s}_2\} \times \mathbb{T}$$
$$\forall \hat{t} \in \mathbb{T}, \quad \phi_{i(\hat{t})} = \mathcal{S}[\![\{(\hat{s}_1, \hat{t}), (\hat{s}_2, \hat{t})\}]\!]$$
$$\phi = \{\sum_{j \in i^{\rightarrow}} \pi_3 x_j \mid (x_1, x_2, \ldots, x_{|\mathbb{T}|}) \in \phi_1 \times \phi_2 \times \ldots \times \phi_{|\mathbb{T}|}\}$$

Figure 8.8: Denotation of a SELECT element that is an Arithmetic Addition Expression

The denotation of the result of an SQL query is a graph with a height of 2 where the root node is a string comprising the FROM set elements, the children of the root node are row identifiers and the children of the row identifiers are leaves containing the data pertaining to the SELECT list elements which, when prefixed with the SQL query alias, $\hat{t}_{al}$, serve as the labels of the edges between row identifier nodes and leaf nodes.

Since the denotation of integer and character constants are similar and since arithmatic expressions, the semantics of which are well-known and have the usual meaning, are simply mathematical operations on constants and/or column, the denotations of which are as illustrated in the addition example in Figure 8.8, we do not consider integer constants and arithmetic expressions separately in our denotational semantics. In the final line of Figure 8.8, π_3 is the projection operator that projects out the third component of triple x_j.

The crossproduct of a SELECT set consisting of character constants and columns from simple/derived tables, and a FROM set consisting of simple tables and derived tables yields 4 combinations - (character constant, simple table), (character constant, derived table/sub-query), (column, simple table), and (column, derived table/sub-query). In an SQL query containing one or more of these (SELECT element $\hat{s}$ and FROM element $\hat{tr}$) combinations, there are various scenarios that are possible with each of the 4 combinations and the cases corresponding to the simple table FROM element are discussed below. Please note that in the case of derived table/sub-query in the FROM clause, the procedures remain similar to the ones described below with the only difference being that the denotation of the simple table FROM element is replaced by the denotation of the sub-query. The subquery cases can be found in Appendix C, Figures C.2 and C.4.

1. $\hat{tr}$ **Not previously processed:**

   a. $\hat{s}$ = ***Constant, Previously Processed*** *(Appendix C, Figure C.1, Case ChChEx):* This scenario represents the case where the current constant SELECT element was added in combination with another FROM element in a recursive step, while the current FROM element $\hat{tr}$ was not encountered in any previous step. This case represents a cartesian product (cross join) of the sets of rows from the two participating tables. In our denotational semantics, the two participating entities are the graph that results from the recursive step and the graph that represents the denotation of the current FROM element. Since the current constant SELECT element has already been added in a recursive step, no actual data, other than the row identifiers, is required from the current FROM element. Thus the cross join of the two graphs is achieved by the `JoinGraphs` function which takes the graph from recursive step and the row identifiers of the graph representing the FROM element $\hat{tr}$ denotation as input and returns a graph that represents a cartesian product of the two input graphs.

   b. $\hat{s}$ = ***Constant, Not Previously Processed*** *(Appendix C, Figure C.1, Case ChNoneEx):* In this scenario where neither the SELECT element nor the FROM element were previously encountered in any recursive step, the result is a cartesian product of the two tables, one of which is represented by the recursive step graph $\phi'_{\mathcal{S}}$, and the other is represented by the graph that results from addition of a new edge, created using the `CreateEdges` function, for every row identifier in the graph, $\phi'_{\mathcal{S}}$ resulting from the recursive step. `CreateEdges` function generates an edge, (v_1', l', v_2'), for every row identifier node, v_1, in $\phi'_{\mathcal{S}}$, such that $v_1' = v_1$, $l = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ (the query and Constant SELECT element aliases), and v_2 =Constant SELECT element. The cartesian product of the two graphs is achieved using the `JoinGraphs` function as described in item 1a above.

c. $\hat{s}$ = ***Table.Column, Table*** = $\hat{tr}$ *(Appendix C, Figure C.3, Case ClTMchNoEx):* When the SELECT element is a column in the same simple/derived table as the current FROM element, the edges corresponding to the column SELECT element are extracted from the graph representing the denotation of the current FROM element $\hat{tr}$, using `ExtColEdges`. `ExtColEdges` returns a forest of single-edged trees where each edge is labeled with the column SELECT element. Further, there is with one edge per row identifier in $\hat{tr}$'s graph. If a particular row identifier does not have an edge corresponding to the current column SELECT element, a null edge, i.e., an edge with v_1 =row identifier, $l = \hat{cl}_{al}$, and $v_2 = d_{null}$, is added for that row identifier. Lastly, the edge labels of extracted edges are replaced with the query and column alias names, i.e., with $\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and the extracted edges are cross joined with the graph ϕ'_S from the recursive step using the `JoinGraphs` functon.

d. $\hat{s}$ = ***Table.Column, Table*** $\neq \hat{tr}$ *(Appendix C, Figure C.3, Case ClTNoMch-NoEx):* When the SELECT element is a column that does not belong to the simple/derived table that is the current FROM element, and when the FROM element table was not previously encountered, the scenario represents a limited cartesian product, achieved using `JoinGraphs`, between two graphs, the first of which is the recursive step graph, ϕ'_S, and the other is the denotation of the current FROM element where no actual data, other than the row identifiers, is used in the cartesian product since the column SELECT element does not belong to the FROM table element.

2. $\hat{tr}$ **Previously processed:**

a. $\hat{s}$ = ***Constant, Previously Processed*** *(Appendix C, Figure C.1, Case ChAllEx):* Since an SQL query is a cross-product of the SELECT and FROM sets, it is possible that the current SELECT element was added in combination with another

FROM element in a recursive step. Further, the FROM element could also have been encountered in combination with some other SELECT element. Under such a scenario no further action is required and the result obtained from the recursive step remains unchanged.

b. $\hat{s}$ = ***Constant, Not Previously Processed*** *(Appendix C, Figure C.1, Case ChTblEx):* This scenario represents the case where the current FROM element was added in combination with another FROM element in a recursive step, while the current Constant SELECT element was not encountered in any previous step. This case is straightforward and involves the addition of a new edge, created using the `CreateEdges` function, for every row identifier in the graph $\phi'_{\mathcal{S}}$ resulting from the recursive step. For every row identifier node, v_1, in $\phi'_{\mathcal{S}}$, an edge (v_1', l', v_2') is added where $v_1' = v_1$, $l = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ (the query and Constant SELECT element aliases), and v_2 =Constant SELECT element.

c. $\hat{s}$ = ***Table.Column, Table*** $= \hat{tr}$ *(Appendix C, Figure C.3, Case ClTMchEx):* Since the FROM element, $\hat{tr}$, was processed in a recursive step, the row identifiers of $\hat{tr}$ are present in $\phi'_{\mathcal{S}}$, the graph resulting from the recursive step. Therefore, this scenario does not result in any new row identifiers in $\phi'_{\mathcal{S}}$ and only results in addition of edges to existing row identifiers in $\phi'_{\mathcal{S}}$. Thus, in this scenario, the edges corresponding to the column SELECT element are extracted from the graph representing the denotation of $\hat{tr}$, replaced with the aliased name as described in item 1b, and added to the appropriate row identifiers in $\phi'_{\mathcal{S}}$ (i.e., each edge, (v_1, l, v_2) obtained through `ExtColEdges`, with a row identifier v_1, is added to those row identifiers, v_1', in $\phi'_{\mathcal{S}}$ which have v_1 as a substring) using the function `AttachEdge`.

d. $\hat{s}$ = ***Table.Column, Table*** $\neq \hat{tr}$ *(Appendix C, Figure C.3, Case ClTNoMchEx):* In this scenario where the FROM table element was previously encountered in combination with another SELECT element in a recursive step and the column SELECT element does not belong to the current FROM table element, no further action is required and the result obtaind from the recursive step remains unchanged.

8.2.3 SPARQL Semantics

Similar to our SQL Semantics, we define a SPARQL Query as a function from graphs to graphs. SPARQL queries generated by R2D's semantics-preserving query translation process is based on the SPARQL 1.1 language (Harris and Seaborne 2011). Our SPARQL query model also consists of two sets, the SELECT items (Variables) set and the WHERE items (BGP) set. The Variables set contains character and integer constants, variables from sub-queries or instances belonging to a particular class, and arithmetic expressions. The BGP set contains basic graph pattern matching expressions or sub-queries. Basic Graph Patterns (or BGPs) can be of two kinds: type patterns where the predicate of the pattern is always `type` and the object is an element of the `rdfs:class` set, and non-type patterns where the predicate is not `type` and the object is not an element of the `rdfs:class` set. SPARQL queries are modeled as a crossproduct of the elements of the Variables and BGP sets.

Denotational Semantics of a SPARQL Query

The denotation of a Type BGP element, of the form (s `type` c), where $c =$ `rdfs:class`, in a SPARQL WHERE set, on an input graph ϕ, like its SQL simple table counterpart, is a forest where every tree in the forest has a height of 1. There is one tree in the forest for every s of type c in ϕ, i.e., for every v_1 of (v_1, l, v_2) where $v_2 = c$ and $l =$ `type`, and the root of each tree, which serves as a row identifier, is set to the value of the instance element, i.e., $r_{id} = v_1 = s$. The edges of each tree in the forest are attributes of the instance element that forms the root of the tree. In other words every edge, (v_1', l', v_2'), where $v_1' = r_{id} = s$, is added, with the label, $l' \neq$ `type`, prefixed with the class name, to the tree rooted at r_{id}. Lastly, in the case of class elements (c) that are not of the concatenated type, $c_1\_c_2$ or c_l, representing one-to-many (1:N) or many-to-many (N:M) relationships, the instance value, s, is also added as an edge to its corresponding tree. Thus, each tree in the forest represents an instance of the class identified by c.

The denotation of a SPARQL variable, $\dot{v}$ `AS` $\dot{v}_{al}$, that represents an instance's attribute, p obtained using the BGP pair $((s,\ \texttt{type},\ c),\ (s,\ p,\ \dot{v}))$, in a SPARQL SELECT set is simply the edges in the forest comprising the denotation of the class c that are labeled with the tag c_p, the edge labels of which are subsequently renamed with the variable alias name, $\dot{v}_{al}$. The denotation of a constant (character or integer) SELECT element ($\hat{c}$ `AS` $\dot{v}_{al}$) in conjunction with a Type BGP in the WHERE set is simply the addition of 'n' edges of the form (v_1, l, v_2) where 'n' is the number of trees in the forest comprising the denotation of the class, c, in the Type BGP, from which the constant is to be selected. Thus there is one edge per tree where v_1, the row identifier, is the root of the tree, i.e., $v_1 = r_{id}$, $l = \dot{v}_{al}$, and $v_2 = \hat{c}$.

The denotation of the result of a SPARQL query is a graph of with a height of 2 where the root node is a string comprising the class elements, cs, and sub-query alias elements in the WHERE set, the children of the root node are row identifiers and the children of the row identifiers are leaves containing the data pertaining to the SELECT list elements whose aliases serve as the labels of the edges between row identifier nodes and leaf nodes.

The crossproduct of a SELECT set consisting of character constants and variables from sub-queries or variables corresponding to predicates belonging to instances of classes (since the denotation of constants are similar and since arithmatic expressions are simply mathematical operations on constants and/or column we do not consider integer constants and arithmetic expressions separately in our denotational semantics), and a WHERE set consisting of BGP expressions or sub-queries yields, as in the SQL case, 4 combinations: (character constant, Type BGP), (character constant, sub-query), (variable, Type and Non-Type BGP pair), and (variable, sub-query). The various scenarios that are possible with each of the above 4 combinations in a SPARQL query containing one or more of these (SELECT element, $\dot{spsl}$, and WHERE element(s)) combinations, are very similar to the scenarios possible in SQL queries and are discussed below.

1. $\dot{w}$ = **Type BGP or Type and Non-Type BGP Pair -** c **in Type BGP not previously processed:**

a. $\dot{spsl}$ = ***Constant, Previously Processed*** *(Appendix C, Figure C.7, Case ChChEx):* This case is similar to item 1a in section 8.2.2 with $\hat{tr}$ replaced with the classelem c in the Type BGP.

b. $\dot{spsl}$ = ***Constant, Not Previously Processed*** *(Appendix C, Figure C.7, Case ChNoneEx):* This case is similar to item 1b in section 8.2.2 with $\hat{tr}$ replaced with the classelem c in the Type BGP.

c. $\dot{spsl}$ $= \dot{v}$ `AS` $\dot{v}_{al}$***, Object in Non-Type BGP*** $= \dot{v}$ ***and Subject in Type BGP = Subject in Non-Type BGP*** *(Appendix C, Figure C.9, Case ClTwMchNoEx):* This scenario signifies the fact that the required variable corresponds to an attribute of instances belonging to the class element c in the Type BGP and is handled in a manner similar to item 1c in section 8.2.2 with $\hat{tr}$ replaced with c.

d. $\dot{spsl}$ $= \dot{v}$ `AS` $\dot{v}_{al}$***, Object in Non-Type BGP*** $= \dot{v}$ ***and Subject in Type BGP*** $\neq$ ***Subject in Non-Type BGP*** *(Appendix C, Figure C.9, Case ClTwNoMchNoEx):* This scenario signifies the fact that the required variable corresponds to an attribute of instances that do not belong to the class element c in the Type BGP and is handled in a manner similar to item 1d in section 8.2.2 with $\hat{tr}$ replaced with c.

2. $\dot{w} = \dot{spq}_{sq}$ **- Not previously processed** *(Appendix C, Figures C.8 and C.10):* This scenario for either SELECT element, Constant or $\dot{v}$, is similar to the corresponding scenarios in items 1a/1b and 1c/1d, respectively, (depending on whether (1a) or not (1b) the SELECT element has been previously encountered in the case of a Constant SELECT element), in section 8.2.2. In the case of $\dot{v}$ SELECT element, further processing happens (in a manner similar to item 1c in section 8.2.2) if $\dot{v}$ is an element of the SELECT set of the SPARQL sub-query $\dot{spq}_{sq}$, and 1d if not.

3. $\dot{w}$ = **Type BGP or Type and Non-Type BGP Pair -** c **in Type BGP previously processed:**

a. $\dot{spsl}$ = ***Constant- Previously processed*** *(Appendix C, Figure C.7, Case ChAllEx):* When the Constant SELECT element as well as the class element c in the Type

BGP have been previously encountered, and, thus, processed, in combination with other SELECT and WHERE elements in some recursive step, no further action is required and the result obtained from the recursive step remains unchanged, similar to item 2a in section 8.2.2.

b. $\dot{spsl}$ = ***Constant- Not previously processed*** *(Appendix C, Figure C.7, Case ChTblEx):* This scenario for a Constant SELECT element is similar to the corresponding scenario in item 2b in section 8.2.2, with $\hat{tr}$ replaced with the class element c in the Type BGP, and $\hat{cl}_{al}$ replaced with `RemoveFirstChar`$(\dot{v}_{al})$.

c. $\dot{spsl} = \dot{v}$ `AS` $\dot{v}_{al}$***, Object in Non-Type BGP*** $= \dot{v}$ ***and Subject in Type BGP = Subject in Non-Type BGP*** *(Appendix C, Figure C.9, Case ClTwMchEx):* : This scenario is similar to the one detailed in item 2c in section 8.2.2, with $\hat{tr}$ replaced with the class element c in the Type BGP, and $\hat{cl}_{al}$ replaced with `RemoveFirstChar`$(\dot{v}_{al})$ and is processed in the same manner.

d. $\dot{spsl} = \dot{v}$ `AS` $\dot{v}_{al}$***, Object in Non-Type BGP*** $= \dot{v}$ ***and Subject in Type BGP*** $\neq$ ***Subject in Non-Type BGP*** *(Appendix C, Figure C.9, Case ClTwNoMchEx):* This scenario signifies the fact that the required variable corresponds to an attribute of instances that do not belong to the class element, c, in the Type BGP and is handled in a manner similar to item 1d in section 8.2.2 with $\hat{tr}$ replaced with c.

4. $\dot{w} = \dot{spq}_{sq}$ **- Previously processed** *(Appendix C, Figures C.8 and C.10):* This scenario for either SELECT element, Constant or $\dot{v}$, is similar to the corresponding scenarios in items 2a/2b and 2c/2d respectively, (depending on whether (2a) or not (2b) the SELECT element has been previously encountered in the case of a Constant SELECT element), in section 8.2.2. In the case of $\dot{v}$ SELECT element, further processing happens (in a manner similar to item 2c in section 8.2.2) if $\dot{v}$ is an element of the SELECT set of the SPARQL sub-query $\dot{spq}_{sq}$, and 2d if not.

8.3 Translation Functions

As stated earlier, R2D's objective of semantics-preserving translation is achieved through two components, the Schema Mapping component and the Query Transformation component, each of which consists a transformation function that enables the realization of R2D's objectives. The schema mapping component, responsible for translating the canonical form of any given RDF graph into an equivalent relational schema, is accomplished using a transformation function, f, from RDF graphs in canonical form, ϕ, to relational schemas, $\mathbb{R}$. The SQL-to-SPARQL Query transformation component is realized using a function, h, from $\hat{sql}$ to $\dot{spq}$.

8.3.1 Schema Mapping/Transformation Function, f

The schema transformation function, f, which takes the canonical form of an RDF graph as input and returns an equivalent relational schema as output, encodes the RDFMapFileGenerator and DBSchemaGenerator Algorithms that were discussed at length in Chapter 4. However, since RDFMapFileGenerator algorithm caters to RDF graphs with and without RDF Schema (ontological) information and is data-agnostic, there are several differences between the algorithm and f. Although the RDF specification does not prevent the creation of graphs without RDF Schema information, such graphs constitute only a small minority of universe of RDF graphs in existence. Thus, f focuses only on those graphs that constitute the majority, i.e., RDF Graphs with RDFS Class information.

Another difference between the RDFMapFileGenerator Algorithm and f is in the way blank nodes are handled. f operates on the canonical form of the input RDF graph where no blank nodes exist due to the fact that each blank node chain is collapsed into a single edge. Collapsing blank nodes has two advantages: Firstly, it keeps the complexity of f independent of the number of blank nodes in the graph as well as the length of blank node chains in the graph, and, secondly, it greatly simplifies the number of cases that need to be addressed during the schema mapping process. However, the collapsing of blank nodes causes, in some cases, grouping information to be split across several entities during the transformation into a relational schema. This scenario can be better understood through the *"Phone"* blank node in Chapter 3, Figure 3.2. Collapsing of blank nodes will result in *"Work_ Phone"* being a column in the *"Student"* table while the *"Cell Phone"* values, by virtue of being multi-valued, get translated into a separate MVA table called, for example, *"Student_PhoneCell"*. The RDFMapFileGenerator algorithm operates directly on RDF Graphs and thus encounters, and classifies, blank nodes into various categories as detailed in Chapter 3. Thus, the *"Phone"* blank node in Figure 3.2, classified as a *r2d:ComplexLiteralBlankNode*, is transformed into a more intuitive, separate table called *"Phone"*, which has *"Student_PK"*, *"Phone_Type"*, and *"Phone_Value"* as columns. However, classification of blank nodes into categories is limited to non-nested blank nodes as the process becomes quickly intractable for nested blank nodes. Therefore, RDFMapFileGenerator, and, consequently, DBSchemaGenerator, consider nested blank nodes as MixedBlankNodes the transformation of which is as detailed in Chapters 3 and 4.

Lastly, since f is not data agnostic, in addition to non-temporary tables, under certain scenarios it generates temporary tables unlike RDFMapFileGenerator and DBSchemaGenerator which generates only non-temporary tables at all times. Temporary table cells are created when f encounters resource/instance-attribute edges before encountering class-resource/instance edges. Under such a circumstance, since the class to which the resource belongs (and therefore the table into which the attribute and object/data value are to be added as a column and row data respectively) is unknown, a temporary table, named after the resource, is created and the attribute and object/data is added to the temporary table. When the class-resource/instance edge is encountered by f at some point in the future, the cells from the appropriate temporary tables are copied over to the table corresponding to the class element and the temporary tables are removed from the schema. RDFMapFileGenerator and DBSchemaGenerator, on the other hand, have been implemented in a manner where class-resource/instance edges are always processed prior to resource/instance-attribute edges and, thus, do not require the use of temporary tables.

8.3.2 Query Transformation Function, h

The query tranformation function, h, illustrated in Figures 8.9 and 8.10, takes the canonical form of an SQL statement as input and produces an equivalent SPARQL query as output and encodes the third algorithm comprising the R2D framework, the SQL-to-SPARQL translation algorithm in Chapter 4. As mentioned earlier, in order to keep the semantics-preserving query transformation proofs tractable and readable, a representative sub-set of the SQL language was chosen as the core language that is supported by R2D. Consequently, the mathematically defined function, h, is a simplification of the implemented SQL-to-SPARQL translation algorithm and models the transformation of SQL queries with a SELECT set consisting of constants, columns from simple/derived tables, and arithmetic expressions and a FROM set consisting of simple tables and derived tables or sub-queries. The query translation algorithm implemented in Chapter 4, on the other hand, handles, in addition to everything that h does, WHERE clauses, aggregrate functions, and pattern matching functions such as the LIKE operator. However, a correctness proof that encompasses all of these features would be too large to be easily comprehensible and, as a result, we chose not to encode these features in h. It can be seen that encoding these features is a straightforward process as, in most cases, the features translate directly into appropriate FILTER, REGEX, and aggregate functions within the SPARQL grammar, and, therefore, is an prospect for future work.

8.4 Proofs

R2D's transformation framework is semantics-preserving and therefore $\mathcal{S}[\![\hat{sql}]\!]\mathcal{E}[\![f(\phi)]\!] = \mathcal{Q}[\![h(\hat{sql})]\!]\phi$. This (semantics-preserving) characteristic of R2D's translation framework is, to recapitulate, established using two theorems, Theorem 1 and Theorem 2 described below.

$$\frac{}{h_w(\cdot, \hat{t}_{al}) = \cdot}(h_w \text{ SQL FROM }\textsc{List Base})$$

$$\frac{\hat{tr} = \hat{dt} \quad \hat{dt} = (\hat{sql}, \hat{t}_{al}) \quad \hat{sql} = (\rho_{\mathbb{SL}}, \rho_{\mathbb{TR}}) \quad (\rho_{\text{SpSL}}, \rho_{\text{SpW}_\text{t}}) = h(\hat{sql} = (\rho_{\mathbb{SL}}, \rho_{\mathbb{TR}}), \hat{t}_{al}) \quad \dot{spq} = (\rho_{\text{SpSL}}, \rho_{\text{SpW}_\text{t}})}{h_w(\hat{tr}, \hat{t}_{al}) = \dot{spq}}(h_w \text{ SQL }\textsc{From - SubQuery})$$

$$\frac{\hat{tr} = \hat{t} \quad \dot{w} = (?\hat{t}\_\texttt{type type } \hat{t})}{h_w(\hat{tr}, \hat{t}_{al}) = \dot{w}}(h_w \text{ SQL }\textsc{From - Table})$$

$$\frac{\dot{w} = h_w(\hat{tr}, \hat{t}_{al}) \quad \rho_{\text{SpW}} = h_w(\rho_{\mathbb{TR}}, \hat{t}_{al})}{h_w(\hat{tr} :: \rho_{\mathbb{TR}}, \hat{t}_{al}) = \dot{w} :: \rho_{\text{SpW}}}(h_w \text{ SQL }\textsc{From Clause})$$

$$\frac{h_s(\rho_{\mathbb{SL}}, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\rho_{\text{SpSL}}, \rho_{\text{SpW}_s}) \quad h_w(\rho_{\mathbb{TR}}, \hat{t}_{al}) = \rho_{\text{SpW}} \quad \dot{spq} = (\rho_{\text{SpSL}}, \rho_{\text{SpW}_s} @ \rho_{\text{SpW}})}{h(\hat{sql} = (\rho_{\mathbb{SL}}, \rho_{\mathbb{TR}}), \hat{t}_{al}) = \dot{spq}}(h \text{ SQL }\textsc{Query Translation})$$

Figure 8.9: SQL-to-SPARQL Translation function - h, `and` h_w

$$\frac{}{h_s(\cdot, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\cdot, \cdot)}(h_s \text{ SQL SELECT \textsc{List Base}})$$

$$\frac{\begin{array}{c} n\hat{v}e_a = \hat{t}.\hat{cl} \texttt{ AS } \hat{cl}_{al} \quad \texttt{IsTable}(\rho_{\mathbb{TR}}, \hat{t}) = \texttt{true} \quad \dot{spsl} = ?\hat{t}_{\text{-}T_{id}}\hat{cl} \texttt{ AS } ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al} \\ \dot{w}_s = (?\hat{t}_{\text{-}T_{id}}\texttt{type } \hat{cl} \; ?\hat{t}_{\text{-}T_{id}}\hat{cl}) \end{array}}{h_s(n\hat{v}e_a, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\dot{spsl}, \dot{w}_s)}(h_s \text{ TC\textsc{olumn} A\textsc{l}SQ})$$

$$\frac{\begin{array}{c} n\hat{v}e_a = \hat{t}.\hat{cl} \texttt{ AS } \hat{cl}_{al} \quad \texttt{IsTable}(\rho_{\mathbb{TR}}, \hat{t}) = \texttt{false} \\ \dot{spsl} = ?\hat{t}_{\text{-}T_{id}}\hat{cl} \texttt{ AS } ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al} \end{array}}{h_s(n\hat{v}e_a, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\dot{spsl}, \cdot)}(h_s \text{ DTC\textsc{olumn} A\textsc{l}SQ})$$

$$\frac{n\hat{v}e_a = \hat{i} \texttt{ AS } \hat{cl}_{al} \quad \dot{spsl} = \hat{i} \texttt{ AS } ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}}{h_s(n\hat{v}e_a, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\dot{spsl}, \cdot)}(h_s \text{ I\textsc{nt} A\textsc{l}SQ})$$

$$\frac{n\hat{v}e_a = \hat{c} \texttt{ AS } \hat{cl}_{al} \quad \dot{spsl} = \hat{c} \texttt{ AS } ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}}{h_s(n\hat{v}e_a, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\dot{spsl}, \cdot)}(h_s \text{ C\textsc{har} A\textsc{l}SQ})$$

$$\frac{(\dot{spsl}, \dot{w}_s) = h_s(n\hat{v}e_a, \hat{t}_{al}) \quad h_s(\rho_{\mathbb{SL}}, \hat{t}_{al}) = (\rho_{\text{SpSL}}, \rho_{\text{SpW}_s})}{h_s(n\hat{v}e_a :: \rho_{\mathbb{SL}}, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\dot{spsl} :: \rho_{\text{SpSL}}, \dot{w}_s :: \rho_{\text{SpW}_s})}(h_s \text{ SQL S\textsc{elect} C\textsc{lause}})$$

Figure 8.10: SQL-to-SPARQL Translation function - h_s

8.4.1 Semantic Preservation of Schema Translation

Theorem 1, given below, proves the semantics-preserving transformation performed by the schema mapping module.

Theorem 1: $\mathcal{E}[\![f(\phi)]\!] = \phi$

where ϕ is the canonical form of an RDF graph, $\mathbb{G}$, $\mathcal{E}[\![\mathbb{R}]\!]$ represents the denotational semantics of a relational schema, $\mathbb{R}$, and $f(\phi)$ is Schema mapping/transformation function that transforms the canonical form of $\mathbb{G}$ to $\mathbb{R}$

f transforms ϕ into $\mathbb{R}$ one edge at a time and the transformation of an edge depends on the type of edge encountered. The following are the types of edges that comprise any given ϕ.

1. Instance Triple (IT) Edges
2. Literal Triple (LT) Edges
3. Resource Triple With One-to-One Relationship (RT-Non1N) Edges
4. Resource Triple With One-to-Many Relationship (RT-1N) Edges
5. MultiValuedAttribute Instance (MVAI) Edges
6. MultiValueAttribute Resource (MVART) Edges
7. MultiValueAttribute Attribute (MVALT) Edges
8. Many-to-Many Relationship Instance (NMI) Edges
9. Many-to-Many Relationship Resource (NMRT) Edges

Since there are several scenarios possible with each of the above edge types, the number of cases that comprise the proof of Theorem 1 are large enough to prohibit their coverage in this section. Therefore, the discussion in the remainder of this section is retricted to a few of the more interesting cases while the complete proof is included to Appendix B.

Case (NMRT AllOK), given below, corresponds to Many-to-Many Relationship resource edges of the type (r_r', p, r'') where (r_r') is an instance of class, c.

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_r', p, r'') \quad \texttt{GetRegTblID}(\mathbb{R}, r'') \neq \{\} \\ \texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') = 2 \\ \texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r') \neq \{\} \end{array}}{\begin{array}{c} f(\phi = \phi' \uplus e) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_r', \tau_{tmp_{nm}})) \\ \uplus\, \texttt{AddNMTblRow}_{\texttt{RP}}(r\_r', p, r'') \end{array}} \text{(NMRT AllOK)}$$

NMRT Edge details are translated into (Non-Temporary) NM Tables if and only if all three edges that participate in the NM Relationship (i.e., one NMI edge and two NMRT Edges) are available (identified using the function $\texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r')$), and information regarding the classes to which the two resources (r and r') participating in the NM relationship belong is available (identified using the functions $\texttt{GetRegTblID}(\mathbb{R}', r'')$ and $\texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r') \neq \{\}$). If any of these required details are not available, the rest of the NMI/NMRT Edges information is stored in temporary tables named after the NM resource (r_r'). When an NM edge arrives that, together with other related information in a temporary NM table, makes available all required information about the NM relationship that exists between the pair of resources in the arriving edge, information from the appropriate temporary table is obtained, using the $\texttt{GetTmpTblRow}(\mathbb{R}', (r\_r', \tau_{tmp_{nm}}))$, and removed from the relational schema, $\mathbb{R}$, and added as a new row into the appropriate non-temporary NM table using the function, $\texttt{AddNMTblRow}_{\texttt{RP}}(\mathbb{R}', r\_r', p, r'')$. Thus, this case results in movement of applicable cells from temporary NM tables into non-temporary NM Tables and the addition of the value r'' into the column corresponding to p in r's row in the non-temporary NM Table that corresponds to the class to which the resource r_r' of the arriving NMRT edge belongs. The complete proof for **Case** (NMRT AllOK) is presented below.

Case (NMRT AllOK) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_r', p, r'')$, $\texttt{GetRegTblID}$ $(\mathbb{R}', r'') \neq \{\}$, $\texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') = 2$, and $\texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r')$ $\neq \{\}$. By Rule (NMRT AllOK),

$$f(\phi) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_r', \tau_{tmp_{nm}})) \uplus \texttt{AddNMTblRow}_{\texttt{RP}}(\mathbb{R}', r\_r', p, r'')$$

By definition of $\texttt{GetTmpTblRow}$ and $\texttt{AddNMTblRow}_{\texttt{RP}}$, $f(\phi) = \mathbb{R}' - \{d\} \uplus \{e\}$ where

$$d = \{(((r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), c), (((r\_r', \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r''')\}$$

, and

$$e = \{(((c, \tau_{nm}), (k_{id}, \kappa_{fk}), w), r'''), (((c, \tau_{nm}), (p, \kappa_{fk}), w), r'')\}$$

for some unoccupied row, w. By Rule (NMTmpT), (NMTTC), and (NMTNTC), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' - \{d\} \uplus \{e\}]\!] = \mathcal{E}[\![\mathbb{R}']\!] - \{(r\_r', \texttt{type}, c), (r\_r', k_{id}, r''')\} \cup \{(r\_r', \texttt{type}, c), (r\_r', k_{id}, r'''), (r\_r', p, r'')\} = \mathcal{E}[\![\mathbb{R}']\!] \uplus \{(r\_r', p, r'')\} = \mathcal{E}[\![f(\phi')]\!] \uplus \{e\}$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \phi' \uplus \{e\} = \phi$.

Another case, **Case** (IT), given below, is one of the more interesting cases in $f(\phi)$ as its proof follows from a composite of several lemmas.

$$\frac{f(\phi') = \mathbb{R}' \quad e = (r, \texttt{type}, c) \quad w = \texttt{GetNewRegTblRow}(c) + 1}{f(\phi = \phi' \uplus e) = (((((\mathbb{R}' - \texttt{GetTmpTblLitCols}(\mathbb{R}', r))}\,(\text{IT})$$
$$- \texttt{GetTmpRColsWithP}(\mathbb{R}', r)) - \texttt{GetTmpRTblsWithrObj}(\mathbb{R}', r))$$
$$- \texttt{GetValidTmpMVATbls}(\mathbb{R}', r) - \texttt{GetValidTmpNMTbls}(\mathbb{R}', r))$$
$$\uplus \texttt{AddTmpLColstoRegTbl}(\mathbb{R}', c, r, w) \uplus \texttt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}', r, c, w)$$
$$\uplus \texttt{AddTmpRColsWithPtoRegTbl}(\mathbb{R}', c, r, w) \uplus \texttt{UpdRRegTblsWithrObj}(\mathbb{R}', r, c, w)$$
$$\uplus \texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}', r, classelem, w)$$

Instance Triple (IT) edges, of the form $(r, \texttt{type}, c)$, are the edges that result in the creation of a new table corresponding to the class, c, if one does not already exist in the relational schema, $\mathbb{R}$, and addition of a new row in the table with the primary key cell in the row set of r. Prior to the availablity of a resource's (r's) class information (i.e. a resource's IT edge), all attribute edges (LT, RT-Non1N, RT-1N) pertaining to the resource are stored in temporary tables named, in the case of LT and Non-RT edges, after the resource, r, and in the case of RT-1N, named after the object resource (r'). The arrival of an IT edge results in the following transformations in the relational schema, $\mathbb{R}$.

- Literal value cells populated by LT edges corresponding to r are removed, using the function `GetTmpTblLitCols`($\mathbb{R}'$, r) from r's temporary table, if one exists, and added into appropriate cells, using the function `AddTmpLColstoRegTbl`($\mathbb{R}'$, c, r, w), in r's row in the table corresponding to c.

- Non-1N resource value cells, populated by RT-Non1N edges, with object resources whose class information is available, that are housed in r's temporary table are removed, using the function `GetTmpRColsWithP`($\mathbb{R}'$, r), and added into the appropriate cells, using `AddTmpRColsWithPtoRegTbl`($\mathbb{R}'$, c, r, w)

- 1N resource value cells, populated by RT-1N edges, with object resources whose class information is available, that are housed in the object resource's (r''s) temporary table are removed, using the function `GetTmpRTblsWithrObj`($\mathbb{R}'$, r), and added into the appropriate cells, using `UpdRRegTblsWithrObj`($\mathbb{R}'$, r, c, w)

- Data from temporary MVA tables that have r as an element in one of their cells and that have all information pertaining to all three edges (MVAI, MVART, and MVALT) available in the temporary MVA table, is removed from the temporary table using the function `GetValidTmpMVATbls`($\mathbb{R}'$, r) and added as a new row into the appropriate non-temporary MVA Table using `AddTmpMVATblRowstoMVATbl`($\mathbb{R}'$, r, c, w)

- Data from temporary NM tables that have r as an element in one of their cells, and that have all information pertaining to all three edges (NMLInstance, two NMRT edges) available in the temporary NM table along with the class information of the other resource (r') participating in the NM relationship with r, is removed from the temporary table using the function $\texttt{GetValidTmpNMTbls}(\mathbb{R}', r)$ and added as a new row into the appropriate non-temporary NM Table using $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}', r, c, w)$

Each of the above transformations consists of function pairs that, as can be seen from the respective function definitions, together, effectively cause the addition of a new row corresponding to r in the table corresponding to c and the movement of applicable cells from temporary tables to non-temporary tables. As a result, proof of **Case** (IT) follows from the proof that the denotation of the transformations effected by each function pair consolidates into a single edge, $(r, \texttt{type}, c)$. Thus there is one proof for each of the 5 function pairs ennumerated above. As an illustration of the transformations effected by the function pairs, the proof of lemma 1, which illustrates the transformations resulting from the first function pair, $\texttt{GetTmpTblLitCols}(\mathbb{R}', r)$ and $\texttt{AddTmpLColstoRegTbl}(\mathbb{R}', c, r, w)$, is included below while lemmas corresponding to the rest of the function pairs can be found in Appendix B.

Lemma 1 *For every instance edge e of type $(r, \texttt{type}, c)$, and an unoccupied row, w, in the table corresponding to c in $\mathbb{R}$,*

$$\mathcal{E}[\![\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w) - \texttt{GetTmpTblLitCols}(\mathbb{R}, r)]\!] = \{e\}$$

[$\Longrightarrow$]: Let $((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_r), w), \mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w')))$ be an element in the set $\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w)$. Let $\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w')$ $= d$. From the definition of $\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w)$, it can be observed that $((r, \tau_{tmp}), (k_{id}, \kappa_r), w') \in \mathbb{R}^{\leftarrow}$ and $d \in \mathcal{L}$. However, from the definition of $\texttt{GetTmpTblLitCols}(\mathbb{R}, r)$, it can be observed that, whenever $((r, \tau_{tmp}), (k_{id}, \kappa_r), w') \in \mathbb{R}^{\leftarrow}$ and $d \in \mathcal{L}$,
$(((r, \tau_{tmp}), (k_{id}, \kappa_r), w'), d)$ is an element of the set $\texttt{GetTmpTblLitCols}(\mathbb{R}, r)$.

Thus, for every element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_r), w), d)) \in$ $\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w)$, there exists an element $e'' = (((r, \tau_{tmp}), (k_{id}, \kappa_r), w'),$ $d) \in \texttt{GetTmpTblLitCols}(\mathbb{R}, r)$.

[$\Longleftarrow$]: Let $\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w') = d$ (where $d \in \mathcal{L}$) be an element in the set $\texttt{GetTmpTblLitCols}(\mathbb{R}, r)$. As can be observed from the definition of $\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w)$, the values r and $\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w')$ are added to columns $\texttt{type}$ and k_{id}, respectively, in row w of table, c, i.e. $\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) = r$ and $\mathbb{R}((c, \tau_r), (k_{id}, \kappa_r), w) = d$.

Thus, for every element $e'' = (((r, \tau_{tmp}), (k_{id}, \kappa_r), w'), d) \in \texttt{GetTmpTblLitCols}(\mathbb{R}, r)$, there exists an element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_r), w), d)) \in \texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w)$.

For any corresponding pair e' and e'', we have

$\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w) - \texttt{GetTmpTblLitCols}(\mathbb{R}, r) = \{\pi_1 e', \pi_2 e'\} - \{e''\} =$

$\{(((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_r), w), d)\} - \{(((r, \tau_{tmp}), (k_{id}, \kappa_r), w'), d)\}$.

From Rule (TmpTN1N), it can be observed that $\mathcal{E}[\![\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w') \mapsto d]\!] = \{(r, k_{id}, d)\}$. From Rule (KCRT), it can be observed that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) \mapsto r]\!] = \{(r, \texttt{type}, c)\}$ and from Rule (NKCN1NRT), it can be seen that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (k_{id}, \kappa_r), w) \mapsto d]\!] = \{(r, k_{id}, d)\}$.

Thus, $\mathcal{E}[\![\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w) - \texttt{GetTmpTblLitCols}(\mathbb{R}, r)]\!] = \mathcal{E}[\![\{\pi_1 e', \pi_2 e'\} - \{e''\}]\!] =$
$\{(r, \texttt{type}, c), (r, k_{id}, d)\} - \{(r, k_{id}, d)\} = \{(r, \texttt{type}, c)\} = \{e\}$.

8.4.2 Semantic Preservation of Query Translation

Theorem 2, given below, is used to prove that R2D's SQL-to-SPARQL Query Transformation process is semantics-preserving.

Theorem 2: $\mathcal{S}[\![\hat{sql}]\!]\phi = \mathcal{Q}[\![h(\hat{sql})]\!]\phi$

where $\mathcal{S}[\![\hat{sql}]\!]$ represents the denotational semantics of an SQL query, $\hat{sql}$, $\mathcal{Q}[\![\dot{spq}]\!]$ represents the denotational semantics of a SPARQL Query, $\dot{spq}$, and $h(\hat{sql})$ is R2D's translation from $\hat{sql}$ to $\dot{spq}$.

As stated earlier, we chose to model SQL queries as a cross-product of the SELECT and FROM sets comprising the query. Thus an SQL query is a tuple the first element of which is a set of tuples where each element of the set is a tuple that consists of one SELECT set element and one FROM set element and the second element of which is the query alias. For the SQL core language that we have considered for the purposes of proving semantics-preserving query translation, the cross-product results in the following tuple combinations.

- ($\hat{s}$=Constant, $\hat{tr}$=Simple Table($\hat{t}$))
- ($\hat{s}$=Constant, $\hat{tr}$=Sub-Query ($\hat{dt}$))
- ($\hat{s}$=Table.Column, $\hat{tr}$=Simple Table($\hat{t}$))
- ($\hat{s}$=Table.Column, $\hat{tr}$=Sub-Query ($\hat{dt}$))

Each of the above combinations has 4 possibilities as discussed in Section 8.2.2 and there exists one case in the proof of Theorem 2 for each of these possibilities. Inclusion of the entire proof in this section would be too verbose and, therefore, only a few cases are presented here. The complete proof including all cases is presented in Appendix C.

The case, **Case** ($\mathcal{S}$ ChNoneEx), given below, addresses the scenario where neither the SELECT set element, $\hat{s}$, which is a character constant, nor the FROM set element, $\hat{tr}$, which is a simple table, $\hat{t}$, have been encountered or processed in previous recursive steps (as evidenced by the functions $\texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{false}$ and $\texttt{TableExists}(r_{\mathcal{S}}', \hat{t}) = \texttt{false}$ respectively.

$\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al} \quad \texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{false}$
$r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \quad \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}) = \texttt{false}$

$$\frac{\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}') \quad r_{\mathcal{S}} = \texttt{AddTable}(r_{\mathcal{S}}', \hat{t})}{\begin{array}{l}\mathcal{S}[\![(\{(\hat{s},\hat{t})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{CreateEdges}(\hat{c}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \\ \texttt{AddTableLabel}(\texttt{GetRootNodes}(\mathcal{S}_T[\![\hat{t}]\!]\phi), \hat{t})), \phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{array}} (\mathcal{S}\ \text{ChNoneEx})$$

In this case, character constant edges of the form (v_1, l, v_2), where v_1 = Row identifiers from $\mathcal{S}_T[\![\hat{tr}]\!]$, $l = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, $v_2 = \hat{c}$, are created using the function, `CreateEdges`, and a cartesian product of the resultant graph with the graph from the recursive step, $\phi'_{\mathcal{S}}$, is produced as the final result of the given SQL query using the `JoinGraphs` function.

The SQL-to-SPARQL transformation of the SQL query $(\hat{sql}, \hat{t}_{al}) = (\{\hat{s}, \hat{tr})\} \uplus \hat{sql}', \hat{t}_{al})$, obtained using the function, $h(\hat{sql}, \hat{t}_{al})$, is $\{(\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, (\hat{t}\texttt{\_type type}\ \hat{t}))\} \cup \dot{spq}'$.

By inductive hypothesis, we assume that $\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$. Further, if we consider a variable, c, such that $c = \hat{t}$, since $\hat{t}$ was not previously encountered in $\phi'_{\mathcal{S}}$, it follows that c was not previously encountered in $\phi'_{\mathcal{Q}}$, where $\phi'_{\mathcal{Q}} = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$. Thus, as can be seen from the SPARQL Denotations in Appendix C, the above transformation matches the case **Case** ($\mathcal{Q}$ ChNoneEx) given below.

$$\frac{\begin{array}{l}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al} \quad \dot{w} = (s\ \texttt{type}\ c) \quad \dot{v}_{al} = c''_{\text{-}T_{id}}p \\ \dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) \quad \texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{false} \\ \texttt{TableExists}(r_{\mathcal{Q}}', c) = \texttt{false} \quad r_{\mathcal{Q}}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \\ \qquad \phi'_{\mathcal{Q}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}') \quad r_{\mathcal{Q}} = \texttt{AddTable}(r_{\mathcal{Q}}', c)\end{array}}{\begin{array}{l}\mathcal{Q}[\![\{(\dot{spsl}, \dot{w})\} \uplus \dot{spq}']\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{CreateEdges}(\hat{c}, \dot{v}_{al_{rf}}, \\ \texttt{AddTableLabel}(\texttt{GetRootNodes}(\mathcal{S}_T[\![c]\!]\phi), c)), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})\end{array}} (\mathcal{Q}\ \text{ChNoneEx})$$

Since $c = \hat{t}$, $\hat{s} = \hat{c}$, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$, $\dot{v}_{al_{rf}} = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $r_{\mathcal{S}} = r_{\mathcal{Q}}$, from the two cases above, **Case** ($\mathcal{S}$ ChNoneEx) and **Case** ($\mathcal{Q}$ ChNoneEx), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$. The more formal proof for this case is given below.

Case ($\mathcal{S}$ ChNoneEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{t})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq \texttt{null}$, $\hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{false}$, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}) = \texttt{false}$, $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}}))$, and $r_{\mathcal{S}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t})$. By

Rule ($\mathcal{S}$ ChNoneEx),

$$\begin{aligned}\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = &\texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{CreateEdges}(\hat{c}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \\ &\texttt{AddTableLabel}(\texttt{GetRootNodes}(\mathcal{S}_T[\![\hat{t}]\!]\phi), \hat{t})), \phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{aligned} \tag{8.1}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{c}\;\texttt{AS}\;\hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (h_s(\hat{c}\;\texttt{AS}\;\hat{cl}_{al}, \hat{t}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. By Rule ($h_s$ Char AlSQ), $h_s(\hat{c}\;\texttt{AS}\;\hat{cl}_{al}, \hat{t}, \hat{t}_{al}) = (\hat{c}\;\texttt{AS}\;?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \cdot)$, and by Rule ($h_w$ SQL From - Table), $h_w(\hat{t}, \hat{t}_{al}) = (?\hat{t}\_\texttt{type type}\;\hat{t})$. Thus, $h((\hat{c}\;\texttt{AS}\;\hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (\hat{c}\;\texttt{AS}\;?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, (\hat{t}\_\texttt{type type}\;\hat{t}))$

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Observe that when $c = \hat{t}$, $\dot{spsl} = \hat{c}\;\texttt{AS}\;\hat{cl}_{al}$, $\dot{v}_{al} = ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ we have $\texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{false}$ and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}),\ c) = \texttt{false}$ from our previous assumption (since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ from Inductive Hypothesis) , and $\dot{v}_{al} = c''\text{-}_{T_{id}}\dot{v}_{al_{suffix}}$, where $c'' = \hat{t}_{al}$ and $\dot{v}_{al_{suffix}} = \hat{cl}_{al}$. Further, since $\phi'_{\mathcal{S}} = \phi_{\mathcal{Q}}$' (by Inductive Hypothesis), $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}})) = \phi'_{\mathcal{Q}r}$ and $r_{\mathcal{S}} = r_{\mathcal{Q}}$. Thus, by Rule ($\mathcal{Q}$ ChNoneEx),

$$\begin{aligned}\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = &\texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{CreateEdges}(\hat{c}, \dot{v}_{al_{rf}}, \\ &\texttt{AddTableLabel}(\texttt{GetRootNodes}(\mathcal{S}_T[\![c]\!]\phi), c)), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})\end{aligned} \tag{8.2}$$

.

Since $c = \hat{t}$, $\hat{s} = \hat{c}$, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$, and $r_{\mathcal{S}} = r_{\mathcal{Q}}$, from equations (8.1) and (8.2), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

A case that involves a column SELECT set element and a derived table (sub-query), is **Case** ($\mathcal{S}$ ClDTMchEx), given below.

$\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}}$ $\quad \hat{s} = \hat{t}.\hat{cl}\;\texttt{AS}\;\hat{cl}_{al}$ $\quad \hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$ $\quad \hat{t} = \hat{t}_{alsq_{\hat{sql}}}$
$\phi''_{\mathcal{S}} = \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$ $\quad r_{\mathcal{S}}{}' = \texttt{GetRoot}(\phi'_{\mathcal{S}})$

$$\frac{\texttt{TableExists}(r_{\mathcal{S}}{}', \hat{t}_{alsq_{\hat{sql}}}) = \texttt{true} \quad r_{\mathcal{S}_{dt}} = \texttt{GetRoot}(\phi''_{\mathcal{S}})}{\mathcal{S}[\![(\{(\hat{s}, \hat{dt})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AttachEdge}(\texttt{ReplaceLabel}(\texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi''_{\mathcal{S}}, r_{\mathcal{S}_{dt}})), \phi''_{\mathcal{S}}), \hat{t}_{\text{-}T_{id}}\hat{cl}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}) \times \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}{}'))) \cup \phi'_{\mathcal{S}}} (\mathcal{S}\ \textsc{ClDTMchEx})$$

Here, the SELECT set element, $\hat{s}$, is a column element of the form $\hat{t}.\hat{cl}$ AS $\hat{cl}_{al}$, and the FROM set element, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$, is a derived table (sub-query) with a query alias, $\hat{t}_{alsq_{\hat{sql}}}$. This particular case addresses the scenario where the FROM set subquery element, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$ has been previously encountered, as identified by the function $\texttt{TableExists}(r_{\mathcal{S}}{}', \hat{t}_{alsq_{\hat{sql}}}) = \texttt{true}$, in a recursive step and the SELECT set column element is a column that belongs to the derived table, $\hat{t}_{alsq_{\hat{sql}}}$, that results from the subquery, $\hat{sql}_{sq}$, i.e., $\hat{t} = \hat{t}_{alsq_{\hat{sql}}}$.

In this case, edges corresponding to the SELECT set column element, i.e. edges with an edge label of $\hat{t}_{\text{-}T_{id}}\hat{cl}$, are extracted from the graph that represents the denotation of the derived table, i.e. from $\phi''_{\mathcal{S}} = \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$ and are added to the appropriate row identifier nodes of the graph from the recursive step, $\phi'_{\mathcal{S}}$, using the `AttachEdge` function to arrive at the final result of the given SQL query. `AttachEdge` attaches an extracted edge, (v_1, l, v_2), to those row identifier vertices, $v_1{}'$, in $\phi'_{\mathcal{S}}$ where v_1 is present as a substring.

The SQL-to-SPARQL transformation of the SQL query $(\hat{sql}, \hat{t}_{al}) = (\{\hat{s}, \hat{tr})\} \uplus \hat{sql}', \hat{t}_{al})$, obtained using the function, $h(\hat{sql}, \hat{t}_{al})$, is $\{(?\hat{t}_{\text{-}T_{id}}\hat{cl}$ AS $?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \dot{spq}_{sq})\} \cup \dot{spq}'$.

By inductive hypothesis, we assume that $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ and $\mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi = \mathcal{Q}[\![h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$. Also, the SELECT item translation rule for columns from derived tables (or sub-queries) prefixes the sub-query alias, $\hat{t}_{alsq_{\hat{sql}}}$, to the alias of every corresponding SPARQL Query SELECT element. Thus $\hat{t}_{alsq_{\dot{spq}}} = \texttt{GetSubQueryAlias}(h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}) = \hat{t}_{alsq_{\hat{sql}}}$. Since $\hat{t}_{alsq_{\hat{sql}}}$ was previously encountered in $\phi'_{\mathcal{S}}$, it follows that $\hat{t}_{alsq_{\dot{spq}}}$ was also previously encountered in $\phi'_{\mathcal{Q}}$, where $\phi'_{\mathcal{Q}} = \mathcal{Q}[\![h(\hat{sql}')]\!]\phi$. Thus, as can be seen from the SPARQL Denotations in Appendix C, the above transformation matches the case **Case** ($\mathcal{Q}$ ClSQMchEx) given below.

$$\dfrac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \quad \dot{spsl} = \dot{v} = ?c_{\text{-}T_{id}}p \texttt{ AS } \dot{v}_{al} \quad \dot{v}_{al} = c''{}_{\text{-}T_{id}}p \\ \dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) \quad \hat{t}_{alsq\dot{spq}} = \texttt{GetSubQueryAlias}(\dot{spq}_{sq}) \\ \hat{t}_{alsq\dot{spq}} = c \quad r_{\mathcal{Q}}{}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}{}', \hat{t}_{alsq\dot{spq}}) = \texttt{true} \\ \mathcal{Q}[\![\dot{spq}_{sq}]\!]\phi = \phi''_{\mathcal{Q}} \quad r_{\mathcal{Q}_{dt}} = \texttt{GetRoot}(\phi''_{\mathcal{Q}})\end{array}}{\begin{array}{l}\mathcal{Q}[\![\{(\dot{spsl}, \dot{spq}_{sq})\} \uplus \dot{spq}']\!]\phi = \texttt{AttachEdge}(\texttt{ReplaceLabel}(\texttt{ExtColEdges} \\ (c_{\text{-}T_{id}}p, \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi''_{\mathcal{Q}}, r_{\mathcal{Q}_{dt}})), \phi''_{\mathcal{Q}}), c_{\text{-}T_{id}}p, \dot{v}_{al_{rf}}) \times \\ \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}{}'))) \cup \phi'_{\mathcal{Q}}\end{array}} (\mathcal{Q}\ \textsc{ClSQMchEx})$$

Since $c = \hat{t}$, $\dot{p} = \hat{cl}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ and $\phi''_{\mathcal{S}} = \phi''_{\mathcal{Q}}$ by inductive hypothesis, from the two cases above, **Case** ($\mathcal{S}$ ClDTMchEx) and **Case** ($\mathcal{Q}$ ClSQMchEx), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$. The more formal proof for this case is given below.

Case ($\mathcal{S}$ ClDTMchEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{dt})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq \texttt{null}$, $\hat{s} = \hat{t}.\hat{cl} \texttt{ AS } \hat{cl}_{al}$, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq\hat{sql}})$, $\hat{t} = \hat{t}_{alsq\hat{sql}}$, $\phi''_{\mathcal{S}} = \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq\hat{sql}})]\!]\phi$, and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq\hat{sql}}) = \texttt{true}$. By Rule ($\mathcal{S}$ ClDTMchEx),

$$\begin{aligned}\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi =& \texttt{AttachEdge}(\texttt{ReplaceLabel}(\texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, \\ & \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi''_{\mathcal{S}}, \texttt{GetRoot}(\phi''_{\mathcal{S}}))), \phi''_{\mathcal{S}}), \hat{t}_{\text{-}T_{id}}\hat{cl}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}) \times \\ & \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}})))) \cup \phi'_{\mathcal{S}}\end{aligned} \tag{8.3}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{t}.\hat{cl} \texttt{ AS } \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (h_s(\hat{t}.\hat{cl} \texttt{ AS } \hat{cl}_{al}, \hat{dt}, \hat{t}_{al}), h_w(\hat{dt}, \hat{t}_{al}))$. Since $\hat{dt}$ is not a table, i.e., $\texttt{IsTable}(\hat{dt}) = \texttt{false}$ and $\hat{t}_{al} \neq \texttt{null}$, by Rule ($h_s$ DTColumn AlSQ), $h_s(\hat{t}.\hat{cl} \texttt{ AS } \hat{cl}_{al}, \hat{dt}, \hat{t}_{al}) = (?\hat{t}_{\text{-}T_{id}}\hat{cl} \texttt{ AS } ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \cdot)$, and by Rule ($h_w$ SQL From - SubQuery), $h_w(\hat{dt}, \hat{t}_{al}) = h(\hat{sql}_{sq}, \hat{t}_{alsq\hat{sql}}) = \dot{spq}_{sq}$. Thus, $h((\hat{t}.\hat{cl} \texttt{ AS } \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (?\hat{t}_{\text{-}T_{id}}\hat{cl} \texttt{ AS } ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \dot{spq}_{sq})$. Also, observe that, in the case of sub-queries, where $\hat{t}_{alsq\hat{sql}} \neq \texttt{null}$, the SELECT item translation rules, (h_s Char/Int/{T/DT}Column AlSQ), always prefix the sub-query alias,

$\hat{t}_{alsq_{\hat{sql}}}$, to the alias of every corresponding SPARQL Query SELECT item. Thus $\hat{t}_{alsq\dot{spq}} = \texttt{GetSubQueryAlias}(h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}) = \hat{t}_{alsq_{\hat{sql}}}$.

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ and $\mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$ $= \mathcal{Q}[\![h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$. Substituting c for $\hat{t}$, and $\hat{t}_{alsq\dot{spq}}$ for $\hat{t}_{alsq_{\hat{sql}}}$, we have, $c = \hat{t}_{alsq\dot{spq}}$. Further, by inductive hypothesis, $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ and $\phi''_{\mathcal{S}} = \phi''_{\mathcal{Q}}$, and therefore, $\texttt{TableExists}($ $\texttt{GetRoot}(\phi'_{\mathcal{Q}}), \hat{t}_{alsq\dot{spq}}) = \texttt{true}$. Thus, since $\hat{t}_{al} \neq \texttt{null}$ and $\dot{v}_{al_{rf}} =$ $\texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, substituting $\dot{p}$ for $\hat{cl}$, we have, by Rule ($\mathcal{Q}$ CLSQMCHEX),

$$\begin{aligned}\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi =& \texttt{AttachEdge}(\texttt{ReplaceLabel}(\texttt{ExtColEdges}(c_{\text{-}T_{id}}\dot{p},\\ & \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi''_{\mathcal{Q}}, \texttt{GetRoot}(\phi''_{\mathcal{Q}}))), \phi''_{\mathcal{Q}}), c_{\text{-}T_{id}}\dot{p}, \dot{v}_{al_{rf}})\times\\ & \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, \texttt{GetRoot}(\phi'_{\mathcal{Q}})))) \cup \phi'_{\mathcal{Q}}\end{aligned} \tag{8.4}$$

.

Since $c = \hat{t}$, $\dot{p} = \hat{cl}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ and $\phi''_{\mathcal{S}} = \phi''_{\mathcal{Q}}$ by inductive hypothesis, from equations (8.3) and (8.4), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi =$ $\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Readers are referred to Appendices B and C for a complete listing of all the cases comprising Theorems 1 and 2 and their proofs respectively.

CHAPTER 9
UPDATE-ENABLED TRIPLIFICATION OF RELATIONAL DATA

The idea of a semantically annotated Web that would bestow upon machines enhanced interpretative abilities and enable a perception of the Web as one large database, thereby facilitating large scale data integration and reuse is rapidly gaining popularity. Endeavors to find a way to translate this idea into tangible reality resulted in the Semantic Web initiative which advocates resource description through the use of meta-data rather than keywords thereby enhancing computer understandability and reusability of information in web pages. In order to enable the definition and description of data (or resources) on the Web and the relations between them, several Semantic Web technologies such as Resource Description Framework (RDF), which is considered as one of the fundamental building blocks of the Semantic Web, RDF/S, and Web Ontology Language (OWL) have been proposed by the W3C. These technologies provide a means to integrate disparate data sources and reuse data across applications through the use of ontologies, and their flexibility and ease of adoption have resulted in their pervasive acceptance.

However, given the fact that approximately 70% of the websites that were available in 2007 were backed by relational databases (He, Patel, Zhang, and Chang 2007), and the percentage can only have increased further till date, it follows that the success and longevity of adoption of the Semantic Web depends heavily on enabling access of data within these relational databases to the Semantic Web. Moreover, relational databases, by virtue of having been in existence for several decades now, have the advantage of being equipped with sound, refined, and efficient query optimization, transaction support, data concurrency and security techniques. Thus, in order to reap the benefits offered by Semantic Web technologies while continuing to exploit the advantages of well-established and scalable relational database technologies, a means to enable the integration of the two technologies so as to create a best of both worlds scenario needs to be established. The need to address and provide a solution to this integration problem has prompted the initiation of the RDB2RDF incubator group (Malhotra 2009) whose primary objective is to establish mapping standards that facilitate relational database and RDF interoperability.

The problem of bridging Relational Database Management System (RDBMS) and RDF concepts has been the focus of several research efforts (Bizer and Seaborne 2004; Erling and Mikhailov 2009; Auer, Dietzold, Lehmann, Hellmann, and Aumueller 2009; Cerbah 2008) currently underway, each of which attempt to transform/propagate existing as well as newly added/modified data housed in relational databases into virtual RDF stores. However, almost all current solutions offer merely a read-only view of data from one domain into another. Thus, while one can view the relational data in RDF graph form and can query the resultant RDF triples using SPARQL (Prud'hommeaux and Seaborne 2008), RDF's native query language, data in the underlying relational database cannot be added to, deleted from, or altered in any way through the virtual RDF graph corresponding to the relational database. In this chapter, we propose a solution that eliminates this data modification restriction and allows data flow in either direction. Towards this end, we present D2RQ++ (Ramanujam, Khadilkar, Khan, Seida, Kantarcioglu, and Thuraisingham 2010; Ramanujam, Khadilkar, Khan, Kantarcioglu, Thuraisingham, and Seida 2010), a bi-directional data flow facilitating enhancement to an existing, extensively adopted relational-to-RDF read-only translation tool called D2RQ. This work was published in ***International Journal of Semantic Computing*** 2010, and was co-authored by Vaibhav Khadilkar, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham from University of Texas at Dallas (UTD) and Steven Seida from Raytheon Company. In addition to translating regular RDF triples, D2RQ++ includes the ability to propagate data changes specified in the form of RDF triples containing blank nodes and RDF reification nodes back to the underlying relational database.

Blank nodes, an important component of RDF graphs, are used to represent complex data. They are neither URI references nor literals and they enable association of a set of related properties with a resource, thereby creating a composite relationship.

RDF reification is a means of validating an RDF triple based on the trust level of another statement (Powers 2003) and is an important facility provided by RDF that enables users to make assertions about statements and record provenance data, thereby facilitating appropriate authentication of RDF data. No application or tool that works with RDF data is complete without RDF reification support and, hence, the original version of D2RQ++ (Ramanujam, Khadilkar, Khan, Seida, Kantarcioglu, and Thuraisingham 2010) has been augmented with algorithms that permit insert/update/delete of reification data. As before, when triples cannot explicitly be translated into equivalent concepts within the underlying relational database schema, D2RQ++ continues to adhere to the Open-World Assumption by permitting those triples to be housed in a separate native RDF store. When information on a particular entity is requested, the output returned is a union of the data pertaining to the entity from the relational database as well as any triples that have the entity as the subject and that may exist in the native RDF store. Thus, RDF triples submitted for insertion/update/deletion are never rejected due to mismatches with the underlying relational schema, thereby maintaining the Open-World Assumption of the Semantic Web world while still being able to work with technologies such as RDBMSs which are based on the Closed-World Assumption. To reiterate, the contributions of the D2RQ++ initiative include:

- Algorithms to translate RDF update triples into equivalent relational attributes/tuples thereby enabling DML operations on the underlying relational database schema.
- D2RQ Mapping language extensions and algorithms to support translation of a wide variety of blank node structures to equivalent relational tuples.
- Algorithms to support and permit triples that represent the concept of RDF reification to be propagated back into the underlying relational database schema through DML operations.

- Preservation of the Open-World Assumption by maintaining a separate native RDF store to house triples that are mismatched with the underlying relational database schema.

- Incorporation of the above algorithms and extensions into D2RQ++, an enhanced version of the highly popular D2RQ open-source relational-to-RDF mapping tool, and into D2R++-Server, the update-aware front-end graphical user interface (GUI) through which users can now issue DML requests using RDF data.

9.1 D2RQ++ - Our Approach

As stated earlier, the goal of our enhancement is to make the translation between RDF and RDBMS data stores bi-directional, thereby permitting update activities that can be propagated back to either the underlying RDBMS itself or to a native RDF store. We use the oldest and most recognized Employee-Department-Project relational schema depicted in Figure 9.1 to reinforce concepts wherever applicable in subsequent sections. Figure 9.1's scenario was chosen purely for elucidation purposes. D2RQ++, however, can be used on any relational schema and is not just restricted to the scenario depicted in Figure 9.1

There were several issues that needed to be addressed in order to achieve bi-directional translation and these are discussed in the following sub-sections.

9.1.1 Persistence of Unmapped and/or Duplicate Information

The first issue involved trying to preserve the Open-World assumption expected by Semantic Web applications and standards such as RDF, OWL, etc. during DML operations. In order to address this issue, we chose to maintain a separate native RDF store which would house all those triples that did not have an equivalent entity/attribute mapping within the underlying relational database schema. Even when an equivalent mapping exists, duplicate triples (such as a second *name* attribute value for a given employee) are housed in the native RDF store,

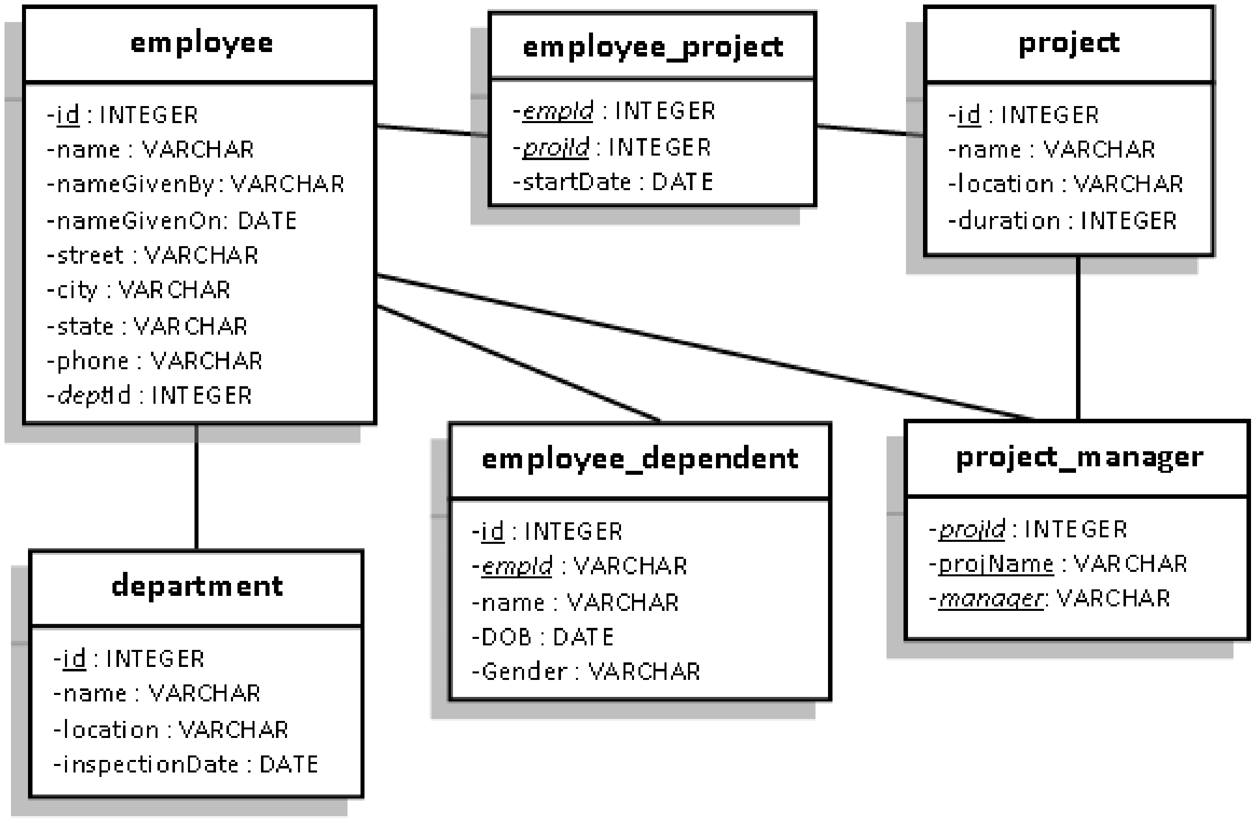


Figure 9.1: Relational Schema used to illustrate D2RQ++

Table 9.1: Extensions To D2RQ's Mapping Constructs

Construct	**Description**
d2rq:SimpleLiteralBlankNode-Property Bridge (SLBNPB)	Blank Nodes that have only literal objects, each with a unique predicate
d2rq:ComplexLiteralBlankNode-Property Bridge (CLBNPB)	Blank Nodes that have only literal objects; however, the predicates are not unique and include repetitions
d2rq:ResourceBlankNode-PropertyBridge (RBNPB)	Blank Nodes that have only resource objects with predicates that may or may not be unique
d2rq:BelongsToBlankNode	Construct that helps link a relational attribute to the parent blank node

instead of overwriting the existing value within the relational database schema and, thereby, losing the earlier information. Subsequent querying of the relevant employee information would return both *name* values (from the RDBMS and the RDF store) to the end user.

9.1.2 Mapping and Persistence of RDF Blank Nodes

The second issue involved arriving at a translation process for RDF blank nodes while continuing to re-use existing mapping languages such as D2RQ since this concept does not have an equivalent relational database mapping. In order to facilitate DML operations involving blank nodes we added several new mapping constructs to D2RQ's mapping language. These constructs are listed in Table 9.1.

The first three constructs essentially identify a specific concept in the relational database schema as a blank node and include information, specified through the "d2rq:pattern" construct, on the format in which object values should be specified for the blank node. These blank node classifications are similar to, and based on, the scenarios and examples discussed in Chapter 4 and more details can be found in the same. The last construct is used to identify those attributes within the relational database schema that together make up the d2rq:{SL/CL/R}BlankNodeProperty-Bridge. This (last) construct is used to associate the attributes comprising a blank node with their parent blank node. In order to better understand how blank nodes are mapped in D2RQ++, let us consider the address attributes in the employee entity, viz., *street, city,* and *state.* If the end user prefers to view (or update) these address attributes in the form of a blank node that contains these attributes as objects, the following are the mapping statements the user would add to D2RQ's map file.

```
map:employee_address a
        d2rqrw:SimpleLiteralBlankNodePropertyBridge;
        d2rq:belongsToClassMap map:employee;
        d2rq:property vocab:employee_address;
        d2rq:propertyDefinitionLabel "employee address";
        d2rq:pattern "@@employee.address_street@@/
                                 @@employee.address_city@@/
                                 @@employee.address_state@@";
        .
map:employee_address_street a d2rq:PropertyBridge;
        d2rqrw:belongsToBlankNode  map:employee_address;
        d2rq:belongsToClassMap map:employee;
        d2rq:property vocab:employee_address_street;
        d2rq:propertyDefinitionLabel "employee address_street";
        d2rq:column "employee.address_street";
```

For readability reasons, the entry corresponding to only one of the address attributes *(street)* is shown above. Similar entries will have to be included for the other two attributes *(city, state)* as well. The *employee_address* blank node is characterized as a SLBNPB since every object belonging to the blank node (i.e., *street, city, state*) is a simple literal and each of these objects has a unique predicate. More details on updating blank nodes are presented in section 9.2 along with the appropriate algorithms.

9.1.3 Mapping and Persistence of RDF Reification Nodes

The third issue, similar to the second issue above, involved arriving at a translation process for RDF reification nodes while continuing to re-use D2RQ's existing mapping language since this concept does not have an equivalent relational database mapping. RDF reification is a feature through which a triple can be encapsulated as a resource in order to enable additional statements about that resource to be made that help establish the degree of confidence and trustworthiness of the triple. This issue was resolved by extending D2RQ's mapping language with new constructs that are capable of handling reification data, and providing two ways in which reification information can be persisted using D2RQ++. The first alternative is trivial and involves storing all reification information directly into the native RDF store. This method is adopted by D2RQ++ when appropriate tables/columns to house reification information do not exist in the underlying relational database or when mapping information pertaining to reification nodes is not available in D2RQ++'s mapping file. The second option involves storing the data within the underlying relational database schema and is available if the schema includes appropriate tables and columns within which reification data can be stored. In-depth details on the mapping and persistence of reification nodes are provided in section 9.3.

9.1.4 Maintenance of Open-World Assumption through Periodic Consolidation

The fourth issue was concerned with establishing the order of update activities (in the case of batch updates) in order to ensure that referential integrity constraints do not force an update rejection due to incorrect update sequences such as an *employee* triple of the form {<*empURI*> <*DeptID*> <*DepartmentID*>} arriving before the actual *department* triple {<*deptURI*> <*ID*> <*DepartmentID*>}. This issue was resolved by blindly accommodating triples that violate referential integrity constraints in the native RDF store and introducing a periodic consolidation/flush algorithm. This consolidation/flush algorithm periodically validates the RDF store contents against the underlying relational database schema to identify

those triples that now have parent key values in the relational schema corresponding to their object values, i.e., to identify those triples that earlier violated foreign key constraints but now no longer do because the parent key is now present in the relational schema. Once these triples are identified, the flush algorithm transfers them into the relational schema by following Algorithm 9.1 and removes them from the native RDF store. The flush algorithm also consolidates duplicate triples in the event the underlying relational database column corresponding to the triples predicate is updated to a null value. These duplicate triples were originally accommodated in the native RDF store because the corresponding column in the underlying relational database had a non-null value. Whenever these columns are updated to null values, the flush algorithm consolidates duplicate triples back into the underlying database and deletes the same from the native RDF store.
The next section presents the various algorithms that were developed to address and resolve the issues presented above.

9.2 D2RQ++ Algorithms for Regular Triples and Blank Nodes

9.2.1 Insert/Update Operations on Regular Triples

Insert/Update operations on regular RDF triples are fairly straightforward and are applicable to simple triples that involve a literal or resource object and that do NOT involve any blank nodes. The only time an INSERT statement is executed against the underlying relational schema is when the predicate exists as a column in the table to which the subject of the triple belongs and the subject value itself does not exist as a primary key value in the same table. When the predicate exists as a column in the table and the subject exists as a primary key value in the same table, the object value is updated (using an SQL UPDATE statement) only if the corresponding cell in the relational schema is empty. If not, under the Open-World Assumption, the object in the input triple is considered to be a duplicate value for the corresponding column and is preserved by housing the triple in the native RDF store. Subsequent querying of that column will return both values, i.e., the cell value stored in the relational database as well as the object value for the corresponding predicate stored in the native RDF store. In the event the predicate of the input triple does not map to an equivalent column in the underlying relational schema as specified in the mapping file, the input triple is always added into the native RDF store. Algorithm 9.1 lists the details of insert/update operations on regular RDF triples.

Algorithm 9.1 INSERT/UPDATEREGULARTRIPLE

Input: An RDF Triple
Output: A successful RDBMS/RDF Store update

```
1: Identify table assoicated with triple's subject
2: if triple is a Blank Node then
3:    Call Insert/UpdateLiteral/ResourceBlankNodeTriple - Algorithms 9.2 and 9.3
4: else
5:    if subject does not exist in RDBMS then
6:       if predicate exists in subject's table in RDBMS then
7:          Insert triple as new tuple in RDBMS and Return
8:       else
9:          Add triple to native RDF Store and Return
10:      end if
11:   else
12:      if predicate exists in subject's table in RDBMS then
13:         if (object.isLiteral()) or (object.isResource() and exists( object as PK in another
            table)) then
14:            if RDBMS table cell value is NULL then
15:               Update triples obj value in column and Return
16:            end if
17:         end if
18:      end if
19:      Add triple to native RDF Store and Return
20:   end if
21: end if
```

9.2.2 Insert/Update Operations on Blank Node Triples

Triples that contain blank node objects are handled using the procedures illustrated in Algorithms 9.2 and 9.3. Algorithm 9.2 illustrates the procedure for simple or complex literal blank nodes while Algorithm 9.3 deals with resource blank nodes.

Algorithm 9.2 Insert/UpdateLiteralBlankNodeTriple

Input: An RDF Triple with Blank Node Object
Output: A successful RDBMS/RDF Store update

```
 1: if BlankNode definition not found in Map File then
 2:     Add Blank Node structure to native RDF Store and Return
 3: end if
 4: if BlankNode.Type = SLBNPB then
 5:     for every predicate off of the blank node do
 6:         if  predicate exists as column in subject table in RDBMS then
 7:             if corresponding column value is NOT NULL then
 8:                 Add entire blank node structure to the native RDF Store and Return
 9:             end if
10:         else
11:             Add triple to native RDF Store and Return
12:         end if
13:     end for
14:     if subject of SLBNPB does not exist in RDBMS then
15:         Insert subject and objects of SLBNPB as a new tuple in RDBMS and Return
16:     else
17:         Update the object values of SLBNPB in corresponding columns against subject
            tuple and Return
18:     end if
19: else if BlankNode.Type = CLBNPB then
20:     if subject of CLBNPB does not exist in RDBMS then
21:         Add CLBNPB to native RDF Store and Return
22:     else
23:         Get RDBMS table corresponding to object of CLBNPB
24:         for every predicate belonging to CLBNPB do
25:             Insert subject, predicates off of CLBNPB, and objects of CLBNPB into RDBMS
                table in the Foreign Key, Type, and Value fields respectively and Return
26:         end for
27:     end if
28: end if
```

SLBNPBs translate into a collection of simple attributes that belong to the same table in the underlying relational schema. An example of an SLBNPB has been discussed in the previous sub-section. SLBNPBs are inserted/updated into the underlying relational database only when every attribute value comprising the SLBNPB is empty. Even if one attribute value comprising the SLBNPB has a non-null value in the relational schema (for example, if the *employee* table has, for a particular employee, a non-null value for the *state* column while the *street* and the *city* columns are null), or if the input SLBNPB has a structure that is different from the specification included in the map file (i.e., it has less/more constituent attributes than the map file specification) the entire SLBNPB structure is housed in the native RDF store instead. However, this condition (of requiring all constituent attributes to be null in order to achieve a successful insert or update activity) is a design choice adopted by us and can be changed into other suitable choices as required by the end-user of the system.
CLBNPBs represent 1:N relationships between a subject and an object concept within the underlying relational database. An example of such a relationship is the *employee-phone* relationship where an employee has 1 or more phone numbers while a phone number belongs to one and only one employee. CLBNPBs map to normalized tables (such as an *employee_phone* table with possible attributes of *EmpID, PhoneType, PhoneNumber*, all of which form a combined primary key with the *EmpID* being a foreign key that references the parent *employee* table) that result from such 1:N relationships. CLBNPBs are inserted into the underlying relational table iff the subject of the CLBNPB exists in the parent table (In our *employee-phone* example, phone details are inserted into the *employee_phone* table iff the corresponding employee URI exists in the parent *employee* table). For every predicate belonging to the CLBNPB a new tuple is inserted into the underlying relational table with the CLBNPB subject, the predicate of the CLBNPB, and the object forming the values of the foreign key, the type, and the value columns of the relational table. In the event the subject of the CLBNPB does not exist in the parent table, the entire CLBNPB structure is housed in the native RDF store and is consolidated into the relational database by the

periodic flush algorithm when the CLBNPB's subject is added into the parent table.

ResourceBlankNodePropertyBridges are blank nodes that comprise only of resource objects that may or may not belong to the same object class. In the case of RBNPBs the objects of the blank node are inserted into the underlying relational database tables iff the RBNPB subject and every one of the RBNPB object values exist as primary key values in their corresponding tables in the underlying relational schema; otherwise the entire RBNPB structure is added into the native RDF store as illustrated in Algorithm 9.3.

Algorithm 9.3 INSERT/UPDATERESOURCEBLANKNODETRIPLE

Input: An RDF Triple with Blank Node Object
Output: A successful RDBMS/RDF Store update

```
 1: if BlankNode definition not found in Map File then
 2:    Add Blank Node structure to native RDF Store & Return
 3: end if
 4: if subject of RBNPB does not exist in RDBMS then
 5:    Add RBNPB structure to native RDF store & Return
 6: end if
 7: for every predicate belonging to RBNPB do
 8:    Get table corresponding to resource object class in RDBMS
 9:    if object value does not exist in RDBMS table then
10:       Add RBNPB structure to native RDF store & Return
11:    end if
12:    if RBNPB's subject and current object class relationship is I:N then
13:       Get 1-side table's field in N-side table
14:       if field value is not NULL then
15:          Add RBNPB structure to native RDF store & Return
16:       end if
17:    end if
18: end for
19: for every predicate belonging to RBNPB do
20:    if RBNPB subject and predicate object relationship = 1:N then
21:       Update 1-side field in N-side table with subject value
22:    else
23:       Insert subject and object values as new tuple in N:M (join) table
24:    end if
25: end for
```

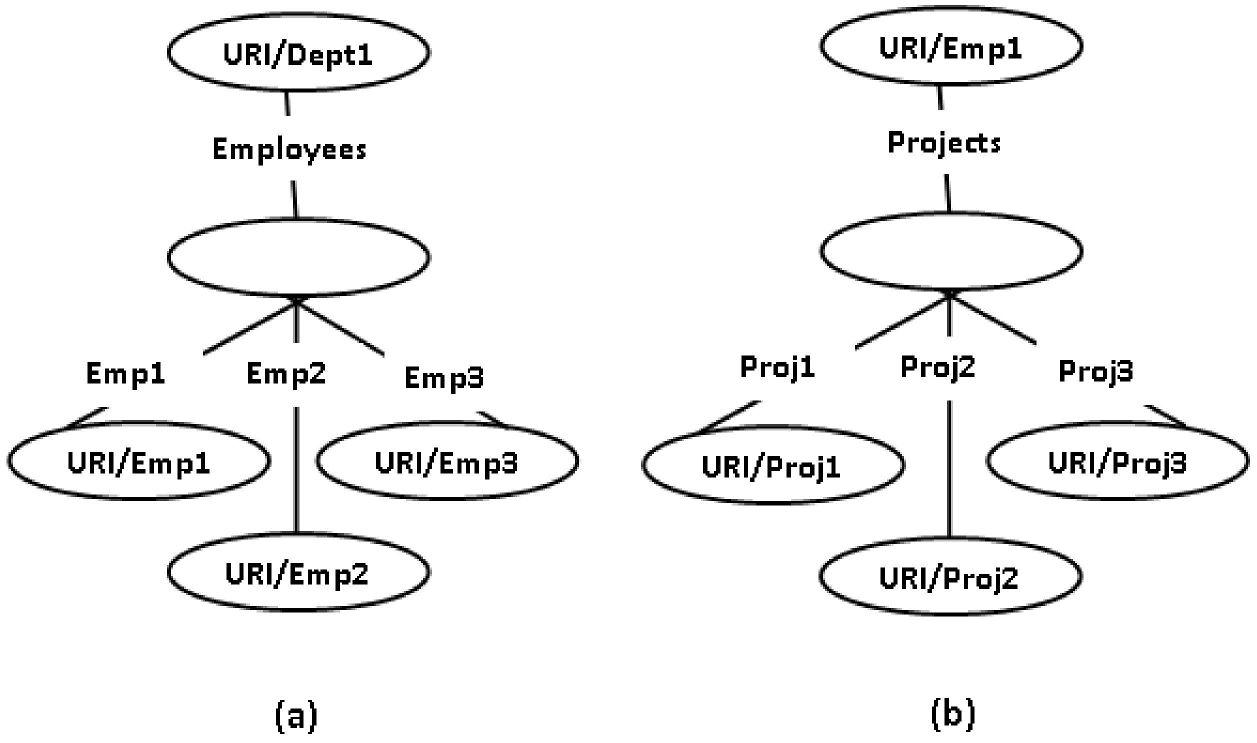


Figure 9.2: Sample RBNPB Scenarios

Two example RBNPBs are illustrated in Figure 9.2.

If the RBNPB subject and object share a 1:N relationship as illustrated in Figure 9.2(a) between *employee* and *department* entities, for every predicate of the RBNPB, the tuple with the object value is located in the object table (i.e., the *employee* table) and the subject value (i.e., *departmentID*) is updated into the appropriate field (*deptId* in *employee* table) in the N-side table (i.e., in the *Employee* table) . If they share an N:M relationship like the one between the *employee* and *project* entities illustrated in Figure 9.2(b), a new tuple is inserted with the subject and object URIs as values in the respective columns (i.e., in *empID* and *projID* columns) in the join table (i.e. *employee_project* table) representing the N:M relationship.

9.2.3 Consolidation Procedure

As stated earlier, the order of insert/update activities may result in certain triples being rejected from the RDBMS and being housed temporarily in the native RDF store instead due to the violation of referential integrity constraints. Periodically, the triples in the native RDF store are validated against the underlying relational database and any triple that no longer violates referential integrity constraints is transmitted back to the relational database and deleted from the native RDF store using the Flush Algorithm, Algorithm 9.4.

Algorithm 9.4 FLUSH (FOR REFERENTIAL INTEGRITY CONSTRAINTS)

Input: Triples in native RDF Store
Output: Possible insert/update in RDBMS and associated delete from RDF Store

```
1: for every triple in RDF Store do
2:   if Object.type = Resource then
3:     if Exists(Predicate in Subject table) then
4:       if Exists(Object value in Object table) then
5:         if Exists(Subject value in Subject table) then
6:           if column corresponding to Object == NULL in Subject table then
7:             Update object value in column in Subject table
8:             Delete triple from native RDF Store and Return
9:           end if
10:        else
11:          Insert subject and object values in appropriate columns in Subject table
12:          Delete triple from native RDF Store and Return
13:        end if
14:      end if
15:    end if
16:  end if
17: end for
```

This algorithm, run periodically, is also quite straightforward and applies to any triple in the native RDF store with a resource object. Whenever the object value of such a triple is found to exist as a primary key in the table corresponding to the Object Resource class, and the predicate of the triple exists as a column in the table corresponding to the subject of the triple, if the subject does not already exist in the subject table in the RDBMS, a new tuple is inserted with the subject and object values of the triple being updated in the appropriate relational columns. If the subject exists as a primary key in the underlying RDBMS table and the current predicate column value is null, it is updated and set to the object value of the triple; otherwise no update happens and the triple continues to remain in the native RDF store. In the event a successful insert/update of the triple was accomplished in the underlying relational schema, the triple is then deleted from the native RDF store. In this manner, referential integrity violations are given opportunities to return to the underlying relational schema periodically. Similar flush procedures exist for SLBNPBs, CLBNPBs, and RBNPBs as well, however, as they are intuitive and repetitive, they are not presented here.

9.2.4 Delete Operations on RDF Triples and Blank Nodes

Algorithm 9.5 highlights the process to delete a regular triple from either the RDBMS or the native RDF store as applicable. The only scenario in which a regular triple deletion fails is if the subject value of the triple is referenced as a foreign key by another relational database table. If either the subject or the predicate of the triple do not exist as a primary key value or column, respectively, in the underlying relational database, the triple is assumed to be housed in the native RDF store and is deleted from that store. If the column corresponding to the triple predicate has a value that differs from the triple's object value, this implies that the triples object value is a secondary value for that column. In this case as well the triple is housed in the native RDF store and, hence, is deleted from the RDF store. If the column value matches the triple's object value and the column corresponding to the triple's predicate is not the primary key column, the tuple corresponding to the subject row is updated with a null value for the column; otherwise the tuple corresponding to the subject row is deleted completely from the underlying table.

Delete procedures for SL/CL/RBNPBs are illustrated in Algorithms 9.6 and 9.7 and are self-explanatory.

The next section discusses the mapping language extensions and algorithms that enable inserts, updates, and deletes of reification information to be either propagated back to the relational database schema or stored in a native RDF store as applicable.

Algorithm 9.5 DeleteRegularTriple

Input: An RDF Triple

Output: Possible successful RDBMS/RDF Store delete

```
 1: if !Exists(Subject) in subject table in RDBMS then
 2:    Find and Delete triple from native RDF Store and Return
 3: else
 4:    if !Exists(Predicate in subject table in RDBMS) then
 5:       Find and Delete triple from native RDF Store and Return
 6:    else
 7:       if subject referenced by another RDBMS table then
 8:          Deny Delete and Return
 9:       else
10:          if object value exists in predicate column in subject table then
11:             if predicate column = Primary Key Column (PK) then
12:                Delete from subject table where PK = subject and Return
13:             else
14:                Update subject table set predicate column = NULL Where PK = subject
                   and Return
15:             end if
16:          else
17:             Find and Delete triple from native RDF Store and Return
18:          end if
19:       end if
20:    end if
21: end if
```

Algorithm 9.6 DELETELITERALBLANKNODETRIPLE

Input: An RDF Triple with Literal Blank Node object
Output: A successful RDBMS/RDF delete

```
1:  if BlankNode definition not found in Map File then
2:      Delete BlankNode structure from native RDF Store and Return
3:  end if
4:  if BlankNode structure does not match Map File definition then
5:      Delete BlankNode structure from native RDF Store and Return
6:  end if
7:  if subject of BlankNode does not exist in RDBMS then
8:      Delete BlankNode structure from native RDF Store and Return
9:  end if
10: if BlankNode.Type = SLBNPB then
11:     for every predicate off of the BlankNode do
12:         if value in column (of subject table in row where PK = SLBNPB's subject) corre-
            sponding to predicate != predicate's object then
13:             Delete BlankNode structure from native RDF Store and Return
14:         end if
15:     end for
16:     UPDATE subject table set values of columns corresponding to SLBNPB's predicates
        to NULL in subject table WHERE PK = SLBNPB's subject
17: else if BlankNode.Type = CLBNPB then
18:     Get RDBMS table corresponding to object of CLBNPB
19:     for every predicate belonging to CLBNPB do
20:         DELETE subject, predicate, and object of CLBNPB FROM RDBMS table from
            the Foreign Key, Type, and Value fields respectively and Return
21:     end for
22: end if
```

Algorithm 9.7 DELETERESOURCEBLANKNODETRIPLE

Input: An RDF Triple with Resource Blank Node object

Output: A successful RDBMS/RDF delete

```
 1: if BlankNode definition not found in Map File then
 2:    Delete BlankNode structure from native RDF Store and Return
 3: end if
 4: if BlankNode structure does not match Map File definition then
 5:    Delete BlankNode structure from native RDF Store and Return
 6: end if
 7: if subject of BlankNode does not exist in RDBMS then
 8:    Delete BlankNode structure from native RDF Store and Return
 9: end if
10: for every predicate belonging to RBNPB do
11:    Get tables corresponding to resource subject and object classes in RDBMS
12:    if object value does not exist in object RDBMS table then
13:       Delete RBNPB structure from native RDF store and Return
14:    end if
15:    if RBNPB's subject and current object class relationship is I:N then
16:       Get object table's field corresponding to subject
17:       if field value != subject value WHERE table_PK = object value then
18:          Delete RBNPB structure from native RDF store and Return
19:       end if
20:    end if
21: end for
22: for every predicate belonging to RBNPB do
23:    if RBNPB subject and predicate object relationship = 1:N then
24:       UPDATE object table SET field corresponding to subject to NULL WHERE object
          table PK = object value of predicate
25:    else
26:       DELETE subject and object values FROM N:M (join) table
27:    end if
28: end for
```

9.3 Bi-directional Translation of RDF Reification Nodes

RDF reification, as described in section 9.1.3, is an important feature that enables the establishment of the degree of confidence and trustworthiness of a triple. RDF reification nodes consist of four mandatory properties: rdf:subject, which identifies the subject in the original triple being reified, rdf:predicate, which identifies the property in the statement being reified, rdf:object, which identifies the object of the statement being reified, and, lastly, rdf:type, which is always rdf:statement since all reified statements are instances of rdf:Statement. These four properties are called the reification quad and these can be accompanied by one or more non-quad properties that detail the actual provenance information. These non-quad predicates can have one or more literal, resource, or blank node objects as illustrated in the sample scenario in Figure 9.3 below.

In the graph, every solid node with outgoing edges, such as *URI/Emp1* and *URI/Dept1*, represents a subject/resource. Solid edges, such as *Street, City, State*, and *Phone*, represent predicates and the solid nodes at the end of the edges, such as *<Street>, <City>, <State>*, and *<Phone>*, represent objects. Empty solid nodes, such as the node at which the *Projects* predicate terminates represent blank nodes. The nodes in dashed lines with the “s”, “p”, “o”, and “t” predicates, amongst others, represent reified nodes. “s”, “p”, “o”, and “t” represent the “rdf:subject”, “rdf:predicate”, “rdf:object”, and the “rdf:type” predicates of the reification quad. Other predicates of the reification nodes (other than “s”, “p”, “o”, and “t” predicates) represent non-quad predicates (NQPs). Empty nodes in dashed lines (other than the reified nodes) that are the objects of non-quad reification predicates, such as the object of the *ManagedBy*, and *NameDetails* predicates, represent reification blank nodes. The non-quad reification properties chosen in this example may not represent actual provenance information. They were primarily chosen to illustrate proof of concept. Elements of Figure 9.3 are used, wherever applicable, to facilitate better comprehension of the information presented in subsequent sections.

Reification information can be persisted using D2RQ++ in two ways as detailed in sec-

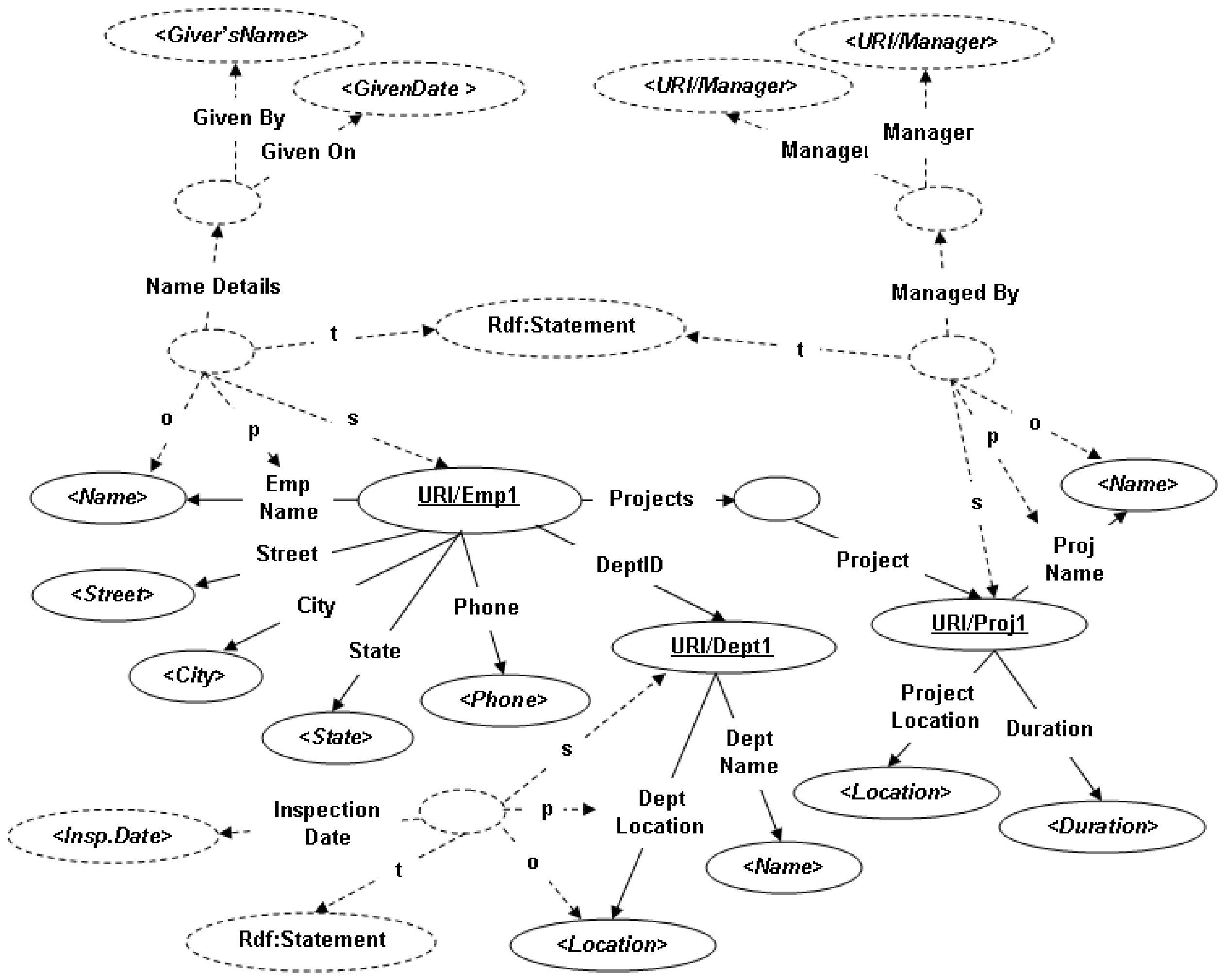


Figure 9.3: Sample RDF Scenario with Reification Nodes

tion 9.1.3. As can be seen in Figure 9.3, reification nodes comprise different types of non-quad predicates. We have classified reification nodes into five broad categories based on the type of non-quad predicates they contain and these categories are described in the following sub-section.

9.3.1 Reification Node Categories and their Relationalization

a) *Reification Nodes (RNs) with non-repeating literal/resource NQPs*

   As the category implies, these RNs contain one or more predicates, each of which is unique, and each of which leads to a literal or resource object. An example of these kinds of RNs is the RN with the *InspectionDate* NQP, leading to a literal object, in Figure 9.3.

   i. *Relational Storage of RNs with literal rdf:objects:* Storage of this kind of reification information when the RN reifies a literal object (i.e. rdf:object is a literal) within a relational database requires columns for each of the non-repeating NQPs in the table corresponding to the rdf:subject of the RN, and storing the NQP objects in those columns. Thus, the object of the *InspectionDate* NQP in Figure 9.3 is stored within the *inspectionDate* column in the *department* table (corresponding to the *URI/Dept1* rdf:subject).

   ii. *Relational Storage of RNs with resource rdf:objects:* In the case of resource reification objects (i.e. resource rdf:objects), if the relationship between rdf:subject and rdf:object of the RN is 1:N the NQPs correspond to columns in the table on the N-side of the relationship; if the relationship is N:M the NQPs correspond to columns in the join table that results from the translation of this N:M relationship.

b) *RNs with single or multiple groups of repeating literal/resource NQPs*

   These RNs also have NQPs leading to literal or resource objects; however, the RN's NQPs need not be unique and can repeat for multiple object values. For example, if a department location has been inspected multiple times, the corresponding reification information in Figure 9.3 will include multiple NQPs, with a repeating predicate of *InspectionDate*, each of which has a unique inspection date object. This is an example of a single group of repeating literal NQPs (since only one predicate is repeated. Multiple groups of repeating predicates are scenarios where more than one predicate, each of which are repeated a number of times, exist.)

i. *Relational Storage of RNs with literal rdf:objects:* RNs in this category with literal rdf:objects are stored within a relational database in a table with the following columns (*rdf:subject Table PK, rdf:predicate column, type, value*) with the primary key comprising all four columns. The first column is a foreign key that references the primary key of the table corresponding to the RN's rdf:subject (the *department* table in our example), the second column is the one in the rdf:subject's table that corresponds to the RNs rdf:predicate (the value stored in this field is the rdf:object value), the third column (*type*) stores the name of each NQP belonging to the RN, and the fourth column (*value*) stores the object value of each of the NQPs.

ii. *Relational Storage of RNs with resource rdf:objects:* To store RNs containing resource rdf:objects in a relational schema, the repeating NQPs and their objects are stored within the *type* and *value* fields in the join table (which also contains the (*rdf:subject table PK, rdf:objectTablePK*) foreign key fields) for the rdf:subject, rdf:object pair.

c) *RNs with Simple Literal Blank Nodes (SLBNs)*

These are RNs whose non-quad predicates comprise simple literal blank nodes and an example of one such RN is the one with the *NameDetails* NQP in Figure 9.3.

i. *Relational Storage of RNs with literal rdf:objects:* Relational storage of RNs in this category with literal rdf:objects involves translating each of the predicates comprising the SLBN non-quad predicate into appropriate columns in the table corresponding to the RN's rdf:subject (similar to the relationalization process in category (a) (i) above). Thus, information pertaining to the *NameDetails* literal SLBN NQP in

Figure 9.3 is stored within the columns *nameGivenBy* and *nameGivenOn* belonging to the *employee* (rdf:subject) table in the relational schema in Figure 9.1.

ii. *Relational Storage of RNs with resource rdf:objects:* The procedure for storing RNs with resource rdf:objects in this category within a relational database is identical to the process detailed in category (a) (ii).

d) *RNs with Complex Literal Blank Nodes (CLBNs)*

RNs whose NQPs comprise of CLBNs fall into this category. A scenario where the *NameDetails* reification SLBN NQP in Figure 9.3 consists of two, instead of one, *GivenBy* predicates and one *GivenOn* predicate is an example of a CLBN NQP since there are multiple non-unique (repeating) predicates belonging to the *NameDetails* blank node.

Relational Storage: Relational storage of predicates belonging to CLBN NQPs of RNs with literal and resource rdf:objects in this category requires the same procedures as detailed in category (b) (i) and (ii) respectively.

e) *RNs with Resource Blank Nodes (RBNs)*

This category comprises RNs whose NQPs consist of resource blank nodes. An example of such an RN in Figure 9.3 is the RN with the *ManagedBy* NQP which is an RBN consisting of multiple *Manager* entities which are resources, possibly of type *employee*.

i. *Relational Storage of RNs with literal rdf:objects:* These RNs are stored within a relational database in a table with the following columns: (*rdf:subject Table PK, rdf:predicate column, predicate of RNs RBN NQP*) where the third column is a foreign key referencing the table corresponding to the objects of the RBN. Thus, the relational translation of the RN with the *ManagedBy* RBN NQP results in the addition of a new tuple in the *project_manager* table with the fields (*projId, projName, managedBy*) where *projId* is a foreign key referencing the primary key of the *project* table and *managedBy* is a foreign key referencing the *employee* table.

ii. *Relational Storage of RNs with resource rdf:objects:* Relationalization of RBN RNs with resource rdf:objects involves identifying the join table (containing the (*rdf:subject* *table PK, rdf:objectTablePK*) foreign key fields) representing the relationship between the rdf:subject and rdf:object classes, identifying the column that is a foreign key referencing the table corresponding to the resource objects of the RBN NQP, and adding the objects of the RBN NQP as new tuples in the identified table and column.

There are several other categories of reification nodes such as RNs that contain multiple object classes, with or without unique predicates, RNs that contain nested blank nodes, and RNs that contain a mixture of literals, resources, and blank nodes. Further, several other scenarios where reification can be applied also exist such as reification of multivalued attributes, reification of triples where the subject or object is a blank node. However, the scope of the current version of D2RQ++ is restricted to the five categories described above and the incorporation of these extended scenarios is planned in subsequent enhancements to D2RQ++.

9.3.2 Mapping Language Extensions for Reification Support

For SPARQL queries and other tools that work with the virtual RDF store generated by D2RQ++ to distinguish between regular tables/columns and tables/columns that cater exclusively to reification data a mapping scheme that permits differentiation between the two kinds of tables/columns needs to be established. In order to achieve this requirement, we further extended D2RQ's mapping language by adding several mapping constructs specifically for reification support. These constructs are listed in Table 9.2.

The first two constructs are used to identify provenance data stored within the relational database schema and map the same as reification nodes to ensure that the appropriate reification quads are generated in the virtual RDF graph generated through the translation of the relational schema. The last construct is used to identify those columns in the relational schema that correspond to a reification node's non-quad predicates and associate those columns with the parent reification node.

In order to better understand the mapping details pertaining to reification information, let us

Table 9.2: RDF Reification Extentions To D2RQ's Mapping Constructs

Construct	**Description**
d2rqrw:ReificationNode (RN)	Construct used to map reification nodes
d2rqrw:ReifiedPropertyBridge	Construct used to associate relational attribute corresponding to the predicate of the triple that is being reified (by the reification node) to the reification node
d2rqrw:belongsToReification-Node	Construct that helps associate a relational attribute corresponding to the reification nodes non-quad predicate to the reification node

consider the attributes *employee.nameGivenBy, employee.nameGivenOn, department.inspectionDate*, and *project_manager.managedBy* in Figure 9.1 that correspond to NQPs in reification nodes. The first two attributes are examples of an RN with an SLBN NQP, the third attribute is an example of an RN with a simple literal NQP, and the last attribute is an example of an RN with an RBN NQP. These attributes are translated into appropriate reification node predicates in the equivalent virtual RDF store using the following mapping statements in D2RQ++'s mapping file.

```
map:employee_name_reif a  d2rqrw:ReificationNode;
        d2rq:belongsToClassMap map:employee;
        d2rqrw:reifiedPropertyBridge map:employee.name;

map:employee_name_nameDetails a  d2rqrw:SimpleLiteralBlankNodePropertyBridge;
        d2rqrw:belongsToReificationNode map:employee_name_reif;
        d2rq:property vocab:employee_name_nameDetails;
        d2rq:propertyDefinitionLabel "employee Name NameDetails";
        d2rq:pattern "@@employee.nameGivenBy@@/
```

```
                                @@employee.nameGivenOn@@";
        .
map:employee_nameGivenBy a d2rq:PropertyBridge;
        d2rqrw:belongsToBlankNode  map:employee_name_nameDetails;
        d2rq:belongsToClassMap map:employee;
        d2rq:property vocab:employee_nameGivenBy;
        d2rq:propertyDefinitionLabel "employee nameGivenBy";
        d2rq:column "employee.nameGivenBy";
```

Since the entry corresponding to *nameGivenOn* is similar to the *nameGivenBy* entry above, it is omitted from the chapter. The mapping entry for *department.inspectionDate* is very similar to the mapping for *employee.nameGivenBy* entry above except that the SLBNPB entry is excluded here since the *inspectionDate* NQP is a direct predicate to its RN without any intermediate blank nodes. The mapping entries for RNs with RBN NQPs such as the one corresponding to the *project_manager.managedBy* attribute are given below.

```
map:project_manager_projName_reif a  d2rqrw:ReificationNode;
        d2rq:belongsToClassMap map:project_manager;
        d2rq:refersToClassMap map:employee;
        d2rqrw:reifiedPropertyBridge map:project_manager.name;

map:project_manager_name_managedBy a  d2rqrw:ResourceBlankNodePropertyBridge;
        d2rqrw:belongsToReificationNode map:project_manager_projName_reif;
        d2rq:property vocab:project_manager_name_managedBy;
        d2rq:propertyDefinitionLabel "project_manager name managedBy";
        d2rq:pattern "@@project_manager.manager@@/
                                @@ project_manager.manager @@";
        .
map:project_manager_manager a d2rq:PropertyBridge;
        d2rqrw:belongsToBlankNode  map: project_manager_name_managedBy;
        d2rq:belongsToClassMap map:project_manager;
```

```
d2rq:refersToClassMap map:employee;
d2rq:property vocab:project_manager_manager;
d2rq:propertyDefinitionLabel "Project_Manager manager";
d2rq:column "project_manager.manager";
```

The above mappings enable reification information stored in relational database schemas to be mapped to appropriate reification nodes with all intermediate edges (such as blank node NQP edges) maintained, thus enabling accurate RDF graph transformations of relational reification attributes . Algorithms to insert, update, and delete information in RDF reification nodes within the underlying relational database schema are presented in the following sub-section.

9.3.3 D2RQ++ Algorithms for Reification Nodes

Insert and Update operations on RNs belonging to categories (a) and (c) in section 9.3.1 are detailed in Algorithm 9.8. As can be seen in the algorithm, if the incoming reification node is not mapped appropriately in D2RQ++'s mapping file or if there is a mismatch in the map file definition and the actual node structure, the reification node is stored in the native RDF store instead of in the relational database schema. This is also the case when the rdf:predicate of the RN does not exist as a column in the table corresponding to the rdf:subject, when the rdf:object value does not exist in the column corresponding to the RN's rdf:predicate, when any one of the NQP predicates (in case (a)) or predicates of the SLBN NQP (in case (b)) do not exist as columns in the rdf:subject's table, or when any of the NQP predicates or predicates of the SLBN NQP have non-null values.
When none of the above conditions exist and if the rdf:subject value of the RN does not exist as a value in the rdf:subject's table, an INSERT statement is issued as illustrated in line 5 of Algorithm 9.8. If the rdf:subject value exists in the appropriate table then an UPDATE statement is issued as illustrated in line 11 of Algorithm 9.8.

Algorithm 9.9 highlights the process to delete reification nodes belonging to categories (a) and (c) from either the RDBMS or the native RDF store as applicable.

Algorithm 9.8 INSERT/UPDATEL/R/SLBNREIFICATIONNODE

Input: An RDF Reification Node with literal/resource or SLBN NQPs
Output: A successful RDBMS/RDF Insert/Update

```
 1: if ReificationNode (RN) definition not found in Map File OR
    RN structure does not match Map File definition OR
    RN's rdf:predicate column not present in table corresponding to RN's rdf:subject OR
    One or more fields corresponding to RN's NQPs or SLBN NQP's predicates not present
    in table then
 2:     Add RN to native RDF Store and Return
 3: end if
 4: if RN's rdf:subject value does not exist in table corresponding to RN's rdf:subject then
 5:     INSERT rdf:subject value, rdf:object value, and values of all NQP objects or objects
        of SLBN NQPs into rdf:subject Table PK, column corresponding to RNs rdf:predicate,
        and columns corresponding to non-quad predicates of RN and Return
 6: end if
 7: if RN's rdf:subject value exists but rdf:object value does not exist in rdf:predicate column
    of table then
 8:     Add RN to native RDF Store and Return
 9: end if
10: if every column corresponding to RN's NQP or SLBN NQP's predicates is NULL then
11:     UPDATE NQP columns SET values = objects of NQPs in rdf:subject table WHERE
        PK = rdf:subject value and Return
12: else
13:     Add RN to native RDF and Return
14: end if
```

Algorithm 9.9 DeleteL/R/SLBNReificationNode

Input: An RDF Reification Node with literal/resource or SLBN NQPs
Output: A successful RDBMS/RDF Update/Removal

```
1: if ReificationNode (RN) definition not found in Map File OR
   RN structure does not match Map File definition OR
   RN's rdf:predicate column not present in table corresponding to RN's rdf:subject OR
   One or more fields corresponding to RN's NQPs or SLBN NQP's predicates not present
   in table then
2:    Remove RN from native RDF Store and Return
3: end if
4: if RN's rdf:subject value does not exist in table corresponding to RN's rdf:subject then
5:    Remove RN from native RDF Store and Return
6: end if
7: if RN's rdf:subject value exists but rdf:object value does not exist in rdf:predicate column
   of table then
8:    Remove RN from native RDF Store and Return
9: end if
10: if every column corresponding to RN's NQP or SLBN NQP's predicates has value =
    corresponding NQP's object value then
11:    UPDATE NQP columns SET values = NULL in rdf:subject table WHERE PK =
       rdf:subject value and Return
12: else
13:    Remove RN from native RDF Store and Return
14: end if
```

The only situation when reification data is removed from the underlying relational database is when every column value in columns corresponding to the RN's NQPs or SLBN NQP's predicates is equal to the object values of the corresponding predicates in the reification node. Under such a scenario, the appropriate columns in the table corresponding to the RN's rdf:subject are updated to NULL. In every other situation, the RN is stored in the native RDF store and hence, is deleted there. The DELETE DML operation is never executed for reification nodes belonging to categories (a) and (c) since the RNs represent additional information about a triple and deleting reification information does not delete the triple which was reified. Thus, the triple, whose subject is stored as the primary key value in the appropriate table continues to exist and, consequently, the corresponding tuple in the relational table continues to exist as well. Thus, though the algorithm implements the DELETE operations, deletion never occurs in the relational database schema.

Insert/Update and Delete procedures can be inferred along similar lines as Algorithms 9.8 and 9.9 for RNs belonging to the other categories described in section 9.3.1 as well. Since each of the algorithms presented in Sections 9.2 and 9.3 operate on individual triples, the time complexity of algorithms pertaining to regular triples is a constant, a, as the algorithms do not involve looping processes. Algorithms pertaining to blank nodes are dependent on the number of predicates that belong to the blank nodes and, hence, have a time complexity that is slightly higher and can be represented as $a + bn$, where n is the number of predicates belonging to the blank node of interest, and b is the time taken to insert or update or delete the information in each predicate of the blank node into the appropriate database (i.e., either the RDBMS or the RDF store).

Each of the algorithms described in the previous sections have been implemented as a wrapper around the original D2RQ application in order to make the relational-to-RDF transformation bi-directional. Screenshots of D2R++-Server, an enhanced version of D2R-Server (Bizer and Cyganiak 2006), which includes the ability to receive insert/update/delete requests from the end users are presented in the next section as evidence of the bi-directionality of the transformation process.

9.4 Implementation Results

The hardware and software platforms used in the implementation of the various algorithms discussed in the previous sections, and the performance experiments conducted using the same, are described below.

9.4.1 Experimental Platform

Ubuntu 10.04 with 3 GB RAM and 2.00 GHz Intel Processor was used as the operating system for our experiments. The translation and database tools used include D2RQ 0.7 to perform the uni-directional translation of a relational database schema to an equivalent virtual RDF store, MySQL 5.1.37 to store the relational database schema to be translated into an equivalent RDF store, and Jena 2.6.3 to house the native RDF store that stores the RDBMS-rejected insert/update triples. Software development platforms used include the Eclipse 3.4.0 IDE and Java 1.6 for the development of the algorithms and procedures detailed in sections 9.2 and 9.3.

9.4.2 Experimental Dataset

The performance experiments conducted and the D2R++-Server GUI outputs presented below are based on a subset of the Employee-Department-Project scenario illustrated in Figure 9.1. Synthetic RDF triple datasets of various sizes corresponding to the relational schema as defined by D2RQ's mapping file were created through a data loading program and populated using a Semantic Web Toolkit, Jena, in order to evaluate performance of the insert/update operations performed by our bi-directional algorithms.

9.4.3 Experimental Results

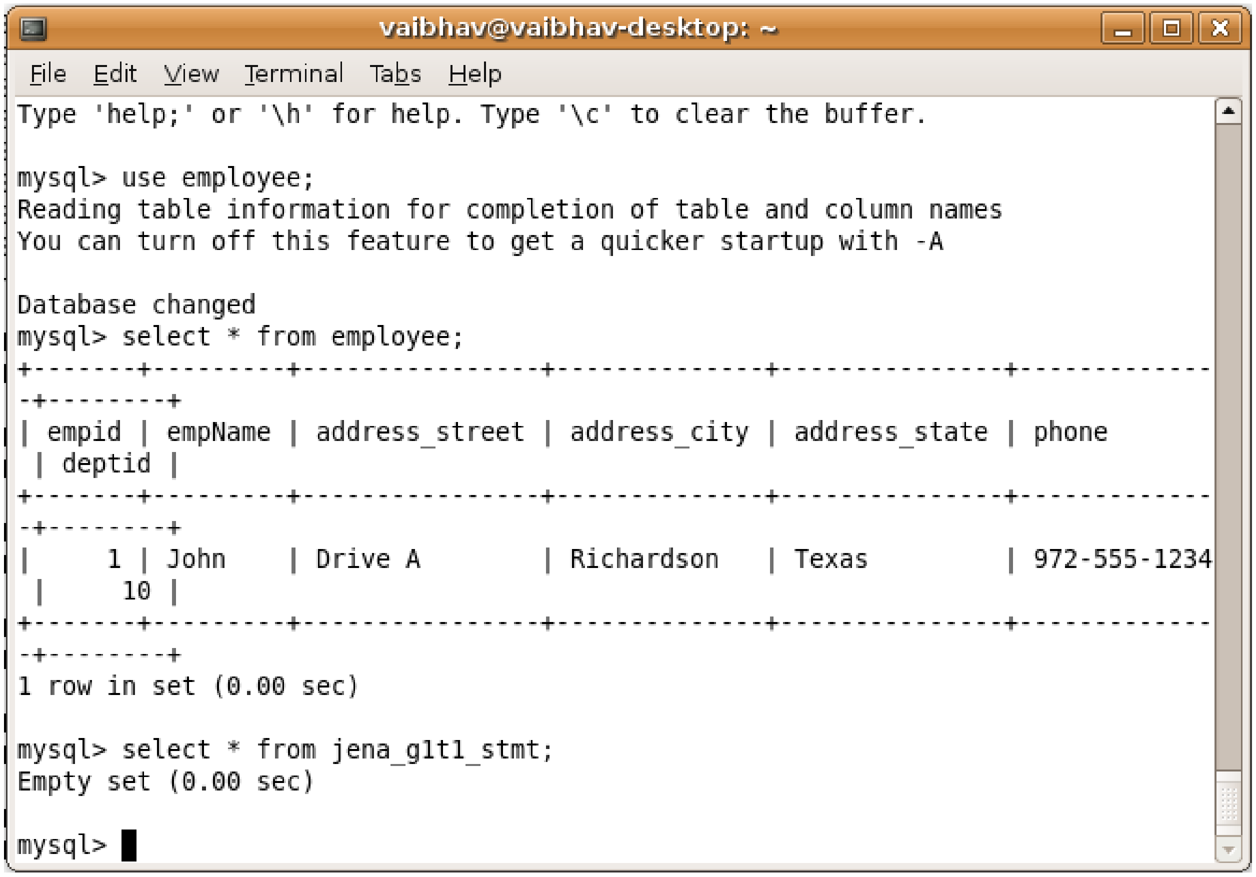
For the purposes of our experimentation and proof of viability, we used Jena's RDB Model as the native RDF store that houses the RDBMS-rejected insert/update triples. Further, in our experiments, the RDB Model is housed in the same MySQL database that houses the actual relational schema. However, in a production environment, the native RDF store will, in all probability, be housed in a completely separate MySQL database. Additionally, the system administrator may also prefer to use an in-memory model rather than a persistent model for the native RDF store. These are design decisions that are left at the administrator's discretion.

In addition to extending the D2RQ application by including our insert/update algorithms, we also extended the D2R-Server front-end GUI application to D2R++-Server which includes provisions to *add/remove* RDF triples. Figures 9.4 through 9.13 illustrate the enhanced D2R++-Server's *add/remove* extensions and D2RQ++'s ability to propagate RDF triples, as new tuples or updates to existing tuples, back to the underlying relational database or native RDF store, as applicable.

Figure 9.4 illustrates the data in the *Employee* table prior to any new insert/update activities. Figure 9.5 illustrates the enhanced D2R++-Server's *Add* option and data in the same table after a triple (with a brand new *EmployeeID* of 2) is added to the equivalent RDF model. As can be seen, per the algorithms described in Section 9.2, this triple is inserted as a new row in the underlying relational database.

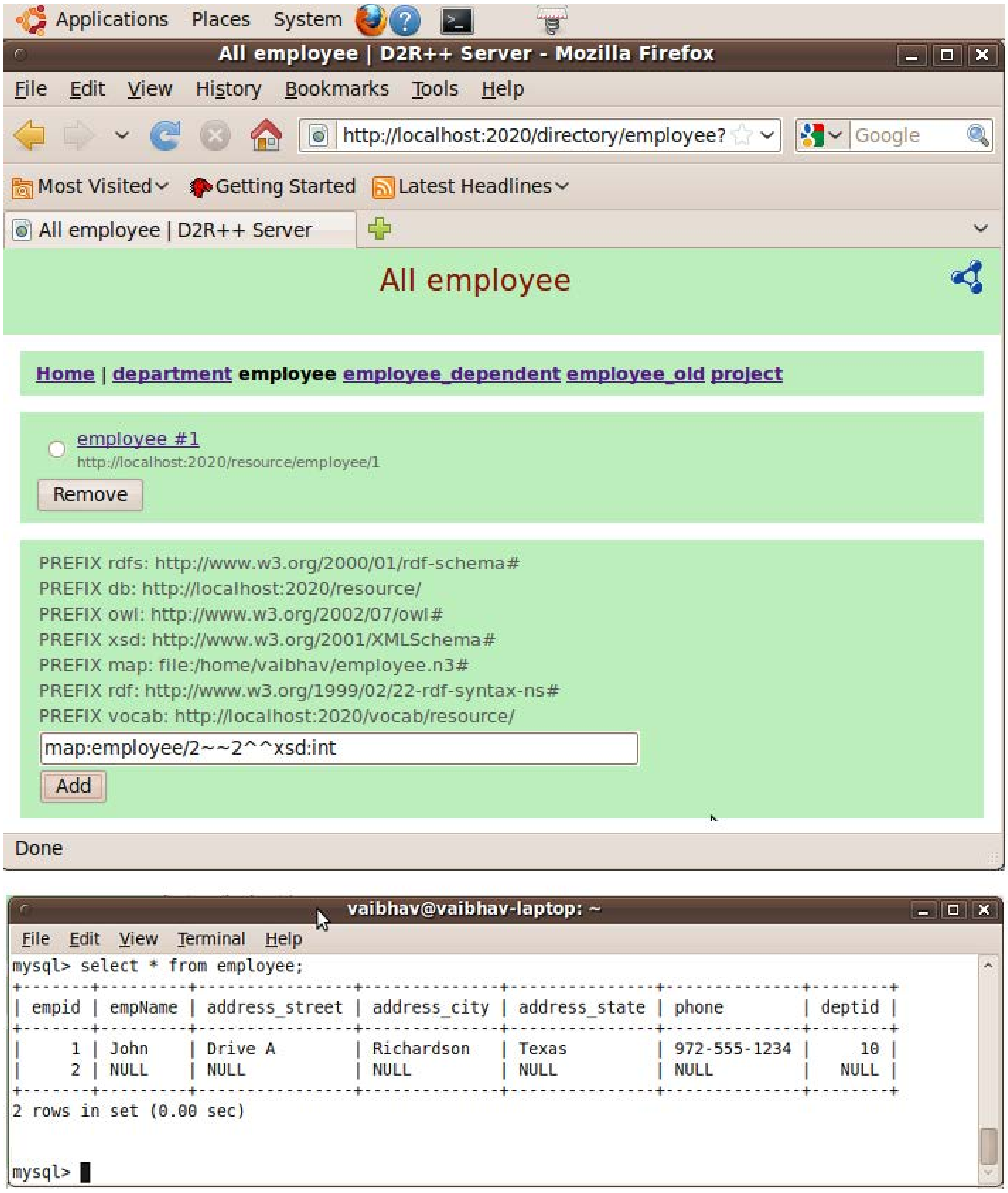
Figures 9.6- 9.8 illustrate a scenario when a duplicate value (a second *empName* for *empID* = 1) is specified for an existing column in the relational database. Under the Closed-World Assumption, this value is either rejected or replaces the original value thereby resulting in loss of information. In D2RQ++, which adheres to the Open-World Assumption, this duplicate value is housed in the native RDF store instead, thereby preserving both the original value as well as the new value as can be seen in the front-end screenshot illustrated in Figure 9.8.

Figures 9.9- 9.11 illustrate the *remove* operation which translates, wherever applicable, to either a simple *remove* operation in the native RDF store (as in the scenario where the second *empName* value of "Doe" is removed from *empID* = 1) or to an *update* (as in the scenario where the original *empName* value of "John" is removed from *empID* = 1). In the first case, since the triple to be removed exists in the RDF store, it is simply removed from the store; in the second case, the triple translates into a specific row and column in



```
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use employee;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from employee;
+-------+---------+----------------+-------------+--------------+------------
-+--------+
| empid | empName | address_street | address_city | address_state | phone
 | deptid |
+-------+---------+----------------+-------------+--------------+------------
-+--------+
|     1 | John    | Drive A        | Richardson   | Texas         | 972-555-1234
 |     10 |
+-------+---------+----------------+-------------+--------------+------------
-+--------+
1 row in set (0.00 sec)

mysql> select * from jena_g1t1_stmt;
Empty set (0.00 sec)

mysql> 
```

Figure 9.4: *Employee* Table Data prior to DML Operations


Applications Places System
All employee | D2R++ Server - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:2020/directory/employee?
Google
Most Visited Getting Started Latest Headlines
All employee | D2R++ Server

All employee

Home | department employee employee_dependent employee_old project

employee #1
http://localhost:2020/resource/employee/1

Remove

PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema#
PREFIX db: http://localhost:2020/resource/
PREFIX owl: http://www.w3.org/2002/07/owl#
PREFIX xsd: http://www.w3.org/2001/XMLSchema#
PREFIX map: file:/home/vaibhav/employee.n3#
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX vocab: http://localhost:2020/vocab/resource/

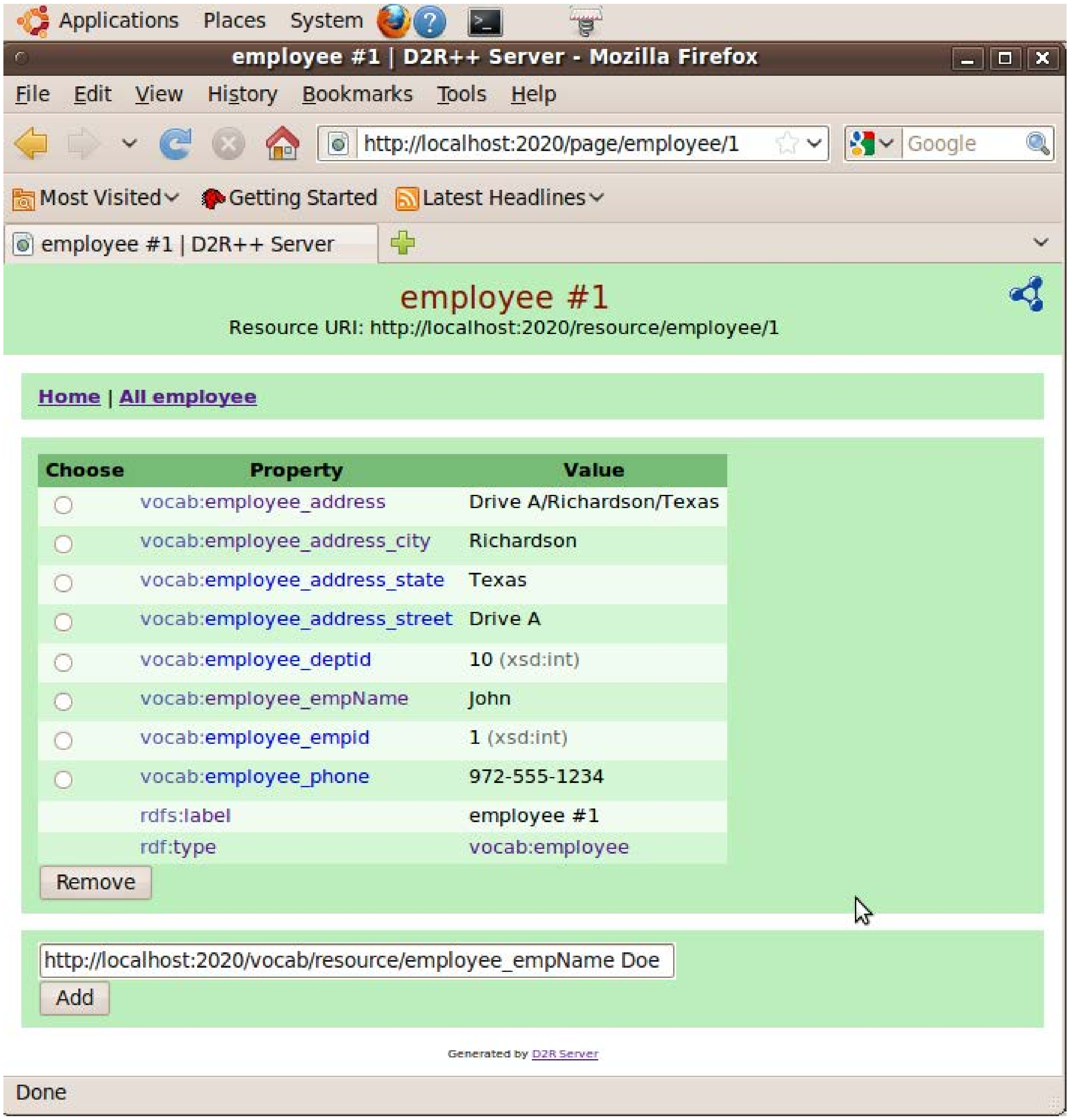
map:employee/2~~2^^xsd:int

Add

Done

```
mysql> select * from employee;
+-------+---------+----------------+--------------+---------------+--------------+--------+
| empid | empName | address_street | address_city | address_state | phone        | deptid |
+-------+---------+----------------+--------------+---------------+--------------+--------+
|     1 | John    | Drive A        | Richardson   | Texas         | 972-555-1234 |     10 |
|     2 | NULL    | NULL           | NULL         | NULL          | NULL         |   NULL |
+-------+---------+----------------+--------------+---------------+--------------+--------+
2 rows in set (0.00 sec)

mysql> 
```

Figure 9.5: Addition of a Second *Employee* Record



employee #1

Resource URI: http://localhost:2020/resource/employee/1

Home | All employee

Choose	Property	Value
○	vocab:employee_address	Drive A/Richardson/Texas
○	vocab:employee_address_city	Richardson
○	vocab:employee_address_state	Texas
○	vocab:employee_address_street	Drive A
○	vocab:employee_deptid	10 (xsd:int)
○	vocab:employee_empName	John
○	vocab:employee_empid	1 (xsd:int)
○	vocab:employee_phone	972-555-1234
	rdfs:label	employee #1
	rdf:type	vocab:employee

Remove

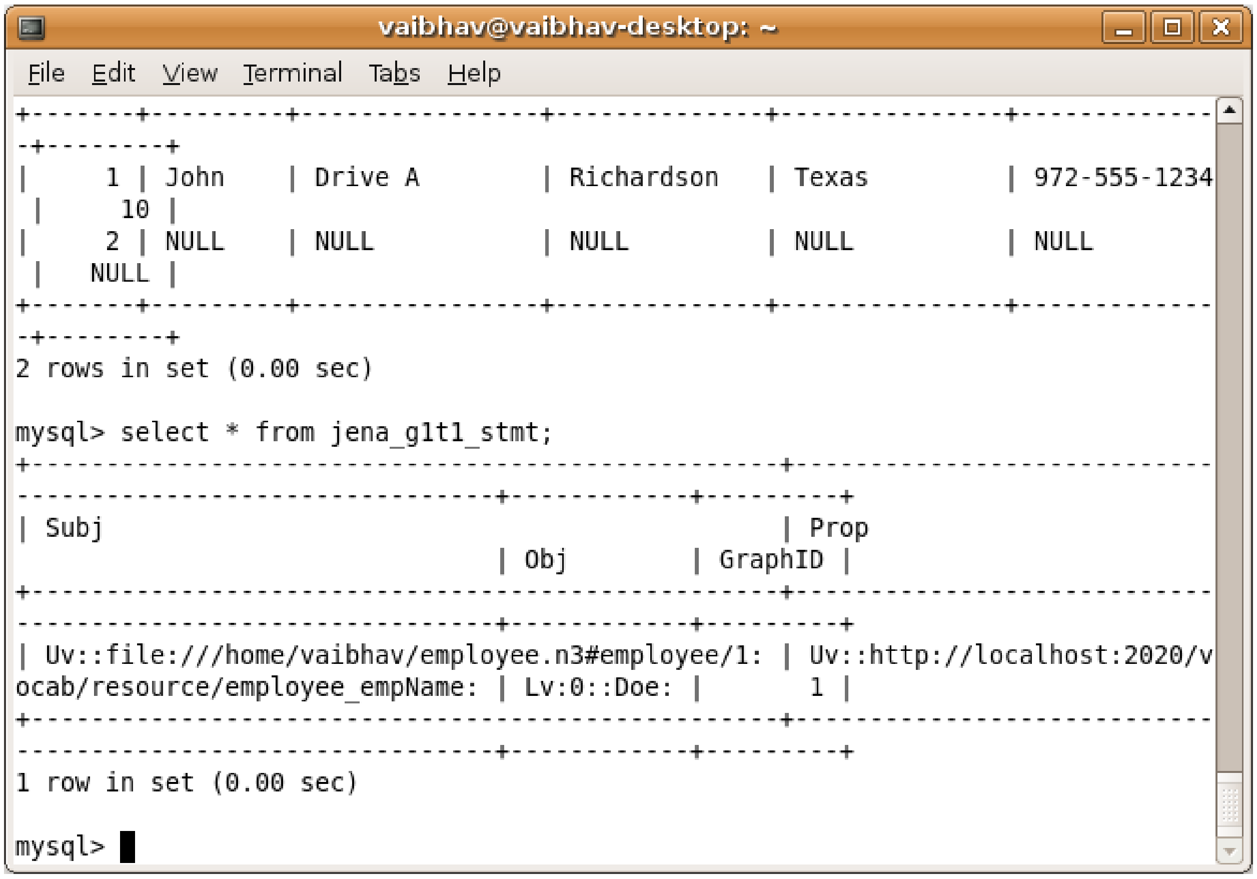
http://localhost:2020/vocab/resource/employee_empName Doe

Add

Generated by D2R Server

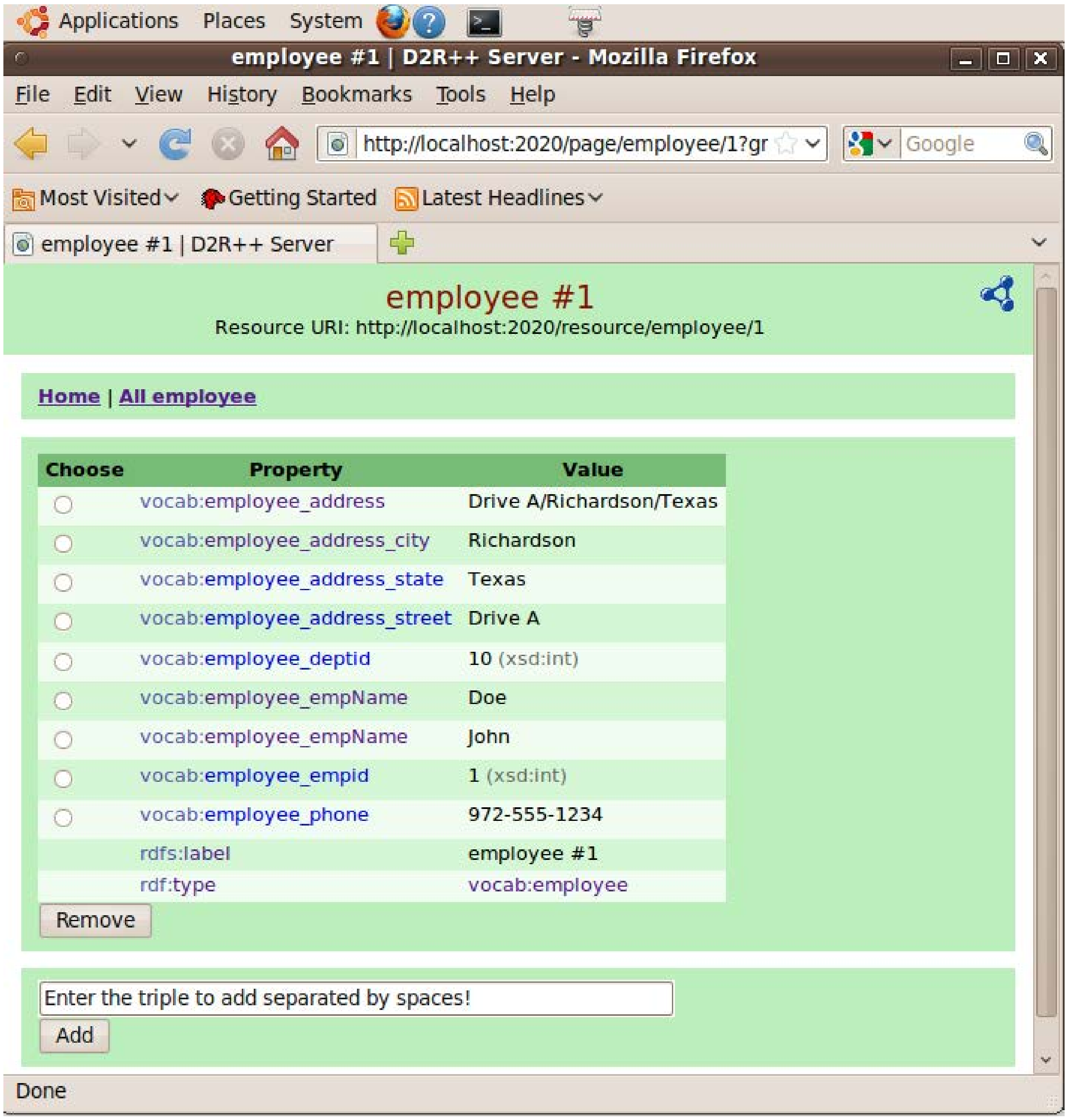
Done

Figure 9.6: D2R++Server Screen illustrating Second *Employee Name* Addition



```
+-------+---------+----------------+-------------+--------------+------------
-+--------+
|     1 | John    | Drive A        | Richardson  | Texas        | 972-555-1234
 |     10 |
|     2 | NULL    | NULL           | NULL        | NULL         | NULL
 |   NULL |
+-------+---------+----------------+-------------+--------------+------------
-+--------+
2 rows in set (0.00 sec)

mysql> select * from jena_g1t1_stmt;
+-----------------------------------------------+----------------------------
--------------------------------+-----------+---------+
| Subj                                          | Prop
                                | Obj       | GraphID |
+-----------------------------------------------+----------------------------
--------------------------------+-----------+---------+
| Uv::file:///home/vaibhav/employee.n3#employee/1: | Uv::http://localhost:2020/v
ocab/resource/employee_empName: | Lv:0::Doe: |       1 |
+-----------------------------------------------+----------------------------
--------------------------------+-----------+---------+
1 row in set (0.00 sec)

mysql> 
```

Figure 9.7: MySQL database after Second *Employee Name* Addition


Applications Places System
employee #1 | D2R++ Server - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:2020/page/employee/1?gr
Google
Most Visited Getting Started Latest Headlines
employee #1 | D2R++ Server

employee #1

Resource URI: http://localhost:2020/resource/employee/1

Home | All employee

Choose	Property	Value
○	vocab:employee_address	Drive A/Richardson/Texas
○	vocab:employee_address_city	Richardson
○	vocab:employee_address_state	Texas
○	vocab:employee_address_street	Drive A
○	vocab:employee_deptid	10 (xsd:int)
○	vocab:employee_empName	Doe
○	vocab:employee_empName	John
○	vocab:employee_empid	1 (xsd:int)
○	vocab:employee_phone	972-555-1234
	rdfs:label	employee #1
	rdf:type	vocab:employee

Figure 9.8: D2R++Server Screen after Second *Employee Name* Addition

the underlying relational database and, hence, the *remove* operation is propagated to the database as an *update* operation that sets the appropriate column value to *null*.

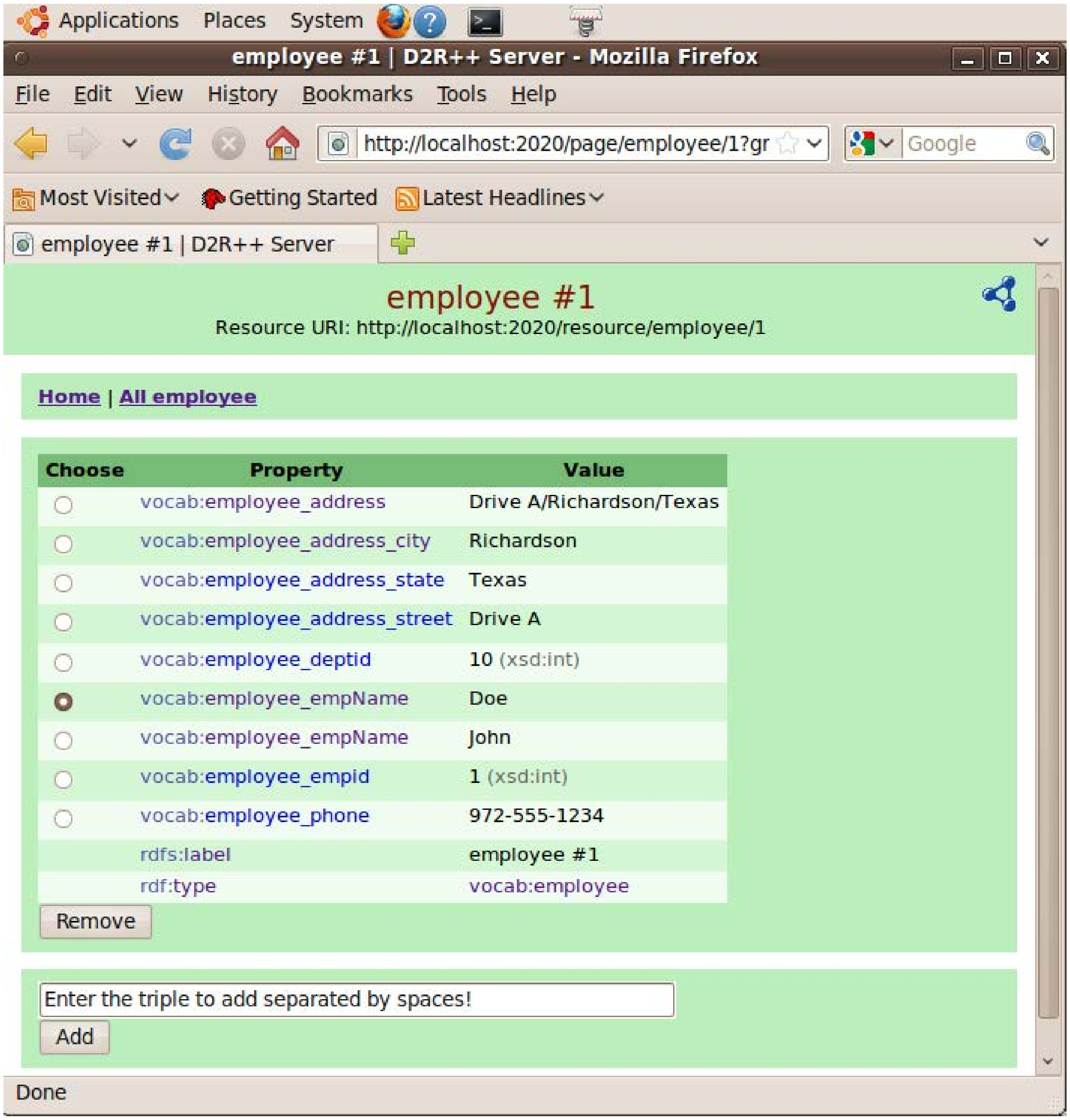
Figures 9.12 and 9.13 illustrate the *remove* operation that translates to a *delete* operation in the underlying relational database schema where the employee with *EmployeeID* = 2 is removed.

Figures 9.14 through 9.21 illustrate the enhanced D2R++-Server's *add/remove* extensions and D2RQ++'s ability to propagate RDF blank nodes and reification nodes, as new tuples or updates to existing tuples, back to the underlying relational database or native RDF store, as applicable. Data in the *employee* table prior to any new insert/update activities pertaining to blank/reification nodes included a single record for an employee named "John" with no address information as can be seen in Figure 9.14.

Figures 9.15 and 9.16 illustrate the addition of the first address information in the form of an SLBN through the D2R++-Server application, along with the corresponding updates within the appropriate RDBMS schema or native RDF Store as applicable. Since the columns corresponding to the predicates of the SLBN are all null initially, the objects of the SLBN are updated against the "John" tuple within the relational database schema table, *employee*, as illustrated in the MySQL Window in Figure 9.16.

Figures 9.17 and 9.18 illustrates the addition of a second address for "John" (through the D2R++-Server application). As can be seen from the appropriate backend database queries in the MySQL window, since the relational table *employee* has an address for "John" already, the new address is added to the native RDF store.

Figures 9.19 - 9.21 illustrates D2RQ++'s reification support through the output from a simple SPARQL query to list all triples. Since the relational schema used in our experimentation did not include a column to store the reification data, and, consequently, there was no mapping in the D2RQ++ map file that corresponds to *inspectionDate*, this reification information is stored in the native RDF store instead of in the relational database schema as illustrated in Figure 9.19. As shown in Figures 9.20 and 9.21, the output includes data from the relational database schema (triplified) as well as from the native RDF store.



employee #1

Resource URI: http://localhost:2020/resource/employee/1

Home | All employee

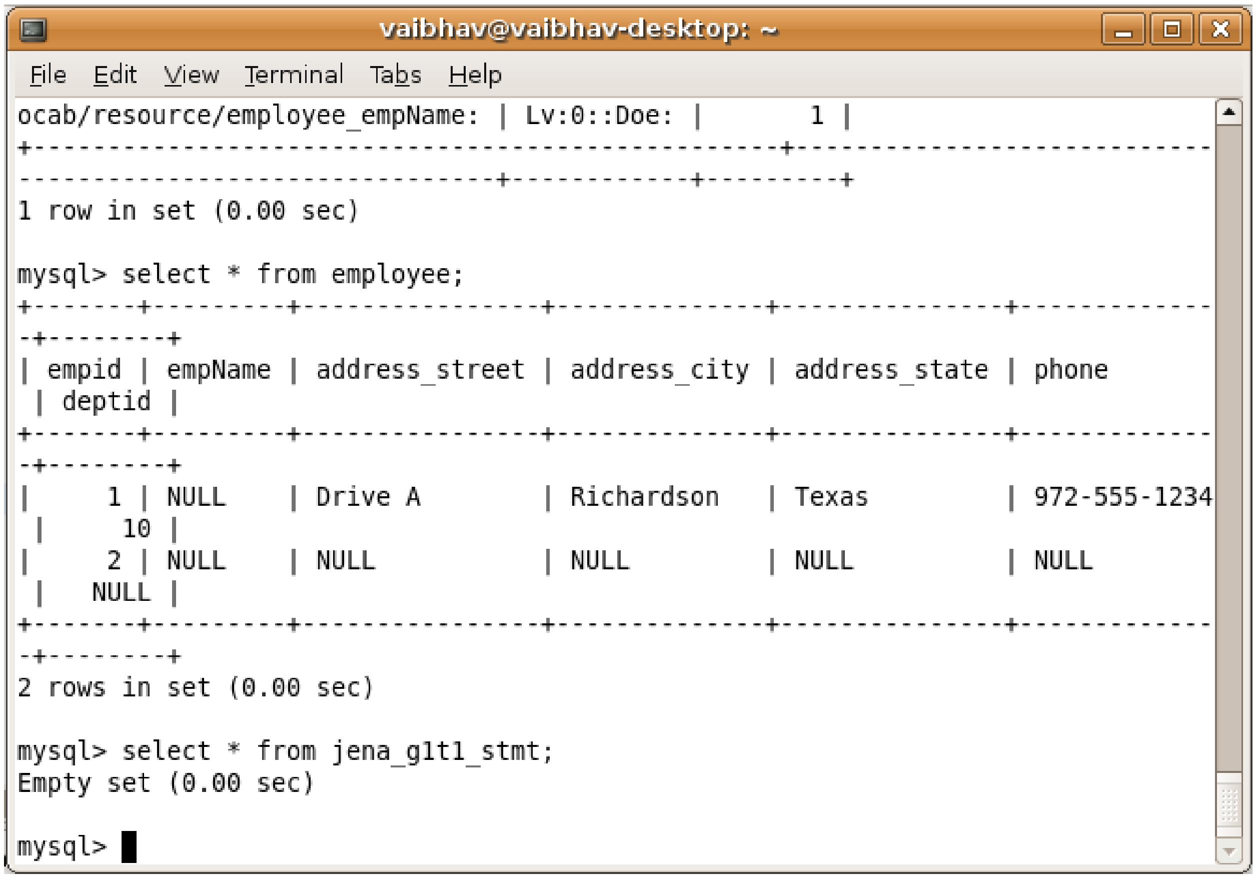
Choose	Property	Value
○	vocab:employee_address	Drive A/Richardson/Texas
○	vocab:employee_address_city	Richardson
○	vocab:employee_address_state	Texas
○	vocab:employee_address_street	Drive A
○	vocab:employee_deptid	10 (xsd:int)
●	vocab:employee_empName	Doe
○	vocab:employee_empName	John
○	vocab:employee_empid	1 (xsd:int)
○	vocab:employee_phone	972-555-1234
	rdfs:label	employee #1
	rdf:type	vocab:employee

Remove

Enter the triple to add separated by spaces!

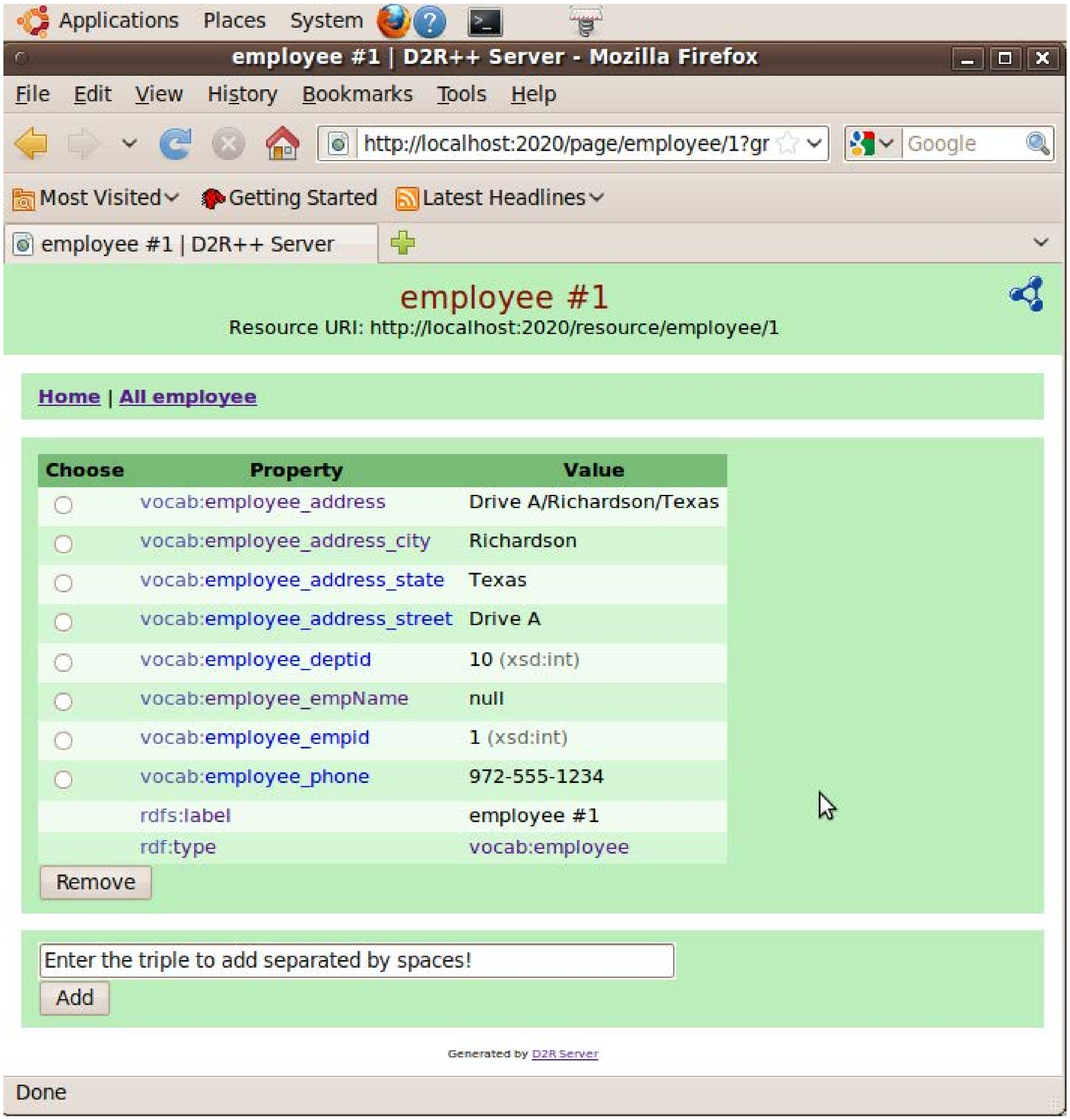
Add

Figure 9.9: D2R++Server Screen illustrating Removal of Second *Employee Name* (“Doe”)



```
ocab/resource/employee_empName: | Lv:0::Doe: |       1 |
+---------------------------------------------------+-----------------------------
--------------------------------+-----------+---------+
1 row in set (0.00 sec)

mysql> select * from employee;
+-------+---------+----------------+-------------+---------------+------------
-+--------+
| empid | empName | address_street | address_city | address_state | phone
 | deptid |
+-------+---------+----------------+-------------+---------------+------------
-+--------+
|     1 | NULL    | Drive A        | Richardson   | Texas         | 972-555-1234
 |     10 |
|     2 | NULL    | NULL           | NULL         | NULL          | NULL
 |   NULL |
+-------+---------+----------------+-------------+---------------+------------
-+--------+
2 rows in set (0.00 sec)

mysql> select * from jena_g1t1_stmt;
Empty set (0.00 sec)

mysql> 
```

Figure 9.10: MySQL database after Deletion of both *Names* for *Employee 1*


Applications Places System
employee #1 | D2R++ Server - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:2020/page/employee/1?gr
Google
Most Visited Getting Started Latest Headlines
employee #1 | D2R++ Server
Home | All employee
Remove
Add
Done

employee #1

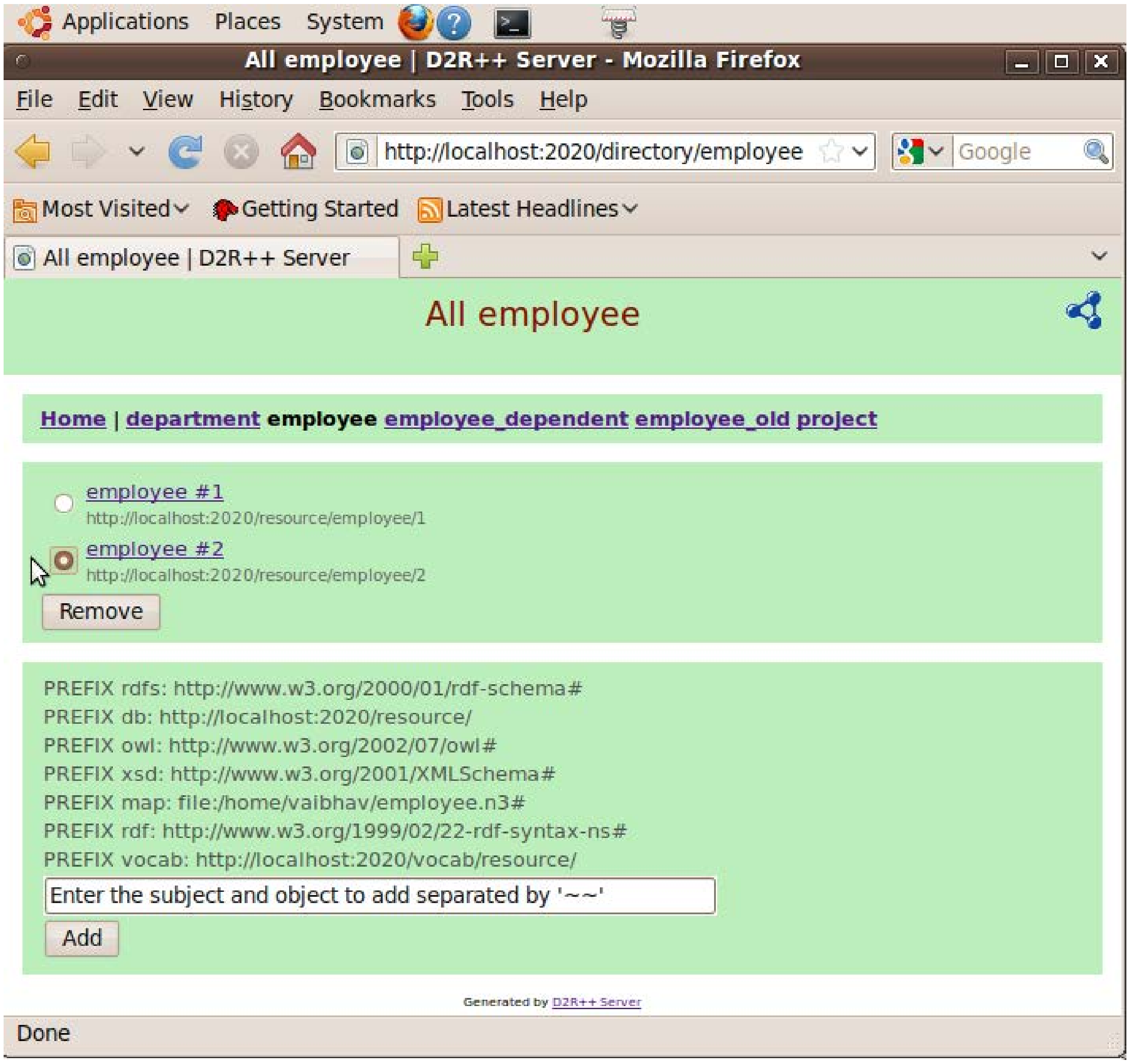
Resource URI: http://localhost:2020/resource/employee/1

Choose	Property	Value
○	vocab:employee_address	Drive A/Richardson/Texas
○	vocab:employee_address_city	Richardson
○	vocab:employee_address_state	Texas
○	vocab:employee_address_street	Drive A
○	vocab:employee_deptid	10 (xsd:int)
○	vocab:employee_empName	null
○	vocab:employee_empid	1 (xsd:int)
○	vocab:employee_phone	972-555-1234
	rdfs:label	employee #1
	rdf:type	vocab:employee

Enter the triple to add separated by spaces!

Generated by D2R Server

Figure 9.11: D2R++Server Screen after Deletion of both *Names* for *Employee 1*


Applications
Places
System
All employee | D2R++ Server - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://localhost:2020/directory/employee
Google
Most Visited
Getting Started
Latest Headlines
All employee | D2R++ Server

All employee

Home | department employee employee_dependent employee_old project

( ) employee #1
http://localhost:2020/resource/employee/1

(●) employee #2
http://localhost:2020/resource/employee/2

Remove

PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema#
PREFIX db: http://localhost:2020/resource/
PREFIX owl: http://www.w3.org/2002/07/owl#
PREFIX xsd: http://www.w3.org/2001/XMLSchema#
PREFIX map: file:/home/vaibhav/employee.n3#
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX vocab: http://localhost:2020/vocab/resource/

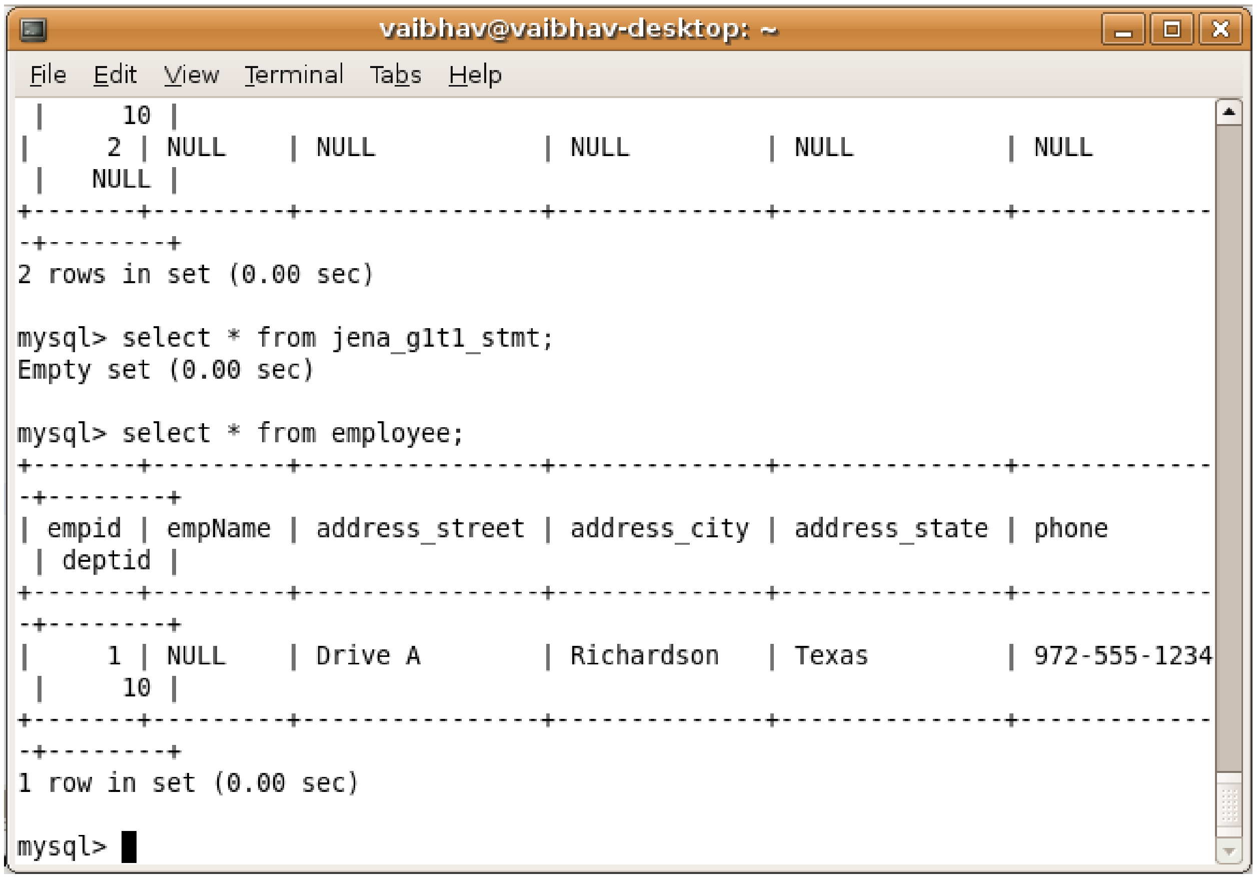
Enter the subject and object to add separated by '~~'

Add

Generated by D2R++ Server

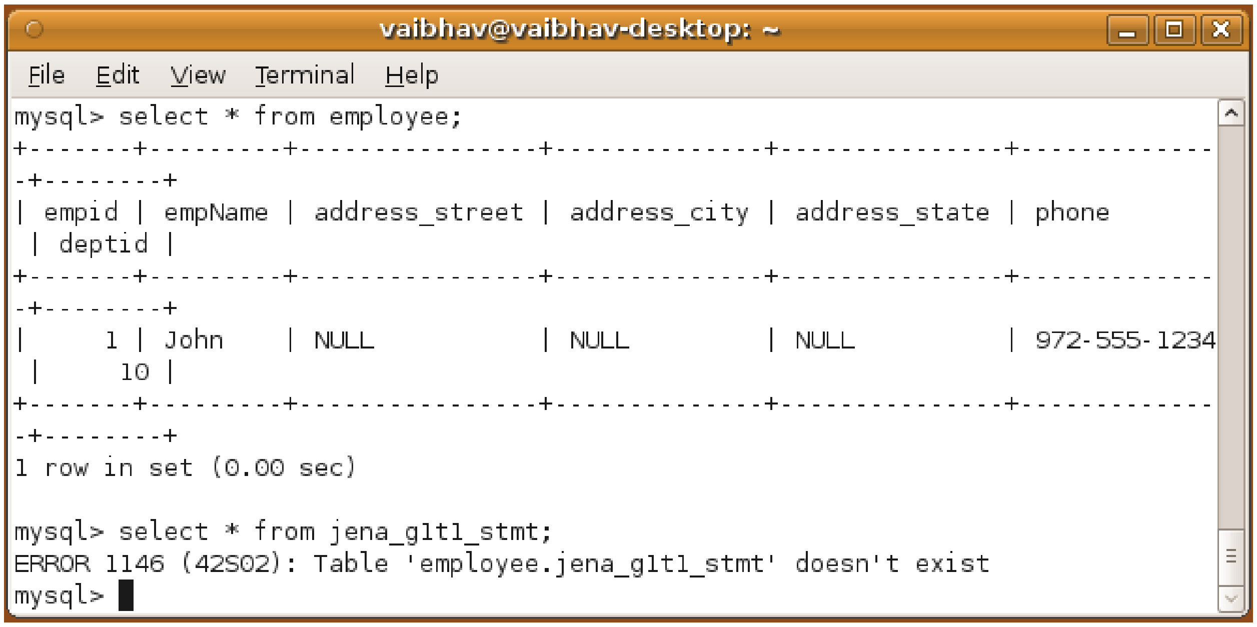
Done

Figure 9.12: D2R++Server Screen illustrating Removal of Second Employee *(Employee 2)*



```
 |     10 |
|      2 | NULL    | NULL            | NULL          | NULL           | NULL
 |   NULL |
+-------+---------+----------------+-------------+--------------+------------
-+--------+
2 rows in set (0.00 sec)

mysql> select * from jena_g1t1_stmt;
Empty set (0.00 sec)

mysql> select * from employee;
+-------+---------+----------------+-------------+--------------+------------
-+--------+
| empid | empName | address_street | address_city | address_state | phone
 | deptid |
+-------+---------+----------------+-------------+--------------+------------
-+--------+
|     1 | NULL    | Drive A         | Richardson   | Texas          | 972-555-1234
 |     10 |
+-------+---------+----------------+-------------+--------------+------------
-+--------+
1 row in set (0.00 sec)

mysql> █
```

Figure 9.13: MySQL database after Deletion of *Employee 2*

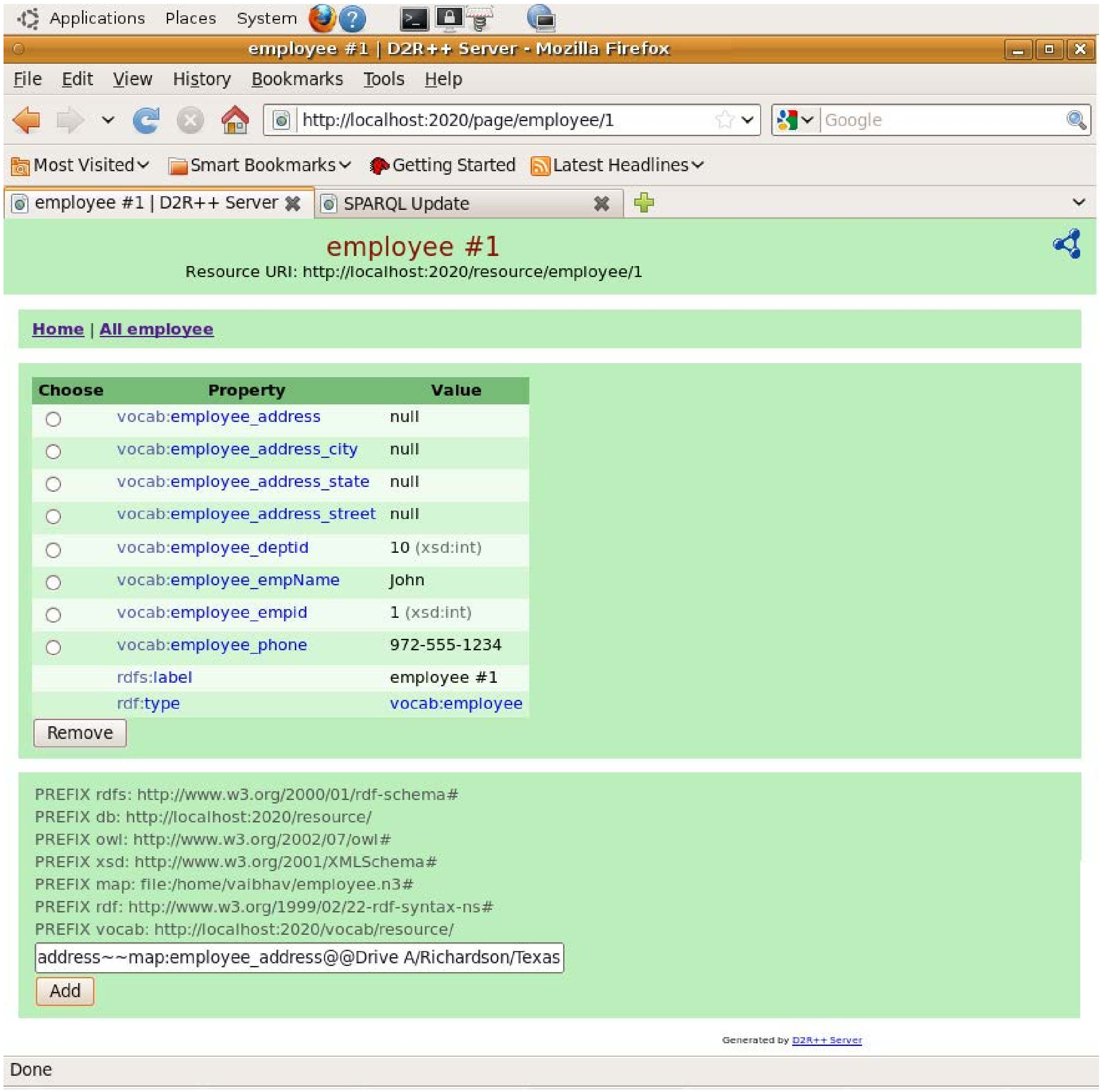


```
mysql> select * from employee;
+-------+---------+----------------+--------------+---------------+-------------
-+--------+
| empid | empName | address_street | address_city | address_state | phone
 | deptid |
+-------+---------+----------------+--------------+---------------+-------------
-+--------+
|     1 | John    | NULL           | NULL         | NULL          | 972-555-1234
 |     10 |
+-------+---------+----------------+--------------+---------------+-------------
-+--------+
1 row in set (0.00 sec)

mysql> select * from jena_g1t1_stmt;
ERROR 1146 (42S02): Table 'employee.jena_g1t1_stmt' doesn't exist
mysql> 
```

Figure 9.14: Initial Data in *employee* Table

The time taken for the insert/update/delete algorithms for regular RDF triples is illustrated in Figure 9.22. For each operation (i.e., insert, update, and delete) performance statistics for three different scenarios were generated. In the first scenario, the operation under consideration only affects data in the underlying relational database; in the second scenario, the only affected data are the ones that exist in the native RDF store; and the third scenario is one in which 50% of the affected data are housed in the underlying relational database while the other 50% is housed in the native RDF store. Since the concept of update in RDF stores is implemented through a delete-insert combination, performance statistics for the second scenario (affected data existing only in the native RDF store) in the second graph (performance statistics for update operations) are obtained by adding the second scenario statistics from graphs 1 (insert) and 3 (delete)

As can be seen from Figure 9.22, the algorithms perform well for small datasets but are somewhat time intensive for larger number of records. Further, while the performance of Insert and Delete operations are similar for smaller datasets, the performance of the Insert


Applications Places System
employee #1 | D2R++ Server - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:2020/page/employee/1
Google
Most Visited Smart Bookmarks Getting Started Latest Headlines
employee #1 | D2R++ Server
SPARQL Update

employee #1

Resource URI: http://localhost:2020/resource/employee/1

Home | All employee

Choose	Property	Value
○	vocab:employee_address	null
○	vocab:employee_address_city	null
○	vocab:employee_address_state	null
○	vocab:employee_address_street	null
○	vocab:employee_deptid	10 (xsd:int)
○	vocab:employee_empName	John
○	vocab:employee_empid	1 (xsd:int)
○	vocab:employee_phone	972-555-1234
	rdfs:label	employee #1
	rdf:type	vocab:employee

Remove

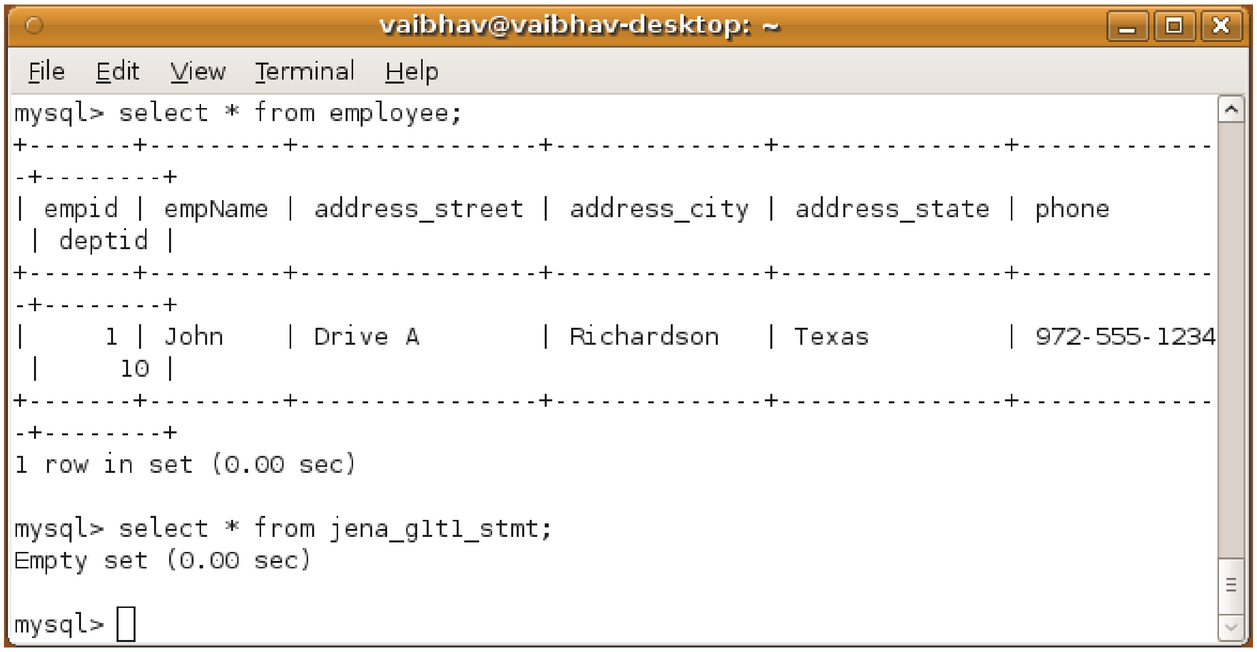
```
PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema#
PREFIX db: http://localhost:2020/resource/
PREFIX owl: http://www.w3.org/2002/07/owl#
PREFIX xsd: http://www.w3.org/2001/XMLSchema#
PREFIX map: file:/home/vaibhav/employee.n3#
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX vocab: http://localhost:2020/vocab/resource/
address~~map:employee_address@@Drive A/Richardson/Texas
```

Add

Generated by D2R++ Server

Done

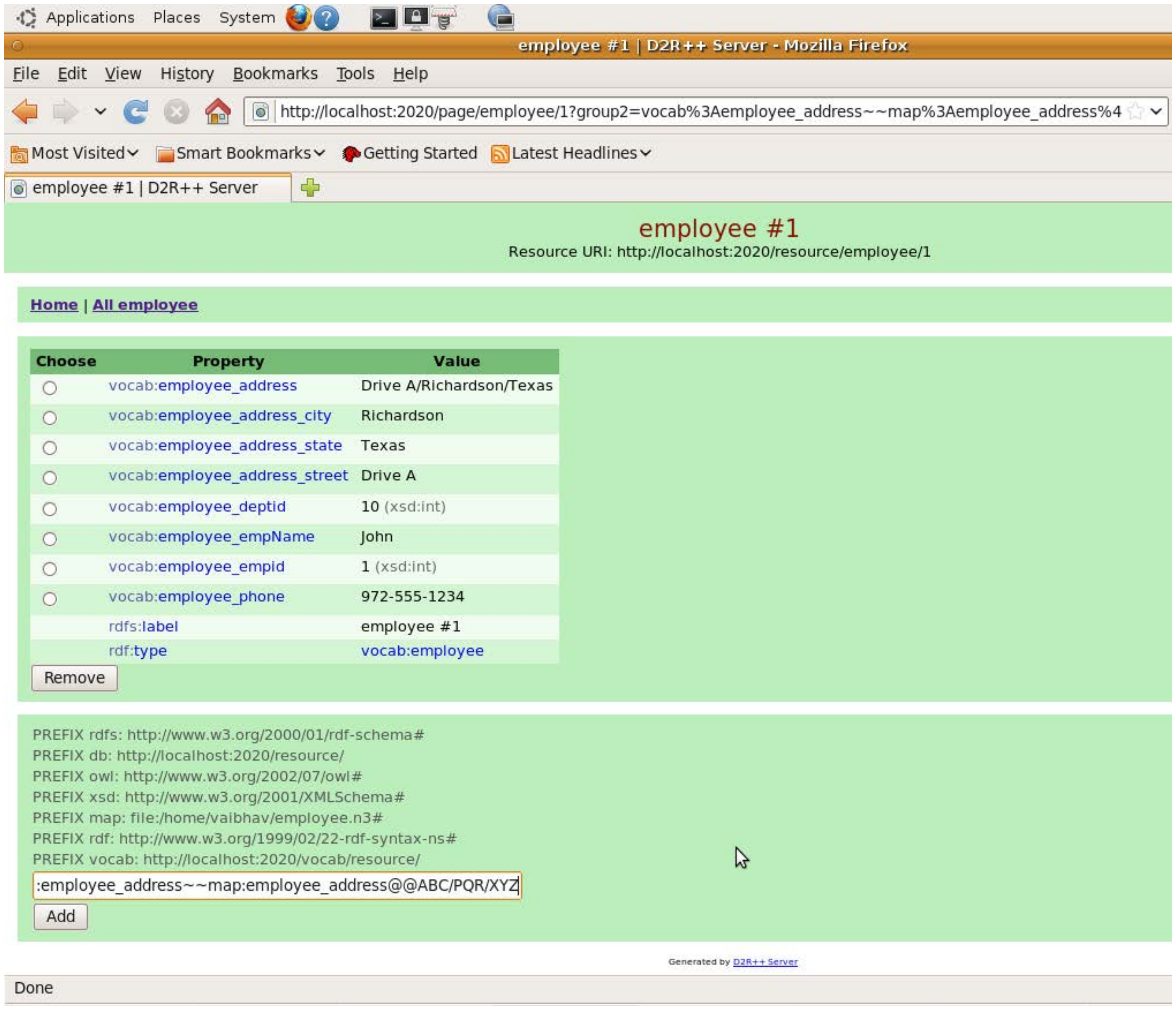
Figure 9.15: D2R++Server Screen illustrating Addition of First *Address* SLBN



```
mysql> select * from employee;
+-------+--------+----------------+--------------+---------------+------------
-+--------+
| empid | empName | address_street | address_city | address_state | phone
 | deptid |
+-------+--------+----------------+--------------+---------------+------------
-+--------+
|     1 | John    | Drive A        | Richardson   | Texas         | 972-555-1234
 |     10 |
+-------+--------+----------------+--------------+---------------+------------
-+--------+
1 row in set (0.00 sec)

mysql> select * from jena_g1t1_stmt;
Empty set (0.00 sec)

mysql> 
```

Figure 9.16: MySQL Database after Addition of First *Address* SLBN

operation degrades faster than that of the Delete operation. This difference in time arises because the Insert and Update algorithms need to check more conditions per triple than the Delete algorithm. Further, as the database size grows, the time taken to identify whether a particular subject or predicate exists in the RDBMS or the RDF store increases due to the increased number of records. In the case of the Delete operation, as the triples are deleted the database size decreases and, therefore, the subject and predicate searches operate on fewer triples which, in turn, reduces the search time. Thus, the performance degradation is not as severe as in the Insert scenario. In either case however, since the main purpose of D2RQ++ is to provide insert/update/delete functionality in OnLine Transaction Processing (OLTP) kind of applications with few insert/update/delete operations per transaction and not to provide bulk data loading functionality, this performance degradation with increasing data volumes is of no consequence. For the same reason, generating performance statistics for DML operations on larger datasets is also considered unnecessary and beyond D2RQ++'s objectives. It is expected that for bulk data loading, performance-optimized


Applications
Places
System
employee #1 | D2R++ Server - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://localhost:2020/page/employee/1?group2=vocab%3Aemployee_address~~map%3Aemployee_address%4
Most Visited
Smart Bookmarks
Getting Started
Latest Headlines
employee #1 | D2R++ Server

employee #1

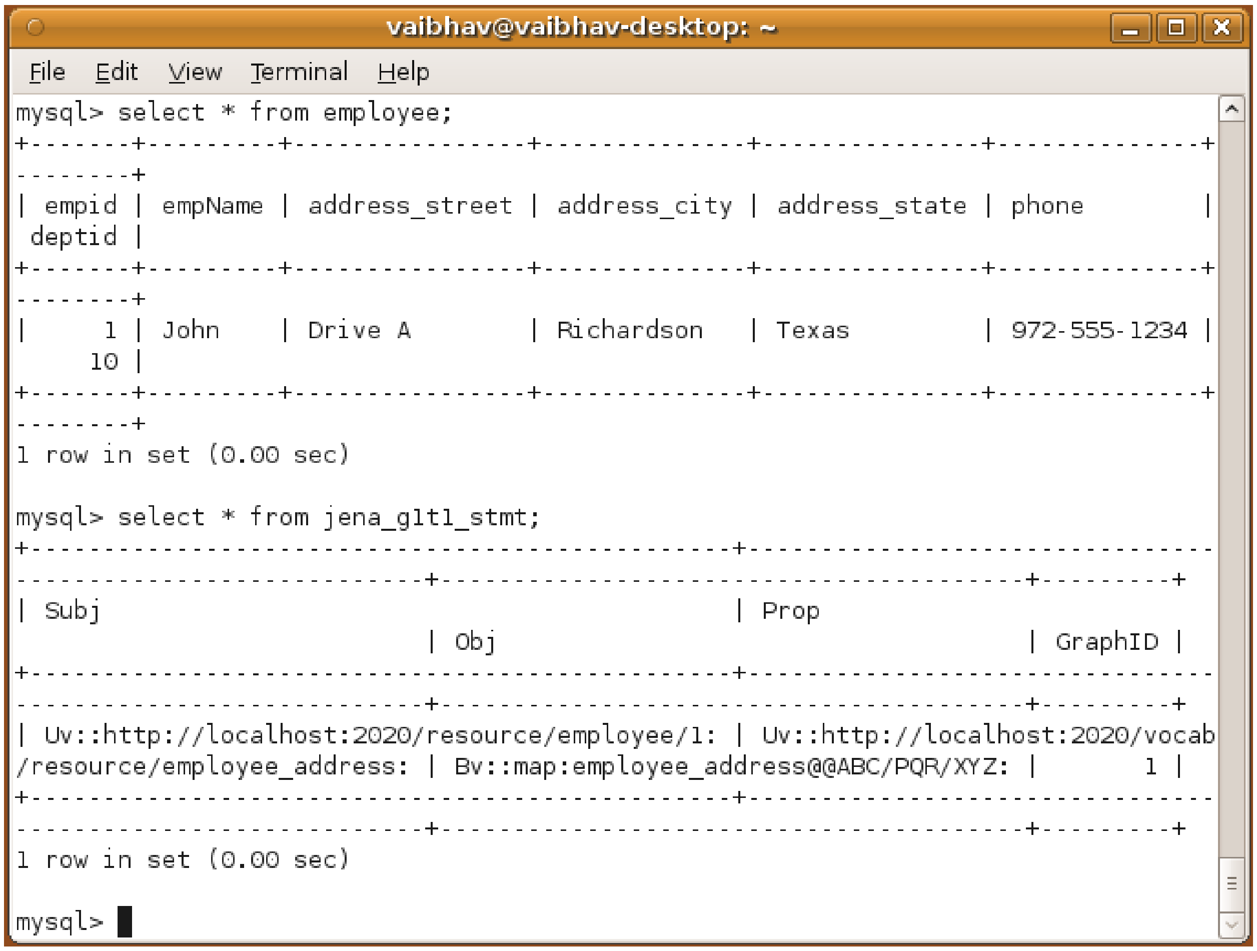
Resource URI: http://localhost:2020/resource/employee/1

Home | All employee

Choose	Property	Value
○	vocab:employee_address	Drive A/Richardson/Texas
○	vocab:employee_address_city	Richardson
○	vocab:employee_address_state	Texas
○	vocab:employee_address_street	Drive A
○	vocab:employee_deptid	10 (xsd:int)
○	vocab:employee_empName	John
○	vocab:employee_empid	1 (xsd:int)
○	vocab:employee_phone	972-555-1234
	rdfs:label	employee #1
	rdf:type	vocab:employee

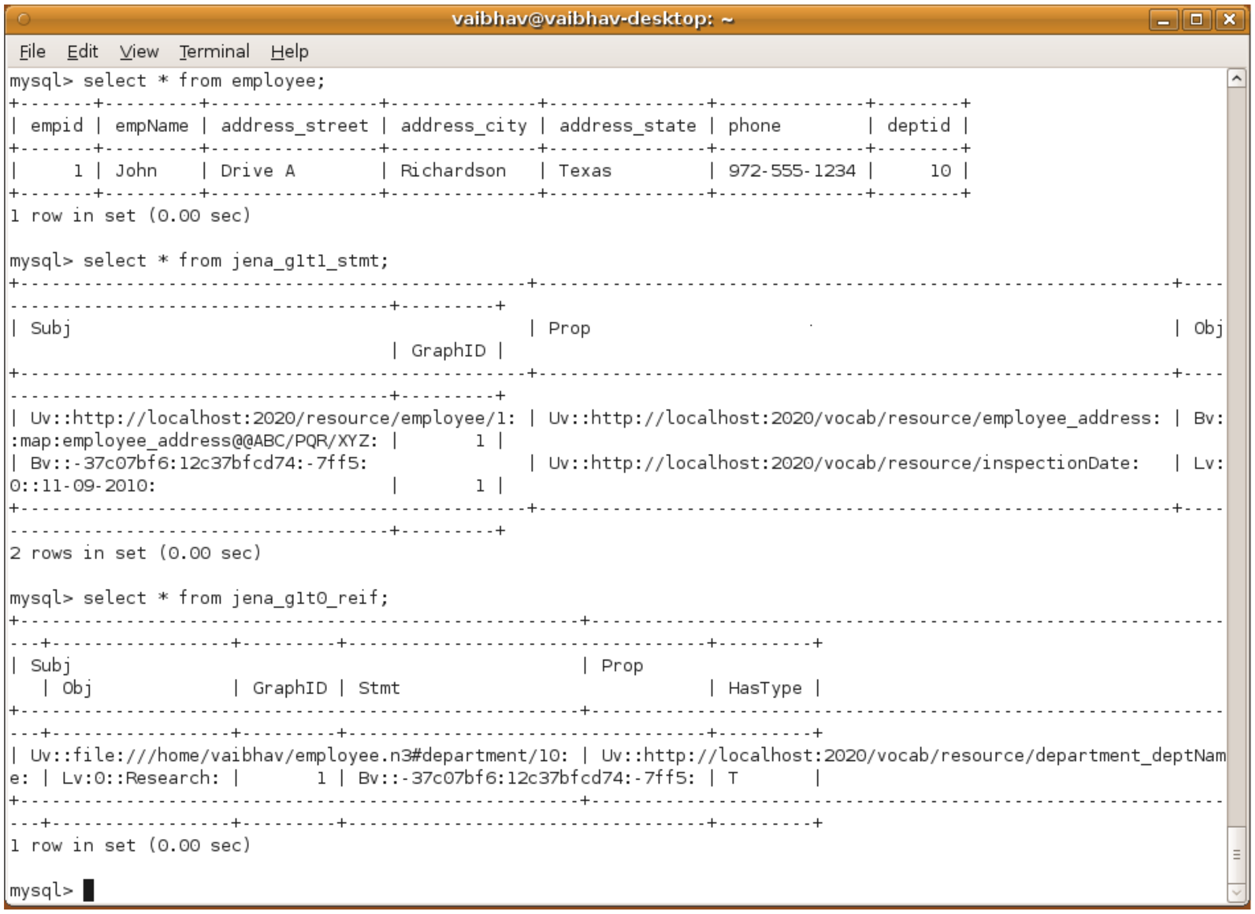
```
PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema#
PREFIX db: http://localhost:2020/resource/
PREFIX owl: http://www.w3.org/2002/07/owl#
PREFIX xsd: http://www.w3.org/2001/XMLSchema#
PREFIX map: file:/home/vaibhav/employee.n3#
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX vocab: http://localhost:2020/vocab/resource/
:employee_address~~map:employee_address@@ABC/PQR/XYZ
```

Figure 9.17: D2R++Server Screen illustrating Addition of Second *Address* SLBN



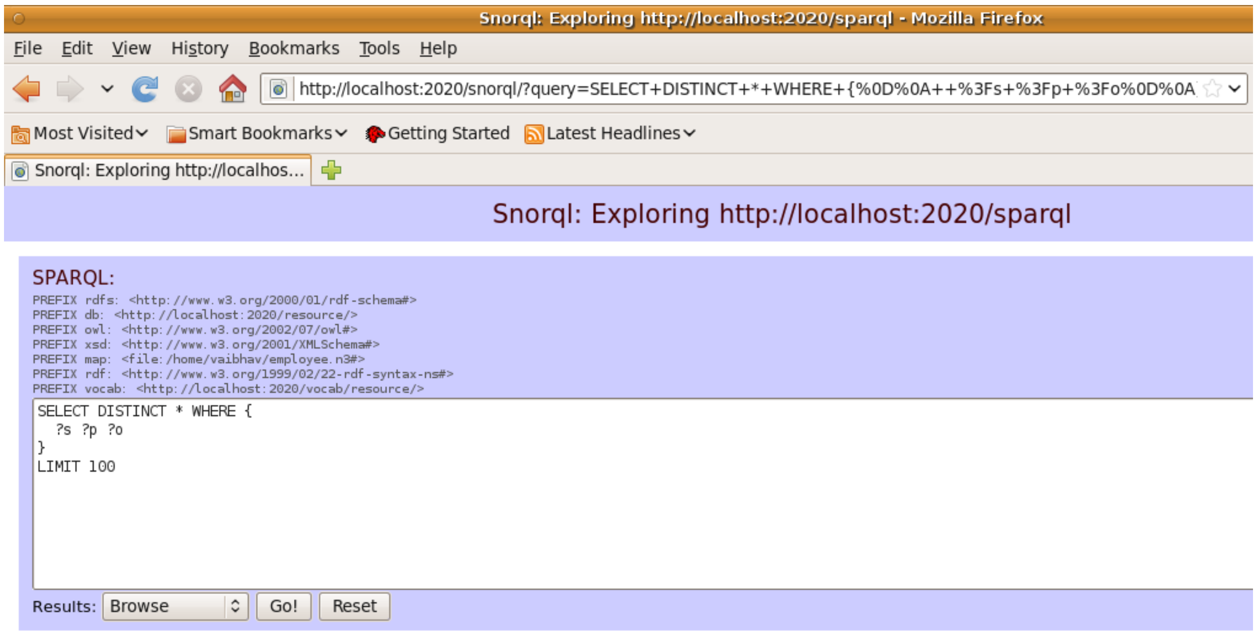
```
mysql> select * from employee;
+-------+---------+----------------+--------------+---------------+--------------+
--------+
| empid | empName | address_street | address_city | address_state | phone        |
 deptid |
+-------+---------+----------------+--------------+---------------+--------------+
--------+
|     1 | John    | Drive A        | Richardson   | Texas         | 972-555-1234 |
     10 |
+-------+---------+----------------+--------------+---------------+--------------+
--------+
1 row in set (0.00 sec)

mysql> select * from jena_g1t1_stmt;
+-------------------------------------------------+--------------------------------
----------------------------+-----------------------------------------+---------+
| Subj                                            | Prop
                            | Obj                                     | GraphID |
+-------------------------------------------------+--------------------------------
----------------------------+-----------------------------------------+---------+
| Uv::http://localhost:2020/resource/employee/1: | Uv::http://localhost:2020/vocab
/resource/employee_address: | Bv::map:employee_address@@ABC/PQR/XYZ: |       1 |
+-------------------------------------------------+--------------------------------
----------------------------+-----------------------------------------+---------+
1 row in set (0.00 sec)

mysql> 
```

Figure 9.18: MySQL Database after Addition of Second *Address* SLBN



```
mysql> select * from employee;
+-------+---------+----------------+--------------+---------------+--------------+--------+
| empid | empName | address_street | address_city | address_state | phone        | deptid |
+-------+---------+----------------+--------------+---------------+--------------+--------+
|     1 | John    | Drive A        | Richardson   | Texas         | 972-555-1234 |     10 |
+-------+---------+----------------+--------------+---------------+--------------+--------+
1 row in set (0.00 sec)

mysql> select * from jena_g1t1_stmt;
+------------------------------------------------+------------------------------------------------------------+----
------------------------------------+---------+
| Subj                                           | Prop                                                       | Obj
                                    | GraphID |
+------------------------------------------------+------------------------------------------------------------+----
------------------------------------+---------+
| Uv::http://localhost:2020/resource/employee/1: | Uv::http://localhost:2020/vocab/resource/employee_address: | Bv:
:map:employee_address@@ABC/PQR/XYZ: |       1 |
| Bv::-37c07bf6:12c37bfcd74:-7ff5:               | Uv::http://localhost:2020/vocab/resource/inspectionDate:   | Lv:
0::11-09-2010:                      |       1 |
+------------------------------------------------+------------------------------------------------------------+----
------------------------------------+---------+
2 rows in set (0.00 sec)

mysql> select * from jena_g1t0_reif;
+-----------------------------------------------------+------------------------------------------------------------
---+-----------------+---------+----------------------------------+---------+
| Subj                                                | Prop
   | Obj             | GraphID | Stmt                             | HasType |
+-----------------------------------------------------+------------------------------------------------------------
---+-----------------+---------+----------------------------------+---------+
| Uv::file:///home/vaibhav/employee.n3#department/10: | Uv::http://localhost:2020/vocab/resource/department_deptNam
e: | Lv:0::Research: |       1 | Bv::-37c07bf6:12c37bfcd74:-7ff5: | T       |
+-----------------------------------------------------+------------------------------------------------------------
---+-----------------+---------+----------------------------------+---------+
1 row in set (0.00 sec)

mysql> 
```

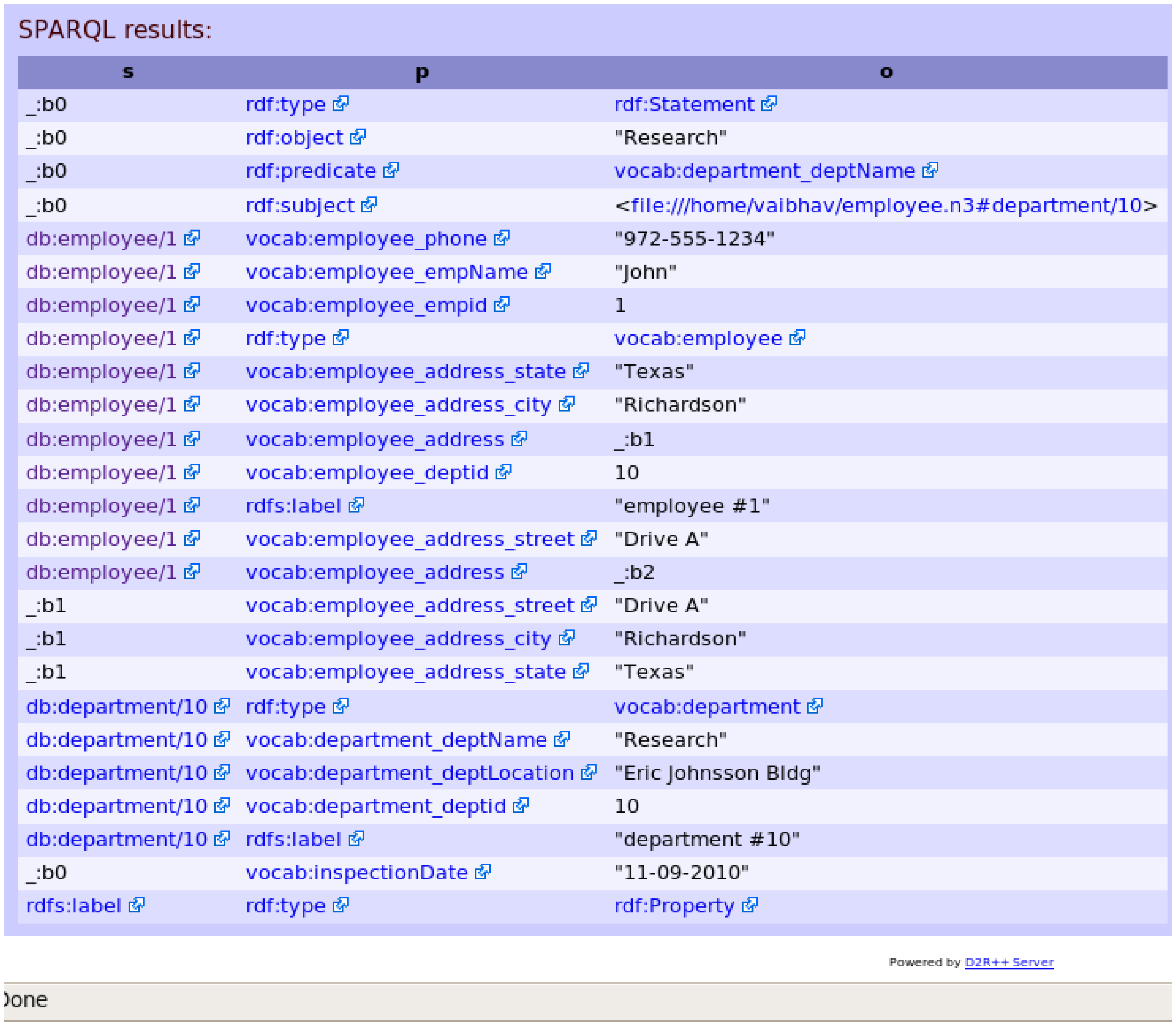
Figure 9.19: MySQL Database after Addition of *inspectionDate* Reification Information to *Department* 10


Snorql: Exploring http://localhost:2020/sparql - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:2020/snorql/?query=SELECT+DISTINCT+*+WHERE+{%0D%0A++%3Fs+%3Fp+%3Fo%0D%0A
Most Visited
Smart Bookmarks
Getting Started
Latest Headlines
Snorql: Exploring http://localhos...
Snorql: Exploring http://localhost:2020/sparql
SPARQL:
Results: Browse
Go!
Reset

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX db: <http://localhost:2020/resource/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX map: <file:/home/vaibhav/employee.n3#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX vocab: <http://localhost:2020/vocab/resource/>
SELECT DISTINCT * WHERE {
  ?s ?p ?o
}
LIMIT 100
```

Figure 9.20: SPARQL Query Extracting RDBMS Data in RDF Form

data loader utilities provided with the underlying relational databases will be used rather than D2RQ++.


SPARQL results:

s	p	o
_:b0	rdf:type	rdf:Statement
_:b0	rdf:object	"Research"
_:b0	rdf:predicate	vocab:department_deptName
_:b0	rdf:subject	<file:///home/vaibhav/employee.n3#department/10>
db:employee/1	vocab:employee_phone	"972-555-1234"
db:employee/1	vocab:employee_empName	"John"
db:employee/1	vocab:employee_empid	1
db:employee/1	rdf:type	vocab:employee
db:employee/1	vocab:employee_address_state	"Texas"
db:employee/1	vocab:employee_address_city	"Richardson"
db:employee/1	vocab:employee_address	_:b1
db:employee/1	vocab:employee_deptid	10
db:employee/1	rdfs:label	"employee #1"
db:employee/1	vocab:employee_address_street	"Drive A"
db:employee/1	vocab:employee_address	_:b2
_:b1	vocab:employee_address_street	"Drive A"
_:b1	vocab:employee_address_city	"Richardson"
_:b1	vocab:employee_address_state	"Texas"
db:department/10	rdf:type	vocab:department
db:department/10	vocab:department_deptName	"Research"
db:department/10	vocab:department_deptLocation	"Eric Johnsson Bldg"
db:department/10	vocab:department_deptid	10
db:department/10	rdfs:label	"department #10"
_:b0	vocab:inspectionDate	"11-09-2010"
rdfs:label	rdf:type	rdf:Property

Figure 9.21: SPARQL Query Output Including *inspectionDate* Reification Information

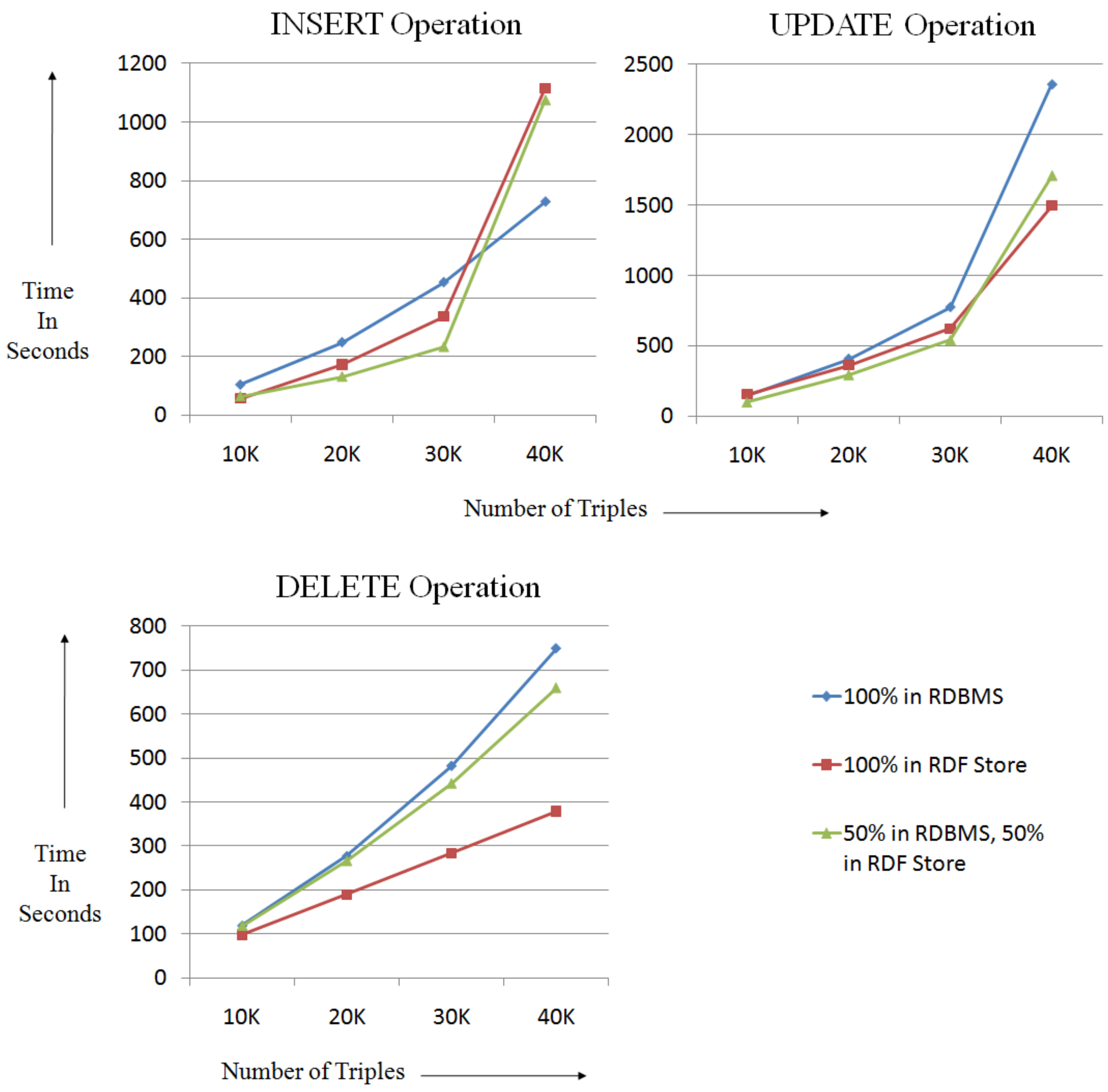


Figure 9.22: Performance of DML Operations

CHAPTER 10
FUTURE WORK AND CONCLUSION

The following subsections discuss future directions that include enhancements to R2D that incorporate entity alignment techniques to enable Duplicate Table Detection (DTD), extending R2D's semantics-preserving schema translation to a wider variety of RDF graphs that include resources belonging to multiple classes and reification information, and extending R2D's semantics-preserving query translation to encompass more features from the SQL language, including ones that do not have a syntactic equivalent in the SPARQL language. Future directions for D2RQ++ include enhancements that enable improvisation of the mapping and bi-directional translation process for mixed blank nodes, which consist of both literal as well as resource objects, and support for SPARQL/Update, an update language for RDF graphs, along with efficient, correct, and complete translation procedures from SPARQL/Update statements into equivalent SQL DML statements. The chapter ends with a brief discussion on R2D and some concluding remarks.

10.1 Future Directions for R2D

10.1.1 Entity Alignment/Matching in RDFMapFileGenerator

In the case of RDF graphs without RDFS/Ontological information, determination of table similarity and duplicate table detection (DTD) is a crucial component of the RDFMapFileGenerator algorithm. In the absence of the DTD component, the algorithm would effectively generate one tablemap for each resource in the RDF graph, thereby resulting in an enormous number of tables, most of which would be duplicates of each other. Further, trivial similarity detection algorithms that identify similarity between a pair of tables only if the two tables are identical do not scale well and do not result in a meaningful equivalent relational schema for a given RDF graph/store. In order to better understand how trivial/rigid similarity detection algorithms fail, let us consider the two graphs, Graph A and Graph B, illustrated in Figure 10.1.

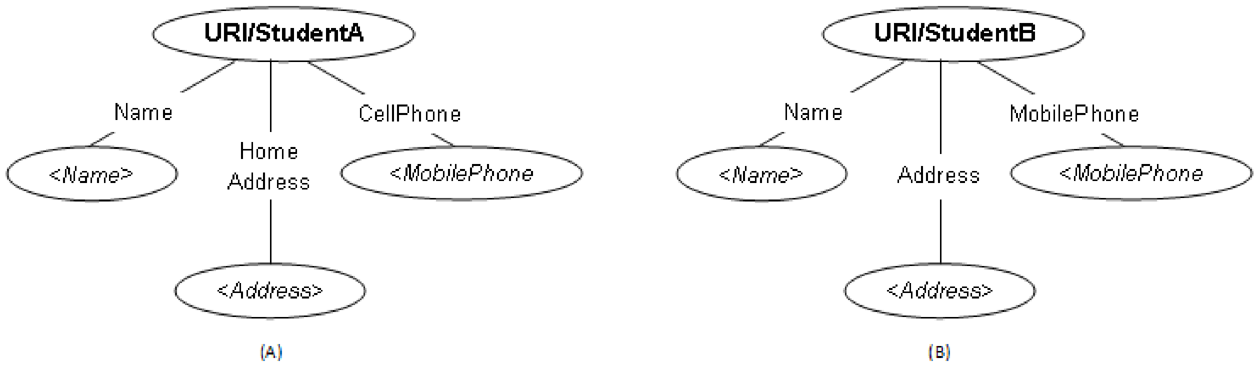
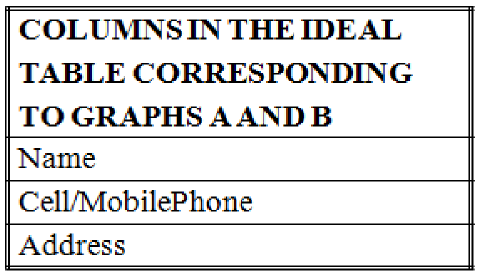


Figure 10.1: Sample RDF Graphs A and B

We, as human experts, realize that, despite minor differences in the nomenclature of predicates and in the way certain data (such as Address information) is represented, both of the above graphs represent the same concept, namely, STUDENT, and, therefore, there should only be one table for both of the graphs. The ideal table, per human intuition would be as shown in Figure 10.2 (a). However, in the absence of any ontology alignment/schema matching techniques, the RDFMapFileGenerator algorithm is not equipped with enough information to arrive at the same conclusions as those of human experts. Thus, there are two equivalent relational r2d:TableMaps, instead of one, generated for the two graphs by the RDFMapFileGenerator with columns as shown in Figure 10.2 (b).

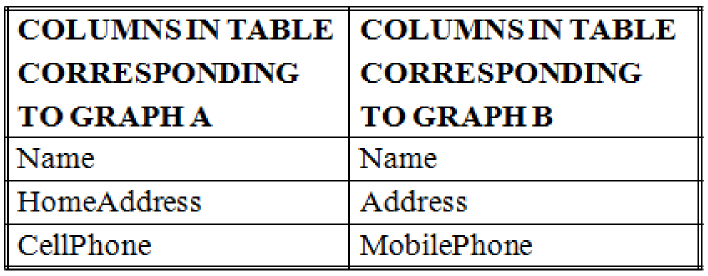
The graphs in Figure 10.3 illustrate examples of 1:N matching between attribute pairs. In this scenario, the *Phone* predicate in Graph B semantically corresponds to both *Phone1* and *Phone2* predicates in Graph A. Further, the *Address* predicate in Graph B is simply a concatenation of the *Street, City,* and *State.*

Thus, intuitively, the ideal table for this scenario should be as shown in Figure 10.4 (a) with the *Phone* predicate of Graph B mapping to either the *Phone1* or the *Phone2* column in



COLUMNS IN THE IDEAL TABLE CORRESPONDING TO GRAPHS A AND B
Name
Cell/MobilePhone
Address

(a)



COLUMNS IN TABLE CORRESPONDING TO GRAPH A	COLUMNS IN TABLE CORRESPONDING TO GRAPH B
Name	Name
HomeAddress	Address
CellPhone	MobilePhone

(b)

Figure 10.2: (a) Columns in the Ideal Table corresponding to Graphs A and B in Figure 10.1; (b) Columns generated by RDFMapFileGenerator in the absence of ontology alignment techniques

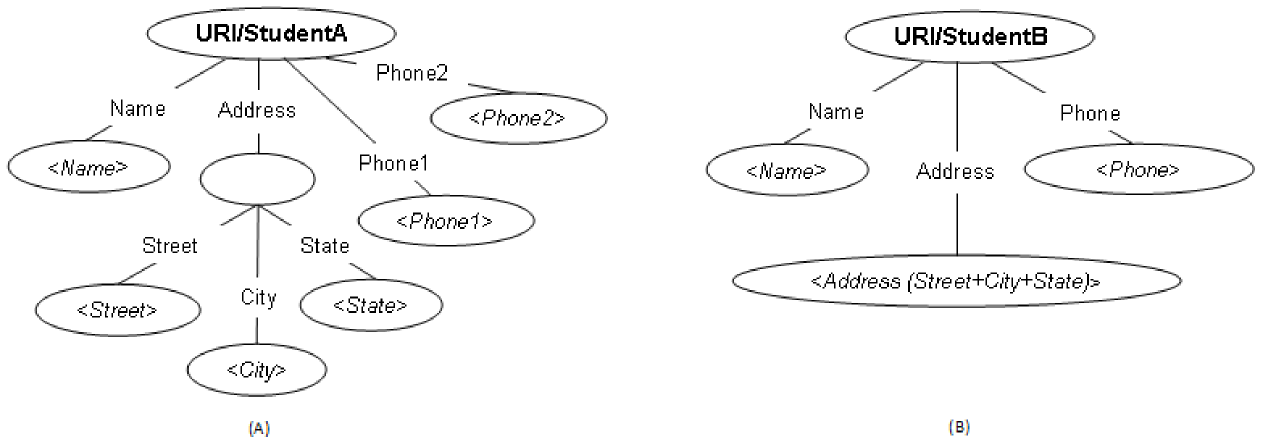
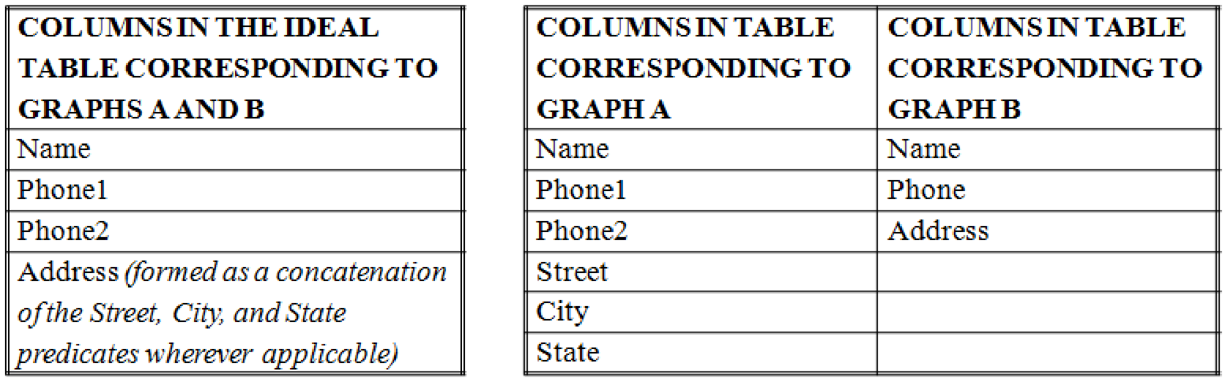


Figure 10.3: Sample RDF Graphs A and B

the ideal table and the *Street, City,* and *State* predicates of Graph A being concatenated to form the values of the *Address* column in the ideal table. However, the actual tables generated by RDFMapFileGenerator without the DTD component are as shown in Figure 10.4 (b).

For large RDF graphs with millions of triples, the number of r2d:TableMaps generated by RDFMapFileGenerator would be massive and meaningless. Therefore, determination and elimination of duplicate tablemaps is essential in order to arrive at a manageable and meaningful relational equivalent of the given RDF graph. Consequently, it is imperative that some form of ontology alignment/schema matching technique is included within the RDFMapFileGenerator algorithm.



COLUMNS IN THE IDEAL TABLE CORRESPONDING TO GRAPHS A AND B
Name
Phone1
Phone2
Address *(formed as a concatenation of the Street, City, and State predicates wherever applicable)*

(a)

COLUMNS IN TABLE CORRESPONDING TO GRAPH A	COLUMNS IN TABLE CORRESPONDING TO GRAPH B
Name	Name
Phone1	Phone
Phone2	Address
Street	
City	
State	

(b)

Figure 10.4: (a) Columns in the Ideal Table corresponding to Graphs A and B in Figure 10.3; (b) Columns generated by RDFMapFileGenerator in the absence of ontology alignment techniques

Fortunately, the problem of information integration, necessitated by the pervasiveness of semantic heterogeneity across data sources, has been, and still is, the topic of numerous research efforts (Dai, Koudas, Ooi, Srivastava, and Venkatasubramanian 2006; Bohannon, Elnahrawy, Fan, and Flaster 2006; Warren and Tompa 2006; Wang, Englebienne, and Schlobach 2008; Wartena and Brussee 2008) aimed at discovering innovative strategies for the same. These strategies include application of name similarity, instance similarity, and relationship similarity techniques in order to semantically align tables from various data sources. Thus our DTD problem can be rephrased and rewritten as a schema matching problem as follows:

Duplicate Table Detection Problem Statement: Given two potential r2d:TableMaps, T_1 and T_2, each of which is composed of a set of attributes (columns) where $\{A_{11}, A_{12}, A_{13}, \ldots, A_{1M}\} = \texttt{Attributes}(T_1)$ and $\{A_{21}, A_{22}, A_{23}, \ldots, A_{2N}\} = \texttt{Attributes}(T_2)$, the goal is to determine the semantic similarity between T_1 and T_2.

As mentioned earlier, there are several techniques proposed by various researchers that attempt to address the above problem statement. Some of these include instance-based schema matching techniques based on comparison of distributions of N-grams among instances of compared attributes (Dai, Koudas, Srivastava, Tung, and Venkatasubramanian 2008), semantic techniques (Partyka, Khan, and Thuraisingham 2009a; Partyka, Khan, and Thuraisingham 2009b) that are free from the syntactic requirements of N-Grams, and techniques such as MultipleMatch (Partyka, Alipanah, Khan, Thuraisingham, and Shekhar 2008) that cater to the 1:N attribute matching scenarios illustrated in Figure 10.3. The next phase with regards to R2D will involve evaluating and, possibly, incorporating one or more of these techniques to address and resolve the DTD problem described above.

10.1.2 Semantics-Preserving Schema Translation Augmentation

At the current time, R2D's semantics-preserving schema translation is restricted to a sub-set of RDF graphs that does not include reification information and the resources of which are assumed to be instances of one and only one RDFS class. Future work includes extending the semantics-preserving framework to include a wider variety of RDF graphs that do not place any of the current restrictions on the resources comprising the graphs.

10.1.3 Semantics-Preserving Query Translation Augmentation

Future directions for R2D's semantics-preserving query translation include expanding the feature set of the SQL core language to include aggregate functions, pattern matching operations, WHERE clauses, and outer join operations that are equivalent to optional graph patterns in SPARQL. Additionally, the feasibility of translating features that, at the current time, do not have syntactic equivalents in the SPARQL language, such as the ones listed in Table 8.2, will be assessed and incorporated in the SQL core language where possible.

10.2 Future Directions For D2RQ++

As mentioned earlier, D2RQ++ was motivated by a need to enable DML operations to be propagated back to the underlying relational database whenever possible and to continue to maintain the Open-World Assumption during the propagation process. One of the requirements of our bi-directional translation work was to reuse and extend existing uni-directional solutions in the translational arena in order to avoid reinventing the wheel wherever possible. The requirements and objectives behind D2RQ++ were achieved by eliminating the read-only restriction on the widely adopted uni-directional translation tool, D2RQ, thereby enabling two-way interaction between the base RDBMS and the virtual RDF layer wrapping the RDBMS. D2RQ++ is, thus, essentially a wrapper around D2RQ that transforms the latter from a read-only application to a read-write application.

The D2RQ++ initiative can be realized by adopting one of two available paths. The first path is one that does not place any restrictions on structural changes to the underlying RDBMS and permits D2RQ++ to make any required changes. This method was considered and rejected as it is believed that structural modifications to a relational schema should be under the complete control of a responsible Database Administrator and should not be initiated through any unsupervised, automated software program. The second alternative, and the one adopted in the current version of D2RQ++, is one that does not permit any structural changes to the underlying database. Some of the enhancements that are currently being researched into with regards to D2RQ++ are as follows.

10.2.1 Provision of SPARQL/Update End Point and Query Translation Algorithms

SPARQL/Update, a companion to the original (query-only) SPARQL Query Language for RDF Stores, is a data manipulation language that enables insert, update, and delete operations on RDF graphs. The SPARQL/Update specification is currently under review with the World Wide Web Consortium and is expected to become a recommendation in due course. As a result, providing support for SPARQL/Update statements in subsequent versions of D2RQ++ is a priority. Further, while there exists a large body of work in syntactically and semantically correct transformation of SPARQL queries into equivalent SQL queries (Wang, Zhang, Miao, and Lu 2010; Elliott, Cheng, Thomas-Ogbuji, and Ozsoyoglu 2009; Chebotko, Lu, and Fotouhi 2009; Wu, Chen, Wang, Wang, Mao, Tang, and Zhou 2006), and while the semantics and complexity of SPARQL have been studied (Pérez, Arenas, and Gutierrez 2006b; Pérez, Arenas, and Gutierrez 2006a), there is no equivalent foundational work done in translation of SPARQL/Update into SQL DML operations. In order to ensure the soundness (correctness and completeness) of the D2RQ++ DML statements translation process, similar rigorous mathematical foundations have to be derived and proved for SPARQL/Update.

10.2.2 Relational-to-RDF Transformation and Update of Nested & Mixed Blank Nodes

At the current time, neither D2RQ's nor D2RQ++'s mapping languages support the translation of relational data into mixed/nested blank nodes. However, since D2RQ is a read-only application, the lack of support for translations into nested/mixed blank nodes is perfectly acceptable. On the other hand, D2RQ++ is an application that accepts triples for update into the corresponding underlying relational database. Therefore, it becomes necessary to arrive at an appropriate translation and update process for such mixed/nested blank nodes. Mapping constructs need to be designed to map, translate, and update such incoming nodes into the underlying relational database.

10.3 Multi-Platform Support

From an implementation perspective, the present version of R2D is tightly integrated with the opensource reporting tool, DataVision. Future efforts will include extending R2D to work with other reporting tools such as Crystal Reports (Reports), SQLMaestro (SQLMaestro), etc. D2RQ++ also has the same shortcoming of being dependent on Jena and its associated API for RDF triples manipulation, and MySQL for housing the RDF triples data within the underlying relational database. Therefore, future directions for D2RQ++ also include providing support for various other platforms such as Sesame (Broekstra, Kampman, and

Harmelen 2002), for RDF manipulation/storage and Oracle, DB2, Sybase, MS SQL Server, etc. for storage of RDF data in the underlying relational database.

10.4 Discussion and Conclusion

The R2D framework presented in this dissertation is an attempt at integrating relational concepts with the new semantic web concepts with the objective of permitting reusability of tools that are based on a relational model. Since current storage methods for RDF stores involve housing the triples in a relational database, some factions may consider R2D to be a "double-wrapping" application that provides a relational wrapper around RDF stores that are, in turn, stored in a relational database. However, almost every storage mechanism involves the creation of a generic, non-application-specific $< s, p, o >$ table that would make the determination of the problem domain addressed by the model difficult without examining the actual data. Further, querying data, using SQL, from such a generic table, to arrive at meaningful information is not a trivial task. It would involve umpteen self-joins on the same table and would require the presence of a domain expert with detailed knowledge of the data. This is because, using these models, it would be impossible for a user to infer the schema and the entities, the attributes, and relationship comprising the same. R2D offers the users the ability to do just this and enables them to actually arrive at a complete Entity-Relationship Diagram using the RDF-to-Relational Schema transformation process and fire SQL queries against the same for information.

Further, R2D, unlike other mapping efforts, can generate an equivalent relational schema even for "sloppy" data (in which ontological constructs/schema definitions are absent) through extensive examination of the data to identify groups of instances that have mostly the same properties associated with them. The degree of accuracy of the generated schema in the absence of structural information may not be as high as when such information is available due to uncertainties regarding similarity of the tables generated in the relational schema. Decisions such as "how similar should two tables be before they are considered to be the same and consolidated" depends, in the absence of structural information, on similarity thresholds set within the algorithm and the accuracy varies depending on the thresholds. Ontology alignment and entity homogeneity/schema matching techniques such as (Dai, Koudas, Srivastava, Tung, and Venkatasubramanian 2008) can be incorporated to improve the accuracy in such scenarios, but are out of the scope of our research.

The R2D framework presented in this dissertation was motivated by a dearth in the number and variety of data modeling, management, and visualization tools for RDF graph data. Though there are a several ongoing research efforts that attempt to address these deficiencies, most of the efforts involve either the painstaking process of creating new tools or the uneconomical alternative of duplicating data into existing relational stores raising a fresh crop of concerns such as resource wastage and synchronization issues (Jiang, Ju, and Xu 2009). The chief goal of R2D is to bridge the gap between RDF data sources and the relational model in order to continue to leverage the benefits offered by existing traditional tools without any customization for RDF. A JDBC interface aimed at accomplishing this goal through a mapping between RDF Graph constructs and their equivalent relational counterparts was presented. A detailed description of the mapping constructs, the system architecture, and the modus operandi of the proposed system was discussed along with in depth discussion on the algorithms comprising the R2D framework. An analogous set of mapping constructs and transformation procedures to enable bi-directional data flow between virtual RDF stores and their corresponding underlying physical relational databases were presented in the form of D2RQ++.

The semantics preservation of R2D's translation framework was presented through the proofs of two theorems, one each for R2D's schema transformation and query transformation processes, and the feasibility of the both proposed frameworks was demonstrated through a variety of experimental results in the form of screenshots and performance graphs.

APPENDIX A

SUPPORTING FUNCTIONS USED IN SEMANTICS-PRESERVATION PROOFS

The various functions used in the proofs of R2D's semantics-preserving characteristic are detailed in this appendix. Tables A.1 through A.3 give a brief description of the functions while the mathematical definitions of the same are highlighted in Figures A.1 through A.10 following the tables.

Table A.1. Theorem 1 - Supporting Functions

Function	Description
`GetRegTblID`$(\mathbb{R}, r)$	Function that returns the name of the table in which r exists as a value in the table's primary key column.
`GetNewRegTblRow`$(\mathbb{R}, (t_{id}, t_t))$	Function that returns the greatest non-empty row number in a regular table with name $= t_{id}$ and table type $= t_t$.
`GetNewNonRegTblRow`$(\mathbb{R}, (t_{id}, t_t), t'_{id})$	Function that returns the greatest non-empty row number in a MVA/N:M (non-regular) table with name $= t_{id}$ and table type $= t_t$.
`GetTmpNMColCnt`$(\mathbb{R}, t_{id})$	Function that returns number of non-empty columns in a temporary N:M table with name $= t_{id}$.
`GetTmpMVAColCnt`$(\mathbb{R}, t_{id})$	Function that returns number of non-empty columns in a temporary MVA table with name $= t_{id}$.

Continued on Next Page...

Table A.1. – Continued

Function	Description
`GetTmpTblRow`$(\mathbb{R}, (t_{id}, t_t))$	Function that returns the complete first row of a temporary table with name = t_{id} and table type = t_t.
`UpdateCell`$(\mathbb{R}, r, p, k_t, d)$	Function that updates the cell identified by row w of the (p, k_t) column in the regular table where r is a primary key column value in row w, and sets the cell value to d.
`UpdateTmpCell`$(\mathbb{R}, t, k, d)$	Function that updates the cell in an empty row in column k of table t and sets the cell value to d.
`AddNMTblRow`$_{\texttt{IP}}(\mathbb{R}, t_{id}, c)$	Function that moves data from a temporary N:M table to an empty row in the appropriate non-temporary N:M table upon the arrival of an NMI Edge that completes all required information corresponding to the temporary N:M table row.
`AddNMTblRow`$_{\texttt{RP}}(\mathbb{R}, t_{id}, p, r)$	Function that moves data from a temporary N:M table to an empty row in the appropriate non-temporary N:M table upon the arrival of an NM Resource (NMRT) Edge that completes all required information corresponding to the temporary N:M table row.
`AddMVATblRow`$_{\texttt{IP}}(\mathbb{R}, t_{id}, c)$	Function that moves data from a temporary MVA table to an empty row in the appropriate non-temporary MVA table upon the arrival of an MVAI Edge that completes all required information corresponding to the temporary N:M table row.

Continued on Next Page…

Table A.1. – Continued

Function	Description
`AddMVATblRow`$_{\texttt{LP}}(\mathbb{R}, t_{id}, p, l)$	Function that moves data from a temporary MVA table to an empty row in the appropriate non-temporary MVA table upon the arrival of an MVA Attribute (Literal) (MVALT) Edge that completes all required information corresponding to the temporary MVA table row.
`AddMVATblRow`$_{\texttt{RP}}(\mathbb{R}, t_{id}, p, r)$	Function that moves data from a temporary MVA table to an empty row in the appropriate non-temporary MVA table upon the arrival of an MVA Resource (MVART) Edge that completes all required information corresponding to the temporary MVA table row.

Table A.2. Theorem 1 - Temporary Triples Manipulation Functions

Function	Description
`GetTmpTblLitCols`$(\mathbb{R}, r)$	Function that returns all literal-valued cells in the temporary table named r.
`AddTmpLColstoRegTbl`$(\mathbb{R}, t_{id}, r, w)$	Function that moves all literal-valued cells in the temporary table named r to the regular table, t_{id}, where r is added as a primary key value.

Continued on Next Page…

Table A.2. – Continued

Function	**Description**
`GetTmpRColsWithP`$(\mathbb{R}, r)$	Function that returns those resource-valued cells in the temporary table named r, where the resource values in the temporary table exist as primary key values in some regular (non-temporary) table.
`AddTmpRColsWithPtoRegTbl`$(\mathbb{R}, t_{id}, r, w)$	Function that moves those resource-valued cells in the temporary table named r, where the resource values exists as primary key values in some regular table, to the regular table, t_{id}, where r is added as a primary key value.
`GetTmpRTblsWithrObj`$(\mathbb{R}, r)$	Function that returns those resource cells that have r as cell values from those temporary tables where the resource after which the temporary table is named exists as a primary key value in some regular, non-temporary table.
`UpdRRegTblsWithrObj`$(\mathbb{R}, r, c, w)$	Function that moves those resource cells that have r as cell values from temporary tables into corresponding regular tables which have, as primary key values, the resources after which the temporary tables are named..

Continued on Next Page…

Table A.2. – Continued

Function	**Description**
`GetValidTmpNMTbls`$(\mathbb{R}, r)$	Function that returns the complete row from those N:M temporary tables that have r as a cell value and that have all other required information (one NMI edge and two NMRT edges with the second resource, r' existing as primary key in some regular table) corresponding to the N:M row available.
`GetClassForTmpNMTblCells`$(\mathbb{R}, t_{id})$	Function that returns the names of the tables in which the resource values in row 1 of table t_{id} exist as primary key values.
`AddTmpNMTblRowstoNMTbl`$(\mathbb{R}, r, c, w)$	Function that adds r as a primary key column value into a new row in table c and moves the complete row from those temporary N:M tables that have r as a cell value, and that have all other required information corresponding to the N:M row available, to an empty row in the appropriate non-temporary N:M table.
`GetValidTmpMVATbls`$(\mathbb{R}, r)$	Function that returns the complete row from those MVA temporary tables that have r as a cell value and that have all other required information (one MVAI edge, one MVALT edge, and the MVART edge that has r as the object value) corresponding to the MVA row available.

Continued on Next Page…

Table A.2. – Continued

Function	Description
`AddTmpMVATblRowstoMVATbl`($\mathbb{R}$, r, c, w)	Function that adds r as a primary key column value into a new row in table c and moves the complete row from those temporary MVA tables that have r as a cell value, and that have all other required information corresponding to the MVA row available, to an empty row in the appropriate non-temporary MVA table.

Table A.3. Theorem 2 - Supporting Functions

Function	Description
$\mathcal{S}_T[\![\hat{t}]\!]\phi$	Denotation of a relational table, $\hat{t}$, that returns a forest from an input graph, ϕ, with one tree per primary key value in the table.
$\mathcal{S}_{F_{pk}}[\![\hat{t}]\!]\phi$	Denotation of a primary key column value which returns a tree, rooted at the primary key value, whose edges are the attributes of the primary key value extracted from the input graph ϕ.

Continued on Next Page…

Table A.3. – Continued

Function	Description
`JoinGraphs`(ϕ_1, ϕ_2)	Function that denotes the cartesian product of two tables. It takes as input the cross-product of the edges of the two graphs representing the two tables, and for each edge-pair in the set ϕ_1-ϕ_2, it replaces the root of each edge with a vertex that is a concatenation of the two original vertices v_1 and v_1'.
`CreateEdges`$(d, S_{r_{\phi_\mathcal{S}}})$	Function that creates an edge, (v_1, l, v_2), for every element, $r_{\phi_\mathcal{S}}$, in the set $S_{r_{\phi_\mathcal{S}}}$ such that $v_1 = r_{\phi_\mathcal{S}}$, and $l = v_2 = d$.
`ExtColEdges`$(\hat{t}_{\text{-}T_{id}}\hat{cl}, r_{\phi_\mathcal{S}} \cup S_{r_{\phi_\mathcal{S}}}, \phi_F)$	Function that extracts the edge labeled $\hat{t}_{\text{-}T_{id}}\hat{cl}$ for every element, $r_{\phi_\mathcal{S}}$, in the set $S_{r_{\phi_\mathcal{S}}}$ from the graph ϕ_F. It corresponds to selecting a column from a table in an SQL query.
`AttachEdge`$(er_{\phi_\mathcal{S}s})$	Function that takes set of tuples with each tuple consisting of an edge, (v_1, l, v_2), and a root node, $r_{\phi_\mathcal{S}}$, and, if v_1 is contained in v_1', creates a new edge, edgeb, such $v_1' = r_{\phi_\mathcal{S}}$, $l' = l$, and $v_2' = v_2$, thereby attaching the edge, (v_1, l, v_2), to the rootnode, $r_{\phi_\mathcal{S}}$.
`1NPrefixExists`(l)	Function that determines whether the given edge label, l, corresponds to an edge representing a 1:N relationship between the vertices connected by the edge label.
`contains`$(\hat{t}, s)$	Function that determines if a pattern, s, is contained within the first argument, $\hat{t}$.
`GetRootNodes`(ϕ)	Function that returns all the root nodes in the forest ϕ.

Continued on Next Page…

Table A.3. – Continued

Function	**Description**
`GetColEdge`$(\hat{t}_{\text{-}T_{id}}\hat{cl}, r_{\phi_S}, \phi)$	Function that returns, from the graph ϕ, the edge rooted at r_{ϕ_S} and labeled $\hat{t}_{\text{-}T_{id}}\hat{cl}$.
`TableExists`$(r, \hat{t})$	Function that determines if the given table, $\hat{t}$, is contained in the root node, r. It is used to determine if $\hat{t}$ has previously been encountered in some recursive step during an SQL query processing.
`ElementExists`(l', v_2', ϕ)	Function that determines if an edge, (v_1, l, v_2), such that $l = l'$ and $v_2 = v_2'$ exists in the given graph ϕ.
`AddTableLabel`$(\phi, \hat{t})$	Function that, for every edge, (v_1, l, v_2), in ϕ, prefixes and suffixes the first vertex, v_1 with the given table name, $\hat{t}$.
`RemoveTableLabels`(ϕ)	Function that removes the concatenation character used by `AddTableLabel` when prefixing and suffixing the first vertex, v_1, of every edge, (v_1, l, v_2), with a table name, $\hat{t}$.
`GetRoot`(ϕ)	Function that returns the root of the given graph ϕ.
`RemoveRoot`(ϕ, r)	Function that returns a graph that is obtained by removing the root element, r, from the given graph ϕ.
`AddRoot`(ϕ, r)	Function that adds the given root, r, to the given graph, ϕ, through the addition of edges of the form $(r, \texttt{rowlabel}, v_2)$ for every edge, (v_1', l', v_2'), in ϕ where $v_2 = v_1'$.
`AddTable`$(r, \hat{t})$	Function that adds the given table name, $\hat{t}$, to the root vertex, r.

Continued on Next Page...

Table A.3. – Continued

Function	**Description**
`AddEdge`$(\hat{s}, l, \phi)$	Function that, for every edge, (v_1, l, v_2), in ϕ where v_1 is not the root of the graph, ϕ, adds an edge of the form $(v_1, l, \hat{s})$.
`RemoveFirstChar`$(\dot{v})$	Function that returns $\dot{v}$ with the first character removed.
`GetSubQueryAlias`$(\dot{spq}_{sq})$	Function that returns the alias name of the given sub-query, $\dot{spq}_{sq}$, by extracting the prefix of the alias name of the first SELECT element in the sub-query.
`IsTable`$(\rho_{\mathbb{TR}}, \hat{t})$	Function that returns `true` if the given table name, $\hat{t}$, is a simple table or `false` if it is a derived table (or sub-query).

$$\texttt{GetRegTblID}(\mathbb{R}, r) = \{t_{id} \mid \mathbb{R}((t_{id}, \tau_r), (t_{id}\text{-}\texttt{type}, \kappa_{pk}), w) = r\}$$

$$\texttt{GetNewRegTblRow}(\mathbb{R}, (t_{id}, t_t)) = \texttt{max}\{0, w|((t_{id}, t_t), (t_{id}\text{-}\texttt{type}, \kappa_{pk}), w) \in \mathbb{R}^{\leftarrow}\}$$

$$\texttt{GetNewNonRegTblRow}(\mathbb{R}, (t_{id}, t_t), t'_{id}) = \texttt{max}\{0, w|((t_{id}, t_t), (t'_{id}\text{-}\texttt{type}, \kappa_{pk}), w) \in \mathbb{R}^{\leftarrow}\}$$

$$\texttt{GetTmpNMColCnt}(\mathbb{R}, t_{id}) = |\ ((t_{id}, \tau_{tmp_{nm}}), k, 1) \in \mathbb{R}^{\leftarrow}\ |$$

$$\texttt{GetTmpMVAColCnt}(\mathbb{R}, t_{id}) = |\ ((t_{id}, \tau_{tmp_{mva}}), k, 1) \in \mathbb{R}^{\leftarrow}\ |$$

$$\texttt{GetTmpTblRow}(\mathbb{R}, (t_{id}, t_t)) = \{(((t_{id}, t_t), k, 1), d) \mid (((t_{id}, t_t), k, 1), d) \in \mathbb{R}\}$$

$$\begin{aligned}\texttt{UpdateCell}(\mathbb{R}, r, p, k_t, d) = \{&(((t_{id}, \tau_r), (p, k_t), w), d) \mid \mathbb{R}((t_{id}, \tau_r), (t_{id}\text{-}\texttt{type}, \kappa_{pk}),\\ &w) = r\}\end{aligned}$$

$$\texttt{UpdateTmpCell}(\mathbb{R}, t, k, d) = \{((t, k, w+1), d) \mid w = \texttt{max}\{0, w'|(t, k, w') \in \mathbb{R}^{\leftarrow}\}$$

$$\begin{aligned}\texttt{AddNMTblRow}_{\texttt{IP}}(\mathbb{R}, t_{id}, c) = \{&(((c, \tau_{nm}), (k_{id}, \kappa_{pk}), w), \mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1))\\ &\mid c = t_{id1}\text{-}t_{id2},\\ &w = \texttt{GetNewNonRegTblRow}(\mathbb{R}, (c, \tau_{nm}), t_{id1}),\\ &((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1) \in \mathbb{R}^{\leftarrow}\}\end{aligned}$$

$$\begin{aligned}\texttt{AddNMTblRow}_{\texttt{RP}}(\mathbb{R}, t_{id}, p, r) = \{&(((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w), \mathbb{R}((t_{id}, \tau_{tmp_{nm}}),\\ &(k_{id}, \kappa_{fk}), 1)), (((t_{idnm}, \tau_{nm}), (p, \kappa_{fk}), w), r)\\ &\mid t_{idnm} = \mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1) = t_{id1}\text{-}t_{id2},\\ &w = \texttt{GetNewNonRegTblRow}(\mathbb{R}, (t_{idnm}, \tau_{nm}), t_{id1}),\\ &((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1) \in \mathbb{R}^{\leftarrow},\\ &((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1) \in \mathbb{R}^{\leftarrow}\}\end{aligned}$$

Figure A.1: Theorem 1 - Supporting Functions (a)

$$\begin{aligned}\texttt{AddMVATblRow}_{\texttt{IP}}(\mathbb{R}, t_{id}, c) = \{ & (((c, \tau_{mva}), (k_{id}, \kappa_{pk}), w), \mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1)), \\ & \mid c = t'_{id}\text{-}p,\ k_{id} \neq \texttt{type}, \\ & w = \texttt{GetNewNonRegTblRow}(\mathbb{R}, (c, \tau_{mva}), t'_{id}), \\ & ((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1) \in \mathbb{R}^{\leftarrow} \}\end{aligned}$$

$$\begin{aligned}\texttt{AddMVATblRow}_{\texttt{LP}}(\mathbb{R}, t_{id}, p, l) = \{ & (((t_{idmva}, \tau_{mva}), (k_{id}, \kappa_{pk}), w), \mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1)), \\ & (((t_{idmva}, \tau_{mva}), (p, \kappa_{pk}), w), l) \\ & \mid t_{idmva} = \mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1) = t'_{id}\text{-}p,\ k_{id} \neq \texttt{type}, \\ & w = \texttt{GetNewNonRegTblRow}(\mathbb{R}, (t_{idmva}, \tau_{mva}), t'_{id}), \\ & ((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1) \in \mathbb{R}^{\leftarrow}, \\ & ((t_{id}, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1) \in \mathbb{R}^{\leftarrow} \}\end{aligned}$$

$$\begin{aligned}\texttt{AddMVATblRow}_{\texttt{RP}}(\mathbb{R}, t_{id}, p, r) = \{ & (((t_{idmva}, \tau_{mva}), (k_{id}, \kappa_{pk}), w), \mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1)), \\ & (((t_{idmva}, \tau_{mva}), (p, \kappa_{pk}), w), r) \\ & \mid t_{idmva} = \mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1) = t'_{id}\text{-}p,\ k_{id} \neq \texttt{type}, \\ & w = \texttt{GetNewNonRegTblRow}(\mathbb{R}, (t_{idmva}, \tau_{mva}), t'_{id}), \\ & ((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1) \in \mathbb{R}^{\leftarrow}, \\ & ((t_{id}, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1) \in \mathbb{R}^{\leftarrow} \}\end{aligned}$$

Figure A.2: Theorem 1 - Supporting Functions (b)

$$\begin{aligned}
\texttt{GetTmpTblLitCols}(\mathbb{R}, r) &= \{(((r, \tau_{tmp}), (k_{id}, \kappa_r), w), d) \mid \\
&\quad (((r, \tau_{tmp}), (k_{id}, \kappa_r), w), d) \in \mathbb{R},\ d \in \mathcal{L}\}
\end{aligned}$$

$$\begin{aligned}
\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, t_{id}, r, w) &= \{(((t_{id}, \tau_r), (t_{id}\texttt{\_type}, \kappa_{pk}), w), r), \\
&\quad (((t_{id}, \tau_r), (k_{id}, \kappa_r), w), \mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w')) \mid \\
&\quad ((r, \tau_{tmp}), (k_{id}, \kappa_r), w') \in \mathbb{R}^{\leftarrow}, \\
&\quad \mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w') \in \mathcal{L}\}
\end{aligned}$$

$$\begin{aligned}
\texttt{GetTmpRColsWithP}(\mathbb{R}, r) &= \{(((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w), d) \mid \\
&\quad (((r, \tau_{tmp}), (k_{id}, \kappa_r), w), d) \in \mathbb{R}, \\
&\quad \texttt{GetRegTblID}(\mathbb{R}, d) \neq \{\}\}
\end{aligned}$$

$$\begin{aligned}
\texttt{AddTmpRColsWithPtoRegTbl}(\mathbb{R}, t_{id}, r, w) &= \{(((t_{id}, \tau_r), (t_{id}\texttt{\_type}, \kappa_{pk}), w), r), \\
&\quad (((t_{id}, \tau_r), (k_{id}, \kappa_{fk}), w), d) \mid \\
&\quad d = \mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), \\
&\quad ((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w') \in \mathbb{R}^{\leftarrow}, \\
&\quad \texttt{GetRegTblID}(\mathbb{R}, d) \neq \{\}\}
\end{aligned}$$

$$\begin{aligned}
\texttt{GetTmpRTblsWithrObj}(\mathbb{R}, r) &= \{(((t_{id}, \tau_{tmp}), (k_{id}, \kappa_{fk}), w), r) \mid \\
&\quad (((t_{id}, \tau_{tmp}), (k_{id}, \kappa_{fk}), w), r) \in \mathbb{R}, \\
&\quad \texttt{GetRegTblID}(\mathbb{R}, t_{id}) \neq \{\}\}
\end{aligned}$$

$$\begin{aligned}
\texttt{UpdRRegTblsWithrObj}(\mathbb{R}, r, c, w) &= \{(((c, \tau_r), (\texttt{type}, \kappa_{pk}), w), r), \\
&\quad (((t_{id}, \tau_r), (k_{id}, \kappa_{fk}), w'), r) \mid \\
&\quad \mathbb{R}((t'_{id}, \tau_{tmp}), (k_{id}, \kappa_{fk}), w'') = r, \\
&\quad \mathbb{R}((t_{id}, \tau_r), (t_{id}\texttt{\_type}, \kappa_{pk}), w') = t'_{id}\}
\end{aligned}$$

Figure A.3: Theorem 1 - Temporary Literal and Resource Triples Manipulation Functions

$$\begin{aligned}
\texttt{GetValidTmpNMTbls}(\mathbb{R}, r) = \{&(((t_{id}, \tau_{tmp_{nm}}), k, 1), d) \mid \\
&(((t_{id}, \tau_{tmp_{nm}}), k', 1), r) \in \mathbb{R}, \\
&\mid \texttt{GetClassForTmpNMTblCells}(\mathbb{R}, t_{id}) \mid = 1, \\
&\texttt{GetTmpNMColCnt}(\mathbb{R}, t_{id}) = 3\}
\end{aligned}$$

$$\begin{aligned}
\texttt{GetClassForTmpNMTblCells}(\mathbb{R}, t_{id}) = \{&\texttt{GetRegTblID}(\mathbb{R}, r) \mid (((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), \\
&r) \in \mathbb{R}\}
\end{aligned}$$

$$\begin{aligned}
\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w) = \{&(((c, \tau_r), (\texttt{type}, \kappa_{pk}), w), r), \\
&(((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w), \\
&\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1)) \mid \\
&((t_{idnm}, \tau_{nm}), k, w) \notin \mathbb{R}, \\
&t_{idnm} = \mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), \\
&(((t_{id}, \tau_{tmp_{nm}}), k', 1), r) \in \mathbb{R}, \\
&\mid \texttt{GetClassForTmpNMTblCells}(\mathbb{R}, t_{id}) \mid = 1, \\
&\texttt{GetTmpNMColCnt}(\mathbb{R}, t_{id}) = 3\}
\end{aligned}$$

$$\begin{aligned}
\texttt{GetValidTmpMVATbls}(\mathbb{R}, r) = \{&(((t_{id}, \tau_{tmp_{mva}}), k, 1), d) \mid \mathbb{R}((t_{id}, \tau_{tmp_{mva}}), k', 1) \\
&= r, \texttt{GetTmpMVAColCnt}(\mathbb{R}, t_{id}) = 3\}
\end{aligned}$$

$$\begin{aligned}
\texttt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}, r, c, w) = \{&(((c, \tau_r), (\texttt{type}, \kappa_{pk}), w), r), \\
&(((t_{idmva}, \tau_{mva}), (k_{id}, \kappa_{pk}), w), \\
&\mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1)) \mid \\
&k_{id} \neq \texttt{type},\ ((t_{idmva}, \tau_{mva}), k, w) \notin \mathbb{R}^{\leftarrow}, \\
&((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1) \in \mathbb{R}^{\leftarrow}, \\
&t_{idmva} = \mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1), \\
&(((t_{id}, \tau_{tmp_{mva}}), k', 1), r) \in \mathbb{R}, \\
&\texttt{GetTmpMVAColCnt}(\mathbb{R}, t_{id}) = 3\}
\end{aligned}$$

Figure A.4: Theorem 1 - Temporary N:M and MVA Triples Manipulation Functions

$$\frac{}{\texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, \emptyset, \phi_F) = \emptyset}\,(\textsc{Base})$$

$$\frac{\begin{array}{c}\phi_e = \texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, S_{r_{\phi_{\mathcal{S}}}}, \phi_F) \\ \texttt{GetColEdge}(\hat{t}_{\text{-}T_{id}}\hat{cl}, r_{\phi_{\mathcal{S}}}, \phi_F) = \{\}\end{array}}{\texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, \{r_{\phi_{\mathcal{S}}}\} \cup S_{r_{\phi_{\mathcal{S}}}}, \phi_F) =}\,(\rho_{r_{\phi_{\mathcal{S}}}}\ \text{IC - }\textsc{NoMatch})$$
$$\{\texttt{AddTableLabel}(\texttt{RemoveTableLabels}(r_{\phi_{\mathcal{S}}}), \hat{t}), \hat{t}_{\text{-}T_{id}}\hat{cl}, d_{null})\} \cup \phi_e$$

$$\frac{\begin{array}{c}\phi_e = \texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, S_{r_{\phi_{\mathcal{S}}}}, \phi_F) \\ \texttt{GetColEdge}(\hat{t}_{\text{-}T_{id}}\hat{cl}, r_{\phi_{\mathcal{S}}}, \phi_F) = (v_1, l, v_2)\end{array}}{\texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, \{r_{\phi_{\mathcal{S}}}\} \cup S_{r_{\phi_{\mathcal{S}}}}, \phi_F) =}\,(\rho_{r_{\phi_{\mathcal{S}}}}\ \text{IC - }\textsc{Match})$$
$$\{(\texttt{AddTableLabel}(\texttt{RemoveTableLabels}(r_{\phi_{\mathcal{S}}}), \hat{t}), l, v_2)\} \cup \phi_e$$

Figure A.5: Theorem 2 Supporting Functions - `ExtColEdges`

$$\frac{}{\texttt{AttachEdge}(\emptyset) = \emptyset}\,(\textsc{Base})$$

$$\frac{\begin{array}{c}\phi_e = \texttt{AttachEdge}(er_{\phi_{\mathcal{S}}s}) \quad e = (v_1, l, v_2) \\ \texttt{contains}(r_{\phi_{\mathcal{S}}}, v_1) = \texttt{false}\end{array}}{\texttt{AttachEdge}(\{(e, r_{\phi_{\mathcal{S}}})\} \cup er_{\phi_{\mathcal{S}}s}) = \phi_e}\,(\text{IC - }\textsc{NoContains})$$

$$\frac{\phi_e = \texttt{AttachEdge}(er_{\phi_{\mathcal{S}}s}) \quad e = (v_1, l, v_2) \quad \texttt{contains}(r_{\phi_{\mathcal{S}}}, v_1) = \texttt{true}}{\texttt{AttachEdge}(\{(e, r_{\phi_{\mathcal{S}}})\} \cup er_{\phi_{\mathcal{S}}s}) = \{(r_{\phi_{\mathcal{S}}}, l, v_2)\} \cup \phi_e}\,(\text{IC - }\textsc{Contains})$$

Figure A.6: Theorem 2 Supporting Functions - `AttachEdge`

$$\frac{}{\texttt{IsTable}(\cdot, \hat{t}) = \texttt{false}}\,(\textsc{IsTable Base})$$

$$\frac{\texttt{IsTable}(\rho_{\mathbb{TR}}, \hat{t}) = \texttt{TruthValue} \quad \mathbb{TR} = \hat{dt} \quad \hat{t} = \hat{t}_{al}}{\texttt{IsTable}(\mathbb{TR} :: \rho_{\mathbb{TR}}, \hat{t}) = \texttt{false}}\,(\textsc{IsTable SQMatch})$$

$$\frac{\texttt{IsTable}(\rho_{\mathbb{TR}}, \hat{t}) = \texttt{TruthValue} \quad \mathbb{TR} = \hat{dt} \quad \hat{t} \neq \hat{t}_{al}}{\texttt{IsTable}(\mathbb{TR} :: \rho_{\mathbb{TR}}, \hat{t}) = \texttt{TruthValue}}\,(\textsc{IsTable SQNoMatch})$$

$$\frac{\texttt{IsTable}(\rho_{\mathbb{TR}}, \hat{t}) = \texttt{TruthValue} \quad \mathbb{TR} = \hat{t}' \quad \hat{t} = \hat{t}'}{\texttt{IsTable}(\mathbb{TR} :: \rho_{\mathbb{TR}}, \hat{t}) = \texttt{true}}\,(\textsc{IsTable TMatch})$$

$$\frac{\texttt{IsTable}(\rho_{\mathbb{TR}}, \hat{t}) = \texttt{TruthValue} \quad \mathbb{TR} = \hat{t}' \quad \hat{t} \neq \hat{t}'}{\texttt{IsTable}(\mathbb{TR} :: \rho_{\mathbb{TR}}, \hat{t}) = \texttt{TruthValue}}\,(\textsc{IsTable TNoMatch})$$

Figure A.7: Theorem 2 Supporting Functions - `IsTable`

$$\frac{s\dot{p}sl = ?\dot{v}\ \texttt{AS}\ \dot{v}_{al} \quad \hat{t}_{alsqs\dot{p}q} = \dot{v}_{al}.\texttt{substring}(0, \texttt{indexOf}(\_T_{id}) - 1)}{\texttt{GetSubQueryAlias}(s\dot{p}q_{sq} = (s\dot{p}sl :: \rho_{\text{SpSL}}, \rho_{\text{SpW}})) = \hat{t}_{alsqs\dot{p}q}}$$

Figure A.8: Theorem 2 Supporting Functions - `GetSubQueryAlias`

$$\mathcal{S}_{F_{pk}}[\![\hat{t}]\!]\phi = \{v_1 \mid (v_1, l, v_2) \in \phi,\ l = \texttt{type},\ v_2 = \hat{t}\}$$

$$\begin{aligned}\mathcal{S}_T[\![\hat{t}]\!]\phi = &\{(v_1, \hat{t}\_T_{id}l, v_2) \mid (v_1, l, v_2) \in \phi,\ v_1 \in \mathcal{S}_{F_{pk}}[\![\hat{t}]\!]\phi,\\ &l \neq \texttt{type},\ \texttt{1NPrefixExists}(l) = \texttt{false}\}\\ &\cup\ \{(v_2, \hat{t}\_T_{id}l, v_1) \mid (v_1, l, v_2) \in \phi,\ v_2 \in \mathcal{S}_{F_{pk}}[\![\hat{t}]\!]\phi,\\ &l \neq \texttt{type},\ \texttt{1NPrefixExists}(l) = \texttt{true}\}\\ &\cup\ \{(v_1, \hat{t}\_T_{id}l, v_1) \mid v_1 \in \mathcal{S}_{F_{pk}}[\![\hat{t}]\!]\phi,\\ &\texttt{contains}(\hat{t}, \_T_{id}) = \texttt{false}\}\end{aligned}$$

$$\begin{aligned}\texttt{JoinGraphs}(\phi_1, \phi_2) = &\{(v_1\_v_1', l, v_2), (v_1\_v_1', l', v_2') \mid (v_1, l, v_2) \in \phi_1,\\ &(v_1', l', v_2') \in \phi_2\}\end{aligned}$$

$$\texttt{CreateEdges}(d, S_{r_{\phi_\mathcal{S}}}) = \{(r_{\phi_\mathcal{S}}, d, d \mid r_{\phi_\mathcal{S}} \in S_{r_{\phi_\mathcal{S}}}\}$$

$$\texttt{1NPrefixExists}(l) = \{l.\texttt{substring}(0,3) == \texttt{1N\_}\}$$

$$\texttt{contains}(\hat{t}, s) = \{\hat{t}.\texttt{indexOf}(s) \neq -1\}$$

$$\begin{aligned}\texttt{ReplaceLabel}(\phi, l_{old}, l_{new}) = &\{(v_1, l_{new}, v_2) \mid (v_1, l, v_2) \in \phi, l = l_{old}\}\\ &\cup\ \{(v_1, l, v_2) \mid (v_1, l, v_2) \in \phi,\ l \neq l_{old}\}\end{aligned}$$

$$\texttt{GetRootNodes}(\phi) = \{v_1 \mid (v_1, l, v_2) \in \phi\}$$

$$\texttt{GetColEdge}(\hat{t}\_T_{id}\hat{cl}, r_{\phi_\mathcal{S}}, \phi) = \{(v_1, l, v_2) \mid (v_1, l, v_2) \in \phi,\ v_1 = r_{\phi_\mathcal{S}},\ l = \hat{t}\_T_{id}\hat{cl}\}$$

$$\texttt{TableExists}(r, \hat{t}) = \{r.\texttt{indexOf}(\_T_{id}\hat{t}\_T_{id}) \neq -1\}$$

$$\texttt{ElementExists}(l', v_2', \phi) = \{\{(v_1, l, v_2) \mid (v_1, l, v_2) \in \phi,\ l = l',\ v_2 = v_2'\} \neq \{\}\}$$

Figure A.9: Theorem 2 - Other Support Functions (a)

$$\texttt{AddTableLabel}(\phi, \hat{t}) = \{\hat{t}_{\_T_{id}} v_1 {}_{\_T_{id}} \hat{t} \mid v_1 \in \phi\}$$

$$\texttt{RemoveTableLabels}(\phi) = \{v_1.\texttt{replaceAll}({}_{\_T_{id}}, \text{``''}) \mid (v_1, l, v_2) \in \phi\}$$

$$\texttt{GetRoot}(\phi, r) = \{v_1 \mid (v_1, l, v_2) \in \phi,\ \{(v_1'', l', v_1) \in \phi\} = \emptyset\}$$

$$\texttt{RemoveRoot}(\phi, r) = \{(v_1, l, v_2) \mid (v_1, l, v_2) \in \phi,\ v_1 \neq r\}$$

$$\texttt{AddRoot}(\phi, r) = \{(r, \texttt{rowlabel}, v_2) \mid (v_2, l, v_2') \in \phi\} \cup \phi$$

$$\texttt{AddTable}(r, \hat{t}) = \{r.\texttt{concat}({}_{\_T_{id}} \hat{t}_{\_T_{id}}\}$$

$$\texttt{AddEdge}(\hat{s}, l, \phi) = \{(v_1, l, \hat{s}) \mid (v_1, l', v_2) \in \phi,\ v_1 \neq \texttt{GetRoot}(\phi)\} \cup \phi$$

$$\texttt{RemoveFirstChar}(\dot{v}) = \{\dot{v}.\texttt{substring}(1)\}$$

Figure A.10: Theorem 2 - Other Support Functions (b)

APPENDIX B

SEMANTICS-PRESERVING SCHEMA MAPPING PROOFS

The denotational semantics of a relational schema and the proof rules comprising the transformation of the canonical form of an RDF Graph into an equivalent relational schema are listed in this appendix along with proofs of Theorem 1 and its five associated lemmas.

$$\frac{}{\mathcal{E}[\![\emptyset]\!] = \emptyset}\text{(EmptySchema)}$$

$$\frac{\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_r) \quad k = (k_{id}, \kappa_{pk})}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d, \texttt{type}, t_{id})\} \cup \phi)}\text{(KCRT)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_r) \quad k = (k_{id}, k_t) \quad k_t \neq \kappa_{pk} \\ \texttt{1NPrefixExists}(k_{id}) = \texttt{false} \quad \mathbb{R}(t, (t_{id}\_\texttt{type}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d', k_{id}, d)\} \cup \phi}\text{(NKCN1NRT)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_r) \quad k = (k_{id}, k_t) \quad k_t \neq \kappa_{pk} \\ \texttt{1NPrefixExists}(k_{id}) = \texttt{true} \quad \mathbb{R}(t, (t_{id}\_\texttt{type}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d, k_{id}, d')\} \cup \phi}\text{(NKC1NRT)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{mva}) \quad t_{id} = t'_{id}\_p \quad k = (k_{id}, \kappa_{pk}) \\ k_{id} = t'_{id}\_\texttt{type} \quad \mathbb{R}(t, (k'_{id}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d\_d', \texttt{type}, t_{id}), (d\_d', k_{id}, d)\} \cup \phi}\text{(MVATTC)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{mva}) \quad t_{id} = t'_{id}\_p \\ k = (k_{id}, \kappa_{pk}) \quad k_{id} \neq t'_{id}\_\texttt{type} \quad \mathbb{R}(t, (t'_{id}\_\texttt{type}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d'\_d, k_{id}, d)\} \cup \phi}\text{(MVATNTC)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{nm}) \quad t_{id} = t_{id1}\_t_{id2} \quad k = (k_{id}, \kappa_{pk}) \\ k_{id} = t_{id1}\_\texttt{type} \quad \mathbb{R}(t, (k'_{id}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d\_d', \texttt{type}, t_{id}), (d\_d', k_{id}, d)\} \cup \phi}\text{(NMTTC)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{nm}) \quad t_{id} = t_{id1}\_t_{id2} \\ k = (k_{id}, \kappa_{pk}) \quad k_{id} \neq t_{id1}\_\texttt{type} \quad \mathbb{R}(t, (t_{id1}\_\texttt{type}, \kappa_{pk}), w) = d'\end{array}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d'\_d, k_{id}, d)\} \cup \phi}\text{(NMTNTC)}$$

Figure B.1: Relational Schema Denotational Semantics - Non-Temporary Tables

$$\frac{\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{tmp}) \quad k = (k_{id}, k_t) \quad \texttt{1NPrefixExists}(k_{id}) = \texttt{false}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(t_{id}, k_{id}, d)\} \cup \phi}(\textsc{TmpTN1N})$$

$$\frac{\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{tmp}) \quad k = (k_{id}, k_t) \quad \texttt{1NPrefixExists}(k_{id}) = \texttt{true}}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(d, k_{id}, t_{id})\} \cup \phi}(\textsc{TmpT1N})$$

$$\frac{\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{tmp_{mva}}) \quad k = (k_{id}, k_t)}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(t_{id}, k_{id}, d)\} \cup \phi}(\textsc{MVATmpT})$$

$$\frac{\mathcal{E}[\![\mathbb{R}]\!] = \phi \quad t = (t_{id}, \tau_{tmp_{nm}}) \quad k = (k_{id}, k_t)}{\mathcal{E}[\![\mathbb{R}(t, k, w) \mapsto d]\!] = \{(t_{id}, k_{id}, d)\} \cup \phi}(\textsc{NMTmpT})$$

Figure B.2: Relational Schema Denotational Semantics - Temporary Tables

$$\frac{}{f(\phi = \cdot) = \cdot}\,(\textsc{BaseEmpty})$$

$$\frac{f(\phi') = \mathbb{R}' \quad e = (r, \mathtt{type}, c) \quad w = \mathtt{GetNewRegTblRow}(c) + 1}{f(\phi = \phi' \uplus e) = (((((\mathbb{R}' - \mathtt{GetTmpTblLitCols}(\mathbb{R}', r)) - \mathtt{GetTmpRColsWithP}(\mathbb{R}', r))}\,(\text{IT }\,)$$

$$-\mathtt{GetTmpRTblsWithrObj}(\mathbb{R}', r)) - \mathtt{GetValidTmpMVATbls}(\mathbb{R}', r) - \mathtt{GetValidTmpNMTbls}(\mathbb{R}', r))$$

$$\uplus\mathtt{AddTmpLColstoRegTbl}(\mathbb{R}', c, r) \uplus \mathtt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}', r)$$

$$\uplus\mathtt{AddTmpRColsWithPtoRegTbl}(\mathbb{R}', c, r) \uplus \mathtt{UpdRRegTblsWithrObj}(\mathbb{R}', r)$$

$$\uplus\mathtt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}', r)$$

$$\frac{f(\phi') = \mathbb{R}' \quad e = (r, p, l) \quad \mathtt{GetRegTblID}(\mathbb{R}', r) = \{\}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \mathtt{UpdateTmpCell}(\mathbb{R}', (r, \tau_{tmp}), (p, \kappa_r), l)}\,(\text{LT }\textsc{NoClass})$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r, p, l) \quad \mathtt{GetRegTblID}(\mathbb{R}', r) \neq \{\} \\ \mathtt{1NPrefixExists}(p) = \mathtt{false} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \mathtt{UpdateCell}(\mathbb{R}', r, p, \kappa_r, l)}\,(\text{LT }\textsc{WithClass})$$

Figure B.3: Translation of Instance Triples and Non-MVA Literal Triples

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r,p,r') \quad \texttt{GetRegTblID}(\mathbb{R}',r) = \{\} \\ \texttt{1NPrefixExists}(p) = \texttt{false} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}', (r, \tau_{tmp}), (p, \kappa_{fk}), r')}\text{(RTNoCls(a)N1N)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r,p,r') \quad \texttt{GetRegTblID}(\mathbb{R}',r) = \{\} \\ \texttt{1NPrefixExists}(p) = \texttt{true} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}', (r', \tau_{tmp}), (p, \kappa_{fk}), r)}\text{(RTNoCls(a)1N)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r,p,r') \quad \texttt{GetRegTblID}(\mathbb{R}',r) \neq \{\} \\ \texttt{GetRegTblID}(\mathbb{R}',r') = \{\} \quad \texttt{1NPrefixExists}(p) = \texttt{false} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}', (r, \tau_{tmp}), (p, \kappa_{fk}), r')}\text{(RTNoCls(b)N1N)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r,p,r') \quad \texttt{GetRegTblID}(\mathbb{R}',r) \neq \{\} \\ \texttt{GetRegTblID}(\mathbb{R}',r') = \{\} \quad \texttt{1NPrefixExists}(p) = \texttt{true} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}', (r', \tau_{tmp}), (p, \kappa_{fk}), r)}\text{(RTNoCls(b)1N)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r,p,r') \quad \texttt{GetRegTblID}(\mathbb{R}',r) \neq \{\} \\ \texttt{GetRegTblID}(\mathbb{R}',r') \neq \{\} \quad \texttt{1NPrefixExists}(p) = \texttt{false} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateCell}(\mathbb{R}', r, p, \kappa_{fk}, r')}\text{(RTClsN1N)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r,p,r') \quad \texttt{GetRegTblID}(\mathbb{R}',r) \neq \{\} \\ \texttt{GetRegTblID}(\mathbb{R}',r') \neq \{\} \quad \texttt{1NPrefixExists}(p) = \texttt{true} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateCell}(\mathbb{R}', r', p, \kappa_{fk}, r)}\text{(RTCls1N)}$$

Figure B.4: Translation of Non-N:M Resource Triples

$$\frac{f(\phi') = \mathbb{R}' \quad e = (r\_l, \texttt{type}, c) \quad \texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) < 2}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), c)}\text{(MVAI CC<2)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_l, \texttt{type}, c) \quad \texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) = 2 \\ \texttt{GetRegTblID}(\mathbb{R}', r) = \{\} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), c)}\text{(MVAI NoRC)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_l, \texttt{type}, c) \quad \texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) = 2 \\ \texttt{GetRegTblID}(\mathbb{R}', r) \neq \{\} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_l, \tau_{tmp_{mva}}))}\text{(MVAI AllOK)}$$

$$\uplus \texttt{AddMVATblRow}_{\texttt{IP}}(\mathbb{R}', r\_l, c)$$

$$\frac{f(\phi') = \mathbb{R}' \quad e = (r\_l, p, l) \quad \texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) < 2}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), l)}\text{(MVALT CC<2)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_l, p, l) \quad \texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) = 2 \\ \texttt{GetRegTblID}(\mathbb{R}, r) = \{\} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), l)}\text{(MVALT NoRC)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_l, p, l) \quad \texttt{GetRegTblID}(\mathbb{R}, r) \neq \{\} \\ \texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) = 2 \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_l, \tau_{tmp_{mva}}))}\text{(MVALT AllOK)}$$

$$\uplus \texttt{AddMVATblRow}_{\texttt{LP}}(\mathbb{R}', r\_l, p, l)$$

Figure B.5: Multi-Valued (MVA) Literal Triples Translation (a)

$$\frac{f(\phi') = \mathbb{R}' \quad e = (r\_l, p, r) \quad \texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) < 2}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), r)} \text{(MVART CC<2)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_l, p, r) \quad \texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) = 2 \\ \texttt{GetRegTblID}(\mathbb{R}, r) = \{\} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), r)} \text{(MVART NoRC)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_l, p, r) \quad \texttt{GetRegTblID}(\mathbb{R}, r) \neq \{\} \\ \texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) = 2 \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_l, \tau_{tmp_{mva}}))} \text{(MVART AllOK)}$$
$$\uplus \texttt{AddMVATblRow}_{\texttt{RP}}(\mathbb{R}', r\_l, p, r)$$

Figure B.6: Multi-Valued (MVA) Literal Triples Translation (b)

Theorem 1 $\mathcal{E}[\![f(\phi) = \phi]\!]$

Proof.

Proof is by induction on the edge set of graph ϕ. Please note that π_i, used in some of the proofs that follow, is the project operator that is used to extract the i^th component from the tuple element.

Base Case: Assume $\phi = \emptyset$. By Rule BASEEMPTY, $f(\phi) = \emptyset$ and by Rule EMPTYSCHEMA, $\mathcal{E}[\![\emptyset]\!] = \emptyset$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \phi$.

Inductive Hypothesis: Assume that for every graph ϕ' that has strictly fewer edges than ϕ, $\mathcal{E}[\![f(\phi')]\!] = \phi'$.

There is one inductive proof case for each case in the recursive definition of f:

Case (IT) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r, \texttt{type}, c)$, and $w = \texttt{GetNewRegTblRow}(c) + 1$.

Proof follows from Lemmas 1-5.

$$\frac{f(\phi') = \mathbb{R}' \quad e = (r\_r', \texttt{type}, c) \quad \texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') < 2}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), c)} \text{(NMI CC<2)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_r', \texttt{type}, c) \quad \texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') = 2 \\ \mid \texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r') \mid \neq 2 \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), c)} \text{(NMI NoRC)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_r', \texttt{type}, c) \quad \texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') = 2 \\ \mid \texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r') \mid = 2 \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_r', \tau_{tmp_{nm}}))} \text{(NMI AllOK)}$$

$$\uplus \texttt{AddNMTblRow}_{\texttt{IP}}(\mathbb{R}', r\_r', c)$$

$$\frac{f(\phi') = \mathbb{R}' \quad e = (r\_r', p, r'') \quad \texttt{GetRegTblID}(\mathbb{R}, r'') = \{\}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_r', \tau_{tmp_{nm}}), (p, \kappa_{fk}), r'')} \text{(NMRT NoOC)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_r', p, r'') \quad \texttt{GetRegTblID}(\mathbb{R}, r'') \neq \{\} \\ \texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') < 2 \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_r', \tau_{tmp_{nm}}), (p, \kappa_{fk}), r'')} \text{(NMRT CC<2)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_r', p, r'') \quad \texttt{GetRegTblID}(\mathbb{R}, r'') \neq \{\} \quad \texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') = 2 \\ \texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r') = \{\} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_r', \tau_{tmp_{nm}}), (p, \kappa_{fk}), r'')} \text{(NMRT NoSOC)}$$

$$\frac{\begin{array}{c} f(\phi') = \mathbb{R}' \quad e = (r\_r', p, r'') \quad \texttt{GetRegTblID}(\mathbb{R}, r'') \neq \{\} \\ \texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') = 2 \\ \texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r') \neq \{\} \end{array}}{f(\phi = \phi' \uplus e) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_r', \tau_{tmp_{nm}}))} \text{(NMRT AllOK)}$$

$$\uplus \texttt{AddNMTblRow}_{\texttt{RP}}(r\_r', p, r'')$$

Figure B.7: Translation of Resource Triples sharing a Many-to-Many (N:M) Relationship

Case (LT NoClass) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r, p, l)$, and $\texttt{GetRegTblID}(\mathbb{R}', r) = \{\}$. By Rule (LT NoClass), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}', (r, \tau_{tmp}), (p, \kappa_r), l)$. By definition of $\texttt{UpdateTmpCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r, \tau_{tmp}), (p, \kappa_r), w+1), l)$ for some unoccupied row $w + 1$. From the Canonical Form of RDF Graphs, it can be observed that $\texttt{1NPrefixExists}(p) = \texttt{false}$ for all literal triples. Therefore, by Rule (TmpTN1N), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r, p, l)\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (LT WithClass) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r, p, l)$, $\texttt{GetRegTblID}(\mathbb{R}', r) \neq \{\}$, and $\texttt{1NPrefixExists}(p) = \texttt{false}$. By Rule (LT WithClass), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateCell}(\mathbb{R}', r, p, \kappa_r, l)$. By definition of $\texttt{UpdateCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((t_{id}, \tau_r), (p, \kappa_r), w), l)$ for t_{id} and w such that $\mathbb{R}'((t_{id}, \tau_r), (t_{id}\_\texttt{type}, \kappa_{pk}), w) = r$. By Rule (NKCN1NRT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r, p, l)\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (RTNoCls(a)N1N) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r, p, r')$, $\texttt{GetRegTblID}(\mathbb{R}', r) = \{\}$, and $\texttt{1NPrefixExists}(p) = \texttt{false}$. By Rule (RTNoCls(a)N1N), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r, \tau_{tmp}), (p, \kappa_{fk}), r')$. By definition of $\texttt{UpdateTmpCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r, \tau_{tmp}), (p, \kappa_{fk}), w+1), r')$ for some uncoocupied row $w+1$. By Rule (TmpTN1N), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r, p, r')\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (RTNoCls(a)1N) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r, p, r')$, $\texttt{GetRegTblID}(\mathbb{R}', r) = \{\}$, and $\texttt{1NPrefixExists}(p) = \texttt{true}$. By Rule (RTNoCls(a)1N), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r', \tau_{tmp}), (p, \kappa_{fk}), r)$. By definition of $\texttt{UpdateTmpCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r', \tau_{tmp}), (p, \kappa_{fk}), w+1), r)$ for some uncoocupied row $w+1$. By Rule (TmpT1N), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r, p, r')\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (RTNoCls(b)N1N) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r, p, r')$, $\texttt{GetRegTblID}(\mathbb{R}', r) \neq \{\}$, $\texttt{GetRegTblID}(\mathbb{R}', r') = \{\}$, and $\texttt{1NPrefixExists}(p) = \texttt{false}$. By Rule (RTNoCls(b)N1N), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r, \tau_{tmp}), (p, \kappa_{fk}), r')$. By definition of UpdateTmpCell, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r, \tau_{tmp}), (p, \kappa_{fk}), w + 1), r')$ for some uncoocupied row $w + 1$. By Rule (TmpTN1N), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r, p, r')\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (RTNoCls(b)1N) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r, p, r')$, $\texttt{GetRegTblID}(\mathbb{R}', r) \neq \{\}$, $\texttt{GetRegTblID}(\mathbb{R}', r') = \{\}$, and $\texttt{1NPrefixExists}(p) = \texttt{true}$. By Rule (RTNoCls(b)1N), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r', \tau_{tmp}), (p, \kappa_{fk}), r)$. By definition of UpdateTmpCell, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r', \tau_{tmp}), (p, \kappa_{fk}), w + 1), r)$ for some uncoocupied row $w + 1$. By Rule (TmpT1N), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r, p, r')\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (RTClsN1N) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r, p, r')$, $\texttt{GetRegTblID}(\mathbb{R}', r') \neq \{\}$, $\texttt{GetRegTblID}(\mathbb{R}', r') = \{\}$, and $\texttt{1NPrefixExists}(p) = \texttt{false}$. By Rule (RTCls-N1N), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateCell}(\mathbb{R}', r, p, \kappa_{fk}, r')$. By definition of UpdateCell, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((t_{id}, \tau_r), (p, \kappa_{fk}), w), r')$ for t_{id} and w such that $\mathbb{R}'((t_{id}, \tau_r), (t_{id}\_\texttt{type}, \kappa_{pk}), w) = r$. By Rule (NKCN1NRT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r, p, r')\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (RTCls1N) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r, p, r')$, $\texttt{GetRegTblID}(\mathbb{R}', r') \neq \{\}$, $\texttt{GetRegTblID}(\mathbb{R}', r') = \{\}$, and $\texttt{1NPrefixExists}(p) = \texttt{true}$. By Rule (RTCls1N), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateCell}(\mathbb{R}', r', p, \kappa_{fk}, r)$. By definition of UpdateCell, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((t_{id}, \tau_r), (p, \kappa_{fk}), w), r)$ for t_{id} and w such that $\mathbb{R}'((t_{id}, \tau_r), (t_{id}\_\texttt{type}, \kappa_{pk}), w) = r'$. By Rule (NKC1NRT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r, p, r')\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (MVAI CC<2) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_l, \texttt{type}, c)$, and $\texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) < 2$. By Rule (MVAI CC<2), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), c)$. By definition of $\texttt{UpdateTmpCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1), c)$. By Rule (MVATmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_l, \texttt{type}, c)\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (MVAI NoRC) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_l, \texttt{type}, c)$, and $\texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) = 2$, and $\texttt{GetRegTblID}(\mathbb{R}', r) = \{\}$. By Rule (MVAI NoRC), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), c)$. By definition of $\texttt{UpdateTmpCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1), c)$. By Rule (MVATmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_l, \texttt{type}, c)\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (MVAI AllOK) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_l, \texttt{type}, c)$, $\texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) = 2$, and $\texttt{GetRegTblID}(\mathbb{R}', r) \neq \{\}$. By Rule (MVAI AllOK), $f(\phi) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_l, \tau_{tmp_{mva}})) \uplus \texttt{AddMVATblRow}_{\texttt{IP}}(\mathbb{R}', r\_l, c)$. By definition of $\texttt{GetTmpTblRow}$ and $\texttt{AddMVATblRow}_{\texttt{IP}}$, $f(\phi) = \mathbb{R}' - \{d\} \uplus \{e\}$ where $d = \{(((r\_l, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1), r), (((r\_l, \tau_{tmp_{mva}}), (k'_{id}, \kappa_{pk}), 1), l)\}$, and $e = \{(((c, \tau_{mva}), (k_{id}, \kappa_{pk}), w), r), (((c, \tau_{mva}), (k'_{id}, \kappa_{pk}), w), l)$ for some unoccupied row, w. By Rule (MVATmpT), (MVATTC), and (MVATNTC), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' - \{d\} \uplus \{e\}]\!] = \mathcal{E}[\![\mathbb{R}']\!] - \{(r\_l, k_{id}, r), (r\_l, k'_{id}, l)\} \cup \{(r\_l, \texttt{type}, c), (r\_l, k_{id}, r), (r\_l, k'_{id}, l)\} = \mathcal{E}[\![\mathbb{R}']\!] \cup \{(r\_l, \texttt{type}, c)\} = \mathcal{E}[\![f(\phi')]\!] \cup \{e\}$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \phi' \cup \{e\} = \phi$.

Case (MVALT CC<2) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_l, p, l)$, and $\texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) < 2$. By Rule (MVALT CC<2), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), l)$. By definition of $\texttt{UpdateTmpCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where

$e = (((r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), 1), l)$. By Rule (MVATmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_l, p, l)\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (MVALT NoRC) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_l, p, l)$, and `GetTmpMVAColCnt`$(\mathbb{R}', r\_l) = 2$, and `GetRegTblID`$(\mathbb{R}', r) = \{\}$. By Rule (MVALT NoRC), $f(\phi) = \mathbb{R}' \uplus$ `UpdateTmpCell`$(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), l)$. By definition of `UpdateTmpCell`, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), 1), l)$. By Rule (MVATmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_l, p, l)\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (MVALT AllOK) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_l, p, l)$, and `GetTmpMVAColCnt`$(\mathbb{R}', r\_l) = 2$, and `GetRegTblID`$(\mathbb{R}', r) \neq \{\}$. By Rule (MVALT AllOK), $f(\phi) = \mathbb{R}' -$ `GetTmpTblRow`$(\mathbb{R}', (r\_l, \tau_{tmp_{mva}})) \uplus$ `AddMVATblRow`$_{\texttt{LP}}(\mathbb{R}', r\_l, p, l)$. By definition of `GetTmpTblRow` and `AddMVATblRow`$_{\texttt{LP}}$, $f(\phi) = \mathbb{R}' - \{d\} \uplus \{e\}$ where $d = \{(((r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1), c), (((r\_l, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1), r)\}$, and $e = \{(((c, \tau_{mva}), (k_{id}, \kappa_{pk}), w), r), (((c, \tau_{mva}), (p, \kappa_{pk}), w), l)$ for some unoccupied row, w. By Rule (MVATmpT), (MVATTC), and (MVATNTC), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' - \{d\} \uplus \{e\}]\!] = \mathcal{E}[\![\mathbb{R}']\!] - \{(r\_l, \texttt{type}, c), (r\_l, k_{id}, r)\} \cup \{(r\_l, \texttt{type}, c), (r\_l, k_{id}, r), (r\_l, p, l)\} = \mathcal{E}[\![\mathbb{R}']\!] \cup \{(r\_l, p, l)\} = \mathcal{E}[\![f(\phi')]\!] \cup \{e\}$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \phi' \cup \{e\} = \phi$.

Case (MVART CC<2) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_l, p, r)$, and `GetTmpMVAColCnt`$(\mathbb{R}', r\_l) < 2$. By Rule (MVART CC<2), $f(\phi) = \mathbb{R}' \uplus$ `UpdateTmpCell`$(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), r)$. By definition of `UpdateTmpCell`, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), 1), r)$. By Rule (MVATmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_l, p, r)\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (MVART NoRC) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_l, p, r)$, and $\texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) = 2$, and $\texttt{GetRegTblID}(\mathbb{R}', r) = \{\}$. By Rule (MVART NoRC), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), l)$. By definition of $\texttt{UpdateTmpCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r\_l, \tau_{tmp_{mva}}), (p, \kappa_{pk}), 1), r)$. By Rule (MVATmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_l, p, r)\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (MVART AllOK) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_l, p, l)$, and $\texttt{GetTmpMVAColCnt}(\mathbb{R}', r\_l) = 2$, and $\texttt{GetRegTblID}(\mathbb{R}', r) \neq \{\}$. By Rule (MVART AllOK), $f(\phi) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_l, \tau_{tmp_{mva}})) \uplus \texttt{AddMVATblRow}_{\texttt{LP}}(\mathbb{R}', r\_l, p, r)$. By definition of $\texttt{GetTmpTblRow}$ and $\texttt{AddMVATblRow}_{\texttt{LP}}$, $f(\phi) = \mathbb{R}' - \{d\} \uplus \{e\}$ where $d = \{(((r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1), c), (((r\_l, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1), l)\}$, and $e = \{(((c, \tau_{mva}), (p, \kappa_{pk}), w), r), (((c, \tau_{mva}), (k_{id}, \kappa_{pk}), w), l)$ for some unoccupied row, w. By Rule (MVATmpT), (MVATTC), and (MVATNTC), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' - \{d\} \uplus \{e\}]\!] = \mathcal{E}[\![\mathbb{R}']\!] - \{(r\_l, \texttt{type}, c), (r\_l, k_{id}, l)\} \cup \{(r\_l, \texttt{type}, c), (r\_l, p, r), (r\_l, k_{id}, l)\} = \mathcal{E}[\![\mathbb{R}']\!] \cup \{(r\_l, p, r)\} = \mathcal{E}[\![f(\phi')]\!] \cup \{e\}$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \phi' \cup \{e\} = \phi$.

Case (NMI CC<2) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_r', \texttt{type}, c)$, and $\texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') < 2$. By Rule (NMI CC<2), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), c)$. By definition of $\texttt{UpdateTmpCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), c)$. By Rule (NMTmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_r', \texttt{type}, c)\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (NMI NoRC) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_r', \texttt{type}, c)$, $\texttt{GetTmpNMColCnt}(\mathbb{R}', r\_l) = 2$, and $\mid \texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r') \mid \neq 2$. By Rule (NMI NoRC), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), c)$. By definition of $\texttt{UpdateTmpCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1),$

c). By Rule (NMTmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_r', \texttt{type}, c)\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (NMI AllOK) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_r', \texttt{type}, c)$, and $\texttt{GetTmpNMColCnt}(\mathbb{R}', r\_l) = 2$, and $\mid \texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r') \mid= 2$. By Rule (NMI AllOK), $f(\phi) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_r', \tau_{tmp_{nm}})) \uplus \texttt{AddNMTblRow}_{\texttt{IP}}(\mathbb{R}', r\_r', c)$. By definition of GetTmpTblRow and $\texttt{AddNMTblRow}_{\texttt{IP}}$, $f(\phi) = \mathbb{R}' - \{d\} \uplus \{e\}$ where $d = \{(((r\_r', \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((r\_r', \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r')\}$, and $e = \{(((c, \tau_{nm}), (k_{id}, \kappa_{pk}), w), r), (((c, \tau_{nm}), (k'_{id}, \kappa_{pk}), w), r')\}$ for some unoccupied row, w. By Rule (NMTmpT), (NMTTC), and (NMTNTC), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' - \{d\} \uplus \{e\}]\!] = \mathcal{E}[\![\mathbb{R}']\!] - \{(r\_r', k_{id}, r), (r\_r', k'_{id}, r')\} \cup \{(r\_r', \texttt{type}, c), (r\_r', k_{id}, r), (r\_r', k'_{id}, r')\} = \mathcal{E}[\![\mathbb{R}']\!] \cup \{(r\_r', \texttt{type}, c)\} = \mathcal{E}[\![f(\phi')]\!] \cup \{e\}$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \phi' \cup \{e\} = \phi$.

Case (NMRT NoOC) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_r', p, r'')$, and $\texttt{GetRegTblID}(\mathbb{R}', r'') = \{\}$. By Rule (NMRT NoOC), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_r', \tau_{tmp_{nm}}), (p, \kappa_{fk}), r'')$. By definition of UpdateTmpCell, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r\_r', \tau_{tmp_{nm}}), (p, \kappa_{fk}), 1), r'')$. By Rule (NMTmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_r', p, r'')\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (NMRT CC<2) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_r', p, r'')$, $\texttt{GetRegTblID}(\mathbb{R}', r'') \neq \{\}$, and $\texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') < 2$. By Rule (NMRT CC<2), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_r', \tau_{tmp_{nm}}), (p, \kappa_{fk}), r'')$. By definition of UpdateTmpCell, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r\_r', \tau_{tmp_{nm}}), (p, \kappa_{fk}), 1), r'')$. By Rule (NMTmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_r', p, r'')\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (NMRT NoSOC) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_r', p, r'')$, $\texttt{GetRegTblID}(\mathbb{R}', r'') \neq \{\}$, and $\texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') = 2$, and $\texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r') = \{\}$. By Rule (NMRT NoSOC), $f(\phi) = \mathbb{R}' \uplus \texttt{UpdateTmpCell}(\mathbb{R}, (r\_r', \tau_{tmp_{nm}}), (p, \kappa_{fk}), r'')$. By definition of $\texttt{UpdateTmpCell}$, $f(\phi) = \mathbb{R}' \uplus \{e\}$ where $e = (((r\_r', \tau_{tmp_{nm}}), (p, \kappa_{fk}), 1), r'')$. By Rule (NMTmpT), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' \uplus \{e\}]\!] = \{(r\_r', p, r'')\} \uplus \mathcal{E}[\![\mathbb{R}']\!] = \{e\} \uplus \mathcal{E}[\![f(\phi')]\!]$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \{e\} \uplus \phi' = \phi$.

Case (NMRT AllOK) : Assume $\phi = \phi' \uplus \{e\}$, $f(\phi') = \mathbb{R}'$, $e = (r\_r', p, r'')$, $\texttt{GetRegTblID}(\mathbb{R}', r'') \neq \{\}$, $\texttt{GetTmpNMColCnt}(\mathbb{R}', r\_r') = 2$, and $\texttt{GetClassForTmpNMTblCells}(\mathbb{R}', r\_r') \neq \{\}$. By Rule (NMRT AllOK), $f(\phi) = \mathbb{R}' - \texttt{GetTmpTblRow}(\mathbb{R}', (r\_r', \tau_{tmp_{nm}})) \uplus \texttt{AddNMTblRow}_{\texttt{RP}}(\mathbb{R}', r\_r', p, r'')$. By definition of $\texttt{GetTmpTblRow}$ and $\texttt{AddNMTblRow}_{\texttt{RP}}$, $f(\phi) = \mathbb{R}' - \{d\} \uplus \{e\}$ where $d = \{(((r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), c), (((r\_r', \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r''')\}$, and $e = \{(((c, \tau_{nm}), (k_{id}, \kappa_{pk}), w), r'''), (((c, \tau_{nm}), (p, \kappa_{pk}), w), r'')\}$ for some unoccupied row, w. By Rule (NMTmpT), (NMTTC), and (NMTNTC), $\mathcal{E}[\![f(\phi)]\!] = \mathcal{E}[\![\mathbb{R}' - \{d\} \uplus \{e\}]\!] = \mathcal{E}[\![\mathbb{R}']\!] - \{(r\_r', \texttt{type}, c), (r\_r', k_{id}, r''')\} \cup \{(r\_r', \texttt{type}, c), (r\_r', k_{id}, r'''), (r\_r', p, r'')\} = \mathcal{E}[\![\mathbb{R}']\!] \cup \{(r\_r', p, r'')\} = \mathcal{E}[\![f(\phi')]\!] \cup \{e\}$.

By inductive hypothesis, $\mathcal{E}[\![f(\phi')]\!] = \phi'$. Therefore, $\mathcal{E}[\![f(\phi)]\!] = \phi' \cup \{e\} = \phi$.

□

Lemma 1 *For every instance edge e of type $(r, \texttt{type}, c)$, and an unoccupied row, w, in the table corresponding to c in $\mathbb{R}$,*

$$\mathcal{E}[\![\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w) - \texttt{GetTmpTblLitCols}(\mathbb{R}, r)]\!] = \{e\}$$

Proof.

$\Longrightarrow$: Let $((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_r), w), \mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w')))$ be an element in the set $\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w)$. Let $\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w') =$

d. From the definition of `AddTmpLColstoRegTbl`$(\mathbb{R}, c, r, w)$, it can be observed that $((r, \tau_{tmp}), (k_{id}, \kappa_r), w') \in \mathbb{R}^{\leftarrow}$ and $d \in \mathcal{L}$. However, from the definition of `GetTmpTblLitCols`$(\mathbb{R}, r)$, it can be observed that, whenever $((r, \tau_{tmp}), (k_{id}, \kappa_r), w') \in \mathbb{R}^{\leftarrow}$ and $d \in \mathcal{L}$, $(((r, \tau_{tmp}), (k_{id}, \kappa_r), w'), d)$ is an element of the set `GetTmpTblLitCols`$($ $\mathbb{R}, r)$.

Thus, for every element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_r), w), d)) \in$ `AddTmpLColstoRegTbl`$(\mathbb{R}, c, r, w)$, there exists an element $e'' = (((r, \tau_{tmp}), (k_{id}, \kappa_r), w'),$ $d) \in$ `GetTmpTblLitCols`$(\mathbb{R}, r)$.

$\Longleftarrow$: Let $\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w') = d$ (where $d \in \mathcal{L}$) be an element in the set `GetTmpTblLitCols`$(\mathbb{R}, r)$. As can be observed from the definition of `AddTmpLColstoRegTbl`$(\mathbb{R}, c, r, w)$, the values r and $\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w')$ are added to columns `type` and k_{id}, respectively, in row w of table, c, i.e. $\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}),$ $w) = r$ and $\mathbb{R}((c, \tau_r), (k_{id}, \kappa_r), w) = d$.

Thus, for every element $e'' = (((r, \tau_{tmp}), (k_{id}, \kappa_r), w'), d) \in$ `GetTmpTblLitCols`$(\mathbb{R}, r)$, there exists an element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_r), w), d)) \in$ `AddTmpLColstoRegTbl`$(\mathbb{R}, c, r, w)$.

For any corresponding pair e' and e'', we have `AddTmpLColstoRegTbl`$(\mathbb{R}, c, r, w) -$ `GetTmpTblLitCols`$(\mathbb{R}, r) = \{\pi_1 e', \pi_2 e'\} - \{e''\} = \{(((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id},$ $\kappa_r),\ w), d)\} - \{(((r, \tau_{tmp}), (k_{id}, \kappa_r), w'), d)\}$.

From Rule (TmpTN1N), it can be observed that $\mathcal{E}[\![\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w') \mapsto d]\!] =$ $\{(r, k_{id}, d)\}$. From Rule (KCRT), it can be observed that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) \mapsto$ $r]\!] = \{(r, \texttt{type}, c)\}$ and from Rule (NKCN1NRT), it can be seen that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (k_{id}, \kappa_r), w)$ $\mapsto d]\!] = \{(r, k_{id}, d)\}$.

Thus, $\mathcal{E}[\![\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w) - \texttt{GetTmpTblLitCols}(\mathbb{R}, r)]\!] = \mathcal{E}[\![\{\pi_1 e', \pi_2 e'\} - \{e''\}]\!] =$
$\{(r, \texttt{type}, c), (r, k_{id}, d)\} - \{(r, k_{id}, d)\} = \{(r, \texttt{type}, c)\} = \{e\}$.

$\square$

Lemma 2 *For every instance edge e of type $(r, \textbf{\texttt{type}}, c)$, and an unoccupied row, w, in the table corresponding to c in $\mathbb{R}$,*

$$\mathcal{E}[\![\texttt{AddTmpRColsWithPtoRegTbl}(\mathbb{R}, c, r, w) - \texttt{GetTmpRColsWithP}(\mathbb{R}, r)]\!] = \{e\}$$

Proof.

$\Longrightarrow$:

Let $((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_{fk}), w), r'))$, (where $r' \in \mathcal{I}$), be an element in the set $\texttt{AddTmpRColsWithPtoRegTbl}(\mathbb{R}, c, r, w)$. From the definition of $\texttt{AddTmpRColsWithPtoRegTbl}(\mathbb{R}, c, r, w)$ it can be observed that $r' = \mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_r), w')$, $(((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), r') \in \mathbb{R}$, and $\texttt{GetRegTblID}(\mathbb{R}, r') \neq \{\}$. However, from the definition of $\texttt{GetTmpRColsWithP}(\mathbb{R}, r)$, it can be observed that, whenever $(((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), r') \in \mathbb{R}$, and $\texttt{GetRegTblID}(\mathbb{R}, r') \neq \{\}$, $(((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), r')$ is an element of the set $\texttt{GetTmpRColsWithP}(\mathbb{R}, r)$.

Thus, for every element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_{fk}), w), r')) \in \texttt{AddTmpRColsWithPtoRegTbl}(\mathbb{R}, c, r, w)$, there exists an element $e'' = (((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), r') \in \texttt{GetTmpRColsWithP}(\mathbb{R}, r)$.

$\Longleftarrow$:

Let $\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w') = r'$ (where $r' \in \mathcal{I}$) be an element in the set $\texttt{GetTmpRColsWithP}(\mathbb{R}, r)$. As can be observed from the definition of $\texttt{AddTmpRColsWithPtoRegTbl}(\mathbb{R}, c, r, w)$, the values r and $\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w')$ are added to columns `type` and k_{id}, respectively, in row w of table, c, i.e. $\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) = r$ and $\mathbb{R}((c, \tau_r), (k_{id}, \kappa_{fk}), w) = r'$.

Thus, for every element $e'' = (((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), r') \in \texttt{GetTmpRColsWithP}(\mathbb{R}, r)$, there exists an element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_{fk}), w), r')) \in \texttt{AddTmpRColsWithPtoRegTbl}(\mathbb{R}, c, r, w)$.

For any corresponding pair e' and e'', we have $\texttt{AddTmpRColsWithPtoRegTbl}(\mathbb{R}, c, r, w) - \texttt{GetTmpRColsWithP}(\mathbb{R}, r) = \{\pi_1 e', \pi_2 e'\} - e'' = \{(((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((c, \tau_r), (k_{id}, \kappa_{fk}), w), r')\} - \{(((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), r')\}$.

Case a: ($\texttt{1NPrefixExists}(k_{id}) = \texttt{false}$) :

From Rule (TMPTN1N), it can be observed that $\mathcal{E}[\![\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w') \mapsto r']\!] = \{(r, k_{id}, r')\}$. From Rule (KCRT), it can be observed that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) \mapsto r]\!] = \{(r, \texttt{type}, c)\}$ and from Rule (NKCN1NRT), it can be seen that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (k_{id}, \kappa_{fk}), w) \mapsto r']\!] = \{(r, k_{id}, r')\}$.

Thus, $\mathcal{E}[\![\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w) - \texttt{GetTmpTblLitCols}(\mathbb{R}, r)]\!] = \mathcal{E}[\![\{\pi_1 e', \pi_2 e'\} - \{e''\}]\!] = \{(r, \texttt{type}, c), (r, k_{id}, r')\} - \{(r, k_{id}, r')\} = \{(r, \texttt{type}, c)\} = \{e\}$.

Case b: ($\texttt{1NPrefixExists}(k_{id}) = \texttt{true}$) :

From Rule (TMPT1N), it can be observed that $\mathcal{E}[\![\mathbb{R}((r, \tau_{tmp}), (k_{id}, \kappa_{fk}), w') \mapsto r']\!] = \{(r', k_{id}, r)\}$. From Rule (KCRT), it can be observed that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) \mapsto r]\!] = \{(r, \texttt{type}, c)\}$ and from Rule (NKC1NRT), it can be seen that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (k_{id}, \kappa_{fk}), w) \mapsto r']\!] = \{(r', k_{id}, r)\}$.

Thus, $\mathcal{E}[\![\texttt{AddTmpLColstoRegTbl}(\mathbb{R}, c, r, w) - \texttt{GetTmpTblLitCols}(\mathbb{R}, r)]\!] = \mathcal{E}[\![\{\pi_1 e', \pi_2 e'\} - \{e''\}]\!] = \{(r, \texttt{type}, c), (r', k_{id}, r)\} - \{(r', k_{id}, r)\} = \{(r, \texttt{type}, c)\} = \{e\}$.

□

Lemma 3 *For every instance edge e of type $(r, \textit{\texttt{type}}, c)$, and an unoccupied row, w, in the table corresponding to c in $\mathbb{R}$,*

$$\mathcal{E}[\![\texttt{UpdRRegTblsWithrObj}(\mathbb{R}, r) - \texttt{GetTmpRTblsWithrObj}(\mathbb{R}, r)]\!] = \{e\}$$

Proof.

$\Longrightarrow$:

Let $((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{id}, \tau_r), (k_{id}, \kappa_{fk}), w''), r))$ be an element in the set $\texttt{UpdRRegTblsWithrObj}(\mathbb{R}, r, c, w)$. From the definition of $\texttt{UpdRRegTblsWithrObj}(\mathbb{R}, r, c, w)$, it can be seen that $t_{id} = \{t_{id} \mid \mathbb{R}((t_{id}, \tau_r), (t_{id}\_\texttt{type}, \kappa_{pk}), w'') = r', \mathbb{R}((r', \tau_{tmp}), (k_{id}, \kappa_{fk}), w') = r\}$. However, from the definition of $\texttt{GetTmpRTblsWithrObj}(\mathbb{R}, r)$, it can be observed that, whenever $\mathbb{R}((r', \tau_{tmp}), (k_{id}, \kappa_{fk}), w') = r$ and $\mathbb{R}((t_{id}, \tau_r), (t_{id}\_\texttt{type}, \kappa_{pk}), w'') = r'$ (from the definition of $\texttt{GetRegTblID}(\mathbb{R}, r')$), $(((r', \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), r)$ is an element of the set $\texttt{GetTmpRTblsWithrObj}(\mathbb{R}, r)$.

Thus, for every element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{id}, \tau_r), (k_{id}, \kappa_{fk}), w''), r)) \in \texttt{UpdRRegTblsWithrObj}(\mathbb{R}, r, c, w)$, there exists an element $e'' = (((r', \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), r) \in \texttt{GetTmpRTblsWithrObj}(\mathbb{R}, r)$.

$\Longleftarrow$:

Let $\mathbb{R}((r', \tau_{tmp}), (k_{id}, \kappa_{fk}), w') = r$ (where $r \in \mathcal{I}$) be an element in the set $\texttt{GetTmpRTblsWithrObj}(\mathbb{R}, r)$. As can be observed from the definition of $\texttt{UpdRRegTblsWithrObj}(\mathbb{R}, r, c, w)$, the values r and $\mathbb{R}((r', \tau_{tmp}), (k_{id}, \kappa_{fk}), w')$ are added to column $\texttt{type}$ in row w of table, c, and column k_{id} in row w'' of table, t_{id}, such that $\mathbb{R}((t_{id}, \tau_r), (t_{id}\_\texttt{type}, \kappa_{pk}), w'') = r'$, i.e. $\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) = r$ and $\mathbb{R}((t_{id}, \tau_r), (k_{id}, \kappa_{fk}), w'') = r$, where $\mathbb{R}((t_{id}, \tau_r), (t_{id}\_\texttt{type}, \kappa_{pk}), w'') = r'$.

Thus, for every element $e'' = (((r', \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), r) \in$ `GetTmpRTblsWithrObj`$(\mathbb{R}, r)$, there exists an element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{id}, \tau_r), (k_{id}, \kappa_{fk}), w''), r)) \in$ `UpdRRegTblsWithrObj`$(\mathbb{R}, r, c, w)$.

For any corresponding pair e' and e'', we have `UpdRRegTblsWithrObj`$(\mathbb{R}, r, c, w) -$ `GetTmpRTblsWithrObj`$(\mathbb{R}, r) = \{\pi_1 e', \pi_2 e'\} - \{e''\} = \{((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{id}, \tau_r), (k_{id}, \kappa_{fk}), w''), r)\} - \{(((r', \tau_{tmp}), (k_{id}, \kappa_{fk}), w'), r)\}$.

Case a: (`1NPrefixExists`$(k_{id}) =$ `false`) :

From Rule (TMPTN1N), it can be observed that $\mathcal{E}[\![\mathbb{R}((r', \tau_{tmp}), (k_{id}, \kappa_{fk}), w') \mapsto r]\!] = \{(r', k_{id}, r)\}$. From Rule (KCRT), it can be observed that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) \mapsto r]\!] = \{(r, \texttt{type}, c)\}$ and from Rule (NKCN1NRT), it can be seen that $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_r), (k_{id}, \kappa_{fk}), w'') \mapsto r]\!] = \{(r', k_{id}, r)\}$.

Thus, $\mathcal{E}[\![$`UpdRRegTblsWithrObj`$(\mathbb{R}, r, c, w) -$ `GetTmpRTblsWithrObj`$(\mathbb{R}, r)]\!] = \mathcal{E}[\![\{\pi_1 e', \pi_2 e'\} - \{e''\}]\!] = \{(r, \texttt{type}, c), (r', k_{id}, r)\} - \{(r', k_{id}, r)\} = \{(r, \texttt{type}, c)\} = \{e\}$.

Case b: (`1NPrefixExists`$(k_{id}) =$ `true`) :

From Rule (TMPT1N), it can be observed that $\mathcal{E}[\![\mathbb{R}((r', \tau_{tmp}), (k_{id}, \kappa_{fk}), w') \mapsto r]\!] = \{(r, k_{id}, r')\}$. From Rule (KCRT), it can be observed that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) \mapsto r]\!] = \{(r, \texttt{type}, c)\}$ and from Rule (NKC1NRT), it can be seen that $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_r), (k_{id}, \kappa_{fk}), w) \mapsto r]\!] = \{(r, k_{id}, r')\}$.

Thus, $\mathcal{E}[\![$`UpdRRegTblsWithrObj`$(\mathbb{R}, r, c, w) -$ `GetTmpRTblsWithrObj`$(\mathbb{R}, r)]\!] = \mathcal{E}[\![\{\pi_1 e', \pi_2 e'\} - \{e''\}]\!] = \{(r, \texttt{type}, c), (r, k_{id}, r')\} - \{(r, k_{id}, r')\} = \{(r, \texttt{type}, c)\} = \{e\}$.

$\square$

Lemma 4 *For every instance edge e of type $(r, \texttt{type}, c)$, and an unoccupied row, w, in the table corresponding to c in $\mathbb{R}$,*

$$\mathcal{E}[\![\texttt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}, r, c, w) - \texttt{GetValidTmpMVATbls}(\mathbb{R}, r)]\!] = \{e\}$$

Proof.

$\Longrightarrow$:

Let $((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), \pi_1 e'_1, \pi_2 e'_1)$, be an element in the set $\texttt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}, r, c, w)$. By definition of an MVA Table and $\texttt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}, r, c, w)$, $e'_1 = ((((t_{idmva}, \tau_{mva}), (k_{id}, \kappa_{pk}), w'), r),$ $(((t_{idmva}, \tau_{mva}), (k'_{id}, \kappa_{pk}), w'), l))$, where $k_{id} \neq \texttt{type}$, and $k'_{id} \neq \texttt{type}$. Also seen from the definition of $\texttt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}, r, c, w)$, $t_{idmva} = \mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1)$, and $t_{id} = \{t_{id} \mid (((t_{id}, \tau_{tmp_{mva}}), k, 1), r) \in \mathbb{R}, \texttt{GetTmpMVAColCnt}(\mathbb{R}, t_{id}) = 3\}$. However, from the definition of $\texttt{GetValidTmpMVATbls}(\mathbb{R}, r)$, it can be observed that, whenever $\mathbb{R}((t_{id}, \tau_{tmp_{mva}}), k, 1) = r$, and $\texttt{GetTmpMVAColCnt}(\mathbb{R}, t_{id}) = 3$, $e'' = (((t_{id}, \tau_{tmp_{mva}}), k, 1), d)$ is an element of the set $\texttt{GetValidTmpMVATbls}(\mathbb{R}, r)$. By definition of a Temporary MVA Table, $e'' = \{(((t_{id}, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1), t_{idmva}),$ $(((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1), r), (((t_{id}, \tau_{tmp_{mva}}), (k'_{id}, \kappa_{pk}), 1), l)\}$, where $k_{id} = t'_{id}\texttt{-type} \neq \texttt{type}$, and $k'_{id} \neq \texttt{type}$, and $t_{id} = r\_l$

Thus, for every element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idmva}, \tau_{mva}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idmva}, \tau_{mva}), (k'_{id}, \kappa_{pk}), w'), l)) \in \texttt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}, r, c, w)$, there exists an element $e'' = ((((r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1), t_{idmva}), (((r\_l, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1), r), (((r\_l, \tau_{tmp_{mva}}), (k'_{id}, \kappa_{pk}), 1), l)) \in \texttt{GetValidTmpMVATbls}(\mathbb{R}, r)$.

$\Longleftarrow$:

Let e'' be an element in the set $\texttt{GetValidTmpMVATbls}(\mathbb{R}, r)$. By definition of a Temporary MVA Table, $e'' = ((((t_{id}, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1), t_{idmva}), (((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1), r), (((t_{id}, \tau_{tmp_{mva}}), (k'_{id}, \kappa_{pk}), 1), l))$, where $k_{id} = t'_{id}\texttt{-type} \neq \texttt{type}$, and $k'_{id} \neq \texttt{type}$, and $t_{id} = r\_d$. As can be observed from the definition of $\texttt{AddTmpMVATblRowstoMVATbl}($

$\mathbb{R}', r, c, w)$, the value r is added to column `type` in row w of table, c, and the values $\mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1)$ and $\mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (k'_{id}, \kappa_{pk}), 1)$, where $k_{id} \neq \texttt{type}$, and $k'_{id} \neq \texttt{type}$, are added to the appropriate columns in an unoccupied row, w', in table, t_{idmva}, where $t_{idmva} = \mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1)$.

Thus, for every element $e'' = ((((r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1), t_{idmva}), (((r\_l, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1), r), (((r\_l, \tau_{tmp_{mva}}), (k'_{id}, \kappa_{pk}), 1), l)) \in \texttt{GetValidTmpMVATbls}(\mathbb{R}, r)$, there exists an element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idmva}, \tau_{mva}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idmva}, \tau_{mva}), (k'_{id}, \kappa_{pk}), w'), l)) \in \texttt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}, r, c, w)$, where $k_{id} \neq \texttt{type}$, and $k'_{id} \neq \texttt{type}$.

For any corresponding pair e' and e'', we have $\texttt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}, r, c, w) - \texttt{GetValidTmpMVATbls}(\mathbb{R}, r) = \{\pi_1 e', \pi_2 e', \pi_3 e'\} - \{\pi_1 e'', \pi_2 e'', \pi_3 e''\} = \{((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idmva}, \tau_{mva}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idmva}, \tau_{mva}), (k'_{id}, \kappa_{pk}), w'), l)\} - \{(((r\_l, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1), t_{idmva}), (((r\_l, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1), r), (((r\_l, \tau_{tmp_{mva}}), (k'_{id}, \kappa_{pk}), 1), l)\}$.

From Rule (MVATMPT), it can be observed that $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (\texttt{type}, \kappa_{pk}), 1) \mapsto t_{idmva}]\!] = \{(t_{id}, \texttt{type}, t_{idmva})\}$, $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (k_{id}, \kappa_{pk}), 1) \mapsto r]\!] = \{(t_{id}, k_{id}, r)\}$, and $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_{tmp_{mva}}), (k'_{id}, \kappa_{pk}), 1) \mapsto l]\!] = \{(t_{id}, k'_{id}, l)\}$, where $t_{id} = r\_l$.

From Rule (KCRT), it can be observed that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) \mapsto r]\!] = \{(r, \texttt{type}, c)\}$ and from Rules (MVATTC) and (MVATNTC), it can be seen that $\mathcal{E}[\![\mathbb{R}((t_{idmva}, \tau_{mva}), (k_{id}, \kappa_{pk}), w') \mapsto r]\!] = \{(r\_l, \texttt{type}, t_{idmva}), (r\_l, k_{id}, r)\}$ and $\mathcal{E}[\![\mathbb{R}((t_{idmva}, \tau_{mva}), (k'_{id}, \kappa_{pk}), w') \mapsto l]\!] = \{(r\_l, k'_{id}, l)\}$.

Thus, $\mathcal{E}[\![\texttt{AddTmpMVATblRowstoMVATbl}(\mathbb{R}, r, c, w) - \texttt{GetValidTmpMVATbls}(\mathbb{R}, r)]\!] = \mathcal{E}[\![\{\pi_1 e', \pi_2 e', \pi_3 e'\} - \{\pi_1 e'', \pi_2 e'', \pi_3 e''\}]\!] = \{(r, \texttt{type}, c), (r\_d, \texttt{type}, c'), (r\_d, k_{id}, r), (r\_d, k'_{id}, d)\} - \{(r\_d, \texttt{type}, c'), (r\_d, k_{id}, r), (r\_d, k'_{id}, d)\} = \{(r, \texttt{type}, c)\} = \{e\}$.

$\square$

Lemma 5 *For every instance edge e of type $(r, \texttt{type}, c)$, and an unoccupied row, w, in the table corresponding to c in $\mathbb{R}$,*

$$\mathcal{E}[\![\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w) - \texttt{GetValidTmpNMTbls}(\mathbb{R}, r)]\!] = \{e\}$$

Proof.

$\Longrightarrow$ **Case a: $(t_{id1} = c)$** :

Let $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), \pi_1 e'_1, \pi_2 e'_1)$, be an element in the set $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$. By definition of an N:M Table and $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$, $e'_1 = ((((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idnm}, \tau_{nm}), (k'_{id}, \kappa_{pk}), w'), r'))$, where $t_{idnm} = t_{id1}\_t_{id2}$, and $t_{id1} = c$.

Therefore, in **Case a**, $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idnm}, \tau_{nm}), (c\_\texttt{type}, \kappa_{pk}), w'), r), (((t_{idnm}, \tau_{nm}), (k'_{id}, \kappa_{pk}), w'), r'))$, where $t_{idnm} = t_{id1}\_t_{id2}$

Also seen from the definition of $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$, $t_{idnm} = \mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1)$, and $t_{id} = \{t_{id} \mid (((t_{id}, \tau_{tmp_{nm}}), k, 1), r) \in \mathbb{R}\}$, $\texttt{GetClassForTmpNMTblCells}(\mathbb{R}, t_{id}) = 1, \texttt{GetTmpNMColCnt}(\mathbb{R}, t_{id}) = 3\}$. However, from the definition of $\texttt{GetValidTmpNMTbls}(\mathbb{R}, r)$, it can be observed that, whenever $\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), k, 1) = r$, $\texttt{GetClassForTmpNMTblCells}(\mathbb{R}, t_{id}) = 1$, and $\texttt{GetTmpNMColCnt}(\mathbb{R}, t_{id}) = 3$, $e'' = (((t_{id}, \tau_{tmp_{nm}}), k, 1), d)$ is an element of the set $\texttt{GetValidTmpNMTbls}(\mathbb{R}, r)$. By definition of a Temporary N:M Table, $e'' = ((((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((t_{id}, \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r'))$, where $k_{id} = t_{id1}\_\texttt{type} \neq \texttt{type}$, and $k'_{id} \neq \texttt{type}$, and $t_{id} = r\_r'$

Thus, for every element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idnm}, \tau_{nm}), (k'_{id}, \kappa_{pk}), w'), r')) \in \texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$, there exists an element $e'' = ((((r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((r\_r', \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((r\_r', \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r')) \in \texttt{GetValidTmpNMTbls}(\mathbb{R}, r)$.

$\Longleftarrow$ **Case a:** $(t_{id1} = c)$:

Let e'' be an element in the set $\texttt{GetValidTmpNMTbls}(\mathbb{R}, r)$. By definition of a Temporary NM Table, $e'' = ((((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((t_{id}, \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r'))$, where $t_{idnm} = t_{id1}\_t_{id2}$, and $t_{id1} = c$.

Therefore, in **Case a**, $e'' = ((((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((t_{id}, \tau_{tmp_{nm}}), (c\_\texttt{type}, \kappa_{fk}), 1), r), (((t_{id}, \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r'))$, where $t_{id} = r\_r'$ and $t_{idnm} = c\_t_{id2}$.

As can be observed from the definition of $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$, the value r is added to column $\texttt{type}$ in row w of table, c, and the values $\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (c\_\texttt{type}, \kappa_{fk}), 1)$ and $\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1)$, are added to the appropriate columns in an unoccupied row, w', in table, t_{idnm}, where $t_{idnm} = \mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1) = c\_t_{id2}$.

Thus, for every element $e'' = ((((r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((r\_r', \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((r\_r', \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r')) \in \texttt{GetValidTmpNMTbls}(\mathbb{R}, r)$, there exists an elements $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idnm}, \tau_{nm}), (k'_{id}, \kappa_{pk}), w'), r')) \in \texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$, where $t_{idnm} = c\_t_{id2}$, $k_{id} \neq \texttt{type}$, and $k'_{id} \neq \texttt{type}$.

For any corresponding pair e' and e'', we have $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w) - \texttt{GetValidTmpNMTbls}(\mathbb{R}, r) = \{\pi_1 e', \pi_2 e', \pi_3 e'\} - \{\pi_1 e'', \pi_2 e'', \pi_3 e''\} = \{((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idnm}, \tau_{nm}), (k'_{id}, \kappa_{pk}), w'), r')\} - \{(((r\_r', \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((r\_r', \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((r\_r', \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r')\}$.

From Rule (NMTmpT), it can be observed that $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1) \mapsto t_{idnm}]\!] = \{(t_{id}, \texttt{type}, t_{idnm})\}$, $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (c\_\texttt{type}, \kappa_{fk}), 1) \mapsto r]\!] = \{(t_{id}, c\_\texttt{type}, r)\}$, and $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1) \mapsto r']\!] = \{(t_{id}, k'_{id}, r')\}$, where $t_{id} = r\_r'$ and $t_{idnm} = c\_t_{id2}$.

From Rule (KCRT), it can be observed that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) \mapsto r]\!] = \{(r, \texttt{type}, c)\}$ and from Rules (NMTTC) and (NMTNTC), it can be seen that $\mathcal{E}[\![\mathbb{R}((c\_t_{id2},$

$\tau_{nm}), (c\_\texttt{type}, \kappa_{pk}), w') \mapsto r]\!] = \{(r\_r', \texttt{type}, c\_t_{id2}), (r\_r', c\_\texttt{type}, r)\}$ and $\mathcal{E}[\![\mathbb{R}((c\_t_{id2}, \tau_{nm}), (k'_{id}, \kappa_{pk}), w') \mapsto r']\!] = \{(r\_r', k'_{id}, r')\}$.

Thus, $\mathcal{E}[\![\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w) - \texttt{GetValidTmpNMTbls}(\mathbb{R}, r)]\!] = \mathcal{E}[\![\{\pi_1 e', \pi_2 e', \pi_3 e'\} - \{\pi_1 e'', \pi_2 e'', \pi_3 e''\}]\!] = \{(r, \texttt{type}, c), (r\_r', \texttt{type}, c\_t_{id2}), (r\_r', c\_\texttt{type}, r), (r\_r', k'_{id}, r')\} - \{(r\_r', \texttt{type}, c\_t_{id2}), (r\_r', c\_\texttt{type}, r), (r\_r', k'_{id}, r')\} = \{(r, \texttt{type}, c)\} = \{e\}$.

$\Longrightarrow$ **Case b:** $(t_{id2} = c)$:

Let $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), \pi_1 e'_1, \pi_2 e'_1)$, be elements in the set $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$. By definition of an N:M Table and $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$, $e'_1 = ((((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idnm}, \tau_{nm}), (k'_{id}, \kappa_{pk}), w'), r'))$, where $t_{idnm} = t_{id1}\_t_{id2}$, and $t_{id2} = c$.

Therefore, in **Case b**, $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idnm}, \tau_{nm}), (t_{id1}\_\texttt{type}, \kappa_{pk}), w'), r'))$, where $t_{idnm} = t_{id1}\_c$.

Also seen from the definition of $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$, $t_{idnm} = \mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1)$, and $t_{id} = \{t_{id} \mid (((t_{id}, \tau_{tmp_{nm}}), k, 1), r) \in \mathbb{R}\}$, $\texttt{GetClassForTmpNMTblCells}(\mathbb{R}, t_{id}) = 1, \texttt{GetTmpNMColCnt}(\mathbb{R}, t_{id}) = 3\}$. However, from the definition of $\texttt{GetValidTmpNMTbls}(\mathbb{R}, r)$, it can be observed that, whenever $\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), k, 1) = r$, $\texttt{GetClassForTmpNMTblCells}(\mathbb{R}, t_{id}) = 1$, and $\texttt{GetTmpNMColCnt}(\mathbb{R}, t_{id}) = 3$, $e'' = (((t_{id}, \tau_{tmp_{nm}}), k, 1), d)$ is an element of the set $\texttt{GetValidTmpNMTbls}(\mathbb{R}, r)$. By definition of a Temporary N:M Table, $e'' = ((((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((t_{id}, \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r'))$, where $k'_{id} = t_{id1}\_\texttt{type} \neq \texttt{type}$, and $k_{id} \neq \texttt{type}$, and $t_{id} = r'\_r$

Thus, for every element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idnm}, \tau_{nm}), (k'_{id}, \kappa_{pk}), w'), r')) \in \texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$, there exists an element $e'' = ((((r'\_r, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((r'\_r, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((r'\_r, \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r')) \in \texttt{GetValidTmpNMTbls}(\mathbb{R}, r)$.

$\Longleftarrow$ **Case b:** $(t_{id2} = c)$:

Let e'' be an element in the set $\texttt{GetValidTmpNMTbls}(\mathbb{R}, r)$. By definition of a Temporary NM Table, $e'' = ((((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((t_{id}, \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r'))$, where $t_{idnm} = t_{id1}\_t_{id2}$, and $t_{id2} = c$.

Therefore, in **Case b**, $e'' = ((((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((t_{id}, \tau_{tmp_{nm}}), (t_{id1}\_\texttt{type}, \kappa_{fk}), 1), r'), (((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r))$, where $t_{id} = r'\_r$ and $t_{idnm} = t_{id1}\_c$.

As can be observed from the definition of $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$, the value r is added to column $\texttt{type}$ in row w of table, c, and the values $\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (t_{id1}\_\texttt{type}, \kappa_{fk}), 1)$ and $\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1)$, are added to the appropriate columns in an unoccupied row, w', in table, t_{idnm}, where $t_{idnm} = \mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1) = t_{id1}\_c$.

Thus, for every element $e'' = ((((r'\_r, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((r'\_r, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((r'\_r', \tau_{tmp_{nm}}), (t_{id1}\_\texttt{type}, \kappa_{fk}), 1), r')) \in \texttt{GetValidTmpNMTbls}(\mathbb{R}, r)$, there exists an element $e' = ((((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idnm}, \tau_{nm}), (t_{id1}\_\texttt{type}, \kappa_{pk}), w'), r')) \in \texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w)$, where $t_{idnm} = t_{id1}\_c$, $k_{id} \neq \texttt{type}$, and $k'_{id} = t_{id1}\_\texttt{type} \neq \texttt{type}$.

For any corresponding pair e' and e'', we have $\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w) - \texttt{GetValidTmpNMTbls}(\mathbb{R}, r) = \{\pi_1 e', \pi_2 e', \pi_3 e'\} - \{\pi_1 e'', \pi_2 e'', \pi_3 e''\} = \{(((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w), r), (((t_{idnm}, \tau_{nm}), (k_{id}, \kappa_{pk}), w'), r), (((t_{idnm}, \tau_{nm}), (k'_{id}, \kappa_{pk}), w'), r')\} - \{(((r'\_r, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1), t_{idnm}), (((r'\_r, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1), r), (((r'\_r, \tau_{tmp_{nm}}), (k'_{id}, \kappa_{fk}), 1), r')\}$, where $t_{idnm} = t_{id1}\_c$, and $k'_{id} = t_{id1}\_\texttt{type}$.

From Rule (NMTmpT), it can be observed that $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (\texttt{type}, \kappa_{pk}), 1) \mapsto t_{idnm}]\!] = \{(t_{id}, \texttt{type}, t_{idnm})\}$, $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (t_{id1}\_\texttt{type}, \kappa_{fk}), 1) \mapsto r']\!] = \{(t_{id}, t_{id1}\_\texttt{type}, r')\}$, and $\mathcal{E}[\![\mathbb{R}((t_{id}, \tau_{tmp_{nm}}), (k_{id}, \kappa_{fk}), 1) \mapsto r]\!] = \{(t_{id}, k_{id}, r)\}$, where $t_{id} = r'\_r$ and $t_{idnm} = t_{id1}\_c$.

From Rule (KCRT), it can be observed that $\mathcal{E}[\![\mathbb{R}((c, \tau_r), (c\_\texttt{type}, \kappa_{pk}), w) \mapsto r]\!] = \{(r, \texttt{type}, c)\}$ and from Rules (NMTTC) and (NMTNTC), it can be seen that $\mathcal{E}[\![\mathbb{R}((t_{id1}\_c, \tau_{nm}), (t_{id1}\_\texttt{type}, \kappa_{pk}), w') \mapsto r']\!] = \{(r'\_r, \texttt{type}, t_{id1}\_c), (r'\_r, t_{id1}\_\texttt{type}, r')\}$ and $\mathcal{E}[\![\mathbb{R}((t_{id1}\_c, \tau_{nm}), (k_{id}, \kappa_{pk}), w') \mapsto r]\!] = \{(r'\_r, k_{id}, r)\}$.

Thus, $\mathcal{E}[\![\texttt{AddTmpNMTblRowstoNMTbl}(\mathbb{R}, r, c, w) - \texttt{GetValidTmpNMTbls}(\mathbb{R}, r)]\!] = \mathcal{E}[\![\{\pi_1 e', \pi_2 e', \pi_3 e'\} - \{\pi_1 e'', \pi_2 e'', \pi_3 e''\}]\!] = \{(r, \texttt{type}, c), (r'\_r, \texttt{type}, t_{id1}\_c), (r'\_r, t_{id1}\_\texttt{type}, r'), (r'\_r, k_{id}, r)\} - \{(r'\_r, \texttt{type}, t_{id1}\_c), (r'\_r, t_{id1}\_\texttt{type}, r'), (r'\_r, k_{id}, r)\} = \{(r, \texttt{type}, c)\} = \{e\}$.

$\square$

APPENDIX C

SEMANTICS-PRESERVING QUERY TRANSFORMATION PROOF

The proof rules comprising the denotations of SQL and SPARQL queries and the transformation of SQL queries into equivalent sparql queries are listed in this appendix along with the proof of Theorem 2.

$$\frac{}{\mathcal{S}[\![(\emptyset, \hat{t}_{al})]\!]\phi = \emptyset}(\mathcal{S}\ \textsc{BaseCase})$$

$$\frac{\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al} \quad \texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{true} \\ {r_{\mathcal{S}}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \quad \texttt{TableExists}({r_{\mathcal{S}}}', \hat{t}) = \texttt{true}}{\mathcal{S}[\![(\{(\hat{s}, \hat{t})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}}}(\mathcal{S}\ \textsc{ChAllEx})$$

$$\frac{\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al} \quad \texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{true} \\ {r_{\mathcal{S}}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \quad \texttt{TableExists}({r_{\mathcal{S}}}', \hat{t}) = \texttt{false} \\ \phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, {r_{\mathcal{S}}}') \quad r_{\mathcal{S}} = \texttt{AddTable}({r_{\mathcal{S}}}', \hat{t})}{\mathcal{S}[\![(\{(\hat{s}, \hat{t})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{AddTableLabel}}(\mathcal{S}\ \textsc{ChChEx}) \\ (\texttt{GetRootNodes}(\mathcal{S}_T[\![\hat{t}]\!]\{\phi\}), \hat{t}), \phi'_{\mathcal{S}r}), r_{\mathcal{S}})$$

$$\frac{\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al} \quad \texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{false} \\ {r_{\mathcal{S}}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}})\ \texttt{TableExists}({r_{\mathcal{S}}}', \hat{t}) = \texttt{true} \quad \phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, {r_{\mathcal{S}}}')}{\mathcal{S}[\![(\{(\hat{s}, \hat{t})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{CreateEdges}(\hat{c}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \texttt{GetRootNodes}(\phi'_{\mathcal{S}r})) \cup \phi'_{\mathcal{S}}}(\mathcal{S}\ \textsc{ChTblEx})$$

$$\frac{\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al} \quad \texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{false} \\ {r_{\mathcal{S}}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \quad \texttt{TableExists}({r_{\mathcal{S}}}', \hat{t}) = \texttt{false} \\ \phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, {r_{\mathcal{S}}}') \quad r_{\mathcal{S}} = \texttt{AddTable}({r_{\mathcal{S}}}', \hat{t})}{\mathcal{S}[\![(\{(\hat{s}, \hat{t})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{CreateEdges}}(\mathcal{S}\ \textsc{ChNoneEx}) \\ (\hat{c}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \texttt{AddTableLabel}(\texttt{GetRootNodes}(\mathcal{S}_T[\![\hat{t}]\!]\{\phi\}), \hat{t})), \phi'_{\mathcal{S}r}), r_{\mathcal{S}})$$

Figure C.1: $\mathcal{S}$ - Base and Character Element with Table

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) \\ \texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{true} \\ r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \quad \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) = \texttt{true}\end{array}}{\mathcal{S}[\![(\{(\hat{s}, \hat{dt})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}}}(\mathcal{S}\ \textsc{ChAllExDT})$$

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) \\ \texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{true} \quad r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \\ \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) = \texttt{false} \quad \phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}') \\ r_{\mathcal{S}} = \texttt{AddTable}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) \quad \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}]\!]\phi = \phi''_{\mathcal{S}} \\ r_{\mathcal{S}_{dt}} = \texttt{GetRoot}(\phi''_{\mathcal{S}}) \quad \texttt{RemoveTableLabels}(\texttt{RemoveRoot}(\phi''_{\mathcal{S}}, r_{\mathcal{S}_{dt}})) = \phi_{\mathcal{S}_{dt}}\end{array}}{\begin{array}{c}\mathcal{S}[\![(\{(\hat{s}, \hat{dt})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{AddTableLabel} \\ (\texttt{GetRootNodes}(\phi_{\mathcal{S}_{dt}}), \hat{t}_{alsq_{\hat{sql}}}), \phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{array}}(\mathcal{S}\ \textsc{ChChExDT})$$

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) \\ \texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{false} \quad r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \\ \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) = \texttt{true} \quad \phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}')\end{array}}{\begin{array}{c}\mathcal{S}[\![(\{(\hat{s}, \hat{dt})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{CreateEdges}(\hat{c}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \texttt{GetRootNodes}(\phi'_{\mathcal{S}r})) \\ \cup\phi'_{\mathcal{S}}\end{array}}(\mathcal{S}\ \textsc{ChTblExDT})$$

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) \\ \texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{false} \quad r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \\ \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) = \texttt{false} \quad \phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}') \\ r_{\mathcal{S}} = \texttt{AddTable}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) \quad \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi = \phi''_{\mathcal{S}} \quad r_{\mathcal{S}_{dt}} = \texttt{GetRoot}(\phi''_{\mathcal{S}}) \\ \texttt{RemoveTableLabels}(\texttt{RemoveRoot}(\phi''_{\mathcal{S}}, r_{\mathcal{S}_{dt}})) = \phi_{\mathcal{S}_{dt}}\end{array}}{\begin{array}{c}\mathcal{S}[\![(\{(\hat{s}, \hat{dt})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{CreateEdges} \\ (\hat{c}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \texttt{AddTableLabel}(\texttt{GetRootNodes}(\phi_{\mathcal{S}_{dt}}), \hat{t}_{alsq_{\hat{sql}}})), \phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{array}}(\mathcal{S}\ \textsc{ChNoneExDT})$$

Figure C.2: $\mathcal{S}$ - Character Element with SubQuery

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{t} = \hat{t}' \\ \phi_{\mathcal{S}T} = \mathcal{S}_T[\![\hat{t}]\!]\{\phi\} \quad r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \quad \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}) = \texttt{true}\end{array}}{\begin{array}{c}\mathcal{S}[\![(\{(\hat{s}, \hat{t})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AttachEdge}(\texttt{ReplaceLabel}(\texttt{ExtColEdges} \\ (\hat{t}_{\text{-}T_{id}}\hat{cl}, \texttt{GetRootNodes}(\phi_{\mathcal{S}T}), \phi_{\mathcal{S}T}), \hat{t}_{\text{-}T_{id}}\hat{cl}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}) \\ \times\texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}'))) \cup \phi'_{\mathcal{S}}\end{array}} (\mathcal{S}\ \textsc{ClTMchEx})$$

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{t} = \hat{t}' \quad \phi_{\mathcal{S}T} = \mathcal{S}_T[\![\hat{t}]\!]\{\phi\} \\ r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \quad \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}) = \texttt{false} \\ \phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}') \quad r_{\mathcal{S}} = \texttt{AddTable}(r_{\mathcal{S}}', \hat{t})\end{array}}{\begin{array}{c}\mathcal{S}[\![(\{(\hat{s}, \hat{t})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{ReplaceLabel} \\ (\texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, \texttt{GetRootNodes}(\phi_{\mathcal{S}T}), \phi_{\mathcal{S}T}), \hat{t}_{\text{-}T_{id}}\hat{cl}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}), \phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{array}} (\mathcal{S}\ \textsc{ClTMchNoEx})$$

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{t} \neq \hat{t}' \\ r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \quad \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}) = \texttt{true}\end{array}}{\mathcal{S}[\![(\{(\hat{s}, \hat{t})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}}} (\mathcal{S}\ \textsc{ClTNoMchEx})$$

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{t} \neq \hat{t}' \quad r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \\ \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}) = \texttt{false} \quad \phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}') \\ r_{\mathcal{S}} = \texttt{AddTable}(r_{\mathcal{S}}', \hat{t}) \quad \phi_{\mathcal{S}T} = \mathcal{S}_T[\![\hat{t}]\!]\{\phi\}\end{array}}{\begin{array}{c}\mathcal{S}[\![(\{(\hat{s}, \hat{t})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{AddTableLabel} \\ (\texttt{GetRootNodes}(\phi_{\mathcal{S}T}), \hat{t}), \phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{array}} (\mathcal{S}\ \textsc{ClTNoMchNoEx})$$

Figure C.3: $\mathcal{S}$ - Column Element With Table

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{t}.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) \quad \hat{t} = \hat{t}_{alsq_{\hat{sql}}} \\ \phi''_{\mathcal{S}} = \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi \quad r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \\ \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) = \texttt{true} \quad r_{\mathcal{S}_{dt}} = \texttt{GetRoot}(\phi''_{\mathcal{S}})\end{array}}{\begin{array}{c}\mathcal{S}[\![(\{(\hat{s}, \hat{dt})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AttachEdge}(\texttt{ReplaceLabel}(\texttt{ExtColEdges} \\ (\hat{t}_{\text{-}T_{id}}\hat{cl}, \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi''_{\mathcal{S}}, r_{\mathcal{S}_{dt}})), \phi''_{\mathcal{S}}), \hat{t}_{\text{-}T_{id}}\hat{cl}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al})\times \\ \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}'))) \cup \phi'_{\mathcal{S}}\end{array}}(\mathcal{S}\ \textsc{ClDTMchEx})$$

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{t}.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) \quad \hat{t} = \hat{t}_{alsq_{\hat{sql}}} \\ \phi''_{\mathcal{S}} = \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi \quad r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \\ \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) = \texttt{false} \quad \phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}') \\ r_{\mathcal{S}} = \texttt{AddTable}(r_{\mathcal{S}}', \hat{t}) \quad r_{\mathcal{S}_{dt}} = \texttt{GetRoot}(\phi''_{\mathcal{S}})\end{array}}{\begin{array}{c}\mathcal{S}[\![(\{(\hat{s}, \hat{dt})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{ReplaceLabel} \\ (\texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi''_{\mathcal{S}}, r_{\mathcal{S}_{dt}})), \phi''_{\mathcal{S}}), \hat{t}_{\text{-}T_{id}}\hat{cl}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}), \\ \phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{array}}(\mathcal{S}\ \textsc{ClDTMchNoEx})$$

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{t}.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) \quad \hat{t} \neq \hat{t}_{alsq_{\hat{sql}}} \\ r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \quad \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) = \texttt{true}\end{array}}{\mathcal{S}[\![(\{(\hat{s}, \hat{dt})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}}}(\mathcal{S}\ \textsc{ClDTNoMchEx})$$

$$\frac{\begin{array}{c}\mathcal{S}[\![(\hat{sql}', \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{S}} \qquad \hat{s} = \hat{t}.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al} \quad \hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})\hat{t} \neq \hat{t}_{alsq_{\hat{sql}}} \\ \phi''_{\mathcal{S}} = \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi \quad r_{\mathcal{S}}' = \texttt{GetRoot}(\phi'_{\mathcal{S}}) \quad \texttt{TableExists}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) = \texttt{false} \\ \phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, r_{\mathcal{S}}') \quad r_{\mathcal{S}} = \texttt{AddTable}(r_{\mathcal{S}}', \hat{t}_{alsq_{\hat{sql}}}) \quad r_{\mathcal{S}_{dt}} = \texttt{GetRoot}(\phi''_{\mathcal{S}}) \\ \texttt{RemoveTableLabels}(\texttt{RemoveRoot}(\phi''_{\mathcal{S}}, r_{\mathcal{S}_{dt}})) = \phi_{\mathcal{S}_{dt}}\end{array}}{\begin{array}{c}\mathcal{S}[\![(\{(\hat{s}, \hat{dt})\} \uplus \hat{sql}', \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs} \\ (\texttt{AddTableLabel}(\texttt{GetRootNodes}(\phi_{\mathcal{S}_{dt}}), \hat{t}_{alsq_{\hat{sql}}}), \phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{array}}(\mathcal{S}\ \textsc{ClDTNoMchNoEx})$$

Figure C.4: $\mathcal{S}$ - Column Element With Sub Query

$$\frac{}{h_w(\cdot, \hat{t}_{al}) = \cdot}(h_w \text{ SQL FROM } \textsc{List Base})$$

$$\frac{\hat{tr} = \hat{dt} \quad \hat{dt} = (\hat{sql}, \hat{t}_{al}) \quad \hat{sql} = (\rho_{\mathbb{SL}}, \rho_{\mathbb{TR}}) \quad (\rho_{\mathrm{SpSL}}, \rho_{\mathrm{SpW_t}}) = h(\hat{sql} = (\rho_{\mathbb{SL}}, \rho_{\mathbb{TR}}), \hat{t}_{al}) \quad \dot{spq} = (\rho_{\mathrm{SpSL}}, \rho_{\mathrm{SpW_t}})}{h_w(\hat{tr}, \hat{t}_{al}) = \dot{spq}}(h_w \text{ SQL } \textsc{From - SubQuery})$$

$$\frac{\hat{tr} = \hat{t} \quad \dot{w} = (?\hat{t}\_\texttt{type type } \hat{t})}{h_w(\hat{tr}, \hat{t}_{al}) = \dot{w}}(h_w \text{ SQL } \textsc{From - Table})$$

$$\frac{\dot{w} = h_w(\hat{tr}, \hat{t}_{al}) \quad \rho_{\mathrm{SpW}} = h_w(\rho_{\mathbb{TR}}, \hat{t}_{al})}{h_w(\hat{tr} :: \rho_{\mathbb{TR}}, \hat{t}_{al}) = \dot{w} :: \rho_{\mathrm{SpW}}}(h_w \text{ SQL } \textsc{From Clause})$$

$$\frac{h_s(\rho_{\mathbb{SL}}, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\rho_{\mathrm{SpSL}}, \rho_{\mathrm{SpW}_s}) \quad h_w(\rho_{\mathbb{TR}}, \hat{t}_{al}) = \rho_{\mathrm{SpW}} \quad \dot{spq} = (\rho_{\mathrm{SpSL}}, \rho_{\mathrm{SpW}_s}@\rho_{\mathrm{SpW}})}{h(\hat{sql} = (\rho_{\mathbb{SL}}, \rho_{\mathbb{TR}}), \hat{t}_{al}) = \dot{spq}}(h \text{ SQL } \textsc{Query Translation})$$

Figure C.5: SQL-to-SPARQL Translation function - h, `and` h_w

$$\frac{}{h_s(\cdot, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\cdot, \cdot)}(h_s \text{ SQL SELECT L}\textsc{ist} \text{ B}\textsc{ase})$$

$$\frac{\begin{array}{c} n\hat{v}e_a = \hat{t}.\hat{cl} \texttt{ AS } \hat{cl}_{al} \quad \texttt{IsTable}(\rho_{\mathbb{TR}}, \hat{t}) = \texttt{true} \quad \dot{spsl} = ?\hat{t}_{\text{-}T_{id}}\hat{cl} \texttt{ AS } ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al} \\ \dot{w}_s = (?\hat{t}_{\text{-}T_{id}}\texttt{type } \hat{cl} \; ?\hat{t}_{\text{-}T_{id}}\hat{cl}) \end{array}}{h_s(n\hat{v}e_a, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\dot{spsl}, \dot{w}_s)}(h_s \text{ TC}\textsc{olumn} \text{ A}\textsc{l}\text{SQ})$$

$$\frac{\begin{array}{c} n\hat{v}e_a = \hat{t}.\hat{cl} \texttt{ AS } \hat{cl}_{al} \quad \texttt{IsTable}(\rho_{\mathbb{TR}}, \hat{t}) = \texttt{false} \\ \dot{spsl} = ?\hat{t}_{\text{-}T_{id}}\hat{cl} \texttt{ AS } ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al} \end{array}}{h_s(n\hat{v}e_a, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\dot{spsl}, \cdot)}(h_s \text{ DTC}\textsc{olumn} \text{ A}\textsc{l}\text{SQ})$$

$$\frac{n\hat{v}e_a = \hat{i} \texttt{ AS } \hat{cl}_{al} \quad \dot{spsl} = \hat{i} \texttt{ AS } ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}}{h_s(n\hat{v}e_a, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\dot{spsl}, \cdot)}(h_s \text{ I}\textsc{nt} \text{ A}\textsc{l}\text{SQ})$$

$$\frac{n\hat{v}e_a = \hat{c} \texttt{ AS } \hat{cl}_{al} \quad \dot{spsl} = \hat{c} \texttt{ AS } ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}}{h_s(n\hat{v}e_a, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\dot{spsl}, \cdot)}(h_s \text{ C}\textsc{har} \text{ A}\textsc{l}\text{SQ})$$

$$\frac{(\dot{spsl}, \dot{w}_s) = h_s(n\hat{v}e_a, \hat{t}_{al}) \quad h_s(\rho_{\mathbb{SL}}, \hat{t}_{al}) = (\rho_{\text{SpSL}}, \rho_{\text{SpW}_s})}{h_s(n\hat{v}e_a :: \rho_{\mathbb{SL}}, \rho_{\mathbb{TR}}, \hat{t}_{al}) = (\dot{spsl} :: \rho_{\text{SpSL}}, \dot{w}_s :: \rho_{\text{SpW}_s})}(h_s \text{ SQL S}\textsc{elect} \text{ C}\textsc{lause})$$

Figure C.6: SQL-to-SPARQL Translation function - h_s

$$\frac{}{\mathcal{Q}[\![\emptyset]\!]\phi = \emptyset}(\mathcal{Q}\ \textsc{BaseCase})$$

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \hat{c}\ \texttt{AS}\ \dot{v}_{al} \quad \dot{w} = (s\ \texttt{type}\ c) \quad \dot{v}_{al} = c''\text{-}_{T_{id}}p \\ \dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) \quad \texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{true} \\ r_{\mathcal{Q}}{}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}{}', c) = \texttt{true}\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{w})\} \uplus \dot{spq}]\!]\phi = \phi'_{\mathcal{Q}}}(\mathcal{Q}\ \textsc{ChAllEx})$$

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \hat{c}\ \texttt{AS}\ \dot{v}_{al} \quad \dot{w} = (s\ \texttt{type}\ c) \quad \dot{v}_{al} = c''\text{-}_{T_{id}}p \\ \dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) \quad \texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{true} \\ r_{\mathcal{Q}}{}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}{}', c) = \texttt{false} \\ \phi'_{\mathcal{Q}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}{}') \quad r_{\mathcal{Q}} = \texttt{AddTable}(r_{\mathcal{Q}}{}', c)\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{w})\} \uplus \dot{spq}]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{AddTableLabel}}(\mathcal{Q}\ \textsc{ChChEx})$$
$$(\texttt{GetRootNodes}(\mathcal{S}_T[\![c]\!]\{\phi\}), c), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})$$

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \hat{c}\ \texttt{AS}\ \dot{v}_{al} \quad \dot{w} = (s\ \texttt{type}\ c) \quad \dot{v}_{al} = c''\text{-}_{T_{id}}p \\ \dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) \quad \texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{false} \quad r_{\mathcal{Q}}{}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \\ \texttt{TableExists}(r_{\mathcal{Q}}{}', c) = \texttt{true} \qquad \phi'_{\mathcal{Q}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}{}')\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{w})\} \uplus \dot{spq}]\!]\phi = \texttt{CreateEdges}(\hat{c}, \dot{v}_{al_{rf}}, \texttt{GetRootNodes}(\phi'_{\mathcal{Q}r}))}(\mathcal{Q}\ \textsc{ChTblEx})$$
$$\cup \phi'_{\mathcal{Q}}$$

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \hat{c}\ \texttt{AS}\ \dot{v}_{al} \quad \dot{w} = (s\ \texttt{type}\ c) \quad \dot{v}_{al} = c''\text{-}_{T_{id}}p \\ \dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) \quad \texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{false} \\ r_{\mathcal{Q}}{}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}{}', c) = \texttt{false} \\ \phi'_{\mathcal{Q}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}{}') \quad r_{\mathcal{Q}} = \texttt{AddTable}(r_{\mathcal{Q}}{}', c)\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{w})\} \uplus \dot{spq}]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{CreateEdges}}(\mathcal{Q}\ \textsc{ChNoneEx})$$
$$(\hat{c}, \dot{v}_{al_{rf}}, \texttt{AddTableLabel}(\texttt{GetRootNodes}(\mathcal{S}_T[\![c]\!]\{\phi\}), c)), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})$$

Figure C.7: $\mathcal{Q}$ - Base and Character Element with Type Where

$$\frac{\begin{array}{c}\mathcal{Q}\llbracket \dot{spq}' \rrbracket\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \hat{c}\ \mathtt{AS}\ \dot{v}_{al} \quad \dot{v}_{al} = c''\text{-}_{T_{id}}p \quad \dot{v}_{al_{rf}} = \mathtt{RemoveFirstChar}(\dot{v}_{al}) \\ \hat{t}_{alsq\dot{spq}} = \mathtt{GetSubQueryAlias}(\dot{spq}_{sq}) \quad \mathtt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \mathtt{true} \\ r_{\mathcal{Q}}' = \mathtt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \mathtt{TableExists}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}})\mathtt{true}\end{array}}{\mathcal{Q}\llbracket\{(\dot{spsl}, \dot{spq}_{sq})\} \uplus \dot{spq}\rrbracket\phi = \phi'_{\mathcal{Q}}}(\mathcal{Q}\ \textsc{ChAllExSQ})$$

$$\frac{\begin{array}{c}\mathcal{Q}\llbracket \dot{spq}' \rrbracket\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \hat{c}\ \mathtt{AS}\ \dot{v}_{al} \quad \dot{v}_{al} = c''\text{-}_{T_{id}}p \quad \dot{v}_{al_{rf}} = \mathtt{RemoveFirstChar}(\dot{v}_{al}) \\ \hat{t}_{alsq\dot{spq}} = \mathtt{GetSubQueryAlias}(\dot{spq}_{sq}) \quad \mathtt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \mathtt{true} \\ r_{\mathcal{Q}}' = \mathtt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \mathtt{TableExists}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}}) = \mathtt{false} \quad \phi'_{\mathcal{Q}r} = \mathtt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}') \\ r_{\mathcal{Q}} = \mathtt{AddTable}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}}) \quad \mathcal{Q}\llbracket \dot{spq}_{sq} \rrbracket\phi = \phi''_{\mathcal{Q}} \\ r_{\mathcal{Q}_{dt}} = \mathtt{GetRoot}(\phi''_{\mathcal{Q}}) \quad \mathtt{RemoveTableLabels}(\mathtt{RemoveRoot}(\phi''_{\mathcal{Q}}, r_{\mathcal{Q}_{dt}})) = \phi_{\mathcal{Q}_{dt}}\end{array}}{\begin{array}{c}\mathcal{Q}\llbracket\{(\dot{spsl}, \dot{spq}_{sq})\} \uplus \dot{spq}\rrbracket\phi = \mathtt{AddRoot}(\mathtt{JoinGraphs}(\mathtt{AddTableLabel} \\ (\mathtt{GetRootNodes}(\phi_{\mathcal{Q}_{dt}}), \hat{t}_{alsq\dot{spq}}), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})\end{array}}(\mathcal{Q}\ \textsc{ChChExSQ})$$

$$\frac{\begin{array}{c}\mathcal{Q}\llbracket \dot{spq}' \rrbracket\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \hat{c}\ \mathtt{AS}\ \dot{v}_{al} \quad \dot{v}_{al} = c''\text{-}_{T_{id}}p \\ \dot{v}_{al_{rf}} = \mathtt{RemoveFirstChar}(\dot{v}_{al}) \quad \hat{t}_{alsq\dot{spq}} = \mathtt{GetSubQueryAlias}(\dot{spq}_{sq}) \\ \mathtt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \mathtt{false} \quad r_{\mathcal{Q}}' = \mathtt{GetRoot}(\phi'_{\mathcal{Q}}) \\ \mathtt{TableExists}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}}) = \mathtt{true} \quad \phi'_{\mathcal{Q}r} = \mathtt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}')\end{array}}{\mathcal{Q}\llbracket\{(\dot{spsl}, \dot{spq}_{sq})\} \uplus \dot{spq}\rrbracket\phi = \mathtt{CreateEdges}(\hat{c}, \dot{v}_{al_{rf}}, \mathtt{GetRootNodes}(\phi'_{\mathcal{Q}r})) \cup \phi'_{\mathcal{Q}}}(\mathcal{Q}\ \textsc{ChTblExSQ})$$

$$\frac{\begin{array}{c}\mathcal{Q}\llbracket \dot{spq}' \rrbracket\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \hat{c}\ \mathtt{AS}\ \dot{v}_{al} \quad \dot{v}_{al} = c''\text{-}_{T_{id}}p \quad \dot{v}_{al_{rf}} = \mathtt{RemoveFirstChar}(\dot{v}_{al}) \\ \hat{t}_{alsq\dot{spq}} = \mathtt{GetSubQueryAlias}(\dot{spq}_{sq}) \quad \mathtt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \mathtt{false} \\ r_{\mathcal{Q}}' = \mathtt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \mathtt{TableExists}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}}) = \mathtt{false} \quad \phi'_{\mathcal{Q}r} = \mathtt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}') \\ r_{\mathcal{Q}} = \mathtt{AddTable}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}}) \quad \mathcal{Q}\llbracket \dot{spq}_{sq} \rrbracket\phi = \phi''_{\mathcal{Q}} \quad r_{\mathcal{Q}_{dt}} = \mathtt{GetRoot}(\phi''_{\mathcal{Q}}) \\ \mathtt{RemoveTableLabels}(\mathtt{RemoveRoot}(\phi''_{\mathcal{Q}}, r_{\mathcal{Q}_{dt}})) = \phi_{\mathcal{Q}_{dt}}\end{array}}{\begin{array}{c}\mathcal{Q}\llbracket\{(\dot{spsl}, \dot{spq}_{sq})\} \uplus \dot{spq}\rrbracket\phi = \mathtt{AddRoot}(\mathtt{JoinGraphs}(\mathtt{CreateEdges} \\ (\hat{c}, \dot{v}_{al_{rf}}, \mathtt{AddTableLabel}(\mathtt{GetRootNodes}(\phi_{\mathcal{Q}_{dt}}), \hat{t}_{alsq\dot{spq}})), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})\end{array}}(\mathcal{Q}\ \textsc{ChNoneExSQ})$$

Figure C.8: $\mathcal{Q}$ - Character Element with SubQuery

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \dot{v}\ \texttt{AS}\ \dot{v}_{al} \quad \dot{v} = ?c_{\text{-}T_{id}}p \quad \dot{v}_{al} = c''_{\text{-}T_{id}}p \\ \dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) \quad \dot{w} = \{(?c_{\text{-}T_{id}}\texttt{type}\ p\ \dot{v}), (?c'_{\text{-}T_{id}}\texttt{type}\ \texttt{type}\ c')\} \quad c = c' \\ \phi_{QT} = \mathcal{S}_T[\![c]\!]\{\phi\} \quad r_{\mathcal{Q}}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}', c) = \texttt{true}\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{w})\} \uplus \dot{spq}]\!]\phi = \texttt{AttachEdge}(\texttt{ReplaceLabel}(\texttt{ExtColEdges}}(\mathcal{Q}\ \textsc{ClTwMchEx})$$

$$(c_{\text{-}T_{id}}p, \texttt{GetRootNodes}(\phi_{QT}), \phi_{QT}), c_{\text{-}T_{id}}p, \dot{v}_{al_{rf}}) \times \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}'))) \cup \phi'_{\mathcal{Q}}$$

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \dot{v}\ \texttt{AS}\ \dot{v}_{al} \quad \dot{v} = ?c_{\text{-}T_{id}}p \quad \dot{v}_{al} = c''_{\text{-}T_{id}}p \\ \dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) \quad \dot{w} = \{(?c_{\text{-}T_{id}}\texttt{type}\ p\ \dot{v}), (?c'_{\text{-}T_{id}}\texttt{type}\ \texttt{type}\ c')\} \quad c = c' \\ \phi_{QT} = \mathcal{S}_T[\![c]\!]\{\phi\} \quad r_{\mathcal{Q}}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}', c) = \texttt{false} \\ \phi'_{Qr} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}') \quad r_{\mathcal{Q}} = \texttt{AddTable}(r_{\mathcal{Q}}', c)\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{w})\} \uplus \dot{spq}]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{ReplaceLabel}}(\mathcal{Q}\ \textsc{ClTwMchNoEx})$$

$$(\texttt{ExtColEdges}(c_{\text{-}T_{id}}p, \texttt{GetRootNodes}(\phi_{QT}), \phi_{QT}), c_{\text{-}T_{id}}p, \dot{v}_{al_{rf}}),$$

$$\phi'_{Qr}), r_{\mathcal{Q}})$$

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \dot{v}\ \texttt{AS}\ \dot{v}_{al} \quad \dot{v} = ?c_{\text{-}T_{id}}p \\ \dot{w} = \{(?c_{\text{-}T_{id}}\texttt{type}\ p\ \dot{v}), (?c'_{\text{-}T_{id}}\texttt{type}\ \texttt{type}\ c')\} \quad c \neq c' \\ r_{\mathcal{Q}}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}', c') = \texttt{true}\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{w})\} \uplus \dot{spq}]\!]\phi = \phi'_{\mathcal{Q}}}(\mathcal{Q}\ \textsc{ClTwNoMchEx})$$

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \dot{v}\ \texttt{AS}\ \dot{v}_{al} \quad \dot{v} = ?c_{\text{-}T_{id}}p \\ \dot{w} = \{(?c_{\text{-}T_{id}}\texttt{type}\ p\ \dot{v}), (?c'_{\text{-}T_{id}}\texttt{type}\ \texttt{type}\ c')\} \quad c \neq c' \quad r_{\mathcal{Q}}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \\ \texttt{TableExists}(r_{\mathcal{Q}}', c') = \texttt{false} \quad \phi'_{Qr} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}') \\ r_{\mathcal{Q}} = \texttt{AddTable}(r_{\mathcal{Q}}', c') \quad \phi_{QT} = \mathcal{S}_T[\![c']\!]\{\phi\}\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{w})\} \uplus \dot{spq}]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}}(\mathcal{Q}\ \textsc{ClTwNoMchNoEx})$$

$$(\texttt{AddTableLabel}(\texttt{GetRootNodes}(\phi_{QT}), c'), \phi'_{Qr}), r_{\mathcal{Q}})$$

Figure C.9: $\mathcal{Q}$ - Variable Element With Type Where

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \dot{v} =?c_{\text{-}T_{id}}p\ \texttt{AS}\ \dot{v}_{al} \quad \dot{v}_{al} = c''_{\text{-}T_{id}}p \\ \dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) \quad \hat{t}_{alsq\dot{spq}} = \texttt{GetSubQueryAlias}(\dot{spq}_{sq}) \quad \hat{t}_{alsq\dot{spq}} = c \\ r_{\mathcal{Q}}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}}) = \texttt{true} \\ \mathcal{Q}[\![\dot{spq}_{sq}]\!]\phi = \phi''_{\mathcal{Q}} \quad r_{\mathcal{Q}_{dt}} = \texttt{GetRoot}(\phi''_{\mathcal{Q}})\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{spq}_{sq})\} \uplus \dot{spq}]\!]\phi = \texttt{AttachEdge}(\texttt{ReplaceLabel}(\texttt{ExtColEdges}}(\mathcal{Q}\ \textsc{ClSQMchEx})$$

$$(c_{\text{-}T_{id}}p, \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi''_{\mathcal{Q}}, r_{\mathcal{Q}_{dt}})), \phi''_{\mathcal{Q}}), c_{\text{-}T_{id}}p, \dot{v}_{al_{rf}})\times$$

$$\texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}'))) \cup \phi'_{\mathcal{Q}}$$

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \dot{v} =?c_{\text{-}T_{id}}p\ \texttt{AS}\ \dot{v}_{al} \quad \dot{v}_{al} = c''_{\text{-}T_{id}}p \\ \dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) \quad \hat{t}_{alsq\dot{spq}} = \texttt{GetSubQueryAlias}(\dot{spq}_{sq}) \quad \hat{t}_{alsq\dot{spq}} = c \\ r_{\mathcal{Q}}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}}) = \texttt{false} \\ \phi'_{\mathcal{Q}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}') \quad r_{\mathcal{Q}} = \texttt{AddTable}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}})l \\ \mathcal{Q}[\![\dot{spq}_{sq}]\!]\phi = \phi''_{\mathcal{Q}} \quad r_{\mathcal{Q}_{dt}} = \texttt{GetRoot}(\phi''_{\mathcal{Q}})\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{spq}_{sq})\} \uplus \dot{spq}]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{ReplaceLabel}}(\mathcal{Q}\ \textsc{ClSQMchNoEx})$$

$$(\texttt{ExtColEdges}(c_{\text{-}T_{id}}p, \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi''_{\mathcal{Q}}, r_{\mathcal{Q}_{dt}})), \phi''_{\mathcal{Q}}), c_{\text{-}T_{id}}p, \dot{v}_{al_{rf}}),$$

$$\phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})$$

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \dot{v} =?c_{\text{-}T_{id}}p\ \texttt{AS}\ \dot{v}_{al} \\ \hat{t}_{alsq\dot{spq}} = \texttt{GetSubQueryAlias}(\dot{spq}_{sq}) \quad \hat{t}_{alsq\dot{spq}} \neq c \\ r_{\mathcal{Q}}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}}) = \texttt{true}\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{spq}_{sq})\} \uplus \dot{spq}]\!]\phi = \phi'_{\mathcal{Q}}}(\mathcal{Q}\ \textsc{ClSQNoMchEx})$$

$$\frac{\begin{array}{c}\mathcal{Q}[\![\dot{spq}']\!]\phi = \phi'_{\mathcal{Q}} \qquad \dot{spsl} = \dot{v} =?c_{\text{-}T_{id}}p\ \texttt{AS}\ \dot{v}_{al} \quad \hat{t}_{alsq\dot{spq}} = \texttt{GetSubQueryAlias}(\dot{spq}_{sq}) \\ \hat{t}_{alsq\dot{spq}} \neq c \quad r_{\mathcal{Q}}' = \texttt{GetRoot}(\phi'_{\mathcal{Q}}) \quad \texttt{TableExists}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}}) = \texttt{false} \\ \phi'_{\mathcal{Q}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, r_{\mathcal{Q}}') \quad r_{\mathcal{Q}} = \texttt{AddTable}(r_{\mathcal{Q}}', \hat{t}_{alsq\dot{spq}}) \\ \mathcal{Q}[\![\dot{spq}_{sq}]\!]\phi = \phi''_{\mathcal{Q}} \quad r_{\mathcal{Q}_{dt}} = \texttt{GetRoot}(\phi''_{\mathcal{Q}}) \\ \texttt{RemoveTableLabels}(\texttt{RemoveRoot}(\phi''_{\mathcal{Q}}, r_{\mathcal{Q}_{dt}})) = \phi_{\mathcal{Q}_{dt}}\end{array}}{\mathcal{Q}[\![\{(\dot{spsl}, \dot{spq}_{sq})\} \uplus \dot{spq}]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}}(\mathcal{Q}\ \textsc{ClSQNoMchNoEx})$$

$$(\texttt{AddTableLabel}(\texttt{GetRootNodes}(\phi_{\mathcal{Q}_{dt}}), \hat{t}_{alsq\dot{spq}}), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})$$

Figure C.10: $\mathcal{Q}$ - Variable Element With SubQuery

Theorem 2 $\mathcal{S}[\![\hat{sql}_q = (\hat{sql}, \hat{t}_{al})]\!]\phi = \mathcal{Q}[\![h(\hat{sql}_q = (\hat{sql}, \hat{t}_{al}))]\!]\phi$

Proof. Proof is by(structural) induction on the structure of SQL query $\hat{sql}_q$.

Base Case: Assume $\hat{sql} = \emptyset$. By Rule $\mathcal{S}[\![]\!]$ BASECASE,

$$\mathcal{S}[\![(\emptyset, \hat{t}_{al})]\!] = \emptyset \tag{13.1}$$

. By Rules h BASECASE and $\mathcal{Q}[\![]\!]$ BASECASE we have $h(\emptyset) = \emptyset$ and $\mathcal{Q}[\![\emptyset]\!] = \emptyset$. Therefore, we have

$$\mathcal{Q}[\![h(\emptyset)]\!] = \emptyset \tag{13.2}$$

From equations (13.1) and (13.2), $\mathcal{S}[\![(\emptyset, \hat{t}_{al})]\!]\phi = \mathcal{Q}[\![h(\emptyset)]\!]\phi$.

Inductive Hypothesis: Assume that for every sql statement $\hat{sql}_q{}'$ that is structurally strictly smaller than $\hat{sql}_q$, $\mathcal{S}[\![\hat{sql}_q{}' = (\hat{sql}', \hat{t}_{al})]\!]\phi = \mathcal{Q}[\![h(\hat{sql}_q{}' = (\hat{sql}', \hat{t}_{al}))]\!]\phi$.

Case ($\mathcal{S}[\![]\!]$ ChAllEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{t})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq \texttt{null}$, $\hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{true}$, and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}) = \texttt{true}$. By Rule ($\mathcal{S}[\![]\!]$ CHALLEX),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}} \tag{13.3}$$

By Rule (h SQL QUERY TRANSLATION), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. By Rule ($h_s$ CHAR ALSQ), $h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}, \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \cdot)$, and by Rule ($h_w$ SQL FROM - TABLE), $h_w(\hat{t}, \hat{t}_{al}) = (?\hat{t}\texttt{\_type}\ \texttt{type}\ \hat{t})$. Thus, $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, (\hat{t}\texttt{\_type}\ \texttt{type}\ \hat{t}))$

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Observe that when $c = \hat{t}$, $\dot{spsl} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\dot{v}_{al} =?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ we have $\texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{true}$ from our previous assumption (since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ from Inductive Hypothesis) , and $\dot{v}_{al} = c''\text{-}_{T_{id}}\dot{v}_{al_{suffix}}$, where $c'' = \hat{t}_{al}$ and $\dot{v}_{al_{suffix}} = \hat{cl}_{al}$. Thus, by Rule ($\mathcal{Q}[\![]\!]$ CHALLEX),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{Q}} \tag{13.4}$$

.

Since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$, from equations (13.3) and (13.4), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ChChEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{t})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq \texttt{null}$, $\hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{true}$, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}) = \texttt{false}$, $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}}))$, and $r_{\mathcal{S}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t})$. By Rule ($\mathcal{S}[\![]\!]$ ChChEx),

$$\begin{aligned}\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi =& \texttt{AddRoot(JoinGraphs(AddTableLabel(GetRootNodes(}\mathcal{S}_T[\![\hat{t}]\!]\{\phi\}), \hat{t}), \\ & \phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{aligned} \tag{13.5}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. By Rule ($h_s$ Char AlSQ), $h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}, \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \cdot)$, and by Rule ($h_w$ SQL From - Table), $h_w(\hat{t}, \hat{t}_{al}) = (?\hat{t}\_\texttt{type}\ \texttt{type}\ \hat{t})$. Thus, $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, (\hat{t}\_\texttt{type}\ \texttt{type}\ \hat{t}))$

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Observe that when $c = \hat{t}$, $\dot{spsl} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\dot{v}_{al} = ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ we have $\texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{true}$ and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), c) = \texttt{false}$ from our previous assumption (since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ from Inductive Hypothesis) , and $\dot{v}_{al} = c''\text{-}_{T_{id}}\dot{v}_{al_{suffix}}$, where $c'' = \hat{t}_{al}$ and $\dot{v}_{al_{suffix}} = \hat{cl}_{al}$. Further, since $\phi'_{\mathcal{S}} = \phi_{\mathcal{Q}}$' (by Inductive Hypothesis), $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}})) = \phi'_{\mathcal{Q}r}$ and $r_{\mathcal{S}} = r_{\mathcal{Q}}$. Thus, by Rule ($\mathcal{Q}[\![]\!]$ ChChEx),

$$\begin{aligned}\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi =& \texttt{AddRoot(JoinGraphs(AddTableLabel(GetRootNodes(}\mathcal{S}_T[\![c]\!]\{\phi\}), c), \\ & \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})\end{aligned} \tag{13.6}$$

.

Since $c = \hat{t}$, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$, and $r_{\mathcal{S}} = r_{\mathcal{Q}}$, from equations (13.5) and (13.6), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ChTblEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{t})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq$ `null`, $\hat{s} = \hat{c}$ `AS` $\hat{cl}_{al}$, `ElementExists`$(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) =$ `false`, `TableExists`(`GetRoot`$(\phi'_{\mathcal{S}}), \hat{t}) =$ `true`, and $\phi'_{\mathcal{S}r} =$ `RemoveRoot`$(\phi'_{\mathcal{S}},$ `GetRoot`$(\phi'_{\mathcal{S}}))$. By Rule ($\mathcal{S}[\![]\!]$ ChTblEx),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \texttt{CreateEdges}(\hat{c}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \texttt{GetRootNodes}(\phi'_{\mathcal{S}r})) \cup \phi'_{\mathcal{S}} \tag{13.7}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{c}$ `AS` $\hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (h_s(\hat{c}$ `AS` $\hat{cl}_{al}, \hat{t}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. By Rule ($h_s$ Char AlSQ), $h_s(\hat{c}$ `AS` $\hat{cl}_{al}, \hat{t}, \hat{t}_{al}) = (\hat{c}$ `AS` $?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \cdot)$, and by Rule ($h_w$ SQL From - Table), $h_w(\hat{t}, \hat{t}_{al}) = (?\hat{t}$`_type type` $\hat{t})$. Thus, $h((\hat{c}$ `AS` $\hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (\hat{c}$ `AS` $?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, (\hat{t}$`_type type` $\hat{t}))$

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Observe that when $c = \hat{t}$, $\dot{spsl} = \hat{c}$ `AS` $\hat{cl}_{al}$, $\dot{v}_{al} = ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\dot{v}_{al_{rf}} =$ `RemoveFirstChar`$(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ we have `ElementExists`$(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) =$ `false` and `TableExists`(`GetRoot`$(\phi'_{\mathcal{Q}}), c) =$ `true` from our previous assumption (since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ from Inductive Hypothesis) , and $\dot{v}_{al} = c''_{\text{-}T_{id}}\dot{v}_{al_{suffix}}$, where $c'' = \hat{t}_{al}$ and $\dot{v}_{al_{suffix}} = \hat{cl}_{al}$. Further, since $\phi'_{\mathcal{S}} = \phi_{\mathcal{Q}}$' (by Inductive Hypothesis), $\phi'_{\mathcal{S}r} =$ `RemoveRoot`$(\phi'_{\mathcal{S}},$ `GetRoot`$(\phi'_{\mathcal{S}})) = \phi'_{\mathcal{Q}r}$. Thus, by Rule ($\mathcal{Q}[\![]\!]$ ChTblEx),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \texttt{CreateEdges}(\hat{c}, \dot{v}_{al_{rf}}, \texttt{GetRootNodes}(\phi'_{\mathcal{Q}r})) \cup \phi'_{\mathcal{Q}} \tag{13.8}$$

.

Since $c = \hat{t}$, $\hat{s} = \hat{c}$, and $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$, from equations (13.7) and (13.8), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ChNoneEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{t})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq \texttt{null}$, $\hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{false}$, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}) = \texttt{false}$, $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}}))$, and $r_{\mathcal{S}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t})$. By Rule ($\mathcal{S}[\![]\!]$ ChNoneEx),

$$\begin{aligned}\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = &\texttt{AddRoot(JoinGraphs(CreateEdges(}\hat{c}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \texttt{AddTableLabel}\\ &\texttt{(GetRootNodes(}\mathcal{S}_T[\![\hat{t}]\!]\{\phi\}), \hat{t})), \phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{aligned} \tag{13.9}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. By Rule ($h_s$ Char AlSQ), $h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}, \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \cdot)$, and by Rule ($h_w$ SQL From - Table), $h_w(\hat{t}, \hat{t}_{al}) = (?\hat{t}\_\texttt{type type}\ \hat{t})$. Thus, $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, (\hat{t}\_\texttt{type type}\ \hat{t}))$

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Observe that when $c = \hat{t}$, $\dot{spsl} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\dot{v}_{al} =?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ we have $\texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{false}$ and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), c) = \texttt{false}$ from our previous assumption (since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ from Inductive Hypothesis) , and $\dot{v}_{al} = c''_{\text{-}T_{id}}\dot{v}_{al_{suffix}}$, where $c'' = \hat{t}_{al}$ and $\dot{v}_{al_{suffix}} = \hat{cl}_{al}$. Further, since $\phi'_{\mathcal{S}} = \phi_{\mathcal{Q}}$' (by Inductive Hypothesis), $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}})) = \phi'_{\mathcal{Q}r}$ and $r_{\mathcal{S}} = r_{\mathcal{Q}}$. Thus, by Rule ($\mathcal{Q}[\![]\!]$ ChNoneEx),

$$\begin{aligned}\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = &\texttt{AddRoot(JoinGraphs(CreateEdges(}\hat{c}, \dot{v}_{al_{rf}}, \texttt{AddTableLabel}\\ &\texttt{(GetRootNodes(}\mathcal{S}_T[\![c]\!]\{\phi\}), c)), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})\end{aligned} \tag{13.10}$$

.

Since $c = \hat{t}$, $\hat{s} = \hat{c}$, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$, and $r_{\mathcal{S}} = r_{\mathcal{Q}}$, from equations (13.9) and (13.10), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ChAllExDT) : Assume $\hat{sql} = \{(\hat{s}, \hat{dt})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq \texttt{null}$, $\hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$, $\texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{true}$, and

$\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}}) = \texttt{true}$. By Rule ($\mathcal{S}[\![]\!]$ ChAllExDT),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}} \tag{13.11}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. By Rule ($h_s$ Char AlSQ), $h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}, \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \cdot)$, and by Rule ($h_w$ SQL From - SubQuery), $h_w(\hat{dt}, \hat{t}_{al}) = h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}$. Thus, $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \dot{spq}_{sq})$. Also, observe that, in the case of sub-queries, where $\hat{t}_{alsq_{\hat{sql}}} \neq \texttt{null}$, the SELECT item translation rules, (h_s Char/Int/{T/DT}Column AlSQ), always prefix the sub-query alias, $\hat{t}_{alsq_{\hat{sql}}}$, to the alias of every corresponding SPARQL Query SELECT item. Thus $\hat{t}_{alsq\dot{spq}} = \texttt{GetSubQueryAlias}(h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}) = \hat{t}_{alsq_{\hat{sql}}}$.

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Observe that when $\hat{t}_{alsq_{\hat{sql}}} = \hat{t}_{alsq\dot{spq}}$, $\dot{spsl} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\dot{v}_{al} = ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ we have $\texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{true}$ and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), \hat{t}_{alsq\dot{spq}}) = \texttt{true}$ from our previous assumption (since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ from Inductive Hypothesis) , and $\dot{v}_{al} = c''_{\text{-}T_{id}}\dot{v}_{al_{suffix}}$, where $c'' = \hat{t}_{al}$ and $\dot{v}_{al_{suffix}} = \hat{cl}_{al}$. Thus, by Rule ($\mathcal{Q}[\![]\!]$ ChAllExSQ),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{Q}} \tag{13.12}$$

.

Since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$, from equations (13.11) and (13.12), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ChChExDT) : Assume $\hat{sql} = \{(\hat{s}, \hat{dt})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq \texttt{null}$, $\hat{s} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$ $\texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{true}$, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}}) = \texttt{false}$, $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}}))$, $r_{\mathcal{S}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}})$, $\mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi = \phi''_{\mathcal{S}}$, and $\texttt{RemoveTableLabels}(\texttt{RemoveRoot}($

$\phi''_{\mathcal{S}}, \texttt{GetRoot}(\phi''_{\mathcal{S}}))) = \phi_{\mathcal{S}_{dt}}$. By Rule ($\mathcal{S}[\![]\!]$ CHCHEXDT),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{AddTableLabel}(\texttt{GetRootNodes}(\mathcal{S}_T[\![\phi_{\mathcal{S}_{dt}}]\!]), \hat{t}_{alsq_{\hat{sql}}}), \phi'_{\mathcal{S}r}), r_{\mathcal{S}}) \tag{13.13}$$

By Rule (h SQL QUERY TRANSLATION), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. By Rule ($h_s$ CHAR ALSQ), $h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}, \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \cdot)$, and by Rule ($h_w$ SQL FROM - SUBQUERY), $h_w(\hat{dt}, \hat{t}_{al}) = h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}$. Thus, $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \dot{spq}_{sq})$. Also, observe that, in the case of sub-queries, where $\hat{t}_{alsq_{\hat{sql}}} \neq \texttt{null}$, the SELECT item translation rules, (h_s CHAR/INT/{T/DT}COLUMN ALSQ), always prefix the sub-query alias, $\hat{t}_{alsq_{\hat{sql}}}$, to the alias of every corresponding SPARQL Query SELECT item. Thus $\hat{t}_{alsq\dot{spq}} = \texttt{GetSubQueryAlias}(h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}) = \hat{t}_{alsq_{\hat{sql}}}$.

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ and $\mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi = \mathcal{Q}[\![h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$. Observe that when $\hat{t}_{alsq_{\hat{sql}}} = \hat{t}_{alsq\dot{spq}}$, $\dot{spsl} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\dot{v}_{al} = ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ we have $\texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{true}$ and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), \hat{t}_{alsq\dot{spq}}) = \texttt{false}$ from our previous assumption (since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ from Inductive Hypothesis) , and $\dot{v}_{al} = c''_{\text{-}T_{id}}\dot{v}_{al_{suffix}}$, where $c'' = \hat{t}_{al}$ and $\dot{v}_{al_{suffix}} = \hat{cl}_{al}$. Further, since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ and $\phi''_{\mathcal{S}} = \phi''_{\mathcal{Q}}$ (by Inductive Hypothesis), $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}})) = \phi'_{\mathcal{Q}r}$, $r_{\mathcal{S}} = r_{\mathcal{Q}}$, and $\phi_{\mathcal{S}_{dt}} = \phi_{\mathcal{Q}_{dt}}$, . Thus, by Rule ($\mathcal{Q}[\![]\!]$ CHCHEXSQ),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{AddTableLabel}(\texttt{GetRootNodes}(\phi_{\mathcal{Q}_{dt}}), \hat{t}_{alsq\dot{spq}}), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}}) \tag{13.14}$$

.

Since $\hat{t}_{alsq_{\hat{sql}}} = \hat{t}_{alsq\dot{spq}}$, $\phi_{\mathcal{S}_{dt}} = \phi_{\mathcal{Q}_{dt}}$, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$, and $r_{\mathcal{S}} = r_{\mathcal{Q}}$, from equations (13.13) and (13.14), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ChTblExDT) : Assume $\hat{sql} = \{(\hat{s}, \hat{dt})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq \texttt{null}$, $\hat{s} = \hat{c}$ `AS` $\hat{cl}_{al}$, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$, $\texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{false}$, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}}) = \texttt{true}$, and $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}}))$. By Rule ($\mathcal{S}[\![]\!]$ ChTblExDT),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \texttt{CreateEdges}(\hat{c}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \texttt{GetRootNodes}(\phi'_{\mathcal{S}r})) \cup \phi'_{\mathcal{S}} \tag{13.15}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. By Rule ($h_s$ Char AlSQ), $h_s(\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}, \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \cdot)$, and by Rule ($h_w$ SQL From - SubQuery), $h_w(\hat{dt}, \hat{t}_{al}) = h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}$. Thus, $h((\hat{c}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (\hat{c}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \dot{spq}_{sq})$. Also, observe that, in the case of sub-queries, where $\hat{t}_{alsq_{\hat{sql}}} \neq \texttt{null}$, the SELECT item translation rules, (h_s Char/Int/{T/DT}Column AlSQ), always prefix the sub-query alias, $\hat{t}_{alsq_{\hat{sql}}}$, to the alias of every corresponding SPARQL Query SELECT item. Thus $\hat{t}_{alsq_{\dot{spq}}} = \texttt{GetSubQueryAlias}(h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}) = \hat{t}_{alsq_{\hat{sql}}}$.

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Observe that when $\hat{t}_{alsq_{\hat{sql}}} = \hat{t}_{alsq_{\dot{spq}}}$, $\dot{spsl} = \hat{c}\ \texttt{AS}\ \hat{cl}_{al}$, $\dot{v}_{al} = ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ we have $\texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{false}$ and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), \hat{t}_{alsq_{\dot{spq}}}) = \texttt{true}$ from our previous assumption (since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ from Inductive Hypothesis) , and $\dot{v}_{al} = c''_{\text{-}T_{id}}\dot{v}_{al_{suffix}}$, where $c'' = \hat{t}_{al}$ and $\dot{v}_{al_{suffix}} = \hat{c}$. Further, since $\phi'_{\mathcal{S}} = \phi_{\mathcal{Q}}$' (by Inductive Hypothesis), $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}})) = \phi'_{\mathcal{Q}r}$. Thus, by Rule ($\mathcal{Q}[\![]\!]$ ChTblExSQ),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \texttt{CreateEdges}(\hat{c}, \dot{v}_{al_{rf}}, \texttt{GetRootNodes}(\phi'_{\mathcal{Q}r})) \cup \phi'_{\mathcal{Q}} \tag{13.16}$$

.

Since $\hat{t}_{alsq_{\hat{sql}}} = \hat{t}_{alsq_{\dot{spq}}}$, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$, and $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$, from equations (13.15) and (13.16), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ChNoneExDT) : Assume $\hat{sql} = \{(\hat{s}, \hat{dt})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq$ `null`, $\hat{s} = \hat{c}\,\texttt{AS}\,\hat{cl}_{al}$, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$, $\texttt{ElementExists}(\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \phi'_{\mathcal{S}}) = \texttt{false}$, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}}) = \texttt{false}$, $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}}))$, and $r_{\mathcal{S}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}})$, $\mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi = \phi''_{\mathcal{S}}$, and $\texttt{RemoveTableLabels}(\texttt{RemoveRoot}(\phi''_{\mathcal{S}}, \texttt{GetRoot}(\phi''_{\mathcal{S}}))) = \phi_{\mathcal{S}_{dt}}$. By Rule ($\mathcal{S}[\![]\!]$ ChNone-ExDT),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{CreateEdges}(\hat{c}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \texttt{AddTableLabel}(\texttt{GetRootNodes}(\mathcal{S}_T[\![\phi_{\mathcal{S}_{dt}}]\!]), \hat{t}_{alsq_{\hat{sql}}})), \phi'_{\mathcal{S}r}), r_{\mathcal{S}}) \tag{13.17}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{c}\,\texttt{AS}\,\hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (h_s(\hat{c}\,\texttt{AS}\,\hat{cl}_{al}, \hat{dt}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. By Rule ($h_s$ Char AlSQ), $h_s(\hat{c}\,\texttt{AS}\,\hat{cl}_{al}, \hat{dt}, \hat{t}_{al}) = (\hat{c}\,\texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \cdot)$, and by Rule ($h_w$ SQL From - SubQuery), $h_w(\hat{dt}, \hat{t}_{al}) = h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}$. Thus, $h((\hat{c}\,\texttt{AS}\,\hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (\hat{c}\,\texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \dot{spq}_{sq})$. Also, observe that, in the case of sub-queries, where $\hat{t}_{alsq_{\hat{sql}}} \neq$ `null`, the SELECT item translation rules, (h_s Char/Int/{T/DT}Column AlSQ), always prefix the sub-query alias, $\hat{t}_{alsq_{\hat{sql}}}$, to the alias of every corresponding SPARQL Query SELECT item. Thus $\hat{t}_{alsq_{\dot{spq}}} = \texttt{GetSubQueryAlias}(h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}) = \hat{t}_{alsq_{\hat{sql}}}$.

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ and $\mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi = \mathcal{Q}[\![h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$. Observe that when $\hat{t}_{alsq_{\hat{sql}}} = \hat{t}_{alsq_{\dot{spq}}}$, $\dot{spsl} = \hat{c}\,\texttt{AS}\,\hat{cl}_{al}$, $\dot{v}_{al} = ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$ we have $\texttt{ElementExists}(\dot{v}_{al_{rf}}, \phi'_{\mathcal{Q}}) = \texttt{false}$ and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), \hat{t}_{alsq_{\dot{spq}}}) = \texttt{false}$ from our previous assumption (since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ from Inductive Hypothesis) , and $\dot{v}_{al} = c''\text{-}_{T_{id}}\dot{v}_{al_{suffix}}$, where $c'' = \hat{t}_{al}$ and $\dot{v}_{al_{suffix}} = \hat{cl}_{al}$. Further, since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ and $\phi''_{\mathcal{S}} = \phi''_{\mathcal{Q}}$ (by Inductive Hypothesis), $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}})) = \phi'_{\mathcal{Q}r}$, $r_{\mathcal{S}} = r_{\mathcal{Q}}$, and $\phi_{\mathcal{S}_{dt}} = \phi_{\mathcal{Q}_{dt}}$, .

Thus, by Rule ($\mathcal{Q}[\![]\!]$ ChNoneExSQ),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{CreateEdges}(\hat{c}, \dot{v}_{al_{rf}}, \texttt{AddTableLabel}(\texttt{GetRootNodes}(\phi_{\mathcal{Q}_{dt}}), \hat{t}_{alsqs\dot{p}q})), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}}) \tag{13.18}$$

.

Since $\hat{t}_{alsq\hat{sql}} = \hat{t}_{alsqs\dot{p}q}$, $\phi_{\mathcal{S}_{dt}} = \phi_{\mathcal{Q}_{dt}}$, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$, and $r_{\mathcal{S}} = r_{\mathcal{Q}}$, from equations (13.17) and (13.18), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ClTMchEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{t})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq \texttt{null}$, $\hat{s} = \hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}$, $\hat{t} = \hat{t}'$, $\phi_{ST} = \mathcal{S}_T[\![\hat{t}]\!]\{\phi\}$, and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}) = \texttt{true}$. By Rule ($\mathcal{S}[\![]\!]$ ColAl TMatchExists),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \texttt{AttachEdge}(\texttt{ReplaceLabel}(\texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, \texttt{GetRootNodes}(\phi_{ST}), \phi_{ST}), \hat{t}_{\text{-}T_{id}}\hat{cl}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}), \times\texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}})))) \cup \phi'_{\mathcal{S}} \tag{13.19}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = s\dot{p}q'$, and $h((\hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}), \hat{t}_{al})$ $= (h_s(\hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. Since $\hat{t}$ is a table, i.e., $\texttt{IsTable}(\hat{t}) = \texttt{true}$, by Rule ($h_s$ TColumn AlSQ), $h_s(\hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}, \hat{t}_{al}) = (?\hat{t}'_{\text{-}T_{id}}\hat{cl}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, (?\hat{t}'_{\text{-}T_{id}}\texttt{type}\ \hat{cl}\ ?\hat{t}'_{\text{-}T_{id}}\hat{cl}))$, and by Rule ($h_w$ SQL From - Table), $h_w(\hat{t}, \hat{t}_{al}) = (?\hat{t}_{\text{-}T_{id}}\texttt{type type}\ \hat{t})$. Thus, $h((\hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (?\hat{t}'_{\text{-}T_{id}}\hat{cl}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \{(?\hat{t}'_{\text{-}T_{id}}\texttt{type}\ \hat{cl}\ ?\hat{t}'_{\text{-}T_{id}}\hat{cl}),$ $(?\hat{t}_{\text{-}T_{id}}\texttt{type type}\ \hat{t})\})$

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Substituting c for $\hat{t}'$, c' for $\hat{t}$, c'' for $\hat{t}_{al}$, $\dot{p}$ for $\hat{cl}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al})$ for $\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, we have, $c = c'$, and $\phi_{QT} = \mathcal{S}_T[\![c]\!]\{\phi\} = \phi_{ST}$. Further, by inductive hypothesis, $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$, and therefore, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), c) = \texttt{true}$. Thus, since $\hat{t}_{al} \neq \texttt{null}$, by Rule

($\mathcal{Q}[\![]\!]$ CLTWMCHEX),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \texttt{AttachEdge}(\texttt{ReplaceLabel}(\texttt{ExtColEdges}(c_{\text{-}T_{id}}\dot{p}, \texttt{GetRootNodes}(\phi_{\mathcal{Q}T}), \phi_{\mathcal{Q}T}), c_{\text{-}T_{id}}\dot{p}, \dot{v}_{al_{rf}}) \times \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, \texttt{GetRoot}(\phi'_{\mathcal{Q}})))) \cup \phi'_{\mathcal{Q}} \tag{13.20}$$

.

Since $c = \hat{t}$ and, therefore, $\phi_{\mathcal{S}T} = \phi_{\mathcal{Q}T}$, $\dot{p} = \hat{cl}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ by inductive hypothesis, from equations (13.19) and (13.20), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ClTMchNoEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{t})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{t}_{al} \neq \texttt{null}$, $\hat{s} = \hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}$, $\hat{t} = \hat{t}'$, $\phi_{\mathcal{S}T} = \mathcal{S}_T[\![\hat{t}]\!]\{\phi\}$, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}) = \texttt{false}$, $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}}))$, and $r_{\mathcal{S}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t})$. By Rule ($\mathcal{S}[\![]\!]$ CLTMCHNOEX),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{ReplaceLabel}(\texttt{ExtColEdges}(\hat{t}_{\text{-}T_{id}}\hat{cl}, \texttt{GetRootNodes}(\phi_{\mathcal{S}T}), \phi_{\mathcal{S}T}), \hat{t}_{\text{-}T_{id}}\hat{cl}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}), \phi'_{\mathcal{S}r}), r_{\mathcal{S}}) \tag{13.21}$$

By Rule (h SQL QUERY TRANSLATION), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (h_s(\hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. Since $\hat{t}$ is a table, i.e., $\texttt{IsTable}(\hat{t}) = \texttt{true}$, by Rule ($h_s$ TCOLUMN ALSQ), $h_s(\hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}, \hat{t}_{al}) = (?\hat{t}'_{\text{-}T_{id}}\hat{cl}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, (?\hat{t}'_{\text{-}T_{id}}\texttt{type}\ \hat{cl}\ ?\hat{t}'_{\text{-}T_{id}}\hat{cl}))$, and by Rule ($h_w$ SQL FROM - TABLE), $h_w(\hat{t}, \hat{t}_{al}) = (?\hat{t}_{\text{-}T_{id}}\texttt{type}\ \texttt{type}\ \hat{t})$. Thus, $h((\hat{t}'.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (?\hat{t}'_{\text{-}T_{id}}\hat{cl}\ \texttt{AS}\ ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \{(?\hat{t}'_{\text{-}T_{id}}\texttt{type}\ \hat{cl}\ ?\hat{t}'_{\text{-}T_{id}}\hat{cl}), (?\hat{t}_{\text{-}T_{id}}\texttt{type}\ \texttt{type}\ \hat{t})\})$

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Substituting c for $\hat{t}'$, c' for $\hat{t}$, $\dot{p}$ for $\hat{cl}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al})$ for $\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, we have, $c = c'$ and $\phi_{\mathcal{Q}T} = \mathcal{S}_T[\![c]\!]\{\phi\} = \phi_{\mathcal{S}T}$. Further, by inductive hypothesis, $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$, and therefore,

$\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), c) = \texttt{false}$, $\phi'_{\mathcal{Q}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, \texttt{GetRoot}(\phi'_{\mathcal{Q}})) = \phi'_{\mathcal{S}r}$, and $r_{\mathcal{Q}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), c) = r_{\mathcal{S}}$. Thus, since $\hat{t}_{al} \neq \texttt{null}$, by Rule ($\mathcal{Q}[\![]\!]$ CLTWMCHNOEX),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{ReplaceLabel}(\texttt{ExtColEdges}(c_{\text{-}T_{id}}\dot{p}, \texttt{GetRootNodes}(\phi_{\mathcal{Q}T}), \phi_{\mathcal{Q}T}), c_{\text{-}T_{id}}\dot{p}, \dot{v}_{al_{rf}}), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}}) \tag{13.22}$$

.

Since $c = \hat{t}$ and, therefore, $\phi_{\mathcal{S}T} = \phi_{\mathcal{Q}T}$, $\dot{p} = \hat{cl}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ by inductive hypothesis, and therefore, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$ and $r_{\mathcal{S}} = r_{\mathcal{Q}}$, from equations (13.21) and (13.22), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ClTNoMchEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{t})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{s} = \hat{t}'.\hat{cl}$ $\texttt{AS}$ $\hat{cl}_{al}$, $\hat{t} \neq \hat{t}'$, and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}) = \texttt{true}$. By Rule ($\mathcal{S}[\![]\!]$ CLTNOMCHEX),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}} \tag{13.23}$$

By Rule (h SQL QUERY TRANSLATION), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{t}'.\hat{cl}$ $\texttt{AS}$ $\hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (h_s(\hat{t}'.\hat{cl}$ $\texttt{AS}$ $\hat{cl}_{al}, \hat{t}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. Since $\hat{t}$ is a table, i.e., $\texttt{IsTable}(\hat{t}) = \texttt{true}$, by Rule ($h_s$ TCOLUMN ALSQ), $h_s(\hat{t}'.\hat{cl}$ $\texttt{AS}$ $\hat{cl}_{al}, \hat{t}, \hat{t}_{al}) = (?\hat{t}'_{\text{-}T_{id}}\hat{cl}$ $\texttt{AS}$ $?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, (?\hat{t}'_{\text{-}T_{id}}\texttt{type}\ \hat{cl}$ $?\hat{t}'_{\text{-}T_{id}}\hat{cl}))$, and by Rule ($h_w$ SQL FROM - TABLE), $h_w(\hat{t}, \hat{t}_{al}) = (?\hat{t}_{\text{-}T_{id}}\texttt{type}\ \texttt{type}\ \hat{t})$. Thus, $h((\hat{t}'.\hat{cl}$ $\texttt{AS}$ $\hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (?\hat{t}'_{\text{-}T_{id}}\hat{cl}$ $\texttt{AS}$ $?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \{(?\hat{t}'_{\text{-}T_{id}}\texttt{type}\ \hat{cl}$ $?\hat{t}'_{\text{-}T_{id}}\hat{cl}),$
$(?\hat{t}_{\text{-}T_{id}}\texttt{type}\ \texttt{type}\ \hat{t})\})$

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Substituting c for $\hat{t}'$, c' for $\hat{t}$, we have, $c \neq c'$. Further, by inductive hypothesis, $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$, and therefore, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), c) = \texttt{true}$. Thus, substituting $\dot{p}$ for $\hat{cl}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al})$ for $\hat{cl}_{al}$, we have, by Rule ($\mathcal{Q}[\![]\!]$ CLTWNOMCHEX),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{Q}} \tag{13.24}$$

.

Since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ by inductive hypothesis, from equations (13.23) and (13.24), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ClTNoMchNoEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{t})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{s} = \hat{t}.\hat{cl}$ `AS` $\hat{cl}_{al}$, $\hat{t} \neq \hat{t}'$, and `TableExists(GetRoot(`$\phi'_{\mathcal{S}}$`),` $\hat{t}$`)` = `false`, $\phi'_{\mathcal{S}r}$ = `RemoveRoot(`$\phi'_{\mathcal{S}}$, `GetRoot(`$\phi'_{\mathcal{S}}$`))`, $r_{\mathcal{S}}$ = `AddTable(GetRoot(`$\phi'_{\mathcal{S}}$`),` $\hat{t}$`)`, and $\phi_{\mathcal{S}T} = \mathcal{S}_T[\![\hat{t}]\!]\{\phi\}$. By Rule ($\mathcal{S}[\![]\!]$ ClTNoMchEx),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \texttt{AddRoot(JoinGraphs(AddTableLabel(GetRootNodes(}\phi_{\mathcal{S}T}\texttt{)}, \hat{t}\texttt{)}, \phi'_{\mathcal{S}r}\texttt{)}, r_{\mathcal{S}}\texttt{)} \tag{13.25}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{t}'.\hat{cl}$ `AS` $\hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (h_s(\hat{t}'.\hat{cl}$ `AS` $\hat{cl}_{al}, \hat{t}, \hat{t}_{al}), h_w(\hat{t}, \hat{t}_{al}))$. Since $\hat{t}$ is a table, i.e., `IsTable(`$\hat{t}$`)` = `true`, by Rule (h_s TColumn AlSQ), $h_s(\hat{t}'.\hat{cl}$ `AS` $\hat{cl}_{al}, \hat{t}, \hat{t}_{al}) = (?\hat{t}'_{\text{-}T_{id}}\hat{cl}$ `AS` $?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, (?\hat{t}'_{\text{-}T_{id}}$`type` $\hat{cl}$ $?\hat{t}'_{\text{-}T_{id}}\hat{cl}))$, and by Rule ($h_w$ SQL From - Table), $h_w(\hat{t}, \hat{t}_{al}) = (?\hat{t}_{\text{-}T_{id}}$`type type` $\hat{t})$. Thus, $h((\hat{t}'.\hat{cl}$ `AS` $\hat{cl}_{al}, \hat{t}), \hat{t}_{al}) = (?\hat{t}'_{\text{-}T_{id}}\hat{cl}$ `AS` $?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \{(?\hat{t}'_{\text{-}T_{id}}$`type` $\hat{cl}$ $?\hat{t}'_{\text{-}T_{id}}\hat{cl}),$
$(?\hat{t}_{\text{-}T_{id}}$`type type` $\hat{t})\})$

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Substituting c for $\hat{t}'$, c' for $\hat{t}$, we have, $c \neq c'$ and $\phi_{\mathcal{Q}T} = \mathcal{S}_T[\![c']\!]\{\phi\} = \phi_{\mathcal{S}T}$. Further, by inductive hypothesis, $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$, and therefore, `TableExists(GetRoot(`$\phi'_{\mathcal{Q}}$`),` c`)` = `false`, $\phi'_{\mathcal{Q}r}$ = `RemoveRoot(`$\phi'_{\mathcal{Q}}$`,GetRoot(`$\phi'_{\mathcal{Q}}$`))` = $\phi'_{\mathcal{S}r}$, and $r_{\mathcal{Q}}$ = `AddTable(GetRoot(`$\phi'_{\mathcal{Q}}$`),` c`)` = $r_{\mathcal{S}}$. Thus, substituting $\dot{p}$ for $\hat{cl}$, we have, by Rule ($\mathcal{Q}[\![]\!]$ ClTwNoMchNoEx),

$$\begin{aligned}\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi =&\texttt{AddRoot(JoinGraphs(AddTableLabel(GetRootNodes(}\phi_{\mathcal{Q}T}\texttt{)}, c'\texttt{)}, \phi'_{\mathcal{Q}r}\texttt{)},\\ &r_{\mathcal{Q}}\texttt{)}\end{aligned} \tag{13.26}$$

.

Since $c' = \hat{t}$ and, therefore, $\phi_{\mathcal{S}T} = \phi_{\mathcal{Q}T}$, and since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ by inductive hypothesis and, therefore, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$ and $r_{\mathcal{S}} = r_{\mathcal{Q}}$, from equations (13.25) and (13.26), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ClDTMchEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{dt})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{s} = \hat{t}.\hat{cl}$ `AS` $\hat{cl}_{al}$, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$, $\hat{t} = \hat{t}_{alsq_{\hat{sql}}}$, $\phi''_{\mathcal{S}} = \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$, and `TableExists(GetRoot(`$\phi'_{\mathcal{S}}$`)` , $\hat{t}_{alsq_{\hat{sql}}}$`)` $=$ `true`. By Rule ($\mathcal{S}[\![]\!]$ ClDTMchEx),

$$\begin{aligned}\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi =& \texttt{AttachEdge(ReplaceLabel(ExtColEdges(}\hat{t}_{\text{-}T_{id}}\hat{cl}, \texttt{GetRootNodes}\\ &(\texttt{RemoveRoot(}\phi''_{\mathcal{S}}, \texttt{GetRoot(}\phi''_{\mathcal{S}}\texttt{))), }\phi''_{\mathcal{S}}\texttt{), }\hat{t}_{\text{-}T_{id}}\hat{cl}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}\texttt{)}\\ &\times \texttt{GetRootNodes(RemoveRoot(}\phi'_{\mathcal{S}}, \texttt{GetRoot(}\phi'_{\mathcal{S}}\texttt{)))) } \cup \phi'_{\mathcal{S}}\end{aligned} \tag{13.27}$$

By Rule (h SQL Query Translation), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{t}.\hat{cl}$ `AS` $\hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (h_s(\hat{t}.\hat{cl}$ `AS` $\hat{cl}_{al}, \hat{dt}, \hat{t}_{al}), h_w(\hat{dt}, \hat{t}_{al}))$. Since $\hat{dt}$ is not a table, i.e., `IsTable(`$\hat{dt}$`)` $=$ `false`, by Rule (h_s DTColumn AlSQ), $h_s(\hat{t}.\hat{cl}$ `AS` $\hat{cl}_{al}, \hat{dt}, \hat{t}_{al}) = ?\hat{t}_{\text{-}T_{id}}\hat{cl}$ `AS` $?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and by Rule ($h_w$ SQL From - SubQuery), $h_w(\hat{dt}, \hat{t}_{al}) = h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}$. Thus, $h((\hat{t}.\hat{cl}$ `AS` $\hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (?\hat{t}_{\text{-}T_{id}}\hat{cl}$ `AS` $?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \dot{spq}_{sq})$. Also, observe that, in the case of sub-queries, where $\hat{t}_{alsq_{\hat{sql}}} \neq$ `null`, the SELECT item translation rules, (h_s Char/Int/{T/DT}Column AlSQ), always prefix the sub-query alias, $\hat{t}_{alsq_{\hat{sql}}}$, to the alias of every corresponding SPARQL Query SELECT item. Thus $\hat{t}_{alsq\dot{spq}} =$ `GetSubQueryAlias(` $h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}$`)` $= \hat{t}_{alsq_{\hat{sql}}}$.

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ and $\mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi = \mathcal{Q}[\![h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$. Substituting c for $\hat{t}$, and $\hat{t}_{alsq\dot{spq}}$ for $\hat{t}_{alsq_{\hat{sql}}}$, we have, $c = \hat{t}_{alsq\dot{spq}}$. Further, by inductive hypothesis, $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ and $\phi''_{\mathcal{S}} = \phi''_{\mathcal{Q}}$, and therefore, `TableExists(` `GetRoot(`$\phi'_{\mathcal{Q}}$`)`, $\hat{t}_{alsq\dot{spq}}$`)` $=$ `true`. Thus, since $\dot{v}_{al_{rf}} =$ `RemoveFirstChar(`$\dot{v}_{al}$`)` $= \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, substituting $\dot{p}$ for $\hat{cl}$, we have, by Rule ($\mathcal{Q}[\![]\!]$ CLSQMCHEX),

$$\begin{aligned}\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = &\texttt{AttachEdge(ReplaceLabel(ExtColEdges(}c_{\text{-}T_{id}}\dot{p}, \texttt{GetRootNodes}\\ &(\texttt{RemoveRoot}(\phi''_{\mathcal{Q}}, \texttt{GetRoot}(\phi''_{\mathcal{Q}}))), \phi''_{\mathcal{Q}}), c_{\text{-}T_{id}}\dot{p}, \dot{v}_{al_{rf}})\\ &\times \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, \texttt{GetRoot}(\phi'_{\mathcal{Q}})))) \cup \phi'_{\mathcal{Q}}\end{aligned} \tag{13.28}$$

.

Since $c = \hat{t}$, $\dot{p} = \hat{cl}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ and $\phi''_{\mathcal{S}} = \phi''_{\mathcal{Q}}$ by inductive hypothesis, from equations (13.27) and (13.28), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ClDTMchNoEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{dt})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{s} = \hat{t}.\hat{cl}$ AS $\hat{cl}_{al}$, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$, $\hat{t} = \hat{t}_{alsq_{\hat{sql}}}$, $\phi''_{\mathcal{S}} = \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}}) = \texttt{false}$, $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}}))$, and $r_{\mathcal{S}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}})$. By Rule ($\mathcal{S}[\![]\!]$ CLDTMCHNOEX),

$$\begin{aligned}\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = &\texttt{AddRoot(JoinGraphs(ReplaceLabel(ExtColEdges(}\hat{t}_{\text{-}T_{id}}\hat{cl},\\ &\texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi''_{\mathcal{S}}, \texttt{GetRoot}(\phi''_{\mathcal{S}}))), \phi''_{\mathcal{S}}), \hat{t}_{\text{-}T_{id}}\hat{cl}, \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}),\\ &\phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{aligned} \tag{13.29}$$

By Rule (h SQL QUERY TRANSLATION), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{t}.\hat{cl}$ AS $\hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (h_s(\hat{t}.\hat{cl}$ AS $\hat{cl}_{al}, \hat{dt}, \hat{t}_{al}), h_w(\hat{dt}, \hat{t}_{al}))$. Since $\hat{dt}$ is not a table, i.e., $\texttt{IsTable}(\hat{dt}) = \texttt{false}$, by Rule ($h_s$ DTCOLUMN ALSQ), $h_s(\hat{t}.\hat{cl}$ AS $\hat{cl}_{al}, \hat{dt}, \hat{t}_{al}) = ?\hat{t}_{\text{-}T_{id}}\hat{cl}$ AS ? $\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and by Rule ($h_w$ SQL FROM - SUBQUERY), $h_w(\hat{dt}, \hat{t}_{al}) = h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}$. Thus, $h((\hat{t}.\hat{cl}$ AS $\hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (?\hat{t}_{\text{-}T_{id}}\hat{cl}$ AS $?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}, \dot{spq}_{sq})$. Also, observe that, in the case of sub-queries, where $\hat{t}_{alsq_{\hat{sql}}} \neq \texttt{null}$, the SELECT item translation rules, (h_s CHAR/INT/

{T/DT}COLUMN ALSQ), always prefix the sub-query alias, $\hat{t}_{alsq_{\hat{sql}}}$, to the alias of every corresponding SPARQL Query SELECT item. Thus $\hat{t}_{alsq\dot{spq}} = \texttt{GetSubQueryAlias}(h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}) = \hat{t}_{alsq_{\hat{sql}}}$.

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ and $\mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi = \mathcal{Q}[\![h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$. Substituting c for $\hat{t}$, $\hat{t}_{alsq\dot{spq}}$ for $\hat{t}_{alsq_{\hat{sql}}}$, we have $c = \hat{t}_{alsq\dot{spq}}$. Further, by inductive hypothesis, $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ and $\phi''_{\mathcal{S}} = \phi''_{\mathcal{Q}}$, and therefore, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), \hat{t}_{alsq\dot{spq}}) = \texttt{false}$, $\phi'_{\mathcal{Q}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, \texttt{GetRoot}(\phi'_{\mathcal{Q}})) = \phi'_{\mathcal{S}r}$, and $r_{\mathcal{Q}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), \hat{t}_{alsq\dot{spq}}) = r_{\mathcal{S}}$. Thus, since $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, substituting $\dot{p}$ for $\hat{cl}$, we have, by Rule ($\mathcal{Q}[\![]\!]$ CLSQMCHNOEX),

$$\begin{aligned}\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi =& \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{ReplaceLabel}(\texttt{ExtColEdges}(c_{\text{-}T_{id}}\dot{p},\\ & \texttt{GetRootNodes}(\texttt{RemoveRoot}(\phi''_{\mathcal{Q}}, \texttt{GetRoot}(\phi''_{\mathcal{Q}}))), \phi''_{\mathcal{Q}}), c_{\text{-}T_{id}}\dot{p}, \dot{v}_{al_{rf}}),\\ & \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}})\end{aligned} \tag{13.30}$$

.

Since $c = \hat{t}$, $\dot{p} = \hat{cl}$, and $\dot{v}_{al_{rf}} = \texttt{RemoveFirstChar}(\dot{v}_{al}) = \hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$, and $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ and $\phi''_{\mathcal{S}} = \phi''_{\mathcal{Q}}$ by inductive hypothesis, and therefore, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$ and $r_{\mathcal{S}} = r_{\mathcal{Q}}$, from equations (13.29) and (13.30), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ClDTNoMchEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{dt})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{s} = \hat{t}.\hat{cl}$ AS $\hat{cl}_{al}$, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$, $\hat{t} \neq \hat{t}_{alsq_{\hat{sql}}}$, and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}}) = \texttt{true}$. By Rule ($\mathcal{S}[\![]\!]$ CLDTNOMCHEX),

$$\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}} \tag{13.31}$$

By Rule (h SQL QUERY TRANSLATION), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{t}.\hat{cl}$ AS $\hat{cl}_{al}, \hat{dt}), \hat{t}_{al})$ $= (h_s(\hat{t}.\hat{cl}$ AS $\hat{cl}_{al}, \hat{dt}, \hat{t}_{al}), h_w(\hat{dt}, \hat{t}_{al}))$. Since $\hat{dt}$ is not a table, i.e., $\texttt{IsTable}(\hat{dt}) = \texttt{false}$, by Rule ($h_s$ DTCOLUMN AL/ALSQ), $h_s(\hat{t}.\hat{cl}$ AS $\hat{cl}_{al}, \hat{dt}, \hat{t}_{al})$ $=?\hat{t}_{\text{-}T_{id}}\hat{cl}$ AS $?\dot{v}_{al}$, where

$\dot{v}_{al} = ?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$. By Rule ($h_w$ SQL FROM - SUBQUERY), $h_w(\hat{dt}, \hat{t}_{al}) = h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}$. Thus, $h((\hat{t}.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (?\hat{t}_{\text{-}T_{id}}\hat{cl}\ \texttt{AS}\ ?\dot{v}_{al}, \dot{spq}_{sq})$. Also, observe that, in the case of sub-queries, where $\hat{t}_{alsq_{\hat{sql}}} \neq \texttt{null}$, the SELECT item translation rules, (h_s CHAR/INT/{T/DT}COLUMN ALSQ), always prefix the sub-query alias, $\hat{t}_{alsq_{\hat{sql}}}$, to the alias of every corresponding SPARQL Query SELECT item. Thus $\hat{t}_{alsq_{\dot{spq}}} = \texttt{GetSubQueryAlias}(h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}) = \hat{t}_{alsq_{\hat{sql}}}$.

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ Substituting c for $\hat{t}$, $\hat{t}_{alsq_{\dot{spq}}}$ for $\hat{t}_{alsq_{\hat{sql}}}$, we have, $c \neq \hat{t}_{alsq_{\dot{spq}}}$. Further, by inductive hypothesis, $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$, and therefore, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), \hat{t}_{alsq_{\dot{spq}}}) = \texttt{true}$. Thus, by Rule ($\mathcal{Q}[\![]\!]$ CLSQNOMCHEX),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \phi'_{\mathcal{Q}} \tag{13.32}$$

.

Since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ by inductive hypothesis, from equations (13.31) and (13.32), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

Case ($\mathcal{S}[\![]\!]$ ClDTNoMchNoEx) : Assume $\hat{sql} = \{(\hat{s}, \hat{dt})\} \uplus \hat{sql}'$, $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \phi'_{\mathcal{S}}$, $\hat{s} = \hat{t}.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}$, $\hat{dt} = (\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})$, $\hat{t} \neq \hat{t}_{alsq_{\hat{sql}}}$, and $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}}) = \texttt{false}$, $\phi'_{\mathcal{S}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{S}}, \texttt{GetRoot}(\phi'_{\mathcal{S}}))$, $r_{\mathcal{S}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{S}}), \hat{t}_{alsq_{\hat{sql}}})$, $\phi''_{\mathcal{S}} = \mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$, and $\texttt{RemoveTableLabels}(\texttt{RemoveRoot}(\phi''_{\mathcal{S}}, \texttt{GetRoot}(\phi''_{\mathcal{S}}))) = \phi_{\mathcal{S}_{dt}}$. By Rule ($\mathcal{S}[\![]\!]$ CLDTNOMCHNOEX),

$$\begin{aligned}\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi =& \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{AddTableLabel}(\texttt{GetRootNodes}(\phi_{\mathcal{S}_{dt}}), \hat{t}_{alsq_{\hat{sql}}}),\\ &\phi'_{\mathcal{S}r}), r_{\mathcal{S}})\end{aligned} \tag{13.33}$$

By Rule (h SQL QUERY TRANSLATION), $h(\hat{sql}', \hat{t}_{al}) = \dot{spq}'$, and $h((\hat{t}.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (h_s(\hat{t}.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}, \hat{t}_{al}), h_w(\hat{dt}, \hat{t}_{al}))$. Since $\hat{dt}$ is not a table, i.e., $\texttt{IsTable}(\hat{dt}) = \texttt{false}$, by Rule ($h_s$ DTColumn Al/AlSQ), $h_s(\hat{t}.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}, \hat{t}_{al})\ =?\hat{t}_{\text{-}T_{id}}\hat{cl}\ \texttt{AS}\ ?\dot{v}_{al}$, where $\dot{v}_{al} =?\hat{t}_{al\text{-}T_{id}}\hat{cl}_{al}$. By Rule ($h_w$ SQL From - SubQuery), $h_w(\hat{dt}, \hat{t}_{al}) = h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}$. Thus, $h((\hat{t}.\hat{cl}\ \texttt{AS}\ \hat{cl}_{al}, \hat{dt}), \hat{t}_{al}) = (?\hat{t}_{\text{-}T_{id}}\hat{cl}\ \texttt{AS}\ ?\dot{v}_{al}, \dot{spq}_{sq})$. Also, observe that, in the case of sub-queries, where $\hat{t}_{alsq_{\hat{sql}}} \neq \texttt{null}$, the SELECT item translation rules, (h_s Char/Int/{T/DT}Column AlSQ), always prefix the sub-query alias, $\hat{t}_{alsq_{\hat{sql}}}$, to the alias of every corresponding SPARQL Query SELECT item. Thus $\hat{t}_{alsq_{\dot{spq}}} = \texttt{GetSubQueryAlias}(h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}}) = \dot{spq}_{sq}) = \hat{t}_{alsq_{\hat{sql}}}$.

By Inductive Hypothesis, we have $\mathcal{S}[\![\hat{sql}', \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}', \hat{t}_{al})]\!]\phi$ and $\mathcal{S}[\![(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi = \mathcal{Q}[\![h(\hat{sql}_{sq}, \hat{t}_{alsq_{\hat{sql}}})]\!]\phi$. Substituting c for $\hat{t}$, $\hat{t}_{alsq_{\dot{spq}}}$ for $\hat{t}_{alsq_{\hat{sql}}}$, we have $c \neq \hat{t}_{alsq_{\dot{spq}}}$. Further, by inductive hypothesis, $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ and $\phi''_{\mathcal{S}} = \phi''_{\mathcal{Q}}$, and therefore, $\texttt{TableExists}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), \hat{t}_{alsq_{\dot{spq}}}) = \texttt{false}$, $\phi'_{\mathcal{Q}r} = \texttt{RemoveRoot}(\phi'_{\mathcal{Q}}, \texttt{GetRoot}(\phi'_{\mathcal{Q}})) = \phi'_{\mathcal{S}r}$, $r_{\mathcal{Q}} = \texttt{AddTable}(\texttt{GetRoot}(\phi'_{\mathcal{Q}}), c) = r_{\mathcal{S}}$, and $\phi_{\mathcal{Q}_{dt}} = \texttt{RemoveTableLabels}(\texttt{RemoveRoot}(\phi''_{\mathcal{Q}}, \texttt{GetRoot}(\phi''_{\mathcal{Q}}))) = \phi_{\mathcal{S}_{dt}}$. Thus, by Rule ($\mathcal{Q}[\![]\!]$ ClSQNoMchNoEx),

$$\mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi = \texttt{AddRoot}(\texttt{JoinGraphs}(\texttt{AddTableLabel}(\texttt{GetRootNodes}(\phi_{\mathcal{Q}_{dt}}), \hat{t}_{alsq_{\dot{spq}}}), \phi'_{\mathcal{Q}r}), r_{\mathcal{Q}}) \tag{13.34}$$

.

Since $\phi'_{\mathcal{S}} = \phi'_{\mathcal{Q}}$ by inductive hypothesis and, therefore, $\phi'_{\mathcal{S}r} = \phi'_{\mathcal{Q}r}$ and $r_{\mathcal{S}} = r_{\mathcal{Q}}$, and $\phi''_{\mathcal{S}} = \phi'_{\mathcal{Q}}$, also by inductive hypothesis, and, therefore, $\phi_{\mathcal{S}_{dt}} = \phi_{\mathcal{Q}_{dt}}$, from equations (13.33) and (13.34), $\mathcal{S}[\![\hat{sql}, \hat{t}_{al}]\!]\phi = \mathcal{Q}[\![h(\hat{sql}, \hat{t}_{al})]\!]\phi$

$\square$

REFERENCES

Almendra, V. D. S. and D. Schwabe (2006). Trust policies for semantic web repositories. In *Proceedings of 2nd International Semantic Web Policy Workshop (SWPW'06), at the 5th International Semantic Web Conference (ISWC*, pp. 17–31.

An, Y., A. Borgida, and J. Mylopoulos (2004). Refining semantic mappings from relational tables to ontologies. In *Proceedings of the 2nd International Workshop on Semantic Web and Databases (SWDB'04) in conjunction with Very Large Databases (VLDB'04)*, pp. 84–90. Springer.

An, Y., A. Borgida, and J. Mylopoulos (2006). Discovering the semantics of relational tables through mappings. In S. Spaccapietra (Ed.), *Journal on Data Semantics VII*, Volume 4244 of *Lecture Notes in Computer Science*, pp. 1–32. Springer Berlin / Heidelberg.

Anyanwu, K., A. Maduko, and A. Sheth (2007). Sparq2l: towards support for subgraph extraction queries in rdf databases. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, New York, NY, USA, pp. 797–806. ACM.

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

Beckett, D. (2004, February). Rdf/xml syntax specification (revised). Available at: `http://www.w3.org/TR/REC-rdf-syntax/`.

Berners-Lee, T. (1999). *Weaving the Web.* Orion Publishing Group.

Berners-Lee, T. (2006). Design issues: Linked data. Available at: `http://www.w3.org/DesignIssues/LinkedData.html`.

Bizer, C. (2003). D2r map - a database to rdf mapping language. In *Proceedings of the 12th International World Wide Web Conference*, WWW 2003.

Bizer, C. and R. Cyganiak (2006). D2r server - publishing relational databases on the semantic web. In *Proceedings of the 5th International Semantic Web Conference*, ISWC 2006.

Bizer, C. and R. Cyganiak (2007). D2rq lessons learned. In *Position Paper in W3C Workshop on RDF Access to Relational Databases.*

Bizer, C., R. Cyganiak, J. Garbers, and O. Maresch. The d2rq platform. Available at: `http://www4.wiwiss.fu-berlin.de/bizer/d2rq/`.

Bizer, C. and A. Seaborne (2004). D2rq-treating non-rdf databases as virtual rdf graphs. In *Proceedings of the 3rd International Semantic Web Conference*, ISWC '04.

Blaha, M., W. Premerlani, and H. Shen (1994, may). Converting oo models into rdbms schema. *Software, IEEE 11*(3), 28 –39.

Blakeley, C. (2007). Virtuoso rdf views - getting started guide. Available at: `http://www.openlinksw.co.uk/virtuoso/Whitepapers/pdf/Virtuoso_SQL_to_RDF_Mapping.pdf`.

Bohannon, P., E. Elnahrawy, W. Fan, and M. Flaster (2006). Putting context into schema matching. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pp. 307–318. VLDB Endowment.

Borodaenko, D. (2009, nov.). On-demand rdf to relational query translation in samizdat rdf store. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, Volume 3, pp. 413 –417.

Brickley, D. and R. Guha (2004, February). Rdf vocabulary description language 1.0: Rdf schema. Available at: `http://www.w3.org/TR/rdf-schema/`.

Broekstra, J., A. Kampman, and F. V. Harmelen (2002). Sesame: A generic architecture for storing and querying rdf and rdf schema. In *Proceedings of the International Semantic Web Conference*, pp. 54–68. Springer.

Buneman, P., A. Chapman, and J. Cheney (2006). Provenance management in curated databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, New York, NY, USA, pp. 539–550. ACM.

Cerbah, F. (2008). 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, Berlin, Heidelberg, pp. 777–781. Springer-Verlag.

Chebotko, A., S. Lu, and F. Fotouhi (2009, October). Semantics preserving sparql-to-sql translation. *Data and Knowledge Engineering. 68*, 973–1000.

Chebotko, A., S. Lu, H. M. Jamil, and F. Fotouhi (2006). Semantics preserving sparql-to-sql query translation for optional graph patterns. Technical report.

Chen, H., Z. Wu, H. Wang, and Y. Mao (2006). Rdf/rdfs-based relational database integration. In *In Proceedings of the 22nd International Conference on Data Engineering*, pp. 94.

Chen, L. and N. Yao (2010, july). Publishing linked data from relational databases using traditional views. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, Volume 6, pp. 9 –12.

Cheong, C. P., C. Chatwin, and R. Young (2009, aug.). A rdf-based semantic schema mapping transformation system for localized data integration. In *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on*, pp. 144 –147.

Choi, M.-Y., C.-J. Moon, D.-K. Baik, Y.-J. Wie, and J.-H. Park (2010, aug.). Interoperability between a relational data model and an rdf data model. In *Networked Computing and Advanced Information Management (NCM), 2010 Sixth International Conference on*, pp. 335 –340.

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

Cong, G., W. Fan, F. Geerts, X. Jia, and S. Ma (2007). Improving data quality: consistency and accuracy. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pp. 315–326. VLDB Endowment.

Dai, B. T., N. Koudas, B. C. Ooi, D. Srivastava, and S. Venkatasubramanian (2006, dec.). Rapid identification of column heterogeneity. In *Data Mining, 2006. ICDM '06. Sixth International Conference on*, pp. 159 –170.

Dai, B. T., N. Koudas, D. Srivastava, A. Tung, and S. Venkatasubramanian (2008, april). Validating multi-column schema matchings by type. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pp. 120 –129.

DataVision. Datavision. the open source report writer. Available at: `http://datavision.sourceforge.net/`.

de Laborda, C. P. and S. Conrad (2006). Bringing relational data into the semanticweb using sparql and relational.owl. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, pp. 55.

de Laborda, C. P., M.Zloch, and S. Conrad (2006). Rdquery - querying relational databases on-the-fly with rdf-ql. In *Poster and Demo Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management*, EKAW 2006.

Duarte, J., M. Salazar, C. Quintas, M. Santos, J. Neves, A. Abelha, and J. Machado (2010). Data quality evaluation of electronic health records in the hospital admission process. *Computer and Information Science, ACIS International Conference on 0*, 201–206.

Elliott, B., E. Cheng, C. Thomas-Ogbuji, and Z. M. Ozsoyoglu (2009). A complete translation from sparql into efficient sql. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, IDEAS '09, New York, NY, USA, pp. 31–42. ACM.

Erling, O. and I. Mikhailov (2009). Rdf support in the virtuoso dbms. In T. Pellegrini, S. Auer, K. Tochtermann, and S. Schaffert (Eds.), *Networked Knowledge - Networked Media*, Volume 221 of *Studies in Computational Intelligence*, pp. 7–24. Springer Berlin / Heidelberg.

Even, A. and G. Shankaranarayanan (2007, May). Utility-driven assessment of data quality. *SIGMIS Database 38*, 75–93.

Grant Clark, K. (2005, March). Rdf data access use cases and requirements. Available at: `http://www.w3.org/TR/rdf-dawg-uc/`.

Gruff. Gruff: A grapher-based triple-store browser for allegrograph. Available at: `http://agraph.franz.com/gruff/`.

Guo, Y., Z. Pan, and J. Heflin (2005). Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics 3*(2-3), 158–182.

Han, L., T. Finin, C. Parr, J. Sachs, and A. Joshi (2008). Rdf123: From spreadsheets to rdf. In A. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. Finin, and K. Thirunarayan (Eds.), *The Semantic Web - ISWC 2008*, Volume 5318 of *Lecture Notes in Computer Science*, pp. 451–466. Springer Berlin / Heidelberg.

Harris, S. (2005). Sparql query processing with conventional relational database systems. In *International Workshop on Scalable Semantic Web Knowledge Base Systems*, pp. 235–244.

Harris, S. and A. Seaborne (2011, May). Sparql 1.1 query language. Available at: `http://www.w3.org/TR/sparql11-query/`.

Hayes, P. (2004, February). Rdf semantics. Available at: `http://www.w3.org/TR/rdf-mt/`.

He, B., M. Patel, Z. Zhang, and K. C.-C. Chang (2007, May). Accessing the deep web. *Commun. ACM 50*, 94–101.

Heath, T. and C. Bizer (2011). *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool.

Hendler, J. (2006, December). Rdf due diligence. Available at: `http://civicactions.com/blog/rdf_due_diligence`.

Hert, M., G. Reif, and H. C. Gall (2010). Updating relational data via sparql/update. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, New York, NY, USA, pp. 24:1–24:8. ACM.

Imai, A. and S. Yukita (2003, march). Rdf model and relational metadata. In *Advanced Information Networking and Applications, 2003. AINA 2003. 17th International Conference on*, pp. 534 – 537.

Ismail, M., M. Yaacob, and S. Kareem (2008, aug.). Integration of heterogeneous relational databases: Rdf mapping approach. In *Information Technology, 2008. ITSim 2008. International Symposium on*, Volume 3, pp. 1 –7.

Jiang, H., L. Ju, and Z. Xu (2009, nov.). Upgrading the relational database to the semantic web with hibernate. In *Web Information Systems and Mining, 2009. WISM 2009. International Conference on*, pp. 227 –230.

Klyne, G. and J. Carroll (2004, February). Resource description framework (rdf): Concepts and abstract syntax. Available at: `http://www.w3.org/TR/rdf-concepts/`.

Korotkiy, M. and J. L. Top (2004). From relational data to rdfs models. In *International Conference on Web Engineering. Volume 3140 of LNCS*, pp. 430–434. Springer.

Krishna, M. (2006, dec.). Retaining semantics in relational databases by mapping them to rdf. In *Web Intelligence and Intelligent Agent Technology Workshops, 2006. WI-IAT 2006 Workshops. 2006 IEEE/WIC/ACM International Conference on*, pp. 303 –306.

Lacy, L. W. (2005). *OWL: Representing Information using the Web Ontology Language.* Trafford Publishing.

Ling, H. and S. Zhou (2010, july). Translating relational databases into rdf. In *Environmental Science and Information Application Technology (ESIAT), 2010 International Conference on*, Volume 3, pp. 464 –467.

Lucas, A. (2010, june). Corporate data quality management: From theory to practice. In *Information Systems and Technologies (CISTI), 2010 5th Iberian Conference on*, pp. 1 –7.

Lv, Y. and Z. Ma (2008, oct.). Transformation of relational model to rdf model. In *Systems, Man and Cybernetics, 2008. SMC 2008. IEEE International Conference on*, pp. 506 – 511.

Malhotra, A. (2009, January). W3c rdb2rdf incubator group report. Available at: `http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/`.

Manola, F. and E. Miller (2004, February). Rdf primer. Available at: `http://www.w3.org/TR/rdf-primer/`.

McBride, B. (2002). Jena: A semantic web toolkit. In *Proceedings of IEEE Internet Computing*, pp. 55–59. Available at: `http://jena.sourceforge.net/index.html`.

McGuinness, D. L. and F. v. Harmelen (2004, February). Owl web ontology language overview. Available at: `http://www.w3.org/TR/owl-features/`.

Miettinen, M. and M. Korhonen (2008, june). Information quality in healthcare: Coherence of data compared between organization's electronic patient records. In *Computer-Based Medical Systems, 2008. CBMS '08. 21st IEEE International Symposium on*, pp. 488 –493.

Miller, L., A. Seaborne, and A. Reggiori (2002). Three implementations of squishql, a simple rdf query language. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, London, UK, UK, pp. 423–435. Springer-Verlag.

Muys, A. (2007). Building an enterprisescale database for rdf data.

Myroshnichenko, I. and M. Murphy (2009, sept.). Mapping er schemas to owl ontologies. In *Semantic Computing, 2009. ICSC '09. IEEE International Conference on*, pp. 324 –329.

MySQL. Mysql open source database. Available at: `http://www.mysql.com/`.

Narasimhan, B., S. Navathe, and S. Jayaraman (1994). On mapping er and relational models into oo schemas. In R. Elmasri, V. Kouramajian, and B. Thalheim (Eds.), *Entity-Relationship Approach ER '93*, Volume 823 of *Lecture Notes in Computer Science*, pp. 402–413. Springer Berlin / Heidelberg.

Orenstein, J. A. and D.N.Kamber (1995). Accessing a relational database through an object-oriented database interface. In *Proceedings of the 21st VLDB Conference*, pp. 702–705. Morgan Kaufmann Publishers Inc.

Pan, Z. and J. Heflin (2003). Dldb: Extending relational databases to support semantic web queries. In *In Practical and Scalable Semantic Systems*, pp. 109–113.

Partyka, J., N. Alipanah, L. Khan, B. Thuraisingham, and S. Shekhar (2008). Content-based ontology matching for gis datasets. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, GIS '08, New York, NY, USA, pp. 51:1–51:4. ACM.

Partyka, J., L. Khan, and B. Thuraisingham (2009a). Geographically-typed semantic schema matching. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, New York, NY, USA, pp. 456–459. ACM.

Partyka, J., L. Khan, and B. Thuraisingham (2009b). Semantic schema matching without shared instances. *International Conference on Semantic Computing 0*, 297–302.

Pérez, J., M. Arenas, and C. Gutierrez (2006a). Semantics and complexity of sparql. In *Proc. 5th International Semantic Web Conference (ISWC06*, pp. 30–43. CommunityProjects/LinkingOpenData.

Pérez, J., M. Arenas, and C. Gutierrez (2006b). Semantics of sparql. Technical report, Universidad de Chile.

Powers, S. (2003). *Practical RDF*. O'Reilly Media.

Prud'hommeaux, E. and A. Seaborne (2008, January). Sparql query language for rdf. Available at: `http://www.w3.org/TR/rdf-sparql-query/`.

Ramanujam, S., A. Gupta, L. Khan, S. Seida, and B. Thuraisingham (2009a, sept.). R2d: A bridge between the semantic web and relational visualization tools. In *Semantic Computing, 2009. ICSC '09. IEEE International Conference on*, pp. 303 –311.

Ramanujam, S., A. Gupta, L. Khan, S. Seida, and B. Thuraisingham (2009b). R2d: A framework for the relational transformation of rdf data. *International Journal of Semantic Computing 3*(4), 471–498.

Ramanujam, S., A. Gupta, L. Khan, S. Seida, and B. Thuraisingham (2009c, sept.). R2d: Extracting relational structure from rdf stores. In *Web Intelligence and Intelligent Agent Technologies, 2009. WI-IAT '09. IEEE/WIC/ACM International Joint Conferences on*, Volume 1, pp. 361 –366.

Ramanujam, S., A. Gupta, L. Khan, S. Seida, and B. Thuraisingham (2009d). A relational wrapper for rdf reification. In E. Ferrari, N. Li, E. Bertino, and Y. Karabulut (Eds.), *Trust Management III*, Volume 300 of *IFIP Advances in Information and Communication Technology*, pp. 196–214. Springer Boston.

Ramanujam, S., A. Gupta, L. Khan, S. Seida, and B. Thuraisingham (2009e). Relationalizing rdf stores for tools reusability. pp. 1059–1060.

Ramanujam, S., A. Gupta, L. Khan, S. Seida, and B. Thuraisingham (2010). Relationalization of provenance data in complex rdf reification nodes. *Electronic Commerce Research 10*, 389–421. 10.1007/s10660-010-9061-1.

Ramanujam, S., V. Khadilkar, L. Khan, M. Kantarcioglu, B. M. Thuraisingham, and S. Seida (2010). Update-enabled triplification of relational data into virtual rdf stores. *International Journal Semantic Computing 4*(4), 423–451.

Ramanujam, S., V. Khadilkar, L. Khan, S. Seida, M. Kantarcioglu, and B. Thuraisingham (2010, sept.). Bi-directional translation of relational data into virtual rdf stores. In *Semantic Computing (ICSC), 2010 IEEE Fourth International Conference on*, pp. 268 –276.

Redman, T. C. (1998, February). The impact of poor data quality on the typical enterprise. *Commun. ACM 41*, 79–82.

Reports, C. Sap crystal solutions. Available at: `http://crystalreports.com/`.

Seaborne, A. (2004, January). Rdql - a query language for rdf. Available at: `http://www.w3.org/Submission/RDQL/`.

Seaborne, A. et al. (2008, July). Sparql update - a language for updating rdf graphs. Available at: `http://www.w3.org/Submission/2008/SUBM-SPARQL-Update-20080715/`.

Singh, R., L. S. Iyer, and A. F. Salam (2005, December). The semantic e-business vision - introduction. *Commun. ACM 48*, 38–41.

SQL2003. Bnf grammar for sql-2003. Available at: `http://savage.net.au/SQL/sql-2003-2.bnf.html#query specification`.

SQLMaestro. Sqlmaestro - the shortest path to sql. Available at: `http://www.sqlmaestro.com/`.

Strong, D. M., Y. W. Lee, and R. Y. Wang (1997, May). Data quality in context. *Commun. ACM 40*, 103–110.

Tauberer, J. (2006, July). What is rdf. Available at: `http://www.xml.com/pub/a/2001/01/24/rdf.html`.

Teswanich, W. and S. Chittayasothorn (2007, aug.). A transformation from rdf documents and schemas to relational databases. In *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on*, pp. 38 –41.

Wang, J., J. Lu, Y. Zhang, Z. Miao, and B. Zhou (2009). Integrating heterogeneous data source using ontology. *Journal of Software 4*(8).

Wang, J., Z. Miao, Y. Zhang, and B. Zhou (2009, june). Querying heterogeneous relational database using sparql. In *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*, pp. 475 –480.

Wang, J., Y. Zhang, Z. Miao, and J. Lu (2010, april). Query transformation in ontology-based relational data integration. In *Wearable Computing Systems (APWCS), 2010 Asia-Pacific Conference on*, pp. 303 –306.

Wang, R. Y. and D. M. Strong (1996, March). Beyond accuracy: what data quality means to data consumers. *J. Manage. Inf. Syst. 12*, 5–33.

Wang, S., G. Englebienne, and S. Schlobach (2008). Learning concept mappings from instance similarity. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC '08, Berlin, Heidelberg, pp. 339–355. Springer-Verlag.

Warren, R. H. and F. W. Tompa (2006). Multi-column substring matching for database schema translation. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pp. 331–342. VLDB Endowment.

Wartena, C. and R. Brussee (2008). Instanced-based mapping between thesauri and folksonomies. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC '08, Berlin, Heidelberg, pp. 356–370. Springer-Verlag.

Welzer, T., B. Brumen, I. Golob, and M. Druovec (2002). Medical diagnostic and data quality. In *Computer-Based Medical Systems, 2002. (CBMS 2002). Proceedings of the 15th IEEE Symposium on*, pp. 97 – 101.

Wu, Z., H. Chen, H. Wang, Y. Wang, Y. Mao, J. Tang, and C. Zhou (2006). Dartgrid: a semantic web toolkit for integrating heterogeneous relational databases. In *Semantic Web Challenge at 4th International Semantic Web Conference*, Athens, USA.

Xu, L., S. Lee, and S. Kim (2010, july). E-r model based rdf data storage in rdb. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, Volume 1, pp. 258 –262.

Yan, Y., C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan (2008). Efficiently querying rdf data in triple stores. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, New York, NY, USA, pp. 1053–1054. ACM.

Zhou, L., H. Chen, Y. Zhang, and C. Zhou (2008). A semantic mapping system for bridging the gap between relational database and semantic web. *American Association for Artificial Intelligence*.

Zhou, S. (2009, dec.). Mapping relational database for semantic web. In *BioMedical Information Engineering, 2009. FBIE 2009. International Conference on Future*, pp. 521 –524.

Zhou, S. (2010, july). Exposing relational database as rdf. In *Industrial and Information Systems (IIS), 2010 2nd International Conference on*, Volume 2, pp. 237 –240.

VITA

Sunitha Ramanujam received her Bachelor of Engineering (Hons) degree in Electrical and Electronics and Master of Science (Hons) degree in Software Systems degree from *Birla Institute of Technology and Science (BITS)* in Pilani, India, in 1995 and 1998 respectively and went on to receive two more masters degrees Masters in Engineering Science with specialization in Software Engineering from the *University of Western Ontario (UWO)*, Canada, in 2003, and M.S. in Computer Science from the *University of Texas at Dallas (UTD)*, U.S.A., in 2008. During her academic career at *UWO*, Sunitha was the recipient of several esteemed awards at college, industrial and national levels including the *Special University Scholarship (SUS)* (2002-2004), *IBM Center for Advanced Studies (CAS) Fellowship* (2002-2004), and National Sciences and Engineering Research Council of Canada (NSERC) Scholarship (2004-2006), and was also offered the Ontario Graduate Scholarship (OGS). Sunitha commenced the pursuit of a Ph.D. in Computer Science at *UTD* in Spring, 2007 and was awarded the prestigious *Jonsson School Distinguished Assistantship scholarship* from January, 2007 to December, 2008. Since then she had been employed as a research assistant at the Department of Computer Science and her research has been funded by *Raytheon Company.* Prior to commencing on her academic quest, Sunitha worked for seven years as an Oracle Database Administrator (DBA) at various international organizations such as Citibank, India, where she was the DBA Project Manager heading a 24-member team, and Cummins Engine Company, Indiana .