University of Cincinnati

Date: 8/28/2017

I, Dippy Aggarwal, hereby submit this original work as part of the requirements for the degree of Doctor of Philosophy in Computer Science & Engineering.

It is entitled:
A Provenance-based Approach Towards Impact Assessment of Schema Changes in a Data Warehouse Environment

Student’s name: Dippy Aggarwal

This work and its defense approved by:

Committee chair: Karen Davis, Ph.D.
Committee member: Raj Bhatnagar, Ph.D.
Committee member: Hsiang-Li Chiang, Ph.D.
Committee member: Ali Minai, Ph.D.
Committee member: Carla Purdy, Ph.D.

27973
A Provenance-based Approach Towards Impact Assessment of Schema Changes in a Data Warehouse Environment

Dissertation

Submitted in Partial Fulfillment of the Requirements for the Degree Of
DOCTOR OF PHILOSOPHY

In the Department of
Electrical Engineering and Computing Systems
College of Engineering and Applied Sciences
University of Cincinnati

April 2017

By

Dippy Aggarwal, M.S.
West Chester University of Pennsylvania, Pennsylvania, USA, 2009

Dissertation Committee:

Prof. Karen C. Davis, Advisor
Prof. Raj Bhatnagar
Prof. Roger Chiang
Prof. Ali Minai
Prof. Carla Purdy
Abstract

Graph-based solutions are receiving significant attention recently for two reasons: (1) their ability to capture relationships as first class elements in interconnected domains, and (2) their inherent resemblance to real-world scenarios such as social networks. In our work, we investigate the use of graph databases towards two schema management challenges: (a) impact assessment of schema evolution in a data warehouse environment, and (b) schema mapping and integration. We leverage the explicit capture of relationships to address these challenges.

Data warehouses are a schema-rich, multi-layered environment consisting of many inter-related artifacts. The artifacts include operational, reconciled and warehouse schemas connected by ETL mappings and a set of queries expressed against the schemas. If a user seeks to make a schema change at any level in the architecture, he or she may not be aware of the other artifacts potentially impacted by the change. Previous approaches to data warehouse schema evolution have focused on proposing algorithms for propagating the impact of the change to the related artifacts in an automated manner. While these contributions ease the task for the user by providing a programmatic way of adapting related artifacts, they do not provide support for detailing the potential impact.

We focus on defining and implementing a graph-based model for impact assessment and explanation. Impact assessment involves identification of the artifacts that depend
on the evolved artifact either directly or transitively. The consequences of the change are revealed before actually propagating the change. Our work also allows changes to all schema artifacts in a multi-layered data warehouse architecture, thus addressing multiple evolution problems under one framework. The current contributions are restricted in that they do not address changes to all schema components in the warehouse architecture. We leverage provenance to facilitate user’s understanding for the identified impact. Along with presenting a list of artifacts that will be potentially impacted by the change, we provide a complete trace of how the evolved artifact and the impacted artifacts are related to each other. The schemas in our work follow the relational paradigm, ETL workflows are described using a leading commercial business tool, Pentaho, and the queries are expressed using SQL. We present our framework, illustrate the supporting conceptual model, detail the modeling challenges, and demonstrate the viability of our approach using a case study.

In the context of the second domain of interest (schema mapping and integration) of our work, we describe a system that supports schema integration based on graph databases. Our work first looks at leveraging a graph-based solution for schema mapping. Specifically, we illustrate how schemas expressed in relational and RDF models can be transformed to a property graph to provide an information-preserving, NoSQL-compliant, standardization model for schemas expressed in heterogeneous models. Then we further extend the work by contributing a schema merging algorithm for property graphs. We consider some concrete examples from the literature to highlight how our framework supports integration over property graphs. We illustrate a modular framework that can be further extended and optimized to incorporate different schema mapping and merging algorithms.
Dedicated to my mom Neena Gupta

who did much more than what she should have to support my privileges.
Acknowledgments

First and foremost I want to say a heartfelt thank you to my advisor Prof. Karen Davis for her continuous support and guidance throughout the course of my doctoral studies. Not only she has been my research advisor but also an incredible mentor towards my teaching and academic career. I have been a teaching assistant for some of her courses and it is an inspiration to watch her dedication towards teaching. My first teaching opportunity at the University of Cincinnati came through her recommendation and her confidence in me and it opened the door for many more opportunities during the subsequent semesters. I am thankful for the financial support that she extended to me several times in the form of research/teaching assistantships and the conference travel and also the freedom she gave me in pursuing different internships that have truly enriched my professional career. I gratefully acknowledge her guidance and time in reviewing the numerous drafts towards my dissertation and conference publications, offering valuable scientific insights and feedback on scholarly writing. It was her guidance and motivation that encouraged me to apply for an international research internship in Ireland which was truly a learning experience both professionally and personally. And I can not thank her enough for her understanding and flexibility in accommodating the dissertation defense schedule given the time constraints I have. Without her patience, support, and expertise this PhD would not have been achievable.
I also want to convey my sincere thanks to each of the committee members: Prof. Raj Bhatnagar, Prof. Ali Minai, Prof. Roger Chiang, and Prof. Carla Purdy. I appreciate their time and interest in reviewing my work and for their support, understanding and valuable feedback on the proposal that helped me make progress towards my dissertation.

I am immensely grateful to Ms. Julie Muenchen and Ms. Barbara Carter from the CEAS graduate studies office. They have been a huge support system in my PhD studies and have extended their support time and again. Their willingness to walk an extra mile to make things work for me is something I will always be grateful for. I would also like to thank Dr. Paul Talaga. His dedication towards student teaching has been a joy to watch and I am grateful for the confidence he has always shown in my abilities.

My family - I would like to start by thanking my maternal uncles, Pardeep Gupta, Anuj Gupta, and Rajiv Gupta whose decision and support in bringing me to USA changed my life for ever and for good. From a submissive girl who never had the courage to speak her mind and never had any serious ambition in life, I have grown into someone who knows and understands what she wants in life. My uncles’ decision and conviction has taught me the true meaning of being independent which I think is a life lesson. I also feel blessed to have truly loving maternal grandparents, aunts, cousins and a cute little nephew.

These past seven years have been a long and at times lonely walk for me. During such times, had I not have my friends - Divya Paliwal, Amit Airon, Divya Sardana, Shruti Marwaha, Ravi, Peggy Shannon-Baker, Alejandro Gutierrez, my cousin Bharti Gupta, and brother in-law Ravi Aggarwal by my side, it would have been much harder
to accomplish my work. The warm and welcoming gesture of my friends Shruti and Ravi to invite me to stay with them during the conference IEEE Big Data 2015 was a huge support. Due to the lack of funds, I would not have been able to present my internship work at the conference without their support at that time. I also greatly appreciate the financial support that my team at InfoTrust offered me for presenting my work at a workshop in IEEE Big Data 2015. I worked for them as an intern for only three months and they covered my travel and conference registration cost for the conference. A huge thank you for my manager Prakash Velayutham at Cincinnati Children’s Hospital who has been truly supportive and encouraging throughout my internship.

I feel grateful for that divine intervention that has brought all these people in my life and always weaved a path for me to move forward.

Last but not the least, my mother Neena Gupta. Talking about her contribution in my doctoral studies would be an understatement. She has been my strength and weakness. I would not have been where I am in life today without all the hardships she has endured in supporting me and giving me a fulfilling life. The principles and values she has given me will stay with me forever and have shaped me into a person who I am today. This is why this dissertation is dedicated to her.
# Table of Contents

1. Introduction ......................................................... 1
   1.1 Motivating example ........................................... 4
   1.2 Limitations of the current contributions .................. 11
   1.3 Problem statement ............................................. 13
   1.4 Research objectives .......................................... 13
   1.5 Outline of the dissertation .................................. 15

2. Background Research ............................................. 17
   2.1 Data warehouse schema evolution ............................. 18
   2.2 Provenance ..................................................... 24

3. Conceptual Model .................................................. 29
   3.1 Design considerations .......................................... 33
   3.2 Formal description of the mapping between conceptual model and the target graph implementation ........................ 36
   3.3 Example - Instantiated conceptual model .................... 37

4. Implementation Technologies ..................................... 42
   4.1 Pentaho ....................................................... 42
   4.2 Neo4j ........................................................ 45
5. Architecture and Implementation ........................................... 51
   5.1 Architecture .......................................................... 52
   5.2 Implementation ...................................................... 55
       5.2.1 Transform relational schemas to target graph representation  56
       5.2.2 Transforming ETL workflows to Neo4j ......................... 60
       5.2.3 Transforming queries to CSV format ........................... 64
       5.2.4 Impact assessment .............................................. 67

6. Case Study ................................................................. 71
   6.1 Case study ............................................................. 72
       6.1.1 Operational schemas ........................................... 73
       6.1.2 Reconciled schema ............................................. 74
       6.1.3 Warehouse schema ............................................. 76
       6.1.4 ETL transformations between operational and the reconciled
            schema .......................................................... 79
       6.1.5 ETL transformations between reconciled and the data warehouse
            schema .......................................................... 84
       6.1.6 Queries ........................................................... 89
   6.2 Verification of the design of ETL transformations ................. 92
   6.3 Verification of the generated graph for the artifacts in the case study 96
   6.4 Impact assessment analysis .......................................... 99
       6.4.1 Examples ......................................................... 100
       6.4.2 Verification of the impact assessment results ............... 110
   6.5 Lineage tracking using pentaho .................................... 115
       6.5.1 db-impact ....................................................... 116
       6.5.2 extract-transformation-metadata ............................. 119
   6.6 Impact assessment in pentaho ...................................... 120
   6.7 Summary .............................................................. 121

7. User Interface and Impact Explanation ................................ 124
   7.1 User interface ......................................................... 124
       7.1.1 Converting input artifacts to the target graph representation 125
       7.1.2 Impact assessment - user interface and Neo4j browser .... 135
   7.2 Impact explanation .................................................... 139

8. Employing Graph Databases Towards Addressing Heterogeneity and Integration ................................................. 146
8.1 Background ......................................................... 151
8.1.1 RDF ......................................................... 151
8.1.2 Relational schema ........................................... 151
8.1.3 Neo4j and property graphs ............................... 152
8.2 Transformation Approach ...................................... 153
8.2.1 Relational model to a property graph ................. 155
8.2.2 RDF model to a property graph .......................... 156
8.3 Architecture .................................................... 160
8.4 Evaluation ....................................................... 163
8.5 Schema integration over property graphs ................. 167
8.6 Algorithm ....................................................... 171
8.7 Evaluation ....................................................... 177
8.7.1 Example 1 .................................................... 178
8.7.2 Example 2 .................................................... 181
8.8 Related work .................................................... 183
8.9 Conclusion and future work ................................. 185

9. Conclusion and Future Work ..................................... 187

Appendices .......................................................... 189

A. Input artifacts ...................................................... 189
A.1 Input schemas - Operational, Reconciled and Warehouse Schema .... 189
A.1.1 Schema generation script for operational schema - *sakila_op1* .... 189
A.1.2 Schema generation script for operational schema - *sakila_op2* .... 197
A.1.3 Schema generation script for reconciled schema - *sakila_ids* .... 203
A.1.4 Schema generation script for warehouse schema - *sakila_dwh* .... 212
A.1.5 Instance generation script for operational schemas - *sakila_op1*
and *sakila_op2* .................................................. 219
A.2 ETL workflows ................................................... 223
A.2.1 ETL workflows between the operational and reconciled schema 223
A.2.2 ETL workflows between the reconciled and warehouse schema .... 228
A.3 End user queries (MySQL) ....................................... 236

B. Java code for Chapter 7 (User Interface and Impact Assessment) ......... 239
B.1 Front end code .................................................. 239
B.1.1 index.jsp .................................................... 240
B.1.2 etl.jsp ....................................................... 242
B.1.3 queries.jsp .................................................. 243
B.1.4 lineage.jsp .................................................. 245
B.2 Java servlet code for handling user requests - servlet files ............ 247
B.2.1 SchemaServlet.java ......................................... 247
B.2.2 ETLServlet.java ............................................. 249
B.2.3 QueryServlet.java .......................................... 254
B.2.4 LineageServlet.java ....................................... 256
B.3 Java code for generating CSV files .................................. 262
B.3.1 ETLExcelExporter.java ..................................... 262
B.3.2 QueryExcelExporter.java ................................... 270
B.4 Java code for generating Neo4j graph ................................ 271
B.4.1 SchemaGraphController.java ................................ 272
B.4.2 ETLGraphController.java ................................... 277
B.4.3 QueryGraphController.java .................................. 288
B.5 XML parsers for ETL workflows and queries ............................ 289
B.5.1 XMLParser.java ............................................. 290
B.5.2 TableInputStepParser.java ................................... 296
B.5.3 TableOutputStepParser.java .................................. 297
B.5.4 InsertUpdateStepParser.java ................................ 300
B.5.5 DimensionLookupStepParser.java ................................ 302
B.5.6 CombinationLookupStepParser.java ................................ 306
B.5.7 DBJoinStepParser.java ...................................... 308
B.5.8 DBLookupStepParser.java .................................... 310
B.5.9 QueryXMLParser.java ...................................... 313
B.6 Java objects for various artifacts .................................... 315
B.6.1 TableInputStep.java ......................................... 315
B.6.2 OutputStep.java ............................................. 316
B.6.3 DimLookupUpdateStep.java ................................... 318
B.6.4 DBLookupStep.java .......................................... 320
B.6.5 DBJoinStep.java ............................................. 321
B.6.6 CombinationLookupUpdateStep.java ................................ 322
B.6.7 Query.java ................................................... 324

C. Evaluation Results .................................................. 326

C.1 Drop attribute from operational schema
sakila_op1.staff.first_name ...................................... 326
C.2 Rename attribute from operational schema
sakila_op1.customer.active ....................................... 326
C.3 Drop attribute from reconciled schema
sakila_ids.film.title ............................................. 327
C.4 Rename attribute from reconciled schema
sakila_ids.category.name ............................................. 327
C.5 Change datatype for attribute from reconciled schema
sakila_ids.actor.first_name .......................................... 328
C.6 Drop attribute from the warehouse schema
sakila_dwh.dim_date.day_abbreviation ............................. 335
C.7 Rename attribute from the warehouse schema
sakila_dwh.dim_film.film_in_category_foreign .................... 335
C.8 Change datatype for an attribute in the warehouse schema
sakila_dwh.dim_customer.customer_key ............................. 336
C.9 Rename relation in the operational schema
sakila_op2.film_category ............................................. 336
C.10 Drop relation in the reconciled schema
sakila_ids.film_category ............................................. 337
C.11 Drop relation in the reconciled schema
sakila_ids.store .......................................................... 337
C.12 Rename relation in the warehouse schema
sakila_dwh.dim_store .................................................... 338
C.13 Evolving reconciled schema sakila_ids ......................... 338
C.14 Evolving warehouse schema sakila_dwh ....................... 339

D. Pre-requisites and manual steps for software tool .............. 347
D.1 Prerequisites for transforming schemas to a graph representation: index.jsp ......................................... 347
D.2 Prerequisites and manual steps for transforming ETL workflows to a graph representation: etl.jsp ....................... 347
D.3 Prerequisites and manual steps for transforming SQL queries to a graph representation: query.jsp ....................... 353

Bibliography ............................................................... 355
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>User interaction and impact of our proposed system</td>
<td>10</td>
</tr>
<tr>
<td>1.2</td>
<td>Research objectives, methodology and contributions</td>
<td>14</td>
</tr>
<tr>
<td>6.1</td>
<td>Query characteristics</td>
<td>90</td>
</tr>
<tr>
<td>6.2</td>
<td>Comparison of the number of records</td>
<td>92</td>
</tr>
<tr>
<td>6.3</td>
<td>Experiment for evaluation of ETL workflows</td>
<td>95</td>
</tr>
<tr>
<td>6.4</td>
<td>Generated metadata statistics(Neo4j) for all the input artifacts</td>
<td>96</td>
</tr>
<tr>
<td>6.5</td>
<td>Verification of the modeling of schemas in the implementation graph model - Neo4j</td>
<td>97</td>
</tr>
<tr>
<td>6.6</td>
<td>Evaluation of the ETL transformation in the generated architecture graph as compared to the results from manual inspection</td>
<td>98</td>
</tr>
<tr>
<td>6.7</td>
<td>Manual inspection of impact assessment outcome for an evolution operation over operational schema</td>
<td>114</td>
</tr>
<tr>
<td>6.8</td>
<td>Manual inspection of impact assessment outcome for an evolution operation over reconciled schema</td>
<td>114</td>
</tr>
<tr>
<td>6.9</td>
<td>Manual inspection of impact assessment outcome for an evolution operation over warehouse schema</td>
<td>115</td>
</tr>
<tr>
<td>6.10</td>
<td>Impact assessment results for different artifacts over each of the input schemas</td>
<td>123</td>
</tr>
</tbody>
</table>
8.1 Relational schema excerpt. Top Employee and Bottom Organization . . 154
8.2 Evaluation metrics results for MySQL database - sakila [16] . . . . . . 164
8.3 Evaluation metrics results for MySQL database - sakila using two mapping transformations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 167
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Three-tier data warehouse architecture [65]</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Operational schemas $S1$ (left) and $S2$ (right)</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Reconciled schema</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Data warehouse schema</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>Mappings between sources ($S1$ and $S2$) and the merged schema ($R$)</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Summary of schema evolution research in data warehouse environments</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Conceptual model for our approach</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>A sample ETL transformation depicting inter-step dependencies</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Sample artifacts to illustrate instantiation of the proposed conceptual model</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>A sample instantiation of the conceptual model</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>A simple three-step Pentaho ETL workflow</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Pentaho ETL Operations</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>A simple property graph</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>A subset of movie dataset [17]</td>
<td>48</td>
</tr>
<tr>
<td>4.5</td>
<td>A sample path traversal query and its output [17]</td>
<td>49</td>
</tr>
</tbody>
</table>
5.1 Architecture of our proposed approach .................................. 52
5.2 MySQL query to capture information about all relations in the MySQL
sakila database .............................................................. 56
5.3 MySQL query to capture information about all attributes in the MySQL
sakila database .............................................................. 57
5.4 MySQL query to capture information about all foreign key constraints
in the MySQL sakila database ............................................ 57
5.5 Code structure for generating Neo4j graph for a schema from a set of
CSV files ................................................................. 58
5.6 API call to Neo4j from Java: SchemaGraphController.java ............. 59
5.7 Snippet of the XML representation of Insert/Update step ................. 61
5.8 Overview of code structure for generating Neo4j graph from a set of
XML files ................................................................. 62
5.9 Code snippet for generateCSVData method in the ETLServlet file ...... 63
5.10 Code snippet for XMLParser.java ..................................... 64
5.11 Sample XML structure for two different step types in an ETL transform-
ation in Pentaho (left) InsertUpdate step (right) TableInput step ...... 65
5.12 Code snippet for ETLGraphController to generate Neo4j graph ...... 66
5.13 XML excerpt from GSQLParser for a SQL query ....................... 67
5.14 Overview of Java code components for parsing SQL query into CSV format 68
5.15 Code snippet illustrating parsing of XML representation of a SQL query 69
5.16 User interface for impact assessment ................................... 69
5.17 Cypher query presented to the user after form submission in Figure 5.16 70
6.19 The complete path and the Cypher query illustrating the artifacts impacted by drop attribute, dim_store.store_valid_from country operation 106

6.20 Impacted artifacts for the operation - drop relation, actor from the schema sakila_op1 .......................... 107

6.21 Cypher query for evolution operation 4 .......................... 108

6.22 Impacted artifacts for the operation - drop relation, language from the schema sakila_ids .......................... 109

6.23 Cypher query for evolution operation 5 .......................... 110

6.24 Impacted artifacts for the operation - drop relation, dim_film from the schema sakila_dwh .......................... 111

6.25 Cypher query for evolution operation 6 .......................... 111

6.26 Impacted artifacts for evolving schema sakila_op1 .......................... 112

6.27 Cypher query for evolution operation 7 .......................... 113

6.28 Lineage support in Pentaho - db-impact.ktr [43] .......................... 116

6.29 Lineage support in Pentaho - db-impact.ktr .......................... 117

6.30 Output received from successful execution of db-impact.ktr .......................... 119

6.31 Output after step labelled TableOutput is deleted from script in Figure 6.32 .......................... 120

6.32 Lineage support in Pentaho - extract-transformation-metadata.ktr . . . 121

6.33 Pentaho’s response to a schema change .......................... 122

7.1 User interface front-end files .......................... 125

7.2 The UI page for converting schemas to the target graph model . . . . 126

7.3 Flow diagram for application code driving the functionality in Figure 7.2 127
7.4 Error messages for web page in Figure 7.2 when Neo4j database is not running

7.5 Error messages for web page in Figure 7.2 for an invalid input on schema name

7.6 Error messages for web page in Figure 7.2 when any of the three csv files is missing or malformed

7.7 Operational schema-sakila_op1 successfully transformed to Neo4j graph

7.8 The main web page for transforming ETL workflows to Neo4j

7.9 Java code structure for conversion of ETL workflows to a graph

7.10 The main web page for transforming SQL queries to Neo4j

7.11 Java code structure for conversion of SQL queries to a graph

7.12 The main web page for viewing the impact of a schema change over an artifact

7.14 Cypher query from Figure 7.13b executed in Neo4j browser

7.15 GUI screenshot for source analysis of an operational schema

7.16 Phrases to facilitate impact explanation

7.17 Example of text-based description generated for the accompanied graph

8.1 Trend of web search interest for graph databases

8.2 An example of a property graph

8.3 Excerpts of two heterogeneous schemas originally in different models (left: relational schema, right: RDF schema) unified under a common graph representation

8.4 RDF schema excerpt
8.5 Placeholders $p, r,$ and $u$ for schema mapping between a column-oriented store and a property graph model [7] ........................................... 159

8.6 Architecture of our proposed approach .............................................. 161

8.7 MySQL query to capture foreign key relationships in the MySQL sakila database [16] ................................................................. 162

8.8 Code-snippet illustrating creation of a Neo4j graph for a relational schema programmatically ...................................................... 163

8.9 (a)(top) Modeling attributes as nodes, (b) (bottom) Our approach to modeling attributes as key-value pairs ........................................ 165

8.10 Two schemas (left: Medicine Graph, right: Engineering Graph) modelled using property graphs ................................................. 168

8.11 Module for unifying the node-types from input schemas ...................... 172

8.12 Module for merging the nodes from input schemas ............................. 173

8.13 Module for unifying relationships from input schemas .......................... 175

8.14 Integrated schema ........................................................................ 176


8.16 Integrated schema obtained through our approach (Example 1) ............ 179

8.17 Book and Publication schemas [23] ..................................................... 180

8.18 Integrated schema using our approach (Example 2) ............................. 181

A.1 ETL transformation - load_film_actor .............................................. 223

A.2 ETL transformation - load_store ...................................................... 224

A.3 ETL transformation - load_staff ...................................................... 224

A.4 ETL transformation - load_rental .................................................... 225
A.5 ETL transformation - load_payment ............................... 225
A.6 ETL transformation - load_inventory ............................ 225
A.7 ETL transformation - load_film_category ..................... 225
A.8 ETL transformation - aux_staff_store .......................... 226
A.9 ETL transformation - load_customer ............................ 226
A.10 ETL transformation - load_country ............................ 227
A.11 ETL transformation - load_category ........................... 227
A.12 ETL transformation - load_city ............................... 227
A.13 ETL transformation - load_dim_actor [43] .................... 228
A.14 ETL transformation - load_dim_customer [43] ................. 229
A.15 ETL transformation - load_dim_film [43] ..................... 230
A.16 ETL transformation - load_dim_date [43] ..................... 231
A.17 ETL transformation - load_dim_time [43] ..................... 232
A.18 ETL transformation - load_dim_store [43] .................... 233
A.19 ETL transformation - load_dim_staff [43] .................... 234
A.20 ETL transformation - load_fact_rental [43] .................. 235
A.21 ETL transformation - load_fetch_address [43] ............... 235
B.1 A high-level overview of the frontend code files ............. 240
C.1 The complete path illustrating the artifacts impacted by drop attribute operation on sakila_op1.staff.first_name ...................... 327
C.2 Cypher query for impact of sakila_op1.staff.first_name .......... 329

xxi
C.3 The complete path illustrating the artifacts impacted by drop attribute operation on sakila_op1.customer.active .......................... 330

C.4 Cypher query for impact of sakila_op1.customer.active .................. 331

C.5 The complete path illustrating the artifacts impacted by attribute, sakila_ids.film.title .................................................. 332

C.6 Cypher query for impact of sakila_ids.film.title ............................... 332

C.7 The complete path illustrating the artifacts impacted by sakila_ids.category.name 333

C.8 Cypher query for sakila_ids.category.name ................................. 333

C.9 The complete path illustrating the artifacts impacted by sakila_ids.actor.first_name 334

C.10 Cypher query for sakila_ids.actor.first_name ................................. 334

C.11 The complete path illustrating the artifacts impacted by drop attribute, sakila_dwh.dim_date.day_abbreviation .......................... 335

C.12 Cypher query for sakila_dwh.dim_date.day_abbreviation .................. 336

C.13 The complete path illustrating the artifacts impacted by drop attribute, 
film_in_category_foreign operation ............................................. 337

C.14 Cypher query for sakila_dwh.dim_film.film_in_category_foreign .............. 338

C.15 The complete path illustrating the artifacts impacted by drop attribute 
sakila_dwh.dim_customer.customer_key operation .............................. 339

C.16 Cypher query for sakila_dwh.dim_customer.customer_key .................. 340

C.17 The complete path illustrating the artifacts impacted by relation, sak-
ila_op2.film_category ............................................................ 340

C.18 Cypher query for the impact of rename relation operation on sakila_op2.film_category341

C.19 The complete path illustrating the artifacts impacted by sakila_ids.film_category342
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.20</td>
<td>Cypher query for <code>sakila_ids.film_category</code></td>
<td>342</td>
</tr>
<tr>
<td>C.21</td>
<td>The complete path illustrating the artifacts impacted by <code>sakila_ids.store</code></td>
<td>343</td>
</tr>
<tr>
<td>C.22</td>
<td>Cypher query for <code>sakila_ids.store</code></td>
<td>343</td>
</tr>
<tr>
<td>C.23</td>
<td>The complete path illustrating the artifacts impacted by drop attribute,</td>
<td>344</td>
</tr>
<tr>
<td></td>
<td><code>sakila_dwh.dim_store</code> operation</td>
<td></td>
</tr>
<tr>
<td>C.24</td>
<td>Cypher query for <code>sakila_dwh.dim_store</code></td>
<td>344</td>
</tr>
<tr>
<td>C.25</td>
<td>The complete path illustrating the artifacts impacted by evolving the</td>
<td>345</td>
</tr>
<tr>
<td></td>
<td>reconciled schema, <code>sakila_ids</code></td>
<td></td>
</tr>
<tr>
<td>C.26</td>
<td>Cypher query for impact of evolving <code>sakila_ids</code></td>
<td>345</td>
</tr>
<tr>
<td>C.27</td>
<td>The complete path illustrating the artifacts impacted by evolving the</td>
<td>346</td>
</tr>
<tr>
<td></td>
<td>warehouse schema, <code>sakila_dwh</code></td>
<td></td>
</tr>
<tr>
<td>C.28</td>
<td>Cypher query for impact of evolving <code>sakila_dwh</code></td>
<td>346</td>
</tr>
<tr>
<td>D.1</td>
<td>Sample user input for user interface <code>etl.jsp</code></td>
<td>348</td>
</tr>
<tr>
<td>D.2</td>
<td>Limitation of the generated CSV data for the user interface <code>etl.jsp</code></td>
<td>349</td>
</tr>
<tr>
<td>D.3</td>
<td>The revised CSV file <code>input_steps.csv</code> after performing edits</td>
<td>349</td>
</tr>
<tr>
<td>D.4</td>
<td>An ETL transformation containing two parallel workflows</td>
<td>350</td>
</tr>
<tr>
<td>D.5</td>
<td>Transformation of ETL workflow <code>load_rental</code> to Neo4j using our tool</td>
<td>351</td>
</tr>
<tr>
<td>D.6</td>
<td>Specifying the folder location in the user interface <code>query.jsp</code></td>
<td>354</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Schemas play a central role in structured data management. Defining a schema specifies the contents of a database using a specific format or model by defining the structure and relationships of the data stored in it, and it provides a basis for expressing queries. However, while a schema enforces structure on the underlying data that conforms to it, it is also subject to change. Mediawiki, the software supporting Wikipedia, has experienced more than 300 schema versions since 2003 [57]. Vassiliadis et al. studied the schema evolution history of eight databases including content management systems, web stores, and medical and scientific stores [116]. The number of schema versions across these databases varies from 46 (lifetime \(\sim 10\) years) to 322 (lifetime \(\sim 8\) years).

Changes in a schema can arise due to various reasons. The requirements of an organization drive the schema design process and changes can occur as a result of evolving requirements. The need for organizing the data to provide better query performance or correct mistakes that were made during the design phase are other potential reasons. In our work, we address the problem of schema evolution in data warehouse environments.

Data warehouses constitute a specialized form of databases created by consolidating information from two or more sources in order to facilitate reporting and analysis over
the integrated information. They are maintained for long period of time and this sustainability characteristic makes evolution common in such environments. Changing requirements become inevitable during such long durations.

Figure 1.1 represents a high-level overview of the schema-rich, three-tier architecture for a data warehouse design that we refer to in our work. The environment depicted in Figure 1.1 consists of multiple source schemas, a reconciled schema, and the warehouse schema along with the ETL mappings and a set of queries. The information spread over individual data sources is cleaned and consolidated into a reconciled schema (also referred to as an integrated schema), which is then transformed to a multidimensional schema to structure it in a format that is better suited for analytical queries.

Based on the architecture in Figure 1.1, there exist several avenues for schema changes. Changes are possible at any level including operational sources, the reconciled schema, or the warehouse schema. However, a schema element at an operational schema level is not directly mapped to a data warehouse schema element. The reconciled schema serves as the bridge between them. Thus, the impact of making a change at the operational schema level is not directly apparent on the warehouse schema. Given such a change, a natural question that would be of interest to the warehouse administrator or designer is: Will any of the ETL mappings, queries, or mapped schemas break as a result of the change?

Our work provides a framework that allows changes to all schema artifacts in the three-tier data warehouse architecture as well as identification of the the artifacts that depend on them both directly or transitively. This highlights the first contribution of our work. None of the current contributions [37, 35, 75, 101, 27, 79, 28, 22, 109]
incorporate proposing changes to the schema artifacts at all levels and studying the impact over all components in a multi-layered data warehouse environment under one methodology. A discussion of related work is covered in Chapter 2.

Secondly, the aspect of facilitating users' understanding during the process has been largely ignored. While offering a solution that can identify the related artifacts for a given entity in the framework is useful, without any information on how the impacted and the evolved artifacts are related, there may be unintended consequences. The user may perform a manual investigation of the relationships manually to understand the
results, which would demand time and effort in exhaustively analyzing elements and their relationships in the architecture. We address this concern by investigating and leveraging provenance. Provenance comes from a French word, *provenir*, and is often referred to as lineage or pedigree in the database community. Our provenance-based approach offers a richer impact assessment solution, not just the list of all artifacts that are impacted by the change. The list is complemented by the complete path trace information that explains how the impacted artifacts and the evolved artifact are related to each other. This is the second contribution of our work.

Having such a framework in place is useful for the data warehouse designer and analyst in the following ways: (a) raise confidence in the decision towards making the change since he or she will now have an understanding of the implications, (b) avoid or reduce the undo operations that may be required if the designer realizes later that his or her proposed change has impacted other artifacts in an undesired manner, (c) provide an insight into the dependencies between the elements in the architecture, and (d) allow the user in understanding the implications of the proposed change in a step-by-step manner, thus simplifying the process of understanding the impact of change.

We present a short example in the next section to illustrate the use case discussed above for providing the impact assessment in a pro-active manner before actually propagating the change.

### 1.1 Motivating example

To illustrate our approach, we consider a simple data warehouse focused on book store sales. The environment consists of two operational sources (*S1* and *S2*) shown
in Figure 1.2, a reconciled schema \( R \) presented in Figure 1.3 and a data warehouse schema \( DW \) in Figure 1.4. Sample mappings between operational sources and the reconciled schema are shown in Figure 1.5.

Consider the set of operational schemas. Although there exist several commonalities in the features captured by the schemas \( S_1 \) and \( S_2 \) in Figure 1.2, there are a few differences. The book information is split across four different relations in \( S_1 \) (\( Book, Book\_Category, Book\_Price\_Format, \) and \( Book\_Format \)), whereas \( S_2 \) consolidates it in one relation (\( Book \)). \( S_1 \) captures a customer’s address as a string in the \( Customer \) relation itself whereas in \( S_2 \), the \( Customer \) relation is normalized and has a referential integrity constraint with the \( Address \) relation. Furthermore, \( S_2 \) contains additional information related to sale promotions.

The reconciled schema of the two book store schemas \( S_1 \) and \( S_2 \) is shown in Figure 1.3. We present a few sample mappings between the operational sources (\( S_1 \) and \( S_2 \))
Figure 1.3: Reconciled schema

and the reconciled schema \((R)\) in Figure 1.5. For the sake of clarity, we focus only on a small set of mappings here (Figure 1.5). The dotted lines represent containment relationships and solid lines connect source and target objects for the mapping. We employ the modeling scheme introduced by Bernstein [33] to describe the mappings. Mappings are shown in three levels: \(map_i\) denotes the schema level mappings, \(t_i\) represents table level mappings, and \(a_i\) depicts attribute level correspondences. The two filled boxes represent the referential integrity constraints. The primary keys of each relation are underlined. Figure 1.4 shows an excerpt of a warehouse schema with facts and dimensions obtained from the reconciled schema based on the requirements.
Based on the artifacts described in Figures 1.2-1.5, we consider an example to illustrate our framework. Consider an evolution scenario where the designer decides to drop information about book categories from the operational source $S1$. He or she would like to see the impact of this change on the rest of the artifacts in the data warehouse environment, such as mappings and the data warehouse schema. The designer may ask the question: "Which elements will be impacted by this deletion operation?"

Using the algorithms we propose in our work, we inform the user about the mappings and any elements in the reconciled or the warehouse schema that are impacted, along with any queries that may have become unanswerable as a result of the proposed change. In our example above, the impact is reflected on the following elements:


![Data warehouse schema](image)
2. Mappings \( t_2, a_4, \) and \( a_5 \). These mappings relate the \textit{Book.Category} relations in the operational and reconciled schemas \( S1 \) and \( R \) respectively.

3. Mappings between the reconciled schema and the warehouse schema for the \textit{DW.Book.category} dimension (Figure 1.4).

It is important to note that even though mappings are impacted, the reconciled or the warehouse schemas are not affected. Without automation, the changes would require exhaustively analyzing all the documentation including mappings and other artifacts in the data warehouse environment to see if the relation \( S1.\textit{Book.Category} \) has an impact on them.

Continuing with the example, the user may wish to investigate why the relation \textit{Book.Category} in the reconciled schema \( R \) is not reflecting any change and asks the following question: “Which elements are contributing to the existence of the relation \textit{Book.Category} in the reconciled schema?” This question may arise because the user
thinks that \texttt{S1.Book\_Category} is the sole contributor towards \texttt{R. Book\_Category} relation.

Our approach shows that the relation \texttt{R.Book\_Category} is derivable using two paths (mappings \(t_2\) and \(t_6\)) as shown in Figure 1.5. While the mapping \(t_2\) is impacted by the change, \(t_6\) is untouched. This illustrates the use of provenance in our work. We provide the complete path information between the different artifacts in the architecture thus providing an insight to the user about the identified impact. In summary, our paper makes the following contributions:

1. We develop a graph-based framework that allows schema changes at any layer in the data warehouse architecture and identifies the impact of the change over the rest of the warehouse environment.

2. We illustrate a use-case for employing provenance in the context of lineage. The list of impacted artifacts is complemented by path trace that shows how the evolved and impacted artifacts are related to each other. This facilitates deeper understanding of how the impact assessment results are derived. We illustrate the effectiveness of the approach using a detailed case study.

3. We contribute a case study that covers each component in the data warehouse architecture from Figure 1.1. We leveraged available artifacts from literature and incorporated additional artifacts to offer a comprehensive dataset consisting of all artifacts from Figure 1.1. The artifacts developed by us include the two operational schemas, revised reconciled schema, ETL transformations between the operational and reconciled schema and synthesizing a set of SQL queries.
The ETL transformations between the reconciled and the warehouse schema are provided by Casters et. al [43]).

Table 6.2 presents an overview of the functionalities that we aim to provide in our work. The intuition and real-world scenarios that may necessitate changes to the artifacts at different levels in the warehouse architecture is described in the following table.

Table 1.1: User interaction and impact of our proposed system

<table>
<thead>
<tr>
<th>System feature</th>
<th>Use-case</th>
<th>System Result</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Propose a change in the operational schema</td>
<td>Reorganization of the schema/adding new information</td>
<td>Detect any intra-schema violations, changes on the transformations, the reconciled and warehouse schema, end-user queries</td>
<td>Gain an understanding of the mappings authored by someone else.</td>
</tr>
<tr>
<td>2. Propose a change to the reconciled schema</td>
<td>Changes can be proposed directly to the reconciled version instead of specifying them over independent operational schemas.</td>
<td>Detect changes on the mappings that populate the evolved artifact, any intra-schema impact, end-user queries, and the impact on the warehouse schema including mappings between the reconciled and warehouse schema</td>
<td>Gain insight on the relationship between the operational and reconciled schema, and the mappings between reconciled and the warehouse schema.</td>
</tr>
<tr>
<td>3. Propose changes to warehouse schema</td>
<td>Changes in the end user's analytical needs</td>
<td>Notify the user about the impacted mappings between reconciled schema and the warehouse schema and end-user queries on the warehouse schema</td>
<td>Gain understanding of the relationship between reconciled and the warehouse schema.</td>
</tr>
</tbody>
</table>
System features in Column 1 lists the inputs that our proposed system can receive and Column 3 represents the corresponding results that will be delivered. Column 2 presents the rationale behind supporting the proposed features in the form of use cases. The column Impact presents the usefulness of the obtained results highlighting the contribution of our work.

1.2 Limitations of the current contributions

Most of the previous research efforts have focused on algorithms for propagating changes in an automated manner. While the current contributions simplify the task for the user by providing a programmatic way of adapting related artifacts, there exists additional merit in revealing the impact in a pro-active manner before silently propagating the change.

Consider an environment where the mapping designer and the administrator who maintains the system after it has been setup are two different people (or different teams of people). Consider a change over a schema that is propagated to the related artifacts in the architecture based on algorithms proposed in the literature. It will be much easier for the mapping designer to understand why certain elements in the target schema require adaptation because of his/her understanding of the mappings (being the author) that connect the two schemas. However, for a person who is unaware of the contents of the mappings, solely relying on the proposed algorithms to propagate the change may not be very helpful.

The impact of a change may be much larger than the change that the user is expecting and this can occur due to the user’s lack of understanding of all the mappings. An element might have been renamed as a result of the change that a user proposed but
because it was automatically propagated by the algorithms, the particular impact of the proposed change on that specific element went unnoticed by the user. As a result, at a later point while using the schema, the user may fail to understand the existence of the renamed element. Thus, defining a model that can report the consequences of the proposed changes over other elements before actually propagating the change is a useful contribution. It allows the user to gain an understanding of the mappings.

Furthermore, in a data warehouse environment consisting of multiple operational source schemas, a reconciled schema, a set of information requirements, the warehouse schema along with mappings, a schema element at the operational level is not directly mapped to a data warehouse schema element. Reconciled schema serves as the bridge between them. Thus, the impact of making a change at the operational schema level is not directly apparent to the warehouse schema.

Apart from the lack of an approach that can report the result of proposing a change to an artifact, the current contributions are also restricted in the sense that they mainly address changes to a single component in the warehouse architecture. Based on the architecture in Figure 1.1, changes are possible to schemas at any level including sources, reconciled schema or the warehouse schema. But none of the current contributions incorporates changes to these multiple artifacts under one methodology.

Considering the architecture in Figure 1.1, we are not aware of any existing work that offers a case study consisting of all the artifacts in our architecture. We have surveyed literature for resources and found a set of reconciled and data warehouse schema along with the ETL transformations between them from Casters et al. [43]. However, these still do not cover the operational schemas and ETL transformations between
them and the reconciled schema. Having a set of queries was another requirement to complete our architecture.

Our case study in Chapter 6 presents the design and development of two operational schemas, a revised reconciled schema based on the open-source sakila database provided by the MySQL community and sixteen ETL transformations for mapping of the operational and the reconciled schema. The database scripts for generation of each of these schemas, ETL workflows and queries are available in Appendix A. This serves as another contribution of our work.

1.3 Problem statement

Our research objective is to develop a provenance-based approach for providing an assessment of the impact of schema level changes to the related schemas and mappings in a data warehouse environment.

1.4 Research objectives

In order to develop a methodology for assessing the impact of schema changes in a warehouse environment, specific research objectives are identified here. Table 1.2 enumerates our specific research objectives and the motivation behind addressing each of them and the contributions made.
<table>
<thead>
<tr>
<th>Motivation</th>
<th>Research Objective</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. With heterogeneous artifacts forming our research environment and our research goal focused on identifying dependencies between them, we need to identify a model that can capture all these elements in a uniform and intuitive manner.</td>
<td>Perform a literature survey for existing techniques on modeling schemas, mappings and dependencies between them.</td>
<td>A graph based model is identified to be well suited for our requirements.</td>
</tr>
<tr>
<td>2. In order to support representation of all artifacts of our research environment in a uniform and intuitive manner and keeping the graph based model in mind, a conceptual scheme needs to be selected.</td>
<td>Perform a literature survey for existing techniques on modeling schemas and mappings.</td>
<td>A conceptual formalism based on UML is developed to capture various artifacts in our research environment.</td>
</tr>
<tr>
<td>3. With the conceptual formalism in place, we need to gain confidence in the coverage and expressiveness of our proposed model.</td>
<td>Demonstrate expressiveness of the conceptual model using case study.</td>
<td>A sample instantiation demonstrating the effectiveness of our conceptual model is presented.</td>
</tr>
<tr>
<td>4. With the conceptual formalism defined, the artifacts need to be transformed from the conceptual model to the chosen graph formalisms.</td>
<td>Develop and implement the algorithm that takes schemas and mappings as input and transforms into the selected graph formalism based on the conceptual model.</td>
<td>The schemas, mappings and queries are transformed from their native formats to the graph model.</td>
</tr>
<tr>
<td>5. With the input artifacts in the target graph-based model, the next step involves proposing schema changes and reporting the elements that are impacted by the change.</td>
<td>Develop a methodology to identify the elements that will be affected by the proposed change. This will constitute the idea of impact assessment.</td>
<td>Change specification and impact assessment methodology is determined. A report in the form of graph is generated with the list of elements affected by the change as the result.</td>
</tr>
</tbody>
</table>

*Continued on next page*
Table 1.2 – 

<table>
<thead>
<tr>
<th>Motivation</th>
<th>Research Objective</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. Having identified the impact of the proposed change, the next goal is to provide further information to the user that explains why the identified elements are impacted.</td>
<td>Augment the algorithm developed in Step 5 above to provide explanations detailing why a particular schema element displayed in the output of Step 5 will be impacted by the proposed change.</td>
<td>An algorithm that generates explanations accompanying the notification is developed.</td>
</tr>
<tr>
<td>7. The validation of the effectiveness of proposed algorithms and support for automation creates a need for a software implementation.</td>
<td>Demonstrate effectiveness of our proposed algorithms using a case study and develop an interface to allow designer/end user observe the result of the proposed change in an intuitive manner.</td>
<td>A software prototype and a case study demonstrating the effectiveness and automation of our proposed approach is developed and illustrated.</td>
</tr>
</tbody>
</table>

1.5 Outline of the dissertation

The rest of the dissertation is organized as follows. In Chapter 2 we discuss the related work in the impact assessment in data warehouses and provenance. Chapter 3 describes our conceptual model for modeling different artifacts (relational schemas, ETL workflows and queries) in the architecture. We demonstrate a sample instantiation of the model using a short example consisting of two relations belonging to two different schemas, an ETL transformation between them and a sample query over one of the relations. Given that we are using a graph-based implementation as the target model, the section also discusses how the proposed conceptual model maps to the underlying graph implementation. Chapter 4 provides an overview of the concepts and terms that are used frequently in the paper. These include a discussion of Neo4j.
graph database and Pentaho, a business intelligence and data integration tool used for expressing ETL workflows in our work. Chapter 5 introduces our architecture and describes the implementation methodology of our work. We demonstrate the graph transformation and impact assessment over an open source sakila database expressed in MySQL. The rationale behind choosing sakila database is because of the wide community support and resources available for it in terms of schemas, queries and ETL workflows. The comprehensive case study will be used as a running example to demonstrate the application of our conceptual model. The complete case study including the evaluation results are discussed in Chapter 6. In Chapter 7 we present our user interface that accomplishes the functionality based on the conceptual model from Chapter 3 and illustrated in Chapter 6. While this completes discussion of our work in the context of schema evolution in a data warehouse environment; next we discuss our work on the use of graph databases towards schema integration in Chapter 8. Chapter 9 offers conclusions and future work.
Chapter 2: Background Research

The main theme of the dissertation is to develop a graph-based framework that allows schema changes at any layer in a data warehouse architecture and identifies the impact of the change over other artifacts in the architecture. The complete path from the evolved artifact and each of the artifacts that are impacted by the change is presented, highlighting a use-case for provenance in the context of lineage. In this chapter we present the research contributions which are closely related to our work.

We present our analysis in two sections: (a) state-of-the-art for approaches on schema evolution in warehouses, and (b) contributions in the field of provenance. Our literature study highlights how our work differentiates from the existing efforts in the field and our methodology for contributing to the resolution of some of their limitations.

Schema evolution and data warehouses have been established areas of research for more than two decades [86]. Schema evolution is the ability of the system to make changes to a schema and adapt its dependent artifacts in order to reflect changes in the real-world. This chapter surveys the research efforts in the field in a data warehouse environment which consists of multiple schemas connected by mappings. This chapter contributes a structured summary of the research efforts in the field in a data warehouse environment. While we are aware of the excellent survey papers that provide a detailed summary of the contributions in the field of schema evolution in data warehouses,
our survey augments those efforts in terms of incorporating additional categorization criterion allowing us to perform and present a more comprehensive comparative study.

Research efforts in schema evolution concentrate on the following aspects: (a) changes addressed over stand-alone databases [27, 35, 22, 50, 79, 75, 28], (b) the set of evolution operations covered, (c) propagation of changes to the underlying data and queries, and (d) schema versioning approaches [75, 92, 63]. In the context of a mapped environment, the contributions have mainly focused on two schemas connected by a set of mappings and approaches for adapting those mappings and the related schemas [117, 110, 19, 53, 24]. Although data warehouses fall into the category of mapped environments, managing evolution in warehouses involves additional artifacts such as three levels of schemas, ETL mappings, and the queries expressed over the schemas.

2.1 Data warehouse schema evolution

Table in Figure 2.1 presents a summary of schema evolution research for data warehouse environments. The header *Changes made at* classifies the contributions based on the artifacts on which a change is proposed. Changes are possible to the artifacts at any level of the architecture in Figure 1.1. A change can be proposed to any of the independent operational schemas, the reconciled schema, or the warehouse schema. The header *Change propagated to* classifies the literature based on the elements that they revise when a change is proposed to an artifact. The set of mappings between operational source schemas (Op) and the reconciled schema (R) is labeled as *Op-R ETL*, and the mappings between reconciled (R) and data warehouse (DW) schema are labeled as *R-DW ETL*. The column *Provenance* indicates whether the research leverages provenance. The last column, *Impact assessment* report indicates if the work
provides feedback to the user regarding the elements that will be impacted by the proposed change. Our work is shown as the last entry in the table and we address all of the given criteria.

![Table of schema evolution research](image)

Figure 2.1: Summary of schema evolution research in data warehouse environments

Most of the previous approaches to data warehouse schema evolution focus on proposing changes to a single component in the warehouse architecture [35, 75, 101, 27, 79, 28, 22, 109, 41]. Blaschka et al. [35] propose a formal framework for describing multidimensional schemas and operations to support evolution of dimensions and facts at a conceptual level. Hurtado et al. [75] focus specifically on making changes to the dimensions in the warehouse schema. Quix propose a metamodel for data warehouse evolution and focuses on the quality factors of warehouse schema [101]. Guerrero et al. [27] introduce a multidimensional data definition language to express changes to...
the warehouse schema independent of the database model which is used to implement the schema. Kaas et al. [79] look at schema evolution in data warehouses at a conceptual level. They study evolution operations over star and snowflake schemas and how to adapt underlying instances and queries in the event of a change. Bentayeb et al. [28] focus on adapting end user queries when a change is made to the warehouse schema. Banerjee et al. [22] allow specification of evolution operations over extended dimension hierarchies in addition to the core warehouse constructs at a conceptual level. The authors present algorithms that check for schema consistency whenever an evolution operation is specified. Feki et al. [109] focus on studying the impact of making changes over the warehouse schema on data marts. The authors observe the schemas of warehouse and data marts at different levels of abstraction. Data warehouse schema is considered at logical level consisting of a set of tables representing facts and dimensions. However, the schema of data marts is assumed to be conceptual and multidimensional. The changes to the warehouse schema include adding or dropping a table. The consequences of such a change are observed as a potential addition/deletion of a fact/dimension respectively on data mart. A common trait of all the above contributions is that they allow changes specifically and solely to the warehouse schema.

Similarly, there are contributions that address the impact of changes made to the reconciled schema [37, 29, 24]. Bellahsene focuses on proposing changes over a relational schema and rewriting the views defined for constructing the warehouse schema [24]. The paper discuss adaptation of the existing views by considering each view and examining its clauses (e.g., select-from) to check if the affected elements are present in the clause. Bernstein et al. [29] discuss how model management operators can be
applied to solve problems in data warehouse environments. Given an integrated relational schema, a data warehouse schema, and a set of mappings in the form of views, the authors present a sequence of high-level model operations that can be applied to integrate a new data source and identifying the required, additional mappings in an automated manner. Bouzeghoub et al. [37] propose a logical model for studying the impact of making changes to the sources on the materialized views. The changes on the data sources are restricted to integration and deletion of a source schema. Revising an existing data source schema is not addressed. An et al.[19] present an approach for incrementally maintaining mappings between conceptual and relational schemas under an event of schema change in either of them.

We extend the contributions of these approaches by allowing the warehouse designer to propose schema changes over additional artifacts besides the warehouse or reconciled schema.

Similar in spirit to the idea of slowly changing dimensions in data warehouses, Goller et al. [66] introduce the concept of slowly changing measures. This pertains to the evolution of those functions in ETL process that compute the values for fact’s measures. Our work will supplement their approach of ETL evolution. However, the focus of our work is restricted to schema evolution.

Jovanovic et al. [78] and Mate et al. [89] study the evolution of a warehouse schema based on changes to the information requirements. Jovanovic et al. [78] address the adaptation of the warehouse schema as a result of integrating new information requirements or revising existing ones. A multidimensional representation is developed for each information requirement by identifying the set of concepts from the data sources
that serve a role in fulfilling that requirement. The individual multidimensional representations of each information requirement are integrated to generate a complete data warehouse design. Thus, while data sources contribute in developing the multidimensional representation for an information requirement, but changes in the schema of the sources is not the focus of this paper.

Similarly, Mate et al. [89] propose a set of semantic mappings to relate the elements of the source schema with the concepts specified in the user requirements. Our research context is different. We focus on identifying and notifying the user about how changes to any schema artifact in the architecture influence related schemas, ETL workflows, or queries. Their approaches do not investigate the impact of making changes over source schemas. Our work addresses this need.

Thus, we recognize two open problems. Firstly, the previous approaches manage schematic changes made to a single component in the warehouse architecture, but none of them incorporates changes to multiple schema artifacts under one methodology. Secondly, there are no tools that can provide feedback to the user regarding the changes that are made in the architecture as a result of the proposed modification to a specific artifact. We address both these limitations in our work.

Mate et al. propose a trace metamodel based on model-driven architecture (MDA) to capture relationships and provide traceability between data source concepts and information requirements [88]. Jovanovic et al. [78] also maintain a traceability model relating the information requirement and the corresponding multidimensional schema component and to store the candidate, alternative multidimensional representations. The traceability metadata contains information about the alternative designs possible for the particular information requirement. These approaches draws our attention
towards the relevance of incorporating trace information and we incorporate this idea by employing provenance in our work.

Apart from studying the research contributions on the basis of the artifacts that are addressed for evolution, we look at the models and approaches that are adopted as another perspective to categorize the evolution research. Golfarelli et al. [64] present a UML-based modeling approach for modeling what-if analysis for BI applications. However, their methodology does not concentrate on schema evolution issues. Braunschweig et al. [40] advocate the idea of leveraging a graph-based model to support schema evolution due to the flexibility it offers. Papastefanatos et al. [95, 96] use a graph-based formalism for modeling ETL queries and their dependencies on the elements from the source schemas. We derive inspiration from their graph-based approach but extend their formalism to incorporate additional artifacts including warehouse schemas and other artifacts in the three-tier data warehouse architecture shown in Figure 5.1.

The contributions by Hoang [73], Solodovnikova [108], Maule [90], Butkevicius et al. [41], and Papastefanatos et al. [95] are most closely related to our work. Maule’s work presents novel techniques for identifying the impact of database schema changes over applications written in object-oriented applications. Recently, Butkevicius et al. [41] have developed an ETL maintainence manager which allows for automatic adaptation of ETL processes if a change is detected on the data sources. The authors also leverage property graph model to capture the dependencies between the source schema elements and the ETL transformations. This closely aligns with our work. Solodovnikova [108] presents a data warehouse evolution framework that’s allows for changes at the data sources or the data warehouse schema and propagating the impact on ETL flow and the warehouse schema. Hoang et al. [73] highlight the importance of addressing the
problem of impact assessment and propagation in data warehouse environment. The proposed approach is focused on leveraging semantic web technologies in order to identify and define relationships between different layers in the warehouse. However, we observe two open research problems here. Firstly, two approaches [73, 108] present a framework but the implementation is missing. Secondly, both approaches can be extended to incorporate additional elements, namely a staging layer called the reconciled schema, that also contributes to the operation of a data warehouse. Our approach is a more holistic solution to the problem of schema evolution in data warehouses.

The study of the existing works helps us in identifying opportunities for research in the field of managing schema evolution. We draw following conclusions:

1. The contributions have mainly considered the changes over one component in the data warehouse environment

2. The proposed approaches do not provide any explanation of the various steps along the process. We aim to leverage provenance in order to assist the designer/end user in providing an understanding of how the changes are propagated.

2.2 Provenance

The idea of providing explanations along with the output to enable a user’s understanding has gained research attention recently [83, 105]. Koutrika et al. [83] generate explanations for SQL queries in plain English. Similarly Roy et al. [105] recognize the need for a tool that can assist users in understanding query results. These contributions highlight the notion that solutions should be backed up by further information that can assist user’s understanding of the results. We derive inspiration from these research efforts to support this notion through the idea of provenance.
Provenance of a data item defines the data and the processes that have led to its current state, termed as source and transformation provenance, respectively. It is an area of significant interest as evidenced by numerous research papers [44, 46, 47, 48, 49, 60, 68, 67, 119, 59, 80] and surveys [107, 61, 76].

Research efforts in the field are concentrated on the following aspects: (1) the application/use-case supported by provenance, (2) environment indicating whether provenance is studied in the context of independent schemas or a mapped environment, (3) subject of provenance, (4) models used for representing and storing provenance information, and (5) information that defines provenance. The information can include data such as a timestamp when a particular data item was modified, information about the user who made the changes, or the data that is serving as a source of the data item. Given an artifact for a schema change, we provide a complete path trace of this artifact with all its related artifacts. This constitutes the idea of provenance in our work.

The first criterion, **Application area**, highlights the scenarios where provenance information has been applied. Reporting [58, 100, 81], debugging of schema mappings [47, 60], and forward propagation and forward tracing [76, 68] have been addressed. In the environments that involve connecting information from sources on the web, provenance becomes fundamental for assessing trustworthiness of data coming from a specific source [80, 119]. We are not aware of any work that leverages provenance for the **impact assessment** of schema evolution in data warehouses.

The second category, **Environment**, defines if the evolution is studied over independent schemas or mapped schemas. Our work falls in the mapped schema environment, specifically data warehouses. We are aware of several research efforts [87, 88, 49, 78]
that leverage provenance in data warehouse evolution. Mate et al. [87, 88] address provenance with the idea of relating requirements, data sources, and warehouse schemas. They introduce multiple trace models in order to capture the relationship between different artifacts. Jovanovic et al. [78] maintain a traceability model relating the information requirements and corresponding multidimensional representation for that requirement. Cui et al. [49] leverage provenance in a data warehouse environment to identify sources of a data item in a materialized data warehouse view. While their work focuses on the provenance of data instances, we address provenance of schema elements in a data warehouse environment setting.

The next criterion, \textit{Subject of provenance}, identifies the information that constitutes provenance. It is classified as source provenance and transformation provenance. While source provenance provides information about the data items that lead to creation of the result, transformation provenance identifies the process or mapping that was used to create the result data item. We record both of them in our work. Source provenance in our work corresponds to the schema elements or queries and transformation provenance is composed of ETL mappings.

Provenance in the modeling context refers to the model used for representing provenance information. Simmhan et al. [107] point to the lack of a metadata standard for lineage representation but identify the different formats that are adopted by existing systems. XML and ontologies are two such formats. World Wide Web Consortium [12] has recently proposed a UML-based metamodel, called PROV, for recording provenance with the idea of ensuring interoperability between various heterogeneous schemes under which provenance is captured by different systems. Holland et al. [74] present pros and cons of different modeling schemes for representing provenance. The authors
believe that a semi-structured model such as XML is best suited for provenance due to its relatively flexible schema structure compared to the relational model. Graph-based formalisms and the relational model are also commonly adopted for representing provenance [67, 68, 113, 83, 58]. We employ graph-based model towards our implementation for representing artifacts including provenance. In terms of the method used for storing provenance information, it pertains to the idea of coupling between the provenance data and the actual data that it describes. Simmhan et al. [107] identify three storage strategies: (a) the no-coupling approach, in which provenance information is stored in dedicated repositories that are used for storing lineage information only, (b) tight-coupling, in which provenance information is stored directly with the data for which it is recorded [48, 113], and (c) loose-coupling, in which source data and provenance are stored on a single storage system but logically separated. As an example, in the relational database, logical separation is enforced by defining separate relations for recording the core information and provenance. We follow the tight-coupling approach.

Provenance approaches also differ based on whether the information that they describe is about schemas or data instances. Schema-based provenance is used while working on schema management tasks, such as schema matching, evolution, and merging. Gao et al. [58] differentiate between schema and data provenance by addressing the former as the information about schema modification operators that describe the schema evolution history. Our work is focused on provenance of schema elements.

We illustrate a use-case for provenance metadata by applying it towards impact assessment of schema evolution in a data warehouse environment. Our research provides a framework that will allow changes to all artifacts in the three-tier data warehouse
architecture, thus addressing multiple evolution problems under one framework. Furthermore, in addition to the identification of related artifacts that are impacted by the change, we also provide explanations detailing why a particular schema element displayed in the output will be impacted by the proposed change. The complete path trace from the element where the change was made to the impacted elements offers one form of explanation. We also offer a text-based description of each path in the graph starting from the evolved artifact.
Chapter 3: Conceptual Model

In this section, we present our conceptual model of the three-tier architecture (Figure 1.1). The model captures schemas (operational, reconciled, and data warehouse), ETL transformations, queries and the dependencies among all these artifacts using a UML-based formalism. Having a conceptual model based on a standard formalism such as UML serves two purposes: (1) it facilitates a quick and clear specification based on the standard UML conventions, and (2) it allows representation of different heterogeneous artifacts in a uniform and implementation independent manner. We also discuss design alternatives and the rationale behind our choices for selecting one modeling approach over another.

Before choosing any specific modeling scheme for our work, we perform a survey of the formalisms adopted by other researchers who have addressed schema evolution and/or provenance in their work. Relational [24, 57, 109, 66] and conceptual models [35, 33, 28, 88, 89, 102] are widely used for representing schemas. For mappings, we observe different formalisms adopted by researchers in the past: (a) logic-based formalisms, as tuple generating dependencies [52], (b) conceptual or visual models [85, 102, 33, 99] such as UML or by allowing the user to draw and annotate lines to represent relationships between schema elements, (c) model-specific formalisms such as SQL queries [24] or mappings [56] to capture relationships between hierarchical
schemas such as XML that allow parent-child relationship structure, and (d) graphs [96, 40].

We consider representation of artifacts at three levels: (a) native representation formats, (b) a conceptual representation that facilitates common and quick understanding among different groups of users and offers an implementation independent representation of heterogeneous artifacts, and (c) implementation model for conducting impact assessment. In this chapter, we focus on identifying and presenting a conceptual representation scheme that allows modeling of all artifacts of our research environment in a uniform manner. We leverage a UML-based formalism towards our conceptual model.

The reason we have adopted a conceptual model based on UML class diagrams is the wide acceptance of UML for expressing different kinds of artifacts and a conceptual model facilitates understanding among different groups of users. However, to adapt UML to capture schemas and mappings in a data warehousing environment, we need to extend it to capture the semantics of that environment.

Our methodology builds upon the idea proposed by Luján-Mora et al. [85] for extending UML to represent attributes as first-class modeling artifacts. UML provides extension mechanisms for extending its vocabulary to allow creating new elements from the existing ones such as classes. The new elements are then treated as first class citizens. In the core UML model, attributes are encapsulated inside a class and thus it is not possible to capture relationships between attributes. However, in our work that is based on proposing schematic changes, it becomes important to represent mappings between the attributes. We employ the extension proposed by Lujan-Mora et al. [85] which models attributes as first-class modeling elements. This allows attributes to participate in associations and thus depict attribute-level relationships.
We augment their approach by providing additional extensions. The other extensions come from differentiating between the classes we plan to leverage in our work based on the artifact that they are representing. As an example, all the artifacts including schemas, relations, attributes, ETL transformations, and queries will be represented by classes in UML but in order to differentiate between them, we plan to extend classes using stereotypes. They are specializations of existing model elements. The attributes are modeled using UML’s extension mechanism and are labeled as <<Attribute>>.

Figure 3.1 shows our conceptual model.

Figure 3.1: Conceptual model for our approach

The nodes labelled Artifact focus on the artifacts in our three-tier architecture while the other two UML classes labelled Enumeration serve as references for the other classes. For example, the values for the property steptype in the artifact ETL Step belongs to one of the values from the enumeration StepTypeVal. The specific components of the architecture are annotated using stereotypes Schema, Relation, Attribute,
ETL Step, and Query. The associations denote the dependency relationship, Impacts, between elements. For example, the relationship from the class labelled Schema to the class Relation in our model implies that a change in a schema Impacts the relations that belong to it.

In addition to the dependency relationship Impacts the association also capture the semantics of the relational model. The self-referential arc between the Attribute captures the foreign key relationship between two attributes. Similarly the relationship from the Attribute class to the Relation class presents the entity integrity constraint over the relation. The attribute serves as the primary key for the specific relation.

The ETL transformation and its dependencies to other schema artifacts in the framework are captured by the relationship between nodes Attribute and ETL Step. The UML class with the stereotype ETL Step represents a step in the ETL transformation. There are four properties attached to this node. The property id refers to a system-defined identifier while name is the name of the step specified by the ETL designer while developing the workflows. The property transname captures the name of the transformation that the step belongs to and the property steptype is a string value that represents the type of database-related step. The value for this property is implementation-specific and in our work here it is based on the set of database-related steps available in Pentaho (TableOutput, Insert/Update, Dimension Lookup/Update, Combination Lookup/Update and DBJoin). Pentaho is a leading business intelligence and data integration tool and we employ it to capture the ETL mappings in our work. We provide an overview of the implementation technologies used in our work including Pentaho in Chapter 4. The UML class labelled StepTypeVal holds the values for these different step types in our conceptual model.
Other than schemas and ETL workflows, the conceptual model shows SQL queries which are modelled using the class labelled *Query*. They are impacted by attributes and this is depicted by the relationship between the classes *Query* and *Attribute*.

This concludes the high-level description of our conceptual model. We discussed how the model considers each of the artifacts in our three-tier architecture and how the conceptual model maps to the underlying target graph implementation. In the next section, we discuss the choices we had to make during the design of the conceptual model. There may exist multiple ways of modeling a particular domain and set of application requirements. We explain the rationale behind our choices for selecting one modeling approach over another.

### 3.1 Design considerations

In the process of designing our conceptual model, we considered the following design alternatives: (1) directionality of the dependency relationship, (2) modeling of the entity integrity and referential integrity constraints, (3) the need to model inter-step dependencies in an ETL transformation, and (4) directionality of the relationship between the *ETL Step* nodes and the *Attribute* nodes. We discuss each of these design choices below.

1. **Directionality of the relationship**: In our proposed conceptual model, we have a directed relationship from a *Schema* to a *Relation* class and similarly from a *Relation* to an *Attribute*. The semantics of this relationship is that one class (tail of the relationship) impacts the other artifact. For example, a drop operation on a *Schema* impacts *Relations* that belong to it.
We also considered the possibility of reversing the relationship direction and thus create a directed edge from a relation to a schema, for example. The reason for not choosing the later option is by keeping in mind the relevance and semantics of the relationship. The relationship between our classes in the conceptual model capture an *Impacts* relationship. In other words, if there exists an edge between two classes in our graph implementation, this would read as the following: The tail of the node (source) *impacts* the head node (target).

In this context we notice that while a change to a schema would impact all its constituent relations and further attributes, the reverse is not true. As an example, dropping a relation will not impact its parent schema but dropping of a schema will impact its relations.

2. *Modeling of the primary and foreign key constraints:* During the process of design of our conceptual model, we realized that we could have created additional nodes to explicitly represent these constraints and connect those nodes to the appropriate attribute nodes. For example, instead of having a directed edge from an attribute class labeled *Attribute* to the relation labeled *Relation* in Figure 3.1, the entity integrity constraint could also have been modelled by creating another class with the stereotype *PrimaryKey*. This class *PrimaryKey* could further be connected to the attribute class *Attribute* which serves as the key.

In our conceptual model, we capture these constraints using relationships (directed edges between nodes *Attribute* and *Relation*) more succinctly and without loss of information.
3. **Self-referencing relationship between the Step nodes**: In the context of ETL workflow modeling, it is important to consider inter-step dependencies within a single ETL transformation.

![Figure 3.2: A sample ETL transformation depicting inter-step dependencies](image)

The sample ETL transformation in Figure 3.2 illustrates this idea. The workflow is depicted using Pentaho. It consists of three steps. The directed edges represent that the output of one step serves as the input of the next. Thus, an impact on any of the steps including `actor` and `max_dim_actor_last_update` will consequently impact the `Insert/Update` step in the transformation. This idea is captured by the self-referential relationship between `Step` nodes.

4. **Bidirectional relationship between the Step and Attribute classes**: Another important point to note in the conceptual model is the bidirectional relationship between the `Attribute` and `Step` nodes. Considering ETL transformations between reconciled schema and the warehouse schema as an example, the reason behind modeling this bidirectional relationship is that while the attributes in the reconciled schema serve as the source for an ETL step (Attribute impacts ETL Step), the attributes of the warehouse schema are in fact the targets for an ETL step (ETL Step impacts Attribute). Thus, the relationship between an
Attribute node and a Step node will depend on the schema and the layer in the data warehouse architecture that the attribute belongs to.

This concludes the discussion of our conceptual model. In the next section, we present a formal description of how the conceptual model would map to our underlying graph-based implementation.

3.2 Formal description of the mapping between conceptual model and the target graph implementation

We adopt a graph-based approach as the implementation model for capturing various artifacts and provenance in our work. Graphs not only provide a simple and flexible abstraction for modeling artifacts of different kinds in the form of nodes and edges but also support provenance. The idea of provenance correspond to the complete path trace from the element where the change was made to the impacted elements.

In this section, we describe how the UML conceptual model described above maps to the target graph based implementation. Each of the classes (labelled Artifact) correspond to a node in the graph and the associations map directly to the edges between the corresponding nodes.

The conventional representation of directed graphs is a pair $G=(V, E)$ where

1. $V$ is the set of vertices in the graph, and

2. $E$ is the set of edges representing a binary relation between the nodes such as $e_1 = (v_1, v_2)$ where $v_1, v_2$ are the vertices in the graph.

The vertices in our conceptual model are connected by a binary relation Impacts such that

$$(v_1, v_2) \in E \iff v_1 \text{ Impacts } v_2$$
According to this representation, our conceptual model (Figure 3.1) can be formally described as \( G = (V, E) \) where

1. \( V = \{S, R, A, E, Q\} \), and

2. \( E = \{(S, R), (R, A), (A, R), (A, A), (A, E), (R, E), (E, R), (E, E), (E, A), (S, E), (E, S), (A, Q), (R, Q), (S, Q)\} \)

The symbols in the vertex set \( S, R, A, E, Q \) correspond to schema, relation, attribute, ETL step, and query, respectively. Each pair of elements represent \( v_1 \) impacts \( v_2 \) if a change is made to \( v_1 \).

In the next section, we illustrate the effectiveness and usefulness of our proposed model by presenting a sample instantiation of the model using a small set of artifacts.

### 3.3 Example - Instantiated conceptual model

We consider a subset of the sakila database to illustrate an instantiation of our conceptual model based on the formalism discussed above. Sakila is an open-source database provided by MySQL community. It represents functionality for a fictitious DVD rental store. In this example here, we consider the set of artifacts as shown in Figure 3.3.

The artifacts are described as follows.

1. Two sets of schemas - sakila\_op1 for the operational schema and sakila\_ids representing the reconciled schema.

2. Relations actor and actor in the operational and reconciled schemas, respectively.
3. Attributes `actor_id`, `fname`, `lname`, `last_update`, `ds_actor_id` and `ds_name` for the relation `actor` in the reconciled schema, and Attributes `actor_id`, `fname`, `lname` and `last_update` for the relation `actor` in the operational schema.

4. ETL transformation `load_actor` as shown in Figure 3.2.

5. A SQL query (shown below) retrieving first and last names of all actors whose last name consists of `SON`.

   Q1:
   ```sql
   SELECT actor.fname,
          actor.lname
   FROM actor
   WHERE actor.lname LIKE '%SON%'
   ORDER BY actor.fname;
   ```

   The conceptual model based on the five artifacts above is presented in Figure 3.4.

   In the interest of space, we show only a subset of artifacts in the instantiated conceptual model. Circle 1 in Figure 3.4 shows the relationship between the relation `actor` connected to four of its attributes (`actor_id`, `lname`, `fname`, and `last_update`). Circle 2
Figure 3.4: A sample instantiation of the conceptual model highlights the entity integrity constraint between an attribute (actor_id) and a relation (actor). The relationship Impacts has a property type with the corresponding value as PrimaryKey. A relationship between an attribute (fname) and a query is highlighted by Circle 5. Similarly, the dependencies between an ETL step and attributes is reflected in Circles labeled 4. It shows two examples. In one example, the ETL Step is responsible for inserting data into the attribute while in the other example (upper Circle 4), it shows how an attribute is being read by an ETL step. Circle 6 shows the dependency between ETL steps and relations and a relationship between a
relation \((actor)\) and a query is highlighted by Circle 7. Finally, the Circles labelled 3 show the inter-step dependencies. There are two steps, \(\text{max\_dim\_actor\_last\_update}\) and \(\text{actor}\), of type \(\text{TableInput}\) in Figure 3.4. Each of these steps impact the third step of the transformation, \(\text{Insert Update}\). Although it is a short example, it contains at least one instance of each of the following different relationships described in our conceptual model (\(\text{Inter-step, Relation-Attribute, Attribute-Query, Step-Attribute, Attribute-Step, Entity Integrity Constraint}\)).

Considering the mapping between the conceptual model and the underlying graph implementation described in Section 3.2, the same sets of artifacts (Figure 3.3) can be described using the graph-based formalism as follows.

1. \(V = \{\text{sakila\_op1, sakila\_ids, actor, actor, actor\_id, fname, lname, last_update, actor\_id, first\_name, last\_name, last\_update, ds\_actor\_id, ds\_name, max\_dim\_actor\_last\_update, actor, Insert Update, Q1}\}\), and

2. \(E = \{(\text{sakila\_op1, actor}), (\text{sakila\_ids, actor}), (\text{actor, actor\_id}), (\text{actor, fname}), (\text{actor, lname}), (\text{actor, last_update}), (\text{actor, actor\_id}), (\text{actor, first\_name}), (\text{actor, last\_name}), (\text{actor, last\_update}), (\text{actor, ds\_actor\_id}), (\text{actor, ds\_name}), (\text{last\_update, max\_dim\_actor\_last\_update}), (\text{actor\_id, actor}), (\text{fname, actor}), (\text{lname, actor}), (\text{last\_update, actor}), (\text{first\_name, Insert Update}), (\text{last\_name, Insert Update}), (\text{last\_update, Insert Update}), (\text{actor\_id, Insert Update}), (\text{fname, Q1}), (\text{lname, Q1}), (\text{actor, actor}), (\text{load\_actor, actor}), (\text{actor, sakila-q1})\}\}

The vertices consist of the two schemas - \(\text{sakila\_op1, sakila\_ids}\), the relation - \(\text{actor}\) in each of the schemas, attributes, three steps \(\text{max\_dim\_actor\_last\_update, actor, Insert Update}\) of the ETL transformation (Figure 3.2) and one SQL query \(\langle Q1 \rangle\). Edges
are the set of ordered pair \((u, v)\) such that \(u \text{ Impacts } v\). For example, the tuple \((sakila_op1, \text{actor})\) in the set of edges above represents that the two vertices \(sakila_op1, \text{actor}\) are connected with a \(\text{Impacts}\) relationship with the source node as \(sakila_op1\) and the target node \(\text{actor}\).

This concludes the discussion of our conceptual model. We illustrate the effectiveness in a case study in Chapter 6. In the next chapter, we present an overview of the technologies used to implement a software tool for impact assessment of schema changes in a data warehouse environment.
Chapter 4: Implementation Technologies

In this section, we provide an overview of the implementation technologies that form a foundation for our research. These are: (a) Pentaho, a commercial business intelligence tool used for expressing one of the input artifacts in our work, namely, ETL workflows, and (b) Neo4j graph database that serves as the implementation model. The other input artifacts of schemas and queries are expressed using the relational database, MySQL.

4.1 Pentaho

ETL (Extract-Transform-Load) represents a core component in the data warehouse architecture. It refers to the set of operations that are performed to read data from the operational sources (extract), parse-clean-validate (transform) and load the result into the target artifact. Pentaho offers a variety of operators to support each of these operations. Pentaho is a leading business intelligence and data integration tool available for free download [10]. We use it to represent the ETL workflows in our work.

An ETL transformation in Pentaho is a directed graph consisting of a series of step nodes connected by hops. Figure 4.1 illustrates an example of the ETL transformation. The step Microsoft Excel Input represents the extract step reading from an excel data source. Once the data is read, the null values in one of the fields are filtered using
Figure 4.1: A simple three-step Pentaho ETL workflow

the Filter rows step (transformation) and finally the results are loaded into the target table configured using the Table Output Step (load).

Depending on the type of the data source, there are a variety of operators available for extracting data in Pentaho. Examples include CSV File Input, Microsoft Excel Input, Json file input, Table Input, Amazon AWS S3 storage and so on. We use Table Input operator in our work since our input schema artifacts are expressed in the relational model. Other steps that interact with the database include: Insert/Update step, Table Output step, Lookup step, Dimension Lookup/Update, and Combination-Lookup/Update. A detailed description of each of these steps is available on Pentaho’s documentation website [10].

The key point to capture in each of these database related steps including the Table Input step (extraction phase) is the information about the name of the database, relation and the attributes. This would lay foundation for determining dependencies between the ETL transformation and the schema.

While an ETL transformation also consists of non-database related steps (e.g., concatenate fields), the reason that we are covering only a discussion of the database related steps here is driven by the scope and expected outcomes that we have defined for our work. The assessed impact of a schema change in our work captures the dependent
schema artifacts, the name of the ETL transformations that are impacted, and queries. The database related steps in ETL transformations offer the necessary and sufficient information to determine these pieces of information. The database related steps in an ETL transformation serve as the link between a schema and transformation. As an example, consider a change which is proposed to an artifact in the reconciled schema. We would consider database related steps in two sets of transformations. The first set will correspond to the mappings between the operational and reconciled schemas; and the other set of transformations will model the relationships between the reconciled and the warehouse schema. Database related steps in the first set of transformation will provide information about the artifacts within the operational schemas that feed data into the reconciled schema. Similarly, from the database related steps of the second set of transformations, we get information about how the artifact within the reconciled schema that is considered for a schema change, participates in the mappings with the warehouse schema.

Figure 4.2 presents an example to summarize the above discussion of ETL operations in Pentaho. The transformation aims to populate a fact table (\textit{fact\_interactions}) based on the data in the dimension tables and an external data source [43]. There are three data extraction database steps \textit{Table Input}, \textit{TimeDim}, and \textit{ChannelDim} with the last two representing data read from the dimension tables. The step \textit{fact\_interactions} is an \textit{Insert/Update} step for loading data into the target table in the warehouse schema.

This concludes our brief discussion of Pentaho. In the next section, we provide an overview of our implementation model, Neo4j.
4.2 Neo4j

We leverage a graph-based model to implement our framework. Each of the artifacts are converted from their native formats to the graph representation and the transformation is based on the conceptual model described in Chapter 3. The rationale behind employing graph databases is based on three reasons.

1. The data warehouse environment exhibits heterogeneity by having schemas, ETL workflows, queries all under one architecture. Graphs with their basic constructs of nodes and edges allows to offer a uniform representation to address heterogeneity in the environment.

2. Proposing a change to a schema artifact at one level in a warehouse environment and identifying its impact (direct or transitive) on all the related artifacts
correspond to the path traversal queries in a graph database. Relational model would use the idea of joins for the same purpose. Given that relationships are pre-computed in the graph database compared to joins in a relational model that are computed at query-time, employing graph databases offers a performance advantage [104].

3. Graph representation supports the idea of provenance which in our work corresponds to the path information that offers the trace of how the evolved artifact and the impacted artifacts are related to each other.

Papastefanatos et al. [96, 95] use a graph-based formalism for modeling ETL queries and their dependencies on the elements from the source schemas. We derive inspiration from their graph-based approach but extend their formalism to incorporate additional artifacts including ETL workflows from a leading, industry business intelligence tool, Pentaho and queries. Furthermore, the impact of changes over source schemas in their work is assessed only over ETL queries. Thus, while these contributions propose novel solutions towards managing schema evolution, they are either limited in their coverage of evolution operations or address evolution over a subset of artifacts in the three-tier architecture shown in Figure 1.1. Our work also drives inspiration from Buohali et al. [36] towards employing the concept of property graphs in our work to represents each of the input artifacts in the graph format.

There are several industry implementations of graph databases such as Titan [18], InfiniteGraph [5], and Neo4j [6]. We adopt Neo4j in our work given that it is open-source and has a strong user base and technical support community. Neo4j is a leading
commercial graph-database provided by NeoTechnology. The database is freely available for download at http://neo4j.com/download.

In this section, we provide an introduction to the specific features of Neo4j that are used in our work. These concepts include the concept of property graphs and path analytical queries that are pertinent to the development of our approach.

Figure 4.3: A simple property graph

**Property Graph**

A property graph extends the basic graph constructs of nodes and edges with additional properties and labels. These properties are expressed in the form of key:value pairs and can be applied to both nodes and edges. This allows augmenting the graph nodes and edges with semantics. Similarly labels can be added to both nodes and edges as well. Labels allows to contextualize the nodes or edges and further facilitate improving query performance. All the nodes with the same label form a group and this leads to an improvement in the query efficiency because a query involving labels limits the the search space to the group of nodes or relationships defined by that label instead of searching through the complete graph [104].

Figure 4.3 illustrates a basic example of a property graph with two nodes and one edge. Some key points to note in Figure 4.3 are as follows: (1) the labels, Employee
Organization are depicted in rectangles over the nodes, (2) a node can possess more than one label, and (3) both nodes and relationships may have attributes that are key-value pairs.

**Path Traversal Queries**

In this section, we discuss the path traversal queries using Cypher. Cypher is the declarative query language provided by Neo4j, similar to SQL is the relational databases. Consider the graph in Figure 4.4 for the example path traversal query below. The graph represents the movie dataset reproduced from Neo4j and consists of 14 nodes and 24 relationships.

**Query:** Find all the paths to a particular target node. In our example here, find all the actors who have acted in the movie titled “The Matrix Reloaded”. Figure 4.5 shows the cypher query and the result set.
The search space is reduced based on the node labels *Person* and *Movie* and further specifying the property *title* for the *Movie* node. The query can also be extended to retrieve a path of a certain length. In our example, the *Movie* nodes are acting as sink nodes with no outgoing edges. Hence the result only retrieves the paths of length 1.

In the context of our research goal, which is to identify the related artifacts of a particular node, the query will be useful to answer following questions:

1. Given a specific artifact, which other artifacts in the architecture will be impacted by it.

2. Identify *all* the paths that result in the creation of a certain node.

This concludes our brief discussion of Neo4j and Pentaho that we use as the implementation tools in our work. In the next chapter, we present the architecture and
implementation of our approach for transforming the input artifacts into a property graph model and submitting queries over the integrated graph model for schema evolution operations to identify the impacted artifacts in the framework.
Chapter 5: Architecture and Implementation

In this chapter, we present the architecture and implementation of our approach. This involves a discussion of the methodology that seeks to achieve the following two functions: (1) transform each of the input artifacts into the target implementation model which is a property graph in our work, and (2) determine the impact of a schema change by submitting queries over the integrated graph model for different schema evolution operations. Section 5.1 presents the main architecture components in this context.

The framework is implemented using J2EE (Java 2 Platform, Enterprise Edition) for application programming, MySQL database for the input schemas, Neo4j graph database as the target graph-based model, JSP (Java Servlet Pages), and Bootstrap CSS for front-end development. J2EE offers a framework for developing web applications using Java and JSP is a part of the technology stack of J2EE. It allows building of dynamic web pages. Our application code consists of three thousand lines of code. While the complete code is available in Appendix 2, this chapter covers the application structure and discusses major code snippets (Section 5.2) that implement the different architecture components discussed in Section 5.1.
5.1 Architecture

Figure 5.1 presents the architecture of our approach. It can be broken down into five main modules:

1. The *input artifacts module* labeled as 1 holds native schemas, ETL workflows, and queries.

2. The *processor module* labeled as 2 reads the input artifacts and uses a combination of tools, including scripts developed in MySQL, application code using Java, and an open-source SQL parser, to convert the input artifacts into comma
3. The graph module labeled as 3 uses the Neo4j browser to view the transformed schemas and execute cypher queries to identify the impact of making a change to a particular artifact over other artifacts in the framework.

4. The schema evolution operation specification module labeled as 4 allows the user to mark the artifacts for evolution through a user-interface and the framework generates the corresponding queries which can be executed over the graph. The queries are generated in a proprietary query language, Cypher, used by our underlying graph implementation, Neo4j.

5. The impact assessment module labeled as 5 represents the subgraph that consists of all the artifacts that will be impacted by the proposed change. In addition to the list of artifacts, the subgraph also captures the complete path trace information illustrating how the evolved artifact is related to each of the impacted artifacts.

While having the additional path trace information helps to enhance user’s understanding of the identified impact but the graph with all its nodes and relationships can still be tedious to read by the end user. To add to the complexity, each node and certain relationships in the graph (representing entity integrity and referential integrity constraints) also hold additional information about them in the form of key:value properties. These properties are not visible to the user until the particular node or relationship is clicked. Thus, to further simplify the task
of end user’s understanding of the presented impact, we present a textual explanation of the result graph. This description covers each path from the evolved artifact to each of the impacted artifacts in the result graph and will expand the properties of each of the node and relationships in that path. The path trace and textual description together provide explanations in our work.

We use MySQL as the backend database for relational schemas. The process starts at the input artifacts layer which consists of three components: schemas from a MySQL database, ETL workflows in Pentaho, and the queries expressed in MySQL. The set of schemas include a built-in MySQL database (information_schema) and the user defined operational, reconciled, and data warehouse schemas. The built-in MySQL database (information_schema) is leveraged in the Scripts section in the processing module which consists of three parts:

1. **Scripts**: A set of stored procedures written in MySQL extracts schema information such as a list of relations, attributes, and entity and referential integrity constraints for each of the operational, reconciled, and warehouse schemas. The built-in MySQL database (information_schema) is leveraged in this section for reading the metadata information described above.

2. **Java**: This includes the application programming code developed using technologies of JSP and Servlets available in the Java Enterprise Edition. The goal is to read the ETL workflows from Pentaho and represent them in CSV format. Pentaho allows exporting the workflows in an XML format. We parse the XML data using Java to obtain a CSV representation of the ETL workflow.
3. An open-source tool, GSQLParser: This tool parses queries originally expressed in SQL and generates the metadata information in the form of XML. The XML output from the GSQL parser is further processed by Java to generate data in the CSV format.

We use the CSV file format since both the Neo4j community and our programming interface which uses Java supports CSV files. Furthermore, Neo4j allows batch creation of graph from csv format.

Once all the artifacts are transformed to the target graph-based implementation in Neo4j, the impact of the changes on specific artifacts is assessed by submitting queries to the Neo4j graph. The user submits their requests using a user interface and the queries are generated in the native query language supported by Neo4j, Cypher. These Cypher queries are then executed in Neo4j to determine the impact along with the path trace information.

In the next sections, we discuss the implementation aspects of the architecture components explained above. We present the specific scripts written in MySQL to generate the schemas expressed originally in relational format into CSV format. A discussion of the transformation of ETL workflows and SQL queries to CSV format is also presented in the following sections.

5.2 Implementation

In this section, we elaborate on the processing module of the architecture from Figure 5.1. We focus on the scripts written in MySQL database and code modules developed using Java that are used to transform the artifacts from their native format
to the target graph representation. Detailed discussion of the rest of the modules is deferred to Chapter 6 where they are explained along with a case study.

5.2.1 **Transform relational schemas to target graph representation**

The transformation of relational schemas residing in the MySQL database to the target graph format can be organized into two modules: (a) conversion of native schemas into the intermediate CSV format, and (b) generation of a graph based on the information from the CSV file. The *Scripts* component of the processor module is responsible for the first task and Java is used for transforming data in the CSV files to a Neo4j graph.

This section focuses on the *Scripts* part of the processing module labelled 2 in Figure 5.1. It consists of three queries (Figures 5.2, 5.3, and 5.4) which offer information about all relations, attributes and constraints in each of the input schemas in our work.

The query in Figure 5.2 offers information about all relations in a particular schema. The output is a CSV file consisting of two columns: *table_name* and *schema_name*.

```sql
select table_name, table_schema as schema_name
from information_schema.tables
where table_schema='<name of the schema>' and table_type="BASE TABLE";
```

Figure 5.2: MySQL query to capture information about all relations in the MySQL *sakila* database

The next query as shown in Figure 5.3 captures information about all of the attributes for each of the relations of a particular schema. The information that we
require in our work includes name of the attribute, its datatype, whether it is a primary key, and the relation and schema it belongs to. We use the information about primary keys from this query to model entity integrity constraints in the graph.

```
select c.column_name, c.table_name, c.data_type,
case when c.column_key = 'PRI' then 'YES' else 'NO' end as is_Key,
c.table_schema
from information_schema.columns c
where c.table_schema = '<name of the schema>' and
c.table_name in
  (Select table_name FROM information_schema.'TABLES'
   where table_type LIKE 'BASE TABLE'
   and table_schema LIKE '<name of the schema>')
```

Figure 5.3: MySQL query to capture information about all attributes in the MySQL sakila database

The query in Figure 5.4 outputs all the foreign key constraints in a given schema. The reason for collecting all the foreign key constraints in our database is that we need them to create relationships in our graph model.

```
SELECT
  table_name as relation_name, column_name as attr_name, constraint_name as constraint_name,
  referenced_table_name as referenced_relation_name,
  referenced_column_name as referenced_column_name
FROM
  information_schema.key_column_usage
where
  referenced_table_schema = '<name of the schema>'
```

Figure 5.4: MySQL query to capture information about all foreign key constraints in the MySQL sakila database
The output from the three queries above is exported in CSV format for each of the schemas in our work. Once CSV files are generated, the graph is generated from the custom Java code discussed as follows.

The code implementation in Java for transforming schema CSV files to a graph resides in two main code files: (a) SchemaServlet.java, and (b) SchemaGraphController.java. The former file is responsible for receiving a user’s request to generate the graph and return appropriate success or error messages. The file SchemaController.java reads the CSV data generated from the queries above (Figures 5.2, 5.3, and 5.4) and uses the Neo4j-Java API to generate the graph programatically.

Figure 5.5 shows the high level code structure for transforming CSV files containing schema information to a graph.

![Figure 5.5: Code structure for generating Neo4j graph for a schema from a set of CSV files](image)

Figure 5.5: Code structure for generating Neo4j graph for a schema from a set of CSV files

The conversion process is initiated through a call to the SchemaServlet.java program which in turn calls the code within SchemaController.java. The code reads the CSV
files and creates the Neo4j graph based on the conceptual model and API calls to the graph from within Java.

Figures 5.6 shows a code excerpt for the file `SchemaGraphController.java`. The code highlights how Cypher queries are send to the implementation graph database interface from within Java. In the code excerpt here, the figure illustrates the creation of relationships between `Relation` and `Attribute` nodes. A CSV file with the name `Attributes.csv` is being read and the `Match` clause is used to match the data in the CSV file and the existing the Neo4j graph.

```java
public void generateRelationAttributeRelationships(String excelfilePath)
{
    try{
        Statement stat = con.createStatement();
        {
            System.out.println("********Generating relation-attribute relationships*********");
            ResultSet rs = stat.executeQuery("LOAD CSV WITH HEADERS FROM "+excelfilePath+" Attributes.csv" AS line "+
                " relation:line.table_name, schema: line.table_schema, IsKey: line.is_key)" "+
                " MATCH (src:Relation (name: line.table_name, schema: line.table_schema))" "+
                " WHERE src.name=to.relation and src.schema=to.schema "
                " MERGE (src)-[[:Impacts]->(to)];
        }
    } catch (Exception ex)
    {
        System.out.println(ex.getMessage());
        resultstatus="Error occurred in generating relation-attribute relationships\n";
    }
}
```

Figure 5.6: API call to Neo4j from Java: `SchemaGraphController.java`

This concludes our discussion of how schemas expressed in the native relational format are converted to the target graph representation. In the next section, we discuss how ETL workflows originally described using Pentaho are converted to Neo4j in a semi-automated manner.
5.2.2 Transforming ETL workflows to Neo4j

The transformation of ETL workflows expressed originally in Pentaho to the target Neo4j format involves three steps: (a) using the built-in utility in Pentaho that allows exporting the graphical ETL workflows to an XML format, (b) parsing XML to generate data in the CSV format, and (c) generating Neo4j graph from CSV format. We discuss each of these steps in detail here.

Pentaho, the data integration solution that we use in our work for modeling ETL workflows, uses a GUI. But it also allows exporting the workflows developed in a graphical manner to an XML format which can then further be parsed using Java to generate data in the CSV format. We use this export feature to obtain an XML version of each of the ETL transformations in our work. As an example, consider the ETL workflow from Figure 3.2. Figure 5.7 shows an excerpt of the XML representation for the InsertUpdate step. It holds information about the database connection, relation and attributes (highlighted in rectangle boxes). We use Java to parse this XML to generate data in the CSV format.

The XML representation allows us to retrieve specific information about the interactions between that particular ETL transformation and the database schemas. We develop code using Java to read the XML file and generate data first in the CSV format and then in the target graph representation. This section shows the high level code structure to allow an understanding of how the various code components interact with each other.

The process is described as follows:
Figure 5.7: Snippet of the XML representation of Insert/Update step

1. **ETLServlet**: The conversion process is initiated through a call to the *ETLServlet.java* program. A method `generateCSVData(String <path to the csv file>)` is invoked to generate CSV files corresponding to each of the input XML file. The method contains several method calls with each sub-method creating a CSV file for a particular step type in an ETL transformation. Figure 5.9 shows the code in Java. The statements highlighted in rectangle boxes captures the interaction with two other code components (*XMParse* and *ExcelExporter*) explained next.

2. **XMLParser**: As explained in Chapter 4, a transformation in Pentaho is composed of multiple steps where each step belongs to a specific type. This code component contains method calls which parse XML for specific step types in the ETL transformation. As an example, for *TableOutput* or *Insert Update* step types, the method `parseXMLforOutputSteps` is invoked. Figure 5.10 shows the code in Java.
Figure 5.8: Overview of code structure for generating Neo4j graph from a set of XML files

The files such as InsertUpdateParser, TableOutputStepParser, TableInputStepParser, DBLookupStepParser, DBJoinStepParser, CombinationLookupStepParser and DimensionLookupStepParser handle code for parsing information about each step type in the XML. Figure 5.10 handles the Insert Update and TableOutput steps through method calls in InsertUpdateParser and TableOutputStepParser programs (Highlighted using two arrow pointers). Each step follows a different structure in terms of recording information. Figure 5.11 show the sample XML snippets for two different step types: Insert Update and TableInput.
The parser first generates a list of step objects which contains information about the name of the step, database, relation, and attributes it uses. In the code snippet in Figure 5.10, OutputStep highlighted by the first arrow pointer represents a step object.

3. **ExcelExporter**: Once the list of step objects is determined, the next step involves writing of these objects to a CSV file. This is accomplished using ExcelExporter.java.

4. **ETLGraphController**: As the final step, the data in the CSV format needs to be converted to a graph. Figure 5.12 shows the code that accomplishes this step.
5.2.3 Transforming queries to CSV format

The aim of this section is to illustrate the process of transforming queries written in SQL into CSV format. This section focuses on the GSQLParser part of the processing module.

The transformation of queries to CSV format is a two-step process:

1. The set of queries are parsed and transformed into a custom XML format using a third-party utility called GSQLParser. Figure 5.13 shows an excerpt of SQL query parsed using GSQLParser.
Figure 5.11: Sample XML structure for two different step types in an ETL transformation in Pentaho
(left) InsertUpdate step (right) TableInput step

2. The output from GSQLParser is then converted into a custom CSV format and further into a graph representation using Java code.

We consider a set of twelve queries (included in Appendix A) in our case study and each of them is converted to XML format using GSQLParser. The next step involves developing code in Java to convert XML format to the custom CSV format. While the complete code is available in Appendix 2, Figure 5.14 shows the design of the various code components.
Figure 5.12: Code snippet for ETLGraphController to generate Neo4j graph

Figure 5.15 shows a code snippet for QueryXMLParser.java illustrating how XML is parsed to build a SQL query object. A query object is a custom class we have defined in Java that encapsulates the information that we need to capture for a query. This includes a query title, the attributes that it depends upon, and the relation and schema of each of these attributes.

The graph at this point represents an integrated model for all our input artifacts. The schema evolution operations are specified over specific nodes in the transformed graph and the list of impacted attributes along with the complete path describing how the source and target artifacts are related to each other is determined using the results of the queries written in the Cypher query language. This is illustrated in the section.
5.2.4 Impact assessment

In this section, we briefly explain how our interface allows the user to specify a schema change over an artifact anywhere in the architecture (Figure 1.1) and how our approach reports the impact. Figure 5.16 presents the main webpage for this module.

The user first selects the type of artifact: schema, relation or attribute and then enters the name of the artifact. If a schema is selected as the artifact type, then the textboxes for Relation and Attribute are disabled. The button labelled Determine its impact generates a request for impact assessment of the specified artifact and the application code then returns a Cypher query to the user. This query can be executed
Figure 5.14: Overview of Java code components for parsing SQL query into CSV format in the Neo4j browser to view the graph. Figure 5.17 shows the generated Cypher query and the output from executing the query in Neo4j is shown in Figure 5.18. The different Cypher queries are discussed in detail in Chapter 6 and Appendix 3.

Apart from impact assessment, the interface also offers Source Analysis. This allows the user to view all the artifacts in the architecture which are responsible for populating the specific artifact whose name is specified in the textbox.

This concludes a brief overview of the different architecture components from Figure 5.1. In the next section, we discuss a case study followed by a discussion of the evaluation results.
```java
private List<Query> buildQueryObject(XPath xpath, Document doc, String filename){
    List<String> queryInfoList=new ArrayList<String>();
    List<Query> listOfObjects=new ArrayList<Query>();
    try {
        String attrExpr="/columnImpactResult/file[@name=""+filename+""]/targetColumn/sourceColumn/@name";
        String relExpr="/columnImpactResult/file[@name=""+filename+""]/targetColumn/sourceColumn/tableName";
        String dbExpr="/columnImpactResult/file[@name=""+filename+""]/targetColumn/sourceColumn/tableOwner";
        XPathExpression expr = xpath.compile(attrExpr);
        XPathExpression expr2 = xpath.compile(relExpr);
        XPathExpression db_expr = xpath.compile(dbExpr);
        NodeList attr_nodes = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
        NodeList rel_nodes = (NodeList) expr2.evaluate(doc, XPathConstants.NODESET);
        NodeList db_nodes = (NodeList) db_expr.evaluate(doc, XPathConstants.NODESET);
        for (int i = 0; i < attr_nodes.getLength(); i++){
            String attributeName=attr_nodes.item(i).getNameValue();
            String relName=rel_nodes.item(i).getNameValue();
            String dbName=db_nodes.item(i).getNameValue();
            String queryObjStr=filename,"+dbName+","+relName+","+attributeName;
            if(queryInfoList.contains(queryObjStr)){
                queryInfoList.add(queryObjStr);
                Query obj=new Query();
                obj.setQueryLabel(filename);
                obj.setAttribute(attributeName);
                obj.setRelationName(relName);
                obj.setDbName(dbName);
                listOfObjects.add(obj);
            }
        }
    }
}
```

Figure 5.15: Code snippet illustrating parsing of XML representation of a SQL query

![Image of the UI for impact assessment](image.png)

Figure 5.16: User interface for impact assessment
Figure 5.17: Cypher query presented to the user after form submission in Figure 5.16

Figure 5.18: Cypher query from Figure 5.17 executed in neo4j browser
Chapter 6: Case Study

In this chapter, we demonstrate the feasibility and effectiveness of our approach using a case study to illustrate all of the artifacts in a data warehouse environment. The case study is based on an open source *sakila* database that models a business case of a fictitious DVD rental store. The MySQL community provides a schema and database consisting of 16 entities and relationships between them. We use their schema as the integrated schema and construct two additional schemas from it to represent the two source schemas.

The case study is composed of three parts: (1) discussion of the artifacts including schemas, ETL, and queries, (2) verification of the designed ETL operations and the implemented graph model, and (3) impact assessment results using examples and a tabular summary of the evaluation results.

The case study consists of two operational schemas (*sakila_op1* and *sakila_op2*), a reconciled schema (*sakila_ids*), and a warehouse schema (*sakila_dwh*), two sets of ETL transformations that load data from the source schemas into the reconciled and warehouse schemas, and a set of twelve queries. The schema and instance generation scripts, ETL workflows, and queries are available in Appendix A.
6.1 Case study

This section discusses the input artifacts in three categories: (1) schemas (operational, reconciled and warehouse schema), (2) ETL transformations between the operational, reconciled, and warehouse schemas, and (3) end-user queries expressed over each of the input schemas.
6.1.1 Operational schemas

The MySQL community offers a relational schema that represents entities and relationships of a fictitious DVD rental business. We leverage the available schema and use it to represent reconciled schema in our work. As the first step in illustrating data integration, we require a set of data sources that would load data into this reconciled schema. In this section, first, we present the two operational schemas *sakila_op1*, *sakila_op2* that we have developed to serve as the sources for the integrated *sakila* database. Figures 6.1, 6.2 shows the two schemas.
The differences between the two source schemas are as follows:

1. The operational schema *sakila_op1* consists of 13 relations where as the schema *sakila_op2* captures 10 entities. The relations, *inventory, language, and store* do not exist in the later schema.

2. The information about the language of a film is embedded in the *film* relation itself in the schema *sakila_op2*. In the other schema, *sakila_op1*, it is normalized across relations *film, language*. Also, the second schema, *sakila_op2*, does not contain a *store* relation. It represents schema for a particular store.

3. The *address* relation uses different naming convention for one of its attributes in the two schemas. The street address is captured using the attribute *street_address* in the schema *sakila_op1* while the term *address* is used for the same attribute in the other schema.

### 6.1.2 Reconciled schema

For the reconciled schema, we leverage the available schema (*sakila*) from MySQL community and extend it to incorporate mapping information from multiple data sources, *sakila_op1, sakila_op2*. The schema obtained after revisions in the *sakila* database is called *sakila_ids* in our work. Figure 6.3 shows the reconciled schema.

In the interest of space, we summarize only the changes we have made to the integrated schema. The complete DDL script including our changes are available in Appendix A. The original schema, *sakila* is revised in the following manner to obtain the integrated schema, *sakila_ids* in our work:
1. An attribute, \textit{ds\_name} is added to each of the following relations: \textit{film}, \textit{actor}, \textit{film\_category}, \textit{category}, \textit{customer}, \textit{address}, \textit{film\_actor}, \textit{payment}, \textit{staff}, \textit{rental}, \textit{store}, \textit{inventory}, \textit{language}. This attribute holds one of the two values: \textit{sakila\_op1} or \textit{sakila\_op2}, depending on the data source of the particular row.

2. The following relations also have additional attributes:

   - \textit{actor}, \textit{film\_actor}: \textit{ds\_actor\_id}
   - \textit{film}: \textit{ds\_film\_id}
Some key points to note about the reconciled *sakila_ids* database schema are as follows:

1. The schema captures five subject areas: (1) film including relations *film*, *film_category*, *film_text*, *language*, and *category*, (2) actor including relations *actor* and *film_actor*, (3) business including relations *inventory*, *store*, and *staff*, (4) location including relations *address*, *city*, and *country*, and (5) customer including relations *customer*, *rental*, and *payment*.

2. Each relation has a column *last_update* of the TIMESTAMP datatype. It represents the date and time when a row is inserted or modified.

### 6.1.3 Warehouse schema

The data warehouse schema we use in our work is taken directly from the schema provided by Casters et al. [43]. Casters et al. [43] offer the schema based on *sakila*
along with ETL transformations for extracting data from the original sakila schema and loading it into the target warehouse schema.

The data warehouse star schema is shown in Figure 6.4. The schema consists of a fact table fact_rental which is related to six dimension tables. The fact table contains quantitative metrics such as count_returns, count_rentals and rental_duration to capture
the performance of the business while the other attributes ending with \_key represent the foreign keys to each of the dimension tables.

Some points to note in the schema are as follows:

1. The six dimensions tables provide context to the three quantitative metrics in the fact\_rental table. Each dimension table holds a surrogate primary key labelled as \[dimension-name]\_key such as customer\_key in the dimension table \textit{dim\_customer}.

2. Apart from the surrogate key, each dimension table with the exception of \textit{dim\_date} and \textit{dim\_time} also contains a column identified as \[dimension-name]\_id. This column maps to the primary key column of the reconciled sakila database. For example, the \textit{dim\_film} table in the warehouse schema contains a column \textit{film}\_id which corresponds to the \textit{film}\_id column of the \textit{film} table in the reconciled sakila database. Having these columns is important for the ETL process to determine whether a record in the dimension table needs to be updated or a new row is to be inserted in an event of change in the source schema.

3. The tables \textit{dim\_customer}, \textit{dim\_staff}, and \textit{dim\_store} are implemented as Type 2 slowly changing dimensions (SCD2, hereafter). SCD2 adopt data versioning to preserve the history of data instead of overwriting old values in the event of a change. The columns ending with \texttt{\_version\_number}, \texttt{\_valid\_from}, and \texttt{\_valid\_through} serve to implement this idea of versioning in SCD2.

Next, we discuss the design of eight ETL transformations required to map the two source schemas to the reconciled schema.
6.1.4 ETL transformations between operational and the reconciled schema

While the transformations between reconciled and warehouse schema are provided by Casters et al. [43]; for the transformations between operational and reconciled schema, we designed sixteen workflows. The aim of this section is to illustrate each of these transformations.

load_actor

This transformation (Figure 6.5) is responsible for loading data from the relation `actor` of each of the two operational schemas into the relation `actor` of the reconciled schema, `sakila_ids`. The first step, `max_dim_actor_last_update` determines the most recent value of date/time when an insert or update was made in the target relation. The purpose of this step is to detect if any changes have been made in the source schemas since the last update. The steps, `sakila_op1.actor`, `sakila_op2.actor` read data from each of the two operational schemas which is then appended into one stream using the step, `Append streams`. The last step, `Insert/Update` populates data into the target relation.

Figure 6.5: The load_actor transformation
The design of this transformation is interesting because while one of the source schemas, `sakila_op1`, organizes information about it and the language of each film in the same manner as the integrated schema; the other operational schema `sakila_op2` denormalizes this information. Information about the language of a film is embedded as an attribute, `language` in the `film` relation in it. Thus, the mapping from each of the two sources to the integrated schema is different and for the schema `sakila_op2` particularly, it is not one-to-one. Figure 6.6 shows the ETL transformation.

```sql
SELECT COALESCE(
    MAX(last_update),
    '1970-01-01 00:00:00'
) AS max_language_last_update_id
FROM language
```

```sql
SELECT language_id, name, last_update from language where last_update > ?
```

Figure 6.7: Steps - max_language_last_update and Extract language from ds1

**load_language**

load_language
The steps from `max_language_last_update` to `Insert Update` including them represents the data mapping between `sakila_op1` and `sakila_ids`; while the other three steps `Extract language from ds2`, `Add data source name 2`, `InsertUpdate 2` capture the mapping between `sakila_op2` and `sakila_ids`.

The purpose of the steps – `max_language_last_update`, `Extract language from ds1` is to identify data change in the relation `language` in the source schema `sakila_op1`. The first step, `max_language_last_update` reads the most recent value of date/time when an insert or update was performed in the `language` relation of the `sakila_ids` schema. The next step, `Extract language from ds1` reads only those records from the `language` relation of the `sakila_op1` schema where the value in the column `last_update` is more recent than the value coming from the previous step, `max_language_last_update`. Figure 6.7 shows the query. The question mark in the query on the right side represents the placeholder that reads value returned from the previous step, `max_language_last_update`. The steps `Add data source name` adds a constant value (`sakila_op1` or `sakila_op2` for the column `ds_name`) in the integrated schema. The steps of type `InsertUpdate` are responsible for loading data into the target schema.

**load_address**

This transformation populates the relation `address` in the reconciled schema. Figure 6.8 shows this workflow. The important steps in this transformation are four `Lookup` steps. They are required because the operational schemas contain the fields `city`, `country` as text in the relation `address`. But there are no corresponding attributes in the `address` relation in the reconciled schema. Instead it uses a foreign key, `city_id` to capture this information using two relations, `address` and `city`. Furthermore, the relation,
city in the reconciled schema contains information about the country using a foreign key, country_id. This mapping between the city, country and address relation in the three schemas is achieved using the four lookup steps in Figure 6.8.

![Figure 6.8: The load_address transformation](image)

load_film

Figure 6.9 shows the ETL transformation. There are three key challenges in mapping data for the film relation from each of the two operational schemas to the reconciled schema. First, it loads the information about language (language_id, original_language_id) differently based on the data source. While both the schemas, sakila_op1 and sakila_ids organizes information about film and its language across two relations, it is denormalized in the other source schema, sakila_op2. Secondly, schema sakila_op2 does not store information on original language. As the third key point, the value of the attribute, language_id in the reconciled schema will come from the value, language_id of the relation language. Given that the names of the attributes are the same for language_id in the relations language across the two schemas, sakila_op1, sakila_ids, it can be misconstrued that they form a one-to-one-mapping.
The steps \textit{Lookup language\_id}, \textit{Lookup language\_id 2} provide answers to the first challenge stated above. These steps are of type \textit{Database Lookup} in Pentaho and are used to lookup values in a database by comparing value(s) from the previous step to the columns in the database. For example, the step \textit{Lookup language\_id} compares the values of columns \textit{ds\_language\_id}, \textit{ds\_name} from the relation \textit{sakila\_ids} with the values returned from the previous step \textit{language\_id}, \textit{ds\_name}. These values \textit{language\_id}, \textit{ds\_name} correspond to the actual data from the source schema, \textit{sakila\_op1}. The value of the column \textit{language\_id} is returned from the row which matches the set of the conditions.

Similarly, the step \textit{Lookup language\_id 2} compares the values of columns \textit{language}, \textit{ds\_name} from the relation \textit{sakila\_ids} with the values returned from the previous step \textit{name}, \textit{ds\_name}. The previous step, \textit{Add data source 2} holds values from the schema, \textit{sakila\_op2}.

The solution to the second challenge of loading data for the attribute \textit{original\_language\_id} is achieved using the step \textit{Lookup original\_language\_id}. Since this attribute is only available in the source schema, \textit{sakila\_op1}, it is not a part of the sequence of steps leading to the \textit{Insert Update step 2}.

The solution to third challenge is addressed collectively by the \textit{Lookup} and \textit{Insert Update steps}. Figure 6.10 shows the mapping for the fields \textit{language\_id}, \textit{original\_language\_id} of the reconciled schema. The mapped fields, \textit{ids\_language\_id}, \textit{ids\_original\_language\_id} are obtained from the previous Lookup steps.

These four transformations (Figures 6.5, 6.6, 6.8, 6.9) cover the main concepts that form a part of the sixteen ETL transformations between the source and reconciled schemas. The other 12 transformations are included in the Appendix A. In the
next section, we summarize the key points about ETL transformations between the reconciled and warehouse schema.

6.1.5 ETL transformations between reconciled and the data warehouse schema

We have nine ETL transformations to map and load data from the reconciled schema, sakila_ids to the warehouse schema, sakila_dwh. Eight of these are for loading the dimension tables and one transformation captures the loading of the fact table in the warehouse schema in Figure 6.4. The transformations are provided by Casters et al. [43] and are included in Appendix A. A brief overview of each of them is provided below.

1. load_dim_date - This transformation is responsible for populating the dim_date dimension in the warehouse schema. The dim_date dimension table stores the date when a rental or rental’s return event occurred. The table captures date in multiple formats using different columns. The data is loaded into it once and does not need to be reloaded from the reconciled sakila database.
The transformation is formed of four steps including one database-related step (Load dim_date). The first step labelled Generate 10 years creates a dataset of 3660 rows capturing dates in a particular language and country code. The initial date is specified as a part of configuring this step. The next step which is labelled Day Sequence takes the generated date values from the previous step and adds a field to each row. This additional field is an auto-increment number starting with one. The relevance of this step is observed in the next step labelled Calculate Dimension Attributes. This third step represents a particular step type.
in Pentaho called *Modified Java Script Value Type*. In this step, the incoming values from the Day Sequence Step are processed using JavaScript expressions to generate calendar day in different formats. Finally, the last step labelled *Load dim_date* is of type Insert/Update in Pentaho which generates SQL statements to load data into the dimension table *dim_date*.

In the context of impact assessment, consider that a user attempts to drop a column from the *dim_date* table. Our results will highlight this transformation and the step *Load dim_date* along with the path trace connecting the evolved and impacted artifacts in the output.

2. *load_dim_time*: This transformation is responsible for populating the dim_time table in the warehouse schema. Like the dim_date dimension, the loading of dim_time dimension table is also a one-time operation which generates a certain number of rows and does not needs to be reloaded periodically from the reconciled database. The table is loaded with $24 \times 60 \times 60 = 86400$ rows after successful run of the transformation.

The steps *Generate Hours, Generate Minutes and Generate Seconds* represent the *Generate Rows* step type in Pentaho. They are run in parallel producing 24, 60 and 60 rows respectively. The *Generate Hours* step outputs 24 rows representing the number of hours in a day. Like the *load_dim_date* transformation, the Sequence steps add an autoincrement field to each of the generated rows in the previous steps. The values for this auto-increment field for the *Hours Sequence* step is in the range 0-23. The *Calculate hours12* step is of type *Modified Java Script Value Type* which generates a 12 hour representation augmenting the data.
with AM/PM indicator. The step labelled *Catesian product* joins the hours, minutes and seconds sequences generated in the previous steps to generate all possible combinations for the three datasets. This leads to a total of 86400 rows. The *Table Output dim_time* loads the data into the dimension table *dim_time*.

3. **load_dim_actor**: The load_dim_actor dimension loads the table dim_actor in the warehouse schema. The Insert/Update step in the transformation is used to populate the target table *dim_actor*. If the incoming data from the previous step *actor* exists in the target table but is out of date, the record is updated with the new values. If the record does not exist at all, a new record is inserted with the values of the columns obtained from the incoming field values of the *actor* step. This concept of identifying whether the existing data in the dimension table is the most recent is ensured using the step labelled *max_dim_actor_last_update*. All the three steps in this transformation impact the database schemas and thus would be captured in the graph model in our work.

4. **load_dim_customer**: The transformation load_dim_customer is responsible for loading the *dim_customer* table in the warehouse schema. The relation represents a slowly changing dimension in the warehouse schema. The transformation consists of three database related steps - *max_dim_customer_last_update*, *Customer* and *Load_dim_customer SCD*. The other three steps represent the data cleaning and parsing steps used for the data transformation of the source data.

5. **load_dim_store and load_dim_staff**: These two transformations are very similar to the load_dim_customer transformation. They also implement the slowly changing dimension type 2 concept in loading the table dim_staff and dim_store tables
respectively in the warehouse schema. The step labelled *Store Manager Lookup* in the *load_dim_store* transformation is a database lookup step which is employed here to denormalize the data coming from the store table. This prepares the data for loading into the *dim_store* dimension table.

6. *load_dim_film*: The *load_dim_film* is responsible for populating two tables in the warehouse schema - *dim_film* and *dim_film_actor_bridge*. The steps from *max_dim_film_last_update* till *Load dim_film* are responsible for loading the *dim_film* table whereas the steps after the *Load dim_film* step focus on the *dim_film_actor_bridge* table.

The dimension table *dim_film* contains several fields holding binary values (Yes/No) values. Some of these fields are *film_has_deleted_scenes*, *film_in_category_animation*. The idea behind creating this fields is to normalize the SET valued data in the reconciled sakila database. The film table in the reconciled schema contains a *special_features* column which holds a set of values. A seperate field is created in the *dim_film* dimension table for each of the possible values in the *special_features*. This action is performed using the steps *Normalize Special Features and Denormalize Special Features to Yes Columns* step in the transformation.

The step *Get Film Categories* fulfills a similar role. A film in the reconciled sakila database can belong to several categories based on the relationship between *film* and *film_category* relations. In the warehouse schema, the different categories for a film are captured using multiple Yes/No fields such *film_in_category_family* etc.
7. load_fact_rental: This transformation loads data into the fact table fact_rental in our warehouse schema. The steps in this transformation focus on calculating measures, count_rentals, count_returns and rental_duration, and performing lookups for foreign keys from dimension tables. For example, the step labelled Lookup film_id and store_id from inventory, lookup film_key, lookup store_key together are used to load values for the attributes, store_key, film_key into the fact table. The transformations load_fact_rental and load_dim_film are quite elaborate and a detailed discussion of each of their steps is available in the book by Casters et al. [43].

6.1.6 Queries

We consider a set of twelve queries comprised of a combination of queries pulled from sakila database views, stored procedures and synthetic queries. These queries represent the end-user queries expressed over each of the four input schemas in our work sakila_op1, sakila_op2, sakila_ids, sakila_ids and exhibit variety based on the use of different clauses including select, join, filter, group by, aggregate functions, order by clauses. Table 6.1 shows an overview of the variety in the queries we consider in our work.

While the criteria in Table 6.1 classifies queries on the basis of the number of relations, attributes, joins, functions and any group by/order by clauses it contains, we further consider the variety in queries in the context of parsing. We use a third-party tool, GSQLParser to parse the MySQL queries. The tool generates the output in XML format which is further handled by our code written in Java to construct the corresponding graph in Neo4j.
Table 6.1: Query characteristics

<table>
<thead>
<tr>
<th>Query title</th>
<th>Schema</th>
<th>$N_1$</th>
<th>$N_2$</th>
<th>$N_3$</th>
<th>$N_4$</th>
<th>$N_5$</th>
<th>$N_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sakila-q1</td>
<td>Reconciled</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sakila-q2</td>
<td>Reconciled</td>
<td>4</td>
<td>15</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sakila-q3</td>
<td>Operational</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sakila-q4</td>
<td>Operational</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sakila-q5</td>
<td>Reconciled</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>sakila-q6</td>
<td>Reconciled</td>
<td>4</td>
<td>16</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sakila-q7</td>
<td>Reconciled</td>
<td>6</td>
<td>15</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>sakila-q8</td>
<td>Reconciled</td>
<td>6</td>
<td>12</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>sakila-q9</td>
<td>Reconciled</td>
<td>7</td>
<td>11</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>sakila-q10</td>
<td>Reconciled</td>
<td>7</td>
<td>13</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>sakila-q11</td>
<td>Warehouse</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>sakila-q12</td>
<td>Warehouse</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Key:

- $N_1$ - Number of relations
- $N_2$ - Number of referenced attributes
- $N_3$ - Number of joins
- $N_4$ - Number of functions
- $N_5$ - Group by/Order by clauses
- $N_6$ - Nested subqueries

GSQLParser successfully parses a variety of queries which may include functions, joins and it can also handle any columns which may be listed in the query, prefixed with relation names. For example, in `film.film_id`, the column `film_id` is prefixed with its corresponding relation name `film`. Our code in Java is able to handle any query that is successfully parsed using GSQLParser.
Figure 6.11: Limitation of GSQLParser for parsing certain queries

We observe one limitation of the result generated by GSQLParser. It does not identify columns which are nested more than one-level deep within any function. Figure 6.11 shows this limitation. The column `first_name` in the relation `actor` is not parsed correctly in GSQLParser. When we parse the XML using Java to generate data in CSV format, we manually edit the CSV to address this limitation. The complete set of queries is included in Appendix A.

This concludes a discussion of the artifacts in our case study. As the next step, we illustrate the effectiveness of the design of the sixteen ETL transformations that we developed for mapping the operational schemas to the reconciled schema. The ETL transformations between the reconciled and the warehouse schema are taken directly from Casters et al. [43] and thus are well-tested.
6.2 Verification of the design of ETL transformations

To validate the sixteen transformations designed by us for mapping between the operational and reconciled schemas, we created a few data instances in the two operational schemas. The transformations were then executed to populate data into the reconciled schema. Then we compared the data in the reconciled schema with the results obtained from manual inspection. Table 6.2 shows the results by drawing a comparative analysis of the number of records in each of the relations in all the three schemas.

The dash indicates that the particular schema represented by the column title (sakila_op1, sakila_op2, sakila_ids) does not have a relation which is represented by the

<table>
<thead>
<tr>
<th>Relation</th>
<th>sakila_op1</th>
<th>sakila_op2</th>
<th>sakila_ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>actor</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>film</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>film_actor</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>category</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>film_category</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>address</td>
<td>7</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>payment</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>rental</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>staff</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>store</td>
<td>2</td>
<td>–</td>
<td>2</td>
</tr>
<tr>
<td>inventory</td>
<td>5</td>
<td>–</td>
<td>5</td>
</tr>
<tr>
<td>language</td>
<td>2</td>
<td>–</td>
<td>3</td>
</tr>
<tr>
<td>customer</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>city</td>
<td>–</td>
<td>–</td>
<td>8</td>
</tr>
<tr>
<td>country</td>
<td>–</td>
<td>–</td>
<td>7</td>
</tr>
</tbody>
</table>
corresponding row. For example, the schema, \textit{sakila}\textsubscript{op2} does not contain any relation by the name, \textit{store}.

Table 6.2 shows that the number of records in all relations except \textit{language}, \textit{city}, \textit{country} in the schema \textit{sakila}\textsubscript{ids} are the sum of records from each of the data sources. The relation \textit{language} shows two records in the \textit{sakila}\textsubscript{op1} schema whereas the reconciled schema shows three records. The additional record comes from the value of the \textit{language} attribute which is embedded within the \textit{film} relation in the schema, \textit{sakila}\textsubscript{op2}.

Figure 6.12 further illustrates this idea by showing actual data instances. The figure illustrates the results for the relations \textit{language}, \textit{film} in the reconciled schema.
and illustrates the actual working of the ETL transformation *language*. The records in the relation *language* in the integrated schema, *sakila_ids* come from two sources: *sakila_op1.language* relation and *sakila_op2.film.language* attribute. These are shown as the top two boxes in figure 6.12. The column *language_id* in the integrated schema’s *language* relation is a surrogate key while the column *ds_language_id* corresponds to the language id coming from the data source. The value 0 reflects that the value is not mapped to an actual value for *language_id* in the data source. This is shown in the first record in the relation, *sakila_ids.language* which is highlighted by an arrow with the text, *sakila_ids.language* in Figure 6.12. The last block in the figure marked as *sakila_ids.film* highlights the mapping between the foreign key constraint between the relations *film, language* of the integrated schema. Similarly, the records for the relations, *city* and *country* in the reconciled schema are read from the corresponding columns in the *address* relation of each of the data sources (Table 6.2).

Vassiliadas et al. [115] have proposed a benchmark for ETL workflows. The authors propose three measures for evaluating the effectiveness of the transformations. These measures include: (1) freshness, (2) consistency, and (3) completeness. Table 6.3 show an example to demonstrate how our proposed transformations meet each of these criterias.

*Freshness* is concerned with the ability of the transformation to identify and capture data changes at the sources. Our set of transformations accomplish this using the *last_update* column available in each of the schemas. The relation in the target schema reads the value of this column to determine the date/time when it was last updated. Figure 6.7 illustrates the SQL query in the context of the ETL transformation *load_language*. Only those rows from the sources are read where the value in the
last_update is more recent than the value retrieved from the target schema’s relation. The third action in Table 6.3 illustrate freshness in one of the transformation load_actor which belongs to our set of proposed transformations.

Consistency relates to the number of valid rows that get inserted into the target in the event of failure during the execution of the transformation. Actions 1 and 2 in Table 6.3 confirm the consistency and completeness of the transformation load_actor. Figure 6.13 further shows the output from the last experiment. Each of the actions in Table 6.3 were also executed for other transformations to validate the rest of workflows as well and results confirmed the fact their correct implementation as well.

In summary, based on the results from Table 6.2 and the output of the evaluation metrics proposed by Vassiliadas et al. [115] in Table 6.3 and Figure 6.13 we conclude the correctness of the sixteen transformations developed as a part of our work.
6.3 Verification of the generated graph for the artifacts in the case study

The size of the Neo4j graph containing all the artifacts discussed in our case study is 8.92MB. Table 6.4 provides an insight into different types of nodes and total relationships in our integrated graph.

Table 6.4: Generated metadata statistics (Neo4j) for all the input artifacts

<table>
<thead>
<tr>
<th>Number of schema nodes</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of relation nodes</td>
<td>48</td>
</tr>
<tr>
<td>Number of attribute nodes</td>
<td>390</td>
</tr>
<tr>
<td>Number of ETL transformations</td>
<td>25</td>
</tr>
<tr>
<td>Number of ETL Step nodes</td>
<td>139</td>
</tr>
<tr>
<td>Number of different ETL Step types</td>
<td>8</td>
</tr>
<tr>
<td>Total Nodes</td>
<td>593</td>
</tr>
<tr>
<td>Total Relationships</td>
<td>1758</td>
</tr>
</tbody>
</table>
Once we have all the input artifacts and each of them is converted to Neo4j graph using the architecture and implementation described in Chapter 5, the next step is to ensure correctness of the generated graph. We perform a manual evaluation for this purpose by comparing the input artifacts and their mapping to the Neo4j graph.

Table 6.5 shows the results for each of the four schemas and their mapping to the Neo4j graph.

Table 6.5: Verification of the modeling of schemas in the implementation graph model - Neo4j

<table>
<thead>
<tr>
<th>Schema</th>
<th>Relations</th>
<th>Attributes</th>
<th>PK</th>
<th>FK</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>sakila_op1</td>
<td>13</td>
<td>78</td>
<td>15</td>
<td>20</td>
<td>✔️</td>
</tr>
<tr>
<td>sakila_op2</td>
<td>10</td>
<td>64</td>
<td>12</td>
<td>12</td>
<td>✔️</td>
</tr>
<tr>
<td>sakila_ids</td>
<td>16</td>
<td>119</td>
<td>18</td>
<td>23</td>
<td>✔️</td>
</tr>
<tr>
<td>sakila_dwh</td>
<td>9</td>
<td>129</td>
<td>10</td>
<td>9</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Next, for verification of each of the ETL transformations (in Neo4j) in a systematic manner, we organize the process into several small parts as shown by each column header in the Table 6.6 and determine the result for each of these parts. Table 6.6 shows the results of evaluation and verify that the mapping from the artifacts to the graph is correct.
Table 6.6: Evaluation of the ETL transformation in the generated architecture graph as compared to the results from manual inspection

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Total steps</th>
<th>Database related steps</th>
<th>Inter-step dependencies</th>
<th>Step To Attribute</th>
<th>Attribute To Step</th>
<th>Matches expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.load_actor</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>✓</td>
</tr>
<tr>
<td>2.load_staff</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>28</td>
<td>13</td>
<td>✓</td>
</tr>
<tr>
<td>3.load_address</td>
<td>12</td>
<td>8</td>
<td>7</td>
<td>25</td>
<td>8</td>
<td>✓</td>
</tr>
<tr>
<td>4.load_category</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>✓</td>
</tr>
<tr>
<td>5.load_city</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>6.load_country</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>7.load_customer</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>24</td>
<td>10</td>
<td>✓</td>
</tr>
<tr>
<td>8.load_film</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>47</td>
<td>27</td>
<td>✓</td>
</tr>
<tr>
<td>9.load_film_actor</td>
<td>11</td>
<td>8</td>
<td>7</td>
<td>19</td>
<td>6</td>
<td>✓</td>
</tr>
<tr>
<td>10.load_film_category</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>19</td>
<td>6</td>
<td>✓</td>
</tr>
<tr>
<td>11.load_inventory</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>11</td>
<td>5</td>
<td>✓</td>
</tr>
<tr>
<td>12.load_language</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>✓</td>
</tr>
<tr>
<td>13.load_payment</td>
<td>12</td>
<td>10</td>
<td>9</td>
<td>33</td>
<td>8</td>
<td>✓</td>
</tr>
<tr>
<td>14.load_rental</td>
<td>13</td>
<td>11</td>
<td>10</td>
<td>33</td>
<td>16</td>
<td>✓</td>
</tr>
<tr>
<td>15.load_store</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>✓</td>
</tr>
<tr>
<td>16.aux_staff_store</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>28</td>
<td>13</td>
<td>✓</td>
</tr>
<tr>
<td>17.load_dim_actor</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>✓</td>
</tr>
<tr>
<td>18.load_dim_customer</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>17</td>
<td>10</td>
<td>✓</td>
</tr>
<tr>
<td>19.load_dim_film</td>
<td>21</td>
<td>10</td>
<td>9</td>
<td>3</td>
<td>63</td>
<td>✓</td>
</tr>
<tr>
<td>20.load_dim_staff</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>12</td>
<td>✓</td>
</tr>
<tr>
<td>21.load_dim_store</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>14</td>
<td>16</td>
<td>✓</td>
</tr>
<tr>
<td>22.load_fact_rental</td>
<td>12</td>
<td>8</td>
<td>7</td>
<td>10</td>
<td>28</td>
<td>✓</td>
</tr>
</tbody>
</table>
6.4 Impact assessment analysis

We illustrate the effectiveness of our work using a set of examples and a tabular summary showing the impact analysis results for the artifacts (beyond the ones discussed in examples) that cover each of the four schemas in our work. Our results highlight the two main contributions of our work. First, reporting the impact of a change over all the artifacts in a data warehouse architecture. Secondly, providing complete path trace of how the evolved artifact and the impacted artifacts are related to each other. The path information helps to understand how elements are related when the relationship is not a direct connection and thus apparent. We also offer a text description of the presented graph which elaborates on the conceptual meaning of the artifacts in the graph.
6.4.1 Examples

We analyzed the schema evolution history of the sakila database. The database has evolved over a series of ten versions and there have been 81 different evolution operations performed across these versions. The most common operations have been the data-type changes and adding a new column. The other operations involved deleting and renaming a column, adding a new relation and making changes to stored procedures or triggers. Data-type changes comprise of about 25% of the total 81 schema changes while adding a column constitute about 29% of the schema changes.

In this section, we present the results in the form of short examples presenting the impact of schema change identified using our approach. The three specific examples presented here focus not only on highlighting the result but also explain the graph queries used to determine the impact.

**Example 1**: Consider the possibility of renaming the attribute `language_id` in the relation `language` in the operational schema `sakila_op1`. Before making the change, the user would like to identify all the artifacts in the rest of the architecture that will be impacted by it. Using the system implementation described in Chapter 5, we transform all the artifacts in the architecture (schemas, ETL transformations, and queries) to a Neo4j graph. The impact of the rename operation can be determined by submitting a path analytical query over the Neo4j graph. Figure 6.14 presents the result along with the the query.

The output presents 49 elements that will be potentially impacted by the proposed change. These elements include ETL steps for the transformations between operational and reconciled schema as well as the transformations between the reconciled and the warehouse schema. Two queries, `sakila_q5`, `sakila_12` will be impacted accompanied by attributes in the reconciled and warehouse schema.
The graph in Figure 6.14 not only shows the 49 elements that are impacted by change but also captures the path from the source artifact - `sakila_op1.language.language_id` to each of the impacted artifacts. Having this path trace constitutes provenance information which identifies the lineage of a particular data artifact. The impact can be understood as follows.

1. Box 1 highlights the intra-schema impact highlighting two foreign key violations, `original_language_id, language_id` and one primary key violation with a node of type, `Relation` colored purple.

2. The next step involves identifying all those ETL transformations between the operational and reconciled schema which are reading values from this attribute, `sakila_op1.language.language_id`. Box 2 highlights those two ETL steps. The two steps belong to the transformations, `load_language` and this can be determined by clicking on each of the nodes which shows their properties in the status bar. One of the properties of a step node is `transname` which represents the name of the transformation.

```
1. MATCH (a:Attribute{name:'language_id', relation:'language', schema:'sakila_op1'})
2. WITH a
3. OPTIONAL MATCH (a)-[r1]-(distance1_pk_fk) WHERE r1.type='PrimaryKey' OR r1.type='ForeignKey'
4. WITH a, r1, distance1_pk_fk
5. OPTIONAL MATCH (a)-[rq]->(query:Query)
6. WITH a, r1, distance1_pk_fk, rq, query
```

Figure 6.15: Cypher query snippet (a) for evolution operation 1
3. Based on the ETL steps identified in the previous step, Box 3 identifies the impact on the reconciled schema. The attribute, *name* from the relation, *language* is expected to be impacted in the reconciled schema.

4. Box 3 also highlights the impact of *name* attribute on the ETL transformations between the reconciled schema and the warehouse schema. These are represented by their steps, *Lookup language, Lookup Original Language* which belongs to the transformation, *load_dim_film*.

5. Further consequence of these ETL steps exist on the loading of several attributes in the warehouse schema shown by attribute nodes linked to the node, *Load dim_film*. This is because, these steps impacts the Insert/Update step *Load dim_film* which is responsible for populating the relation *dim_film* in the warehouse schema.

6. The two nodes in red highlight the impact of proposed change on the SQL queries.

Figure 6.15 presents the corresponding query that was created and executed in Neo4j to determine the above impact. We present the query in small snippets followed by an explanation of the snippet.

The query above identifies the intra-schema impact of the attribute, *language_id*. This includes any queries which are expressed over the schema *sakila_op1* and which reference the attribute, *language_id*, or any primary or foreign key violations.

Line 1 matches the artifact on which the change is proposed. The *WITH* clause in Cypher allows all the variables from the previous *MATCH* clauses to be carried over in the subsequent query steps. The variable, *a* in Line 3 is available since it was carried over on Line 2. Identification of the primary and foreign key dependencies is
detected on Line 3. Similarly, the impact on the queries is determined on Line 5. We use the concept of Label on Lines 1 and 5 for pruning the node search in the graph. For example, on Line 5, for the target node in the MATCH clause, only the nodes with the label, Query are considered, thus reducing the number of hits made by the database and reducing the cost of computation.

Figure 6.16 shows the section of the query which identifies the impact on ETL transformations between the operational and reconciled schema, and the reconciled schema.

```cypher
7. OPTIONAL MATCH (a)-[r2]->(distance1_step:Step)
8. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, query
9. OPTIONAL MATCH (distance1_step)-[r3]->(ids_gateway_step:Step)
10. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, query
11. OPTIONAL MATCH (ids_gateway_step)-[r4]->(ids_attributes:Attribute)
12. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r4, query
13. OPTIONAL MATCH (ids_attributes)-[r5]->(ids_pk_fk) where r5.type='PrimaryKey' OR r5.type='ForeignKey'
14. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_fk, r4, query
15. OPTIONAL MATCH (ids_attributes)-[rq2]->(query_ids:Query)
    WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_fk, r4, query, rq2, query_ids
```

Figure 6.16: Cypher query snippet (b) for evolution operation 1

The query in Figure 6.16 can be understood as follows. Lines 7 and 9 identify the impact of the attribute, language_id from the operational schema, on the ETL transformations that exist between the operational and reconciled schema. Lines 11-15 present the impact on the reconciled schema and any queries expressed over it. The variable, ids_gateway_step refer to the ETL steps that are responsible for loading data
into the reconciled schema. Figure 6.17 shows the section of query which identifies the impact on the warehouse schema.

```cypher
16. OPTIONAL MATCH (ids_attributes)-[r6]->(dwph_input_step:Step{layer:ids-dwh})
17. WITH a, r1, distance1_pk_r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_r6, dwph_input_step, rq, query, rq2, query_ids
18. OPTIONAL MATCH (dwph_input_step)-[r7]->(dwph_output_step:Step{layer:ids-dwh})
19. WITH a, r1, distance1_pk_r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_r6, dwph_input_step, r7, dwph_output_step, rq, query, rq2, query_ids
20. OPTIONAL MATCH (dwph_output_step)-[r8]->(dwph_attr:Attribute{schema:sakila_dwh})
21. WITH a, r1, distance1_pk_r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_r6, dwph_input_step, r7, dwph_output_step, r8, dwph_attr, rq, query, rq2, query_ids
22. OPTIONAL MATCH (dwph_attr)-[r9]->(dwph_query:Query)
23. RETURN a, r1, distance1_pk_r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_r6, dwph_input_step, r7, dwph_output_step, r8, dwph_attr, r9, dwph_query, rq, query, rq2, query_ids
```

Figure 6.17: Cypher query snippet (c) for evolution operation 1

**Example 2:** Consider drop operation on the column, `first_name` in the `actor` relation in the **reconciled schema**. Figure 6.18 shows the impact including the lineage information. The impacted artifacts constitute three queries (sakila-q7, sakila-q8, sakila-q10), one relation in the data warehouse schema (dim_actor) and three ETL steps. Box 1 captures the step in an ETL transformation, `load_actor` between operational and reconciled schema. It is impacted since it is responsible for loading values into the attribute, `first_name`. The ETL steps captured in Box 3 belong to a transformation that exists between the reconciled and the warehouse schema.

The query in Figure 6.18 can be understood as follows. While lines 1-5 capture the intra-schema impact, Line 7 identifies all the artifacts and ETL steps between the operational and reconciled schema that are responsible for loading data into the attribute, `actor.first_name`. Line 9 aims to assess the impact of the attribute on the ETL steps between reconciled and warehouse schemas and Lines 11-13 further determine the impact on the warehouse schema including any queries that are expressed on it.
Example 3: Consider the case of dropping an attribute, \textit{store\_valid\_from} from the relation, \textit{dim\_store} in the \textbf{warehouse schema}. Figure 6.19 shows the query and the presented impact.

The next three evolution operations present the analysis for an evolution operation over relations as opposed to attributes. We consider three relations in each of the three schemas - \textit{sakila\_op1}, \textit{sakila\_ids}, \textit{sakila\_dwh}.

Figure 6.18: (a) The complete path illustrating the artifacts impacted by evolving \textit{sakila\_ids\.actor\_first\_name}. (b) Cypher query
Figure 6.19: The complete path and the Cypher query illustrating the artifacts impacted by drop attribute, `dim_store.store_valid_from` country operation

**Example 4:** Figure 6.20 shows the impact of dropping or renaming a relation `actor` in the operational schema, `sakila_op1`.

Figure 6.21 shows the corresponding query. The difference between identifying the impact of a schema change over a relation vs. an attribute is that we identify the dependencies between relations and ETL steps and queries in the former case. For example, Lines 9, 11 illustrate this point. The variable `ids_relation` is contextualized using the label `Relation`. On Line 11, we are identifying the impact of a relation on the query.

The next example captures the impact of dropping the relation, `language` in the reconciled schema, `sakila_ids`.
Figure 6.20: Impacted artifacts for the operation - drop relation, *actor* from the schema *sakila_op1*

**Example 5:** Figure 6.22 shows the impact of dropping or renaming a relation *language* in the operational schema, *sakila_ids*.

Figure 6.23 shows the corresponding query. Lines 1-4 captures all the steps of the transformations between the operational and reconciled schema that load data into the relation, *language*. Similarly, lines 5-6 identifies the impact of the relation on the ETL transformations between reconciled and warehouse schema. The impact on any queries on the reconciled or warehouse schema is captured on Lines 7 and 13.

The next example captures the impact of a schema change over the relation, *dim_film* in the warehouse schema, *sakila_dwh*.

**Example 6:** Figure 6.24 shows the impact of dropping or renaming a relation *dim_film* in the warehouse schema, *sakila_dwh*. 

107
Figure 6.21: Cypher query for evolution operation 4

Figure 6.25 shows the corresponding query. Line 3 captures all the steps of the transformations between the operational and reconciled schema that load data into the relation, `dim.film`. Similarly, Line 5 identifies the impact on any queries expressed over the warehouse schema and the intra-schema impact (referential integrity constraints) is identified on Line 7.

The next example, Example 7 captures the impact of evolving the operational schema `sakila_op1` in our work.
Figure 6.22: Impacted artifacts for the operation - drop relation, \textit{language} from the schema \textit{sakila_ids}

\textbf{Example 7}: Figure 6.26 shows the impact of dropping or renaming the operational schema \textit{sakila_op1}.

Figure 6.27 shows the corresponding query. Line 3 captures the queries which are expressed over \textit{sakila_op1}. Lines 5 -7 shows the impact of evolving the schema \textit{sakila_op1} on the ETL steps between it and the reconciled schema. These ETL steps further impact the reconciled schema \textit{sakila_ids} which is shown in Line 9. The ETL steps between the reconciled and warehouse schema that are impacted by reconciled schema and load into the warehouse schema are captured on Lines 11-14. The impact on warehouse schema and queries is shown in Lines 15 - 19.

This concludes a discussion of seven detailed examples illustrating the result and the queries. We performed schema change operations over additional artifacts and the output graphs are available in Appendix 3.
In order to assess the effectiveness of our results, we compare the identified artifacts from our approach with the expected output determined through manual examination of the data. The evaluation is discussed in two steps: (1) a systematic investigation of the manual inspection of the graph to identify the expected number of artifacts for each evolution operation, (2) a tabular report illustrating the number of impacted artifacts for operations over several attributes.

To ensure that the manual examination of the expected output is accurate, we break the assessment task into a series of questions, which when answered allows to analyze the complete metadata graph in a systematic manner. These questions also highlight our graph traversal strategy for an evolution operation on each of the three schemas - operational, reconciled and warehouse schema. Table 6.7 shows the analysis drawn from the questions listed below.

1. MATCH (rel:Relation{name: 'language', schema: 'sakila_ids'})
2. WITH rel
3. OPTIONAL MATCH (opidsstep:Step{layer: 'opr-ids'}-[sa]-(rel)
4. WITH rel, opidsstep, sa
5. OPTIONAL MATCH (rel)-[r1]-(s:Step{layer: 'ids-dwh'})
6. WITH rel, opidsstep, sa, r1, s
7. OPTIONAL MATCH (rel)-[r2]-(q:Query)
8. WITH rel, opidsstep, sa, r1, s, r2, q
9. OPTIONAL MATCH (s)-[r3]-(s2)
10. WITH rel, opidsstep, sa, r1, s, r2, q, r3, s2
11. OPTIONAL MATCH (s2)-[r4]-(dwh_rel:Relation)
12. WITH rel, opidsstep, sa, r1, s, r2, q, r3, s2, r4, dwh_rel
13. OPTIONAL MATCH (dwh_rel)-[r9]-(dwh_query:Query)
14. WITH rel, opidsstep, sa, r1, s, r2, q, r3, s2, r4, dwh_rel, r9, dwh_query
15. OPTIONAL MATCH (rel)-[attr]-(a:Attribute)-[fk]-(a2:Attribute) where fk type='ForeignKey'
16. RETURN rel, opidsstep, sa, r1, s, r2, q, r3, s2, r4, dwh_rel, r9, dwh_query, attr, a, fk, a2

Figure 6.23: Cypher query for evolution operation 5

### 6.4.2 Verification of the impact assessment results

In order to assess the effectiveness of our results, we compare the identified artifacts from our approach with the expected output determined through manual examination of the data. The evaluation is discussed in two steps: (1) a systematic investigation of the manual inspection of the graph to identify the expected number of artifacts for each evolution operation, (2) a tabular report illustrating the number of impacted artifacts for operations over several attributes.

To ensure that the manual examination of the expected output is accurate, we break the assessment task into a series of questions, which when answered allows to analyze the complete metadata graph in a systematic manner. These questions also highlight our graph traversal strategy for an evolution operation on each of the three schemas - operational, reconciled and warehouse schema. Table 6.7 shows the analysis drawn from the questions listed below.
Figure 6.24: Impacted artifacts for the operation - drop relation, \textit{dim}\_film from the schema \textit{sakila}\_dwh

1. MATCH (rel:Relation\{name: 'dim\_film', schema: 'sakila\_dwh'\})
2. WITH rel
3. OPTIONAL MATCH (idsdwstep:Step\{layer: 'ids-dwh'\})-[sa]->(rel)
4. WITH rel, sa, idsdwstep
5. OPTIONAL MATCH (rel)-[r9]->(dwh\_query:Query)
6. WITH rel, sa, idsdwstep, r9, dwh\_query
7. OPTIONAL MATCH (rel)-[attr]-(a:Attribute)-(fk)->(a2:Attribute) where fk.type='ForeignKey'
8. RETURN rel, sa, idsdwstep, r9, dwh\_query, attr, a, fk, a2

Figure 6.25: Cypher query for evolution operation 6

The column headings in Table 6.7 represent the questions and they are as follows:

\textbf{a} – Intra-schema impact (entity integrity and referential integrity constraints and queries)

\textbf{b} - Number of ETL steps between operational and reconciled schema that read from the evolved attribute, A
Figure 6.26: Impacted artifacts for evolving schema sakila_op1

c – Number of reconciled schema artifacts. Call the list of artifacts as set B.

d - Number of impacted queries/constraints expressed over reconciled schema.

e - Number of ETL artifacts between reconciled and the warehouse schema that read from any of the attributes in set B.

f - Number of impacted attributes in DWH schema

g - Number of impacted queries expressed on the warehouse schema

Similarly, we ask ourselves following questions for assessing the impact of an evolution operation over reconciled attribute. The results are shown in Table 6.8 followed by the corresponding set of questions.

The column headers from Table 6.8 are explained as follows.

a – Intra-schema impact (entity integrity and referential integrity constraints)

b - Number of ETL steps between operational and reconciled schema that load data
Figure 6.27: Cypher query for evolution operation 7

into A.

c – Number of ETL steps between operational and reconciled schema that read from the attribute, A

d - Number of impacted queries/constraints expressed over reconciled schema.

e- Number of ETL steps between reconciled and the warehouse schema that read from A.

f - Number of artifacts in the warehouse schema

g - DWH queries

For assessing the impact of a schema change proposed over a data warehouse artifact,
we ask ourselves following questions and the results are shown in Table 6.9.

   a – Intra-schema impact (entity integrity and referential integrity constraints)

   b - Number of ETL steps between reconciled and the warehouse schema that read from A.

   c - Number of ETL steps between reconciled and the warehouse schema that load data into A.

   d - DWH queries

   Based on these questions, we get the following results.
Table 6.9: Manual inspection of impact assessment outcome for an evolution operation over warehouse schema

<table>
<thead>
<tr>
<th>Artifact</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim_store.store_valid_from</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>dim_date.day_abbreviation</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>✓</td>
</tr>
<tr>
<td>dim_film.film_in_category_foreign</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>fact_rental.count_rentals</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table in Figure 6.10 captures the results of impact assessment on a set of artifacts covering each of the four schemas in our work and exhibiting different evolution operations. This concludes a discussion of our case study. We illustrated the effectiveness of our approach using two methods: (1) through examples which covered evolution operations expressed over different artifact types (attribute, relation), and schemas at different layers in the architecture, and (2) summarizing the results from evolution operations over additional artifacts in the table in figure 6.10. The results highlight the two main contributions of our work: (1) reporting the impact of a change over all the artifacts in a data warehouse architecture, and (2) providing a complete path trace of how the evolved artifact and the impacted artifacts are related to each other.

6.5 Lineage tracking using pentaho

Pentaho, the data integration solution we use for modelling ETL transformations in our work also offers support for lineage tracing. In this section, we discuss their proposed solution and the limitations in the context of our work. Pentaho offers two solutions to determine the database related impact of an ETL transformation. These
ideas are closely related to our goal of assessing the impact of schema changes in a warehouse environment and ETL workflows are a critical component in data warehouses. We briefly present each idea along with the added contribution made by our approach in the same context.

6.5.1 db-impact

This refers to a workflow based solution which reads a set of transformation files (ETL workflows) and identifies the dependency between each transformation and the database information such as the name of the database, table, or attributes. Figure 6.28 shows the workflow provided by Pentaho.

![Image](db-impact.ktr)

Figure 6.28: Lineage support in Pentaho - db-impact.ktr [43]

The first step *Get Filenames* reads a set of ETL transformation files specified in the configuration settings for this step. The step *Modified Javascript Value* uses the class *TransMeta*\(^1\) provided by Pentaho’s core library. This class uses methods (*getTransformationName, getDatabaseName, and getTable*) of another class *DatabaseImpact* to load information about a transformation from its corresponding XML file. Figure 6.29 captures a screenshot of the javascript code that highlights the information (*transname*,

\(^{1}\)http://javadoc.pentaho.com/kettle/org/pentaho/di/trans/TransMeta.html
stepname, db, and table) which is being generated from this step. We added the last step (Microsoft Excel Output) to export and save the generated information into a file.

![Script](image)

```java
try {
    transMeta.analyseImpact(impact, null);
} catch(e) {
    error = e.toString();
}

for (i=0;i<impact.size();i++) {
    var dbi = impact.get(i);
    var newRow = createRowCopy(impact.get(i).size());
    var rowIndex = newRow.getRowMeta().size();
    newRow[rowIndex++] = transname;
    newRow[rowIndex++] = dbi.getStepName();
    newRow[rowIndex++] = dbi.getTypeDesc();
    newRow[rowIndex++] = dbi.getDatabaseName();
    newRow[rowIndex++] = dbi.getTable();
    newRow[rowIndex++] = dbi.getField();
    newRow[rowIndex++] = error;
}
```

Figure 6.29: Lineage support in Pentaho - db-impact.ktr

On executing the workflow from Figure 6.28, we get the output shown in Figure 6.30. We observe following limitations with the presented results.

1. Incorrect parsing of certain SQL queries. As an example, consider the last row in Figure 6.30. It shows the database related information for the step max_dim_actor_last_update. The output lists the field name as max_dim_actor_last_update which is not the actual attribute name. It is the alias used in the query. The actual field/attribute name is actor_last_update.

2. The table name is not retrieved for ETL step of type TableInput. The blank cells in Column F in Figure 6.30 highlights this limitation. The information about
table is only available for steps such as *Insert Update, Database Lookup*. In ETL steps of type, *Table Input*, table related information is embedded within a SQL query whereas for steps such as *Insert Update*, it is captured in a specific field and can be read by the method *getTable* of the class *DatabaseImpact*. Figure 5.11 in Chapter 5 shows how the table related information is specifically captured in an *Insert Update* step type.

3. Certain fields are captured redundantly depending on the specific type. As an example, refer to the rows 2 and 3 in the Figure 6.30. The script does follow the semantics of the *Insert Update* step according to which, the data in the target table is first compared with the data from the incoming stream (or the previous step). This comparison is performed on the basis of a column value (*actor_id*). If the values do not match, then the fields listed in the *Update fields* section in the configuration settings for the step are updated. Since *actor_id* is a part of both the lookup and update sections, it appears twice in the generated excel output. Our work is based on identifying the impact syntactically whereas the above scenario is about having a field whose read/write nature will depend on the actual data values and thus is not within the scope of our work. Furthermore, it introduces cycles in the graph since an attribute can impact a step (read) and vice versa (write).
6.5.2 extract-transformation-metadata

Pentaho offers another built-in script that extracts the metadata (database, table, and attribute information) related to a transformation. Pentaho uses XPathExpressions to read specific information from the XML files for each of the ETL transformations. Figure 6.32 shows the script. The limitation of the script is that it only provides information for a specific steptype (TableOutput).

First approach to overcome this limitation was to eliminate the filter step labelled TableOutput. This would capture data for all the steps. Figure 6.31 illustrates the limitation in adopting this approach. The database information is not captured uniformly for the different step types. This is because each step type uses a different XML schema and thus the XPathExpressions are different.

As the second approach, one could create multiple scripts one for each steptype. However, we developed custom code in Java which uses the same idea of writing XPathExpressions to extract required information. Our Java code offers two additional advantages beyond creating scripts in Pentaho. First, as the goal of developing a unified application that can address all components of our architecture (Figure 1.1), incorporating the parsing of ETL workflows within Java offered an inclusive solution. Using scripts in Pentaho is limited to only a subset of artifacts from our architecture.
Figure 6.31: Output after step labelled *TableOutput* is deleted from script in Figure 6.32.

(ETL workflows and schema). Using Java, we retrieve information about all the step types and not only *TableOutput* and the transformation of ETL workflows (from XML) to CSV is embedded within the user interface. Second, having the code encapsulated within Java and presented to the user through user interface supports abstraction for the end-user and also supports flexibility. The code can be adapted to match another data integration solution than Pentaho without impacting the user application. Also, our code is modular, having independent code modules which parse ETL workflows for each step type.

### 6.6 Impact assessment in pentaho

We examine Pentaho’s behavior in the event of a schema change by renaming an attribute in the relation *dim_actor* in the warehouse schema *sakila_dwh*. Figure 6.33 shows how Pentaho reacts to the change when the ETL transformation *load_dim_actor*
Figure 6.32: Lineage support in Pentaho - extract-transformation-metadata.ktr

is executed. As soon as it determines the error, it stops execution and highlights the current step in red in the results bar.

This methodology also suffers from one of the limitations discussed above for the script extract-transformation-metadata. It is limited to ETL workflows only whereas our approach offers a holistic view of the impact on additional components of the architecture as well. These additional components include schemas and queries. Furthermore, we determine the impact on all the transformations whereas the approach in Figure 6.33 requires the user to manually examine each transformation.

6.7 Summary

We illustrated the effectiveness of our approach using different evolution operations (add a new column, rename an attribute/relation, drop a relation). The results highlight the two main contributions of our work: (1) reporting the impact of a change over all the artifacts in a data warehouse architecture, and (2) providing complete path trace of how the evolved artifact and the impacted artifacts are related to each other.
In the next chapter, we present the user interface that we developed based on the conceptual model from Chapter 3 and the architecture and implementation discussed in Chapter 5. We also present the design of our approach supported by a prototype for text-based descriptions of the presented impact assessment graph.
Table 6.10: Impact assessment results for different artifacts over each of the input schemas

<table>
<thead>
<tr>
<th>Schema</th>
<th>Artifact and evolution operation</th>
<th>Number of impacted artifacts</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operational schema</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sakila_op1</td>
<td>Rename attribute - language.language_id</td>
<td>48</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_op1</td>
<td>Drop attribute - staff.first_name</td>
<td>44</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_op1</td>
<td>Change datatype - customer.active</td>
<td>33</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_op1</td>
<td>Drop relation - actor</td>
<td>11</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_op2</td>
<td>Rename relation - film_category</td>
<td>12</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_op1</td>
<td>Drop schema</td>
<td>70</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Reconciled schema</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sakila_ids</td>
<td>Rename attribute - category.name</td>
<td>43</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_ids</td>
<td>Drop attribute - actor.first_name</td>
<td>10</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_ids</td>
<td>Change datatype - film.title</td>
<td>44</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_ids</td>
<td>Drop relation - film_category</td>
<td>10</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_ids</td>
<td>Rename relation - store</td>
<td>8</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_ids</td>
<td>Drop schema</td>
<td>55</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Warehouse schema</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sakila_dwh</td>
<td>Rename attribute - dim_film.film_in_category_foreign</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_dwh</td>
<td>Drop attribute - dim_date.day_abbreviation</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_dwh</td>
<td>Change datatype - dim_customer.customer_key</td>
<td>5</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_dwh</td>
<td>Drop relation - dim_film</td>
<td>5</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_dwh</td>
<td>Rename relation - dim_store</td>
<td>3</td>
<td>✓</td>
</tr>
<tr>
<td>sakila_dwh</td>
<td>Drop schema</td>
<td>11</td>
<td>✓</td>
</tr>
</tbody>
</table>
Chapter 7: User Interface and Impact Explanation

7.1 User interface

The focus of this chapter is to present a graphical user interface and the prototype we have developed for text-based impact explanations. The user interface is a web application implemented using the following technology stack: J2EE for application programming, Bootstrap CSS for front-end development, GSQLParser for transforming the SQL queries into XML, Neo4j graph database as the target graph model implementation, and MySQL database for our input relational schemas. We have used Eclipse as the development environment for Java; the software consists of three thousand lines of code. The code development and testing was performed on a 64-bit Windows machine, Intel i5-core processor, and 4GB RAM.

The web application’s functionality has two parts: converting input artifacts to the target graph model, and providing impact assessment for a specific artifact. In this section, we describe the screenshots of the user interface (UI) along with a description of the functionality covered by each of them. Figure B.1 shows the main files that implement the interface for each of the features below. The circles A, B, C, and D refer to the modules discussed later in the chapter (Figure 7.3, 7.9, 7.11, and 7.12) describing how each of the jsp files handle the functionality specific to it.
1. Converting input artifacts to the target graph representation (index.jsp, etl.jsp, and query.jsp)

2. Impact assessment for an artifact (lineage.jsp)

### 7.1.1 Converting input artifacts to the target graph representation

The main welcome screen for our application is shown in Figure 7.2. This page allows the user to transform a schema originally expressed using a relational database into the target graph representation (Neo4j). The schema, each of its relations and their attributes are converted to nodes in the graph with specific key:value properties as described in the conceptual model in Chapter 3. A user specifies the name of the schema, selects the schema type from the dropdown menu (operational, reconciled, and data warehouse) and clicks the *Convert To Graph* button. A message is displayed
Figure 7.2: The UI page for converting schemas to the target graph model

depending on whether the schema was successfully converted to the graph or if there were any errors.

A prerequisite step for using the interface to convert schemas to graph is that the set of CSV files that represent schema information about relations, attributes, and foreign keys should reside in a folder with the same name as the schema. When the user access the web interface (Figure 7.2) and specifies the schema name, the folder (holding the same name as the schema) is being read to access the CSV files. If no folder is found, an error message is returned. The successful operation of this part of the web application is based on two conditions: (1) Neo4j graph database is running, and (2) each relational schema that needs to be converted to the graph should be captured as a set of CSV files within a folder which holds the same name as the schema.

Figure 7.3 summarizes the flow diagram for the web page shown in Figure 7.2. The file names on the left with extensions- *jsp* and *java* correspond to the java code files that accomplish the functionality specified at that step. When the user clicks the
button - *Convert To Graph* in the UI, the control transfers to the code within the file *SchemaServlet.java* and proceeds as explained in the flow diagram in Figure 7.3.

![Flow diagram](image)

Figure 7.3: Flow diagram for application code driving the functionality in Figure 7.2

Having discussed the main code components (*index.jsp*, *SchemaServlet.java*, and *SchemaGraphController.java*), we now illustrate the working of the UI application. Figures 7.4, 7.5, and 7.6 show the different error messages that the application may return to the user. These errors can occur in the following three situations:

1. Neo4j database is not running at the time user submits a request either for transforming the schema to Neo4j representation or a request to view the graph.
2. If no folder exists with the same name as the schema specified on the web page exists.

3. If the folder with the schema name exists but does not contain at least one of the required CSV files (Relations.csv, Attributes.csv, and ReferentialIntegrityConstraints.csv).

Figure 7.4 shows the error message received by the user when the Neo4j database is not running. We entered a valid schema name \textit{(sakila\_op1)} and clicked \textit{Convert To Graph}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7_4.png}
\caption{Error messages for web page in Figure 7.2 when Neo4j database is not running}
\end{figure}

Next, we simulated a scenario where we successfully started the Neo4j database but we specified a schema name \textit{sakila\_opr1}. This is an invalid input since the application
could not find a folder with the name, *sakila_opr1*. Figure 7.5 shows the screenshot highlighting this error message.

The screenshot in Figure 7.6 shows the error received by the user when one of the required CSV files (*Relations*, *Attributes*, or *ReferentialIntegrityConstraints*) do not exist or are malformed. In the example here, the application could not find the file *Relations.csv*.

**Figure 7.5:** Error messages for web page in Figure 7.2 for an invalid input on schema name

**Figure 7.7** captures a screenshot when a schema is converted to the Neo4j graph successfully. In the example here, we converted an operational schema- *sakila_opr1* to a graph representation. Figure 7.7 shows the transformed graph. The node representing the schema *sakila_opr1* is shown in the box labeled one. The graph consists of 13 relations, 78 attributes connected by a total of 126 relationships. The relationships exist between the *Schema* node and each of its 13 relations, a *relation* node and each of its attributes, and between attributes depending on referential integrity constraints between them. Box labeled two shows the referential integrity constraint through the
edge between the attribute `rental.rental_id` and `payment.rental_id`. The bidirectional relationship between the node labeled `rental` and attribute labeled `rental_id` captures two points: (1) primary key relationship through the edge from attribute to the relation, and (2) a schema change over the relation impacts its attributes.

This completes a discussion of the UI modules and the corresponding application code that transforms a relational schema to a Neo4j graph. The next step is to transform ETL workflows to Neo4j graph. This step is semi-automated as it requires the end-user to manually execute certain additional queries once the workflows are transformed by the web application. The need for manual intervention comes from certain scenarios that exist in the design of ETL workflows. We document the manual steps in Appendix 4.

Figure 7.8 shows a screenshot of the webpage which allows a user to transform ETL transformations to a graph.

The process works as follows:
1. As a pre-requisite for using this interface, each of the ETL transformations in Pentaho are exported to an XML format using the native utility built in within Pentaho.

2. All the XML files are added to a folder which is read as shown in Figure 7.8.

3. Using code developed in Java, each of the XML files are parsed and CSV files are created. Different CSV files are created for each step type. This corresponds to the functionality behind the button *Convert from XML to CSV* in Figure 7.8.

4. For ETL steps such as *TableInput*, there may be a need for manually editing the generated CSV file. This is required since the configuration for the *TableInput*
Figure 7.8: The main web page for transforming ETL workflows to Neo4j

5. The application then proceeds to generate a graph by reading each of these CSV files. This corresponds to the functionality behind the button Generate Graph in Figure 7.8.

The overall code structure including various Java components involved are shown in Figure 7.9. The node $B$ corresponds to the connector in the flow diagram shown earlier in Figure B.1. Level 1 in the workflow represents the three options (Convert from...
XML to CSV, Generate Graph, and Next) that are available to the user when the page etl.jsp is loaded. While the module ETLServlet.java is invoked in each case but different methods are called within the servlet to handle each functionality. The files included in the dashed rectangles indicate the set of files that are used by ETLServlet.java to handle each case separately. The node C refers to the transition to the next webpage, query.jsp. This happens when the user clicks Next in Figure 7.8.

Figure 7.9: Java code structure for conversion of ETL workflows to a graph

The main reason for having two separate buttons, Convert from XML to CSV, and Generate graph, is because there may be a need for manual editing the CSV files as explained above. Appendix 4 contains details on all the manual steps required in detail but we also list them briefly here.

1. The CSV files input_steps.csv, and dbjoin_steps contain SQL queries. The information about attributes is extracted from these queries manually.
2. Extra relationships between the ETL steps are generated for a few ETL transformations. These transformations include `load_rental`, `load_language`, `load_film`, `load_dim_film`, and `load_fact_rental`. We execute Cypher queries directly in Neo4j browser to delete the extra relationships and the set of queries is listed in Appendix 4. The reason for the generation of these additional relationships is that in a given transformation, we relate an ETL step of type `TableInput`, `Database Lookup`, and `Database Join` to one of the step type: `Insert/Update`, `TableOutput`, `DimensionLookup`, or `CombinationLookup`. The five transformations listed above contain two multiple workflows which are executed in parallel to load data into the target database. The input step type in these transformations is connected to multiple output steps.

Once workflows are converted to a graph, we convert SQL queries to a graph shown in Figure 7.10.

Similar to the need for manual editing of the CSV files in ETL workflows, the CSV files for SQL queries may also require some editing from the user. This is because the utility GSQLParser does not handle nested function calls well and they are missed during the transformation. For example, consider the following expression in a SQL query-

\[
\text{CONCAT} (\text{UCASE} (\text{SUBSTR} (\text{actor.last_name},1,1)))
\]

GSQLParser does not identify the attribute `actor.last_name` in this case.

The overall code structure including is shown in Figure 7.11. The node labeled `C` represents the starting point when the user clicks any of the following three buttons- `Convert from XML to CSV`, `Next`, and `Generate Graph`. The control is transferred to the program `QueryServlet.java`. The set of programs shown in the dashed rectangles are referenced by `QueryServlet.java` for each of the two actions `Convert from XML to CSV`,
Figure 7.10: The main web page for transforming SQL queries to Neo4j

and Generate Graph. When the user clicks Next in the UI, the control is transferred to the code in the program LineageServlet.java represented by the node labeled D. This program handles user interactions for the front end captured by lineage.jsp (Figure 7.12)

7.1.2 Impact assessment - user interface and Neo4j browser

Our web application endpoint-

http://localhost:8080/ImpactAssessmentProject/lineage.jsp

is shown in Figure 7.12. It corresponds to the UI interface that allows a user to select a specific artifact (schema, relation, or attribute) and assess its impact on the rest of the artifacts in the architecture. The application also allows a user to view all the artifacts in the system that are responsible for loading data into that specific artifact.
Thus, the functionality is split across two roles: (1) source analysis, and (2) impact assessment. Impact assessment shows all the artifacts that are populated as a result of the evolved artifact as well as the artifacts that actually populate the evolved artifact. As an example, if an attribute in the reconciled schema undergoes a schema change, then the source analysis will highlight only the ETL artifacts that are responsible for loading the specific attribute. However, the impact assessment will be a union of all the artifacts presented as a part of the source analysis and the additional artifacts (queries on the reconciled schema, ETL transformations between the reconciled and warehouse schemas, and warehouse schema elements) that are dependent on the specified attribute.
The interface in Figure 7.12 proceeds by asking the user to select ArtifactType, which can be a schema, relation, or attribute. The textbox for schema, relation, and attribute name is enabled or disabled based on the user’s selection of the artifact type. For example, if the user selects Relation, then the textbox for Attribute Name will be disabled. Once the name of the specific artifact is entered, a user can request one of the following actions: (1) view all the sources that lead to the population of the specified artifact (determine its sources), or (2) view the impact of the artifact on the rest of the elements in the architecture (determine its impact).
Figures 7.13a and 7.13b presents an example of impact assessment for a schema change over an attribute *first_name* of the relation *actor* in the reconciled schema *sakila_ids*. The screenshot in Figure 7.13b shows the result after the user specifies the artifact (Figure 7.13a) and clicks the button *Determine its impact*. A Cypher query is generated and shown to the user as the result.

Figure 7.14 shows the result of executing the presented Cypher query shown in Figure 7.13b in the Neo4j browser (http://localhost:7474/browser). The box labeled 1 shows the ETL step *InsertUpdate* from the transformation *load_actor*. This step is responsible for populating data into the attribute *first_name*. The nodes in the box labeled 2 show the SQL queries which involve the attribute *first_name*. These queries are expressed directly over the reconciled schema. The box labeled 3 shows the impact on the ETL transformations between the reconciled schema and the warehouse schema. The four nodes on the right represent the attributes in the relation *dim_actor* in the warehouse schema *sakila_dwh*.

Figure 7.15 shows the interface for *source analysis* of an operational schema. Since operational schemas are the starting point for data in a data warehouse environment, no artifacts in the architecture (ETL transformations) are responsible for populating them. This aspect is presented to the user as shown in Figure 7.15. Thus, source analysis of an operational schema does not show a Cypher query as the output.

Appendix C shows screenshots for additional artifacts. These include impact assessment for attributes and relations over each of the schemas (operational, reconciled, and warehouse). An evolution operation directly over each of the schemas (drop/rename a schema) is also shown in Appendix 3.
7.2 Impact explanation

The goal of this section is to illustrate the motivation behind offering an impact explanation module and how we accomplish it in our work. We allow the user to propose a schema change over an artifact and identify its impact on the rest of the elements in the architecture. Instead of providing a list of impacted artifacts, we also show the path trace that highlights how the evolved and the impacted artifacts relate to each other. This offers one form of explanation since the user can traverse the path
starting from the evolved artifact and understand its relationships (which may not be direct) with the impacted artifacts.

However, this still leaves us with one remaining challenge which is about a user’s understanding of the generated graph. This challenge comes from the fact that each node and relationship in our Neo4j graph consists of additional properties in the form of key:value pairs. While this information is presented when the node/relationship is clicked, the user has to examine each of the nodes/relationships in the result graph. As a part of impact explanation, we address this challenge by providing a text description of each path and we expand on the node’s properties in the description. We offer a prototype for impact explanation along with the explanatory phrase-templates and pseudo-code for traversing the impact assessment graph.
We leverage the approach proposed by Koutrika et al. [83] who focus on generating explanations for SQL queries in plain English. We develop a set of phrases that describe each possible relationship between two nodes. The phrase description is created based on the semantics expressed in the conceptual model in Section 2 and they explain the conceptual meaning of each relationship and expand the properties of nodes and relationships. Table 7.16 shows the text descriptions for each pair of nodes and relationship type we have in our work.

Given an impact assessment graph (such as Figures, 6.14 and ??), we offer a text interpretation by traversing the graph (in a depth-first manner) starting from the node where the change was proposed. For each edge, we read the corresponding phrase from the set of phrases in Table 7.16 based on the source and target node type of the edge. The details such as Attribute.name, and Step.transname in each phrase are replaced by
<table>
<thead>
<tr>
<th>Source artifact</th>
<th>Destination artifact</th>
<th>Phrase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute (name, relation, schema, isKey, datatype)</td>
<td>Step (name, transname, steptype, layer)</td>
<td>Step (name, transname, steptype, layer) reads from the attribute named Attribute.name of the relation Attribute.relation in the schema Attribute.schema.</td>
</tr>
<tr>
<td>Step (name, transname, steptype)</td>
<td>Attribute (name, relation, schema, isKey, datatype)</td>
<td>An ETL Step named Step.name belonging to the transformation Step.transname loads/populates the attribute named Attribute.name of the relation Attribute.relation in the schema Attribute.schema.</td>
</tr>
<tr>
<td>Attribute (name, relation, schema, isKey, datatype)</td>
<td>Query (name)</td>
<td>The attribute named Attribute.name from the relation Attribute.relation in the schema Attribute.schema participates in the query Query.name.</td>
</tr>
<tr>
<td>Step (name, transname, steptype, layer)</td>
<td>Step (name, transname, steptype, layer)</td>
<td>An ETL Step (source artifact) named Step.name belonging to the transformation Step.transname impacts another ETL step (destination artifact) named Step.name in the transformation Step.transname.</td>
</tr>
<tr>
<td>Attribute (name, relation, schema, isKey, datatype)</td>
<td>Attribute (name, relation, schema, isKey, datatype)</td>
<td>The attribute (source artifact) named Attribute.name from the relation Attribute.relation in the schema Attribute.schema holds a referential integrity constraint with the attribute (destination artifact) named Attribute.name of the relation Attribute.relation in the schema Attribute.schema.</td>
</tr>
<tr>
<td>Attribute (name, relation, schema, isKey, datatype)</td>
<td>Relation (name, schema)</td>
<td>The attribute named Attribute.name from the relation Attribute.relation in the schema Attribute.schema serves as the primary key in the relation Relation.name.</td>
</tr>
<tr>
<td>Relation (name, schema)</td>
<td>Step (name, transname, steptype, layer)</td>
<td>The ETL Step named Step.name belonging to the transformation Step.transname reads from the relation named Relation.name in the schema Relation.schema.</td>
</tr>
<tr>
<td>Step (name, transname, steptype, layer)</td>
<td>Relation (name, schema)</td>
<td>The ETL Step named Step.name belonging to the transformation Step.transname loads/populates the relation named Relation.name in the schema Relation.schema.</td>
</tr>
<tr>
<td>Relation (name, schema)</td>
<td>Query (name)</td>
<td>The relation named Relation.name in the schema Relation.schema participates in the query Query.name.</td>
</tr>
<tr>
<td>Schema (name)</td>
<td>Step (name, transname, steptype, layer)</td>
<td>The ETL Step named Step.name belonging to the transformation Step.transname reads from the schema named Schema.name.</td>
</tr>
<tr>
<td>Step (name, transname, steptype, layer)</td>
<td>Schema (name)</td>
<td>An ETL Step named Step.name belonging to the transformation Step.transname loads/populates the schema named Schema.name.</td>
</tr>
<tr>
<td>Schema (name)</td>
<td>Query (name)</td>
<td>The query named Query.name is expressed over the schema Schema.name.</td>
</tr>
</tbody>
</table>

Figure 7.16: Phrases to facilitate impact explanation

the actual data from a node’s property values. These phrases offer an explanation that is more intuitive than solely traversing the graph and clicking on a node/relationship to examine its properties.

Figure 7.17 shows an example of the text description generated for the accompanied Neo4j graph. The graph shows the impact of a schema change over the relation film_category in the reconciled schema sakila_ids. The first statement shows that the ETL transformation between operational and reconciled schemas, load_film_category,
Figure 7.17: Example of text-based description generated for the accompanied graph would be impacted since it loads data into the evolved relation. Similarly, lines 2-6 explain the impact of the relation, film\_category on the queries expressed directly over it. The lines 7 and 8 highlight the impact of the relation on the ETL steps that load data into the warehouse schema. Each of the statements are generated based on following three elements: (1) source node type and its properties, (2) target node type and its properties, and (3) the phrases shown in Table 7.16.
The algorithm for graph traversal is shown in Figure 1. We use a stack data structure for the standard depth-first algorithm. Given the graph (which reflects the impact assessment graph) and a node \textit{current\_node} which represents the artifact on which a change is proposed, the algorithm processes the graph in two parts: (1) recording and processing all edges (or paths) that start from an artifact and end at the node, \textit{current\_node} (Lines 6-15), and (2) recording all edges which start from the \textit{current\_node} and process them in a depth-first manner (Lines 16-30). At each step while processing an edge, our algorithm reads the phrase templates as defined in Figure 7.16 and construct complete phrases from them by adding the missing details (such as \textit{Attribute\_name}, \textit{Step\_transname} etc.) in the template. The helper function \texttt{appendPhraseBasedOnTemplate} reads the source and target nodes of the edge and their associated properties. These properties are used to compose phrases based on the templates. Line 31 shows the complete text interpretation of the presented graph.

We observe the potential of extending the current explanations to make them more concise and by categorizing them in a certain manner for ease of readability. For example, all the dependencies between a relation and the queries expressed over it can be grouped together in one statement instead of having multiple statements (Lines 2-6 in Figure 7.17). In our work, we focused on presenting explanations accompanied by a proof of concept but we believe that the idea holds merit towards future work.
**input**: Graph $G = (V, E)$ such that $V = \{\text{Schema, Relation, Attribute, Step, Query}\}$

$n \in V$ artifact on which the change is proposed

$E$ is the set of edges $(v_1, v_2)$ where $v_1, v_2$ are vertices

**output**: Text description of each path from the evolved artifact

**Data:**

1. `begin`
2. `current_node \leftarrow n;`
3. `source_node \leftarrow \text{Empty;}`
4. `dest_node \leftarrow \text{Empty;}`
5. `path_stack \leftarrow \text{Empty;}`
6. `// process all the edges $(v_1, v_2)$ such that $(v_2)$ is the node where change is proposed`
7. `for e \leftarrow (v_1, \text{current_node}) do`
8. `\quad \text{path_stack.push(e);}`
9. `end for`
10. `while \text{path_stack} \neq \emptyset do`
11. `\quad e = \text{path_stack.pop();}`
12. `\quad source_node = \text{getSourceNode(e);}`
13. `\quad dest_node = \text{getDestinationNode(e);}`
14. `\quad str = \text{appendPhraseBasedOnTemplate(source_node, dest_node);}`
15. `end while`
16. `path_stack \leftarrow \text{Empty;}`
17. `// process all the edges $(v_1, v_2)$ such that $(v_1)$ is the node where change is proposed`
18. `for e \leftarrow (current_node, v_2) do`
19. `\quad \text{path_stack.push(e;);}`
20. `end for`
21. `while \text{path_stack} \neq \emptyset do`
22. `\quad e = \text{path_stack.pop();}`
23. `\quad source_node = \text{getSourceNode(e);}`
24. `\quad dest_node = \text{getDestinationNode(e);}`
25. `\quad str = \text{appendPhraseBasedOnTemplate(source_node, dest_node);}`
26. `\quad current_node = dest_node;`
27. `\quad for e \leftarrow (current_node, v_2) do`
28. `\quad \quad \text{path_stack.push(e;);}`
29. `\quad end for`
30. `end while`
31. `Print str;`
32. `end`

**Algorithm 1**: Impact Explanation
Chapter 8: Employing Graph Databases Towards Addressing Heterogeneity and Integration

The advent of big data and NoSQL data stores has led to the proliferation of data models exacerbating the challenges of information integration and exchange. It would be useful to have an approach that allows leveraging both schema-based and schema-less data stores. We present a graph-based solution that attempts to bridge the gap between different data stores using a homogeneous representation. As the first contribution, we present and demonstrate a mapping approach to transform schemas into a homogeneous graph representation. We demonstrate our approach over relational and RDF schemas but the framework is extensible to allow further integration of additional data stores. The second contribution is a schema merging algorithm over property graphs. We focus on providing a modular framework that can be extended and optimized using different schema matching and merging algorithms.

Recently, there has been a shift in the volume of data generated, diversity in data forms (structured, semi-structured and unstructured) and an unprecedented rate at which data is produced. The data possessing these characteristics is termed as big data and the field is forecasted to grow at a 26.4% compound annual growth rate through 2018, according to a report released by Intelligent Data Corporation [1]. The need for faster analysis over big data with current storage and computation power
poses a major challenge for enterprise data infrastructures. Two alternatives to handle analysis over this newer form of data are: (a) scale-up (adding CPU power, memory to a single machine), and (b) scale-out (adding more machines in the system creating a cluster). The main advantage of scaling out vs. scaling up is that more work can be done by parallelizing the workload across distributed machines. Many existing relational database systems are designed to perform efficiently on single-machines. It should not be misconstrued that relational systems cannot scale-out at all. They can, but they lose the features that they are primarily designed such as ACID compliance, for example [2].

NoSQL (Not Only SQL) describes an emerging class of storage models designed for scalable database systems. NoSQL data stores advocate new and relaxed forms of data storage that do not require a schema to be enforced for the underlying data. Instead, a schema is identified and generated at the application side when the data is read from the system. This concept of postponing schema definition to a later point has enabled NoSQL storage models to be applied in many real-world use-cases. However, the popularity has also created a notion that schemaless data management techniques are more suitable for solving emerging data problems than schema-based structures.

While NoSQL data stores offer interesting and novel solutions for managing big data, they are also not a panacea for all data management related scenarios. In scenarios that need query optimization, data governance, and integrity, schema-based stores offer a better solution. Furthermore, a large amount of enterprise data still resides in relational databases. The greater scalability of NoSQL databases over relational databases comes at a price. Most NoSQL systems compromise certain features, such as strong consistency, to achieve efficiency over other critical features of performance.
and availability. Organizations such as Facebook and Hadapt who have widely embraced big data technologies also choose a data store on a use-case basis as opposed to leveraging a single big data storage technique for all their applications and data storage requirements [94]. More recently, a number of SQL implementations have also emerged that are built over platforms and programming models such as MapReduce that support big data [84, 111, 55].

Another field that is growing rapidly is semantic web and linked data technologies. Linked data builds upon the existing web and offers the idea of annotating the web data (and not just the documents) [72] using global identifiers and linking them. The data is published and organized using RDF (Resource Description Framework), which is based on a subject-object-predicate framework. RDF schema specifications and modeling concepts differ from relational databases and NoSQL data models, thus supporting the need and demand for creating schema and data integration solutions. Hitzler et al. [72] identify linked data as a part of the big data landscape.

These recent developments indicate two ideas: (1) the significance and prevalence of structured data in the enterprise world, and (2) the unique advantages possessed by different classes of data stores. In order to reap benefits from all of them, it is important to bring them together under a homogeneous model. This would serve two purposes: (1) offer more complete knowledge by combining data stored in isolated sources, and (2) facilitate harnessing value from each of the data stores (schema-based and schema-less), thus making them complementary and not competitive solutions.

In this chapter, we address this need by adopting graphs as a means towards standardization and integration of different data stores, thus handling the variety characteristic of big data. Our selection of a graph model is based on the following observations:
1. Graph databases are a NoSQL data storage model and thus support the big data processing framework [71].

2. Graphs provide a simple and flexible abstraction for modeling artifacts of different kinds in the form of nodes and edges.

3. Graph databases are attracting significant attention and interest in the past few years as highlighted from the Google Trends analysis shown in Figure 8.1. The values are normalized representing the highest value in the chart as 100% and the x-axis labels are marked in two-year time intervals.

4. The graph model adopted in our work, Neo4j, possesses a query language called Cypher and allows programmatic access using API.

The main contributions of this chapter are as follows:

1. A concept-preserving, integrated graph model that addresses the model heterogeneity and variety dimension of the big data landscape.

2. A software-oriented, automated approach to transform relational and RDF schemas into a graph database.
3. A proof-of-concept that illustrates the potential of graph-based solutions towards addressing diversity in data representations.

4. A framework accompanied by a proof-of-concept for schema merging over property graphs.

The rest of the chapter is organized as follows. Section 8.1 presents an overview of the concepts and terms. These include a discussion of property graphs and Neo4j graph database in particular, and relational and RDF data models as our native models of interest. Section 8.2 describes our transformation rules for converting a relational schema to a property graph. We leverage the approach proposed by Bouhali et al. [36] for converting RDF to a property graph representation and extend it to support additional models. Next, we present a proof-of-concept to illustrate the implementation of our transformation rules. We consider schema excerpts for relational and RDF models and show their corresponding property graphs generated using the transformation rules. Section 8.3 introduces our architecture and the motivation behind mapping non-graph based schemas to a property graph. The evaluation issues for schema mapping are reported in Section 8.4. We next present our approach towards schema integration over property graphs in Section 8.5. The challenges and our proposed solution towards resolving them are illustrated using an example. Section 8.6 presents the schema integration algorithm. In section 8.7, we use two case-studies to evaluate our approach for schema integration. Section 8.8 addresses the related work in graph-based integration and transformations while Section 8.9 offers conclusions and future work.
8.1 Background

In this section, we overview the concepts that form a foundation for our research. These include relational and RDF schemas that serve as input schemas. There are many graph based models; we consider the property graph as our model of interest and introduce it here briefly. We leverage Neo4j [6] in our work.

8.1.1 RDF

RDF stands for Resource Description Framework. It refers to the model and RDF schema is commonly abbreviated as RDFS. RDF allows annotating web resources with semantics. The resources and semantic information is represented in the form of classes and properties which form two core concepts of an RDF schema.

The notion of classes and objects in RDFS differ from the similar concepts that exist in conventional, object-oriented systems [15]. In many systems, classes contain a set of properties and each of the class instances possesses those properties. However, in RDF, properties are described in terms of classes to which they apply. These classes are referred to as domain in RDF schema, and similarly, the values that a certain property can hold is described using range. For example, we could define a property member that has domain Group and its range would be Person. Nejdl et al. [93] and W3C specification [14] offer a summary of RDF classes and properties.

8.1.2 Relational schema

A relational schema is described as a set of relations which consists of a set of attributes and constraints. The relations are connected by different types of relationships with cardinality constraints. There are two major types of constraints: entity integrity
and referential integrity. The entity integrity constraint states that every relation has a set of attributes, termed primary key, that uniquely identifies each of the tuples in the relation. The primary key attribute(s) may not be null. The referential integrity constraint applies to a pair of relations that are associated with each other. Having this constraint signifies that every value of one attribute in one of the participating relations comes from the set of values of a primary key attribute in the other relation.

### 8.1.3 Neo4j and property graphs

Graphs provide a simple and flexible abstraction for modeling artifacts of different kinds in the form of nodes and edges. The graph database adopted in our work, Neo4j [6] supports automation and has a query language. The database is available for download for free and there is vast technical support and a large user base. Neo4j has also been ranked as the most popular graph database by db-engines.com [9].

The graph database in our work, Neo4j, organizes its data as a labelled property graph in which the nodes and relationships possess properties and can be annotated with labels. This allows augmenting the graph nodes and edges with semantics. The concept of properties is analogous to attributes in traditional conceptual models such as the Entity-Relationship Model. In property graphs, properties are key-value pairs. Figure 2 presents an example of data modeling using a property graph.

Some key points to note in Figure 2 are as follows: (1) the labels Person, Book, and Author are depicted in rectangles over the nodes, (2) a node can possess more than one label, and (3) both nodes and relationships may have attributes that are key-value pairs. The name of the relationship is reflected in bold font over the edges.

---

2 [https://neo4j.com/developer/graph-database/](https://neo4j.com/developer/graph-database/)
nodes with the same label form a group and this leads to an improvement in the query efficiency because a query involving labels limits the search space to the group of nodes or relationships defined by that label instead of searching through the complete graph \([6, 11]\). As an example, consider the label *Book* in Figure 8.2. In this case, when a query is specified to list all books, then only the nodes labelled as *Book* are traversed.

![Figure 8.2: An example of a property graph \([11]\)](image)

This concludes our brief discussion of the concepts that form the foundation of our research. In the next section, we present our approach and proof-of-concept for transforming relational and RDF databases into a property graph model.

### 8.2 Transformation Approach

We leverage the graph transformations proposed by Bouhali et al. \([36]\). Bouhali et al. focus on converting RDF data into a graph model whereas we envision an extensible approach that embraces model diversity by allowing multiple models such
Table 8.1: Relational schema excerpt. Top Employee and Bottom Organization

<table>
<thead>
<tr>
<th>Name</th>
<th>SkypeID</th>
<th>Gender</th>
<th>DOB</th>
<th>employeeID</th>
<th>address</th>
<th>ssn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marissa White</td>
<td>marissa yahoo.com</td>
<td>Female</td>
<td>04-091983</td>
<td>mwhite</td>
<td>2600 clifton</td>
<td>1234567890</td>
</tr>
<tr>
<td>Jason Doe</td>
<td>jason yahoo.com</td>
<td>Male</td>
<td>04091973</td>
<td>jdoe</td>
<td>3200 clifton</td>
<td>6789083455</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
<th>orgId</th>
</tr>
</thead>
<tbody>
<tr>
<td>University of Cincinnati</td>
<td>2600 Clifton</td>
<td>uc_org</td>
</tr>
</tbody>
</table>

as relational, RDF, and column-family stores from NoSQL databases, for example, all under one framework. Two issues arise: (1) transformation rules to map native model concepts to the property graph model, and (2) assurance that the individual concepts of native models do not get lost even after all the models are transformed into a graph representation. One of the unique characteristics of our proposed model is its native concept-preserving characteristic. This native concept-preserving characteristic is instrumental in facilitating reverse engineering when the graph representation would need to be expressed in the original model terminology. For example, to publish the data for use in a linked data project, RDF would be the model of choice. Thus, data that is represented using any other format would need to be transformed to the RDF model. Furthermore, in the scenario where the integrated, transformed graph includes information from multiple models, having knowledge about which nodes are originating from a particular model offers an independent view of the data models in use. This lays a foundation for model-specific data extraction or transformation.

We now present transformation rules and proof-of-concept of our graph model representation by considering schema excerpts for relational and RDF schemas. For convenience, we use the general term schema throughout the chapter to refer to both
schema and data when discussing models that have a close coupling of the two. For example, mapping a data source to a Neo4j graph database involves mapping structure and instances, but we refer to this activity as schema mapping. Similarly for merging two graph databases, we refer to this process as schema integration. We intend for the meaning to be clear from the context provided by the discussion and examples.

8.2.1 Relational model to a property graph

We begin by addressing the conversion of a relational schema into a property graph representation. Das et al. [13] have proposed a methodology for converting relational databases to RDF. Given that Bouhali et al. [36] have proposed an algorithm for transforming RDF to graph, it would appear that we can combine these two proposals [13, 36] to transform a relational database to a property graph. However, we do not follow this approach in our work for the following two reasons. First, the final property graph model obtained by Bouhali et al. [36] does not reflect the native model features. In their work, they focus on transforming RDF to a graph model and thus graph nodes implicitly correspond to the RDF schema. Our work applies to multiple models, not only RDF, and in anticipation of the need for reverse engineering from property graphs to native models, it is important to preserve the identity of native models in the graph representation. Second, developing transformations from the relational model to the property graph model offers a direct route to the target format (property graph in our case) instead of creating an RDF representation as an intermediate step. Table 8.1 represents a sample relational schema that we consider for illustrative purposes. The schema excerpt describes two entities, employee and organization, and the relationship
between them using a referential integrity constraint on the attribute `orgId`. We present three transformation rules as follows.

**Rule 1**: Every tuple in the relation is transformed to a node in the property graph. The node is labelled `RelationalResource` and defines a property in the form of name-value pair as type: *Name of the Relation*. The label serves to disambiguate the relational source from the other models that would also be transformed into a graph representation.

**Rule 2**: For each of the attributes in the relations, a property (name/value pair) is added to the corresponding node in the graph. This node would be the one that has the value for the property type equal to the name of the relation.

**Rule 3**: For each foreign key, a relationship is created between the nodes corresponding to the two participating relations.

The dashed rectangle on the left in Figure 8.3 illustrate the relational schema from Table 8.1 as a graph model in Neo4j based on the transformation rules above. The black box at the bottom left shows the properties (name-value pairs) for the employee, “Jason Doe” from the relational schema.

### 8.2.2 RDF model to a property graph

We now focus on the RDF schema. Figure 8.4 presents RDF data based on a schema excerpt from FOAF (Friend Of A Friend) [3] RDF vocabulary. An RDF vocabulary is an RDF schema formed of specific set of classes and properties that define resources of a particular domain. The example in Figure 8.4 describes two entities (*Person* and *Group*), the classes that are used to describe them (*foaf:Person* and *foaf:Group*) and the properties that relate them (*foaf:member*). The properties such as *foaf:name*
Figure 8.3: Excerpts of two heterogeneous schemas originally in different models (left: relational schema, right: RDF schema) unified under a common graph representation and foaf:homepage are applied to a Person entity and their values are either literals or resources described using URI (Uniform Resource Identifier). We show only one individual’s information in Figure 4 to save space, but the graph in Figure 8.3 shows two individual’s information (Marissa White and Jason Doe), their group memberships, and their organization. The black box at the top left shows the properties (key-value pairs) corresponding to the node identified by the name ucshuttletracker and of type Project in the RDF schema.

With the two input schemas transformed to a graph representation, the next natural question to ask is: What is the additional merit that the common graph representation offers compared to the knowledge that could have been derived from the native model representations? Figure 8.3 shows both the relational schema and RDF schema connected by mappings (Maps_With) in a graph model. This provides an insight into the question.
Figure 8.4: RDF schema excerpt

Figure 8.3 highlights how one can obtain more details for an employee if these two separate schemas can be integrated, compared to the information that we originally received from isolated sources. Relating Employee and Person nodes, we can identify his or her details such as name and gender and also information on the groups that he or she is a member of, or the homepage. Notice that the information on the homepage of a person is only captured by the RDF schema in our example. By unifying them based on common attributes such as date of birth or skypeId, an application can benefit from incorporating information from both schemas. This additional information may be harnessed by an organization to develop community-outreach programs based on employee outside interests, for example. Graph models represent a solution for depicting a connected environment. We employ the Neo4j Cypher query language to create mappings (Maps_With) between the appropriate nodes by comparing values of certain attributes. In Figure 8.3, we link the employee and person information coming from relational and RDF schemas, respectively, based on skypeId. In a general context, the
Figure 8.5: Placeholders $p, r,$ and $u$ for schema mapping between a column-oriented store and a property graph model [7]

example helps to illustrate a use case for leveraging a graph-based model towards a common representation scheme for model and schema diversity.

Apart from facilitating an integrated view, another benefit of our approach is that it preserves the native model concepts in the transformed graph model while providing a uniform representation at the same time. By augmenting nodes with the labels (such as RelationalResource and RDFResource), one can easily identify information
that was originally expressed in a particular native model. At the same time, bringing
the individual model and schema concepts under an umbrella of common terms (nodes
and relationships) facilitates linking and querying them using a single query language.

In a blog post, the Neo4j developer team present an approach to transform a
column-oriented data model to a property graph [7]. The goal is to allow loading
of data from a Cassandra data store into Neo4j. The mapping between the source
(column-oriented) and target (property graph) data models is taken as input from the
user and the resulting graph is created by loading the data from Cassandra to a CSV
file. Neo4j supports batch creation of a graph from CSV format. Figure 5 shows
a sample schema in Cassandra with \( p, r, \) and \( u \) as inputs from the user for schema
mapping. The label \( p \) stands for a property, \( r \) for a relationship, and \( u \) for specifying
unique constraint field. Since our work incorporates schemas originally expressed in
multiple heterogeneous models, we can incorporate the Cassandra to Neo4j mapping
by labelling the nodes as \textit{Cassandra Tables}. The approach presented in [7] focuses only
on the mapping between one set of source and target data models. In the next section,
we present the architecture of our approach.

8.3 Architecture

Our framework for transforming schemas to a graph-based format can be broken
down into three main modules:

1. \textit{The database module} holds schemas and exports a database to CSV format to
   support the automation step in the application module.

2. \textit{The application module} offers a presentation layer where a user can select the
   schema that needs to be transformed to Neo4j, and to allow transformation in
a systematic manner. The software implementation in this module employs our transformation rules.

3. *The graph module* uses the Neo4j browser to view the transformed schemas.

Figure 8.6 presents the architecture of our approach.

![Architecture of our proposed approach](image)

Figure 8.6: Architecture of our proposed approach

We use MySQL as the backend database for relational schemas. The process starts at the database layer which consists of three components: schemas, user-defined stored procedures and the MySQL native export tools. Schemas capture a built-in MySQL database (*information_schema*) and any user defined relational schemas. These user defined schemas are the artifacts that will be transformed to a graph model. The stored procedure reads metadata information from the *information_schema* and identifies all the foreign key relationships in our schema of interest. Figure 8.7 illustrates a query
in our stored procedure to capture all referential integrity constraints. The reason for collecting all the foreign keys in our database is that we need them to create relationships in our graph model based on the transformation rules from Section 3.

The user first exports the data in each of the relations in the database as a CSV file using MySQL native export data tool. We use the CSV file format since both the Neo4j community and our programming interface which uses Java support CSV files. Furthermore, Neo4j allows batch creation of a graph from CSV format.

```
select
    concat(table_name, '.', column_name) as 'foreign key',
    concat(referenced_table_name, '.', referenced_column_name) as 'references',
    constraint_name as 'constraint name'
from
    information_schema.key_column_usage
where
    referenced_table_name is not null
    and table_schema = 'sakila';
```

Figure 8.7: MySQL query to capture foreign key relationships in the MySQL sakila database [16]

At the application level we use Java to interact with the database and the graph modules programmatically. A database controller (DBController) manages connection to the database module and a graph management controller (GraphController) handles connection to Neo4j and submits queries to the graph interface through Java. These three application-level components working together along with our transformation rules from Section 3 facilitate automated transformation of a relational schema into a Neo4j property graph.

Figure 8.8 presents a code snippet that reads a relational schema exported to CSV format and converts it to Neo4j graph. The code focuses on generating nodes which
represent the concept of relations in a relational database. Each relation in the schema is exported to a CSV file with the same name as the relation itself. The code reads each of those CSV files and generates nodes with label `RelationalResource`. The Create Set clause in Figure 8.8 adds properties to those nodes. Each field in the CSV file represents one property.

We now have an understanding of the architecture and the input artifacts that are required for software implementation. Our approach is extensible and only requires transformation rules to be defined between any additional models and the property graph.

### 8.4 Evaluation

In this section, we discuss both the qualitative and quantitative analysis of our mapping approach. We leverage the evaluation metrics proposed by Buohali et al. [36] and also discuss qualitative merits of our proposal.

The quantitative evaluation metrics we consider are – **conciseness** and **connectivity** of the graph. Conciseness is given by the total number of nodes and relationships and can be used to calculate the graph size. Connectivity is calculated by dividing the
number of relationships with the total number of nodes. We apply these measures on the generated graph for an open source MySQL database, sakila, in Table 8.2.

Table 8.2: Evaluation metrics results for MySQL database - sakila [16]

<table>
<thead>
<tr>
<th>Total nodes</th>
<th>47273</th>
<th>Conciseness</th>
<th>62682</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total relationships</td>
<td>15409</td>
<td>Connectivity</td>
<td>0.32</td>
</tr>
</tbody>
</table>

Buohali et al. [36] state that for efficient processing over a graph, connectivity should be at least 1.5, which would signify strong connections in the graph. The connectivity value for our graph is quite low from their benchmark perspective. However, on further investigation, we identify why a low value may not always signify a non-desirable characteristic.

First, according to our transformation rules (Rule 3 in Section 3), the only relationship between two nodes that occurs in the target graph model comes from a foreign key relationship in the relational model. This sets the range for the number of relationships between two nodes for a particular constraint to be 0 to max($n_1$, $n_2$) where $n_1$ and $n_2$ correspond to the number of each of the two node types. Thus, based on our transformation rules, the number of relationship instances for a particular relationship type (in our case, a foreign key constraint) cannot exceed the number of nodes and hence the connectivity cannot exceed 1. The reason we have an even lower number is that some relations such as film_text are not even linked to other relations in the schema.

From this investigation, we come to the conclusion that strong connectivity between nodes in a graph certainly is good for processing but it also does not automatically lead to the conclusion that a lower number is not desirable. The two metrics of conciseness
and connectivity can also offer some ideas when we need to make a choice among multiple solutions. A graph with high connectivity is good for processing but if it comes at a price of increasing the graph size (less concise), then this would also lead to an increase in the cost of traversal because of increased path lengths. A larger graph size also implies higher storage requirements.
We evaluated this trade-off between conciseness and connectivity using an alternate mapping and the results are shown in Table 8.2. For the alternate mapping, we considered modeling attributes as nodes instead of properties (key-value pairs). This resulted in an increase in the number of nodes as well as relationships. The additional relationships come from the new edges created between attributes and entity nodes. We take a small example from sakila database to illustrate the two different mappings and their impact on conciseness and connectivity. Figure 8.9 shows two graph representations corresponding to two different mappings.

Figure 8.9a shows a property graph model where attributes are modelled as individual nodes. The conciseness of the graph (which is captured by total number of nodes and relationships) is 161 and connectivity equals 0.98. Our approach shown in Figure 8.9b shows how conciseness is increased based on our proposed set of mappings which model attributes as node properties and not as separate nodes. The connectivity does not show much difference and this is because of the nature of the native model and the types of relationships it exhibits. Edges in the graph are generated by foreign key constraints in the relational model.

Figure 8.9b represents the graph model based on our transformation rules from Section 3; Figure 8.9a captures an alternate mapping where emphasis is placed on increasing the graph connectivity. We modeled the attributes of a relation as separate nodes and created additional relationships between each of those attributes (actor_id, first_name, last_name, and last_update) and the relation node (actor_1). The node labelled actor_1 represents the data tuple from the actor relation that has actor_id equal to one. Similarly, the node film_actor represents the relation film_actor in the sakila database. The actor node has foreign key relationships with 19 film_actor nodes.
Based on the *sakila* database, this signifies that the particular actor has acted in 19 films.

Table 8.3 captures the evaluation metrics from both approaches.

<table>
<thead>
<tr>
<th>Evaluation criteria</th>
<th>Our approach</th>
<th>Alternate mapping (Figure 8.9a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total nodes</td>
<td>47273</td>
<td>62967</td>
</tr>
<tr>
<td>Total relationships</td>
<td>15409</td>
<td>66239</td>
</tr>
<tr>
<td>Conciseness</td>
<td>62682</td>
<td>129206</td>
</tr>
<tr>
<td>Connectivity</td>
<td>0.32</td>
<td>1.05</td>
</tr>
</tbody>
</table>

The results from Table 8.3 and the example from Figure 8.9 illustrate two key ideas: (1) the connectivity depends on the nature of original model, and (2) a higher connectivity may come at the cost of an increase in the graph size. Qualitatively, the merit of our proposal for schema mapping lies in the integration between multiple, heterogeneous models under a common graph framework. The *Maps_With* relationship as shown in Figure 8.3 creates many additional relationships which were not even present when the transformed graph models of relational and RDF schema were studied separately.

### 8.5 Schema integration over property graphs

In this section we propose a framework for schema integration over property graphs. We consider two input schemas expressed as a property graphs and define an algorithm to integrate the two graphs to generate a final integrated schema. The foundation for mapping heterogeneous models to a property graph is established in Section 4.

There have been significant research and industry implementations that offer solutions towards schema integration [106, 26, 32, 82, 25, 77, 31, 23]. Solutions exist in
the areas of specifying or semi-automatically identifying schema mappings which serve as a foundational step in schema integration. The contribution of our proposal lies in providing an infrastructure that leverages existing mapping algorithms but over a new modelling paradigm, the property graph.

We discuss our schema integration approach using two property graphs shown in Figure 8.10. The schemas in both the graphs capture information about entities student, faculty, courses, and department and their relationships. The two schemas correspond to two different departments: uc_ceas and uc_medicine shown as the topmost node in Figure 8.10.
While there are several commonalities as a result of the common domain, the schemas also exhibit differences creating challenges in schema integration. The differences and our proposed solutions for addressing each of them are as follows:

1. Relationship semantics: The property graph \textit{uc\_medicine} allows for a student to have one advisor, if he or she has one (it is optional) while the graph \textit{uc\_ceas} requires a student to have an advisor. A student may not be under a research program and thus may not even have an advisor according to the \textit{uc\_medicine} schema. We model these min/max constraints using properties \textit{(participation and cardinality)} over relationships in our property graphs. As a solution to address this scenario, in the final integrated schema, we impose a min constraint of 0 and max constraint of 1 for the \textit{has\_advisor} relationship. The idea behind picking this set of participation and cardinality constraints is that it leads to information preservation of the two native schemas. Following the \textit{uc\_medicine} schema in Figure 8.10, if a student does not have any advisor, our integrated schema would allow that while also allowing a student to have an advisor if that happens to be the case in the sample schemas.

2. Non-overlapping attributes in two similar entities across the two schemas: this addresses the scenario where an entity in one schema has certain attributes which are not present in the similar entity in the other schema. As an example, the \textit{Faculty Member} node in the \textit{uc\_ceas} schema does not store information about faculty’s \textit{start\_date} and \textit{department} while the \textit{uc\_medicine} schema does. This scenario can even be extended to entities and relationships such that one schema may be capturing additional information about a domain that is not covered in
the other schema. As an example, consider the entity ResearchCredits in the uc_ceas. It is not present in the schema for college of medicine. Our approach adds each of the unique attributes from each of the two schemas to the final integrated schema.

3. Differences in constraints: this difference may occur where the properties from two similar entities in the two schemas have different data type or uniqueness constraints [8]. A uniqueness constraint on a property ensures that no two nodes in the graph hold the same value for that property. As an example of the differences, studentId for a student entity in uc_ceas allows string values while the data type for the corresponding property Id in the uc_medicine only allows integers. Apart from the difference in data types, the uniqueness constraint for one schema may be composite (consisting of multiple properties) while the other schema may be defining a single-attribute constraint.

There are multiple ways to resolve this scenario. As an example, if the difference is in terms of data types, then the data type with a wider range of values (String over integer) can be considered for the final integrated schema. However, consider another scenario where the difference is also in terms of number of attributes representing the uniqueness constraint. One schema may have the constraint defined on a set of attributes (composite) while the other schema uses a single attribute constraint. Considering the possibilities of multiple ways in which differences in the constraints can manifest, we consider the idea of adopting a surrogate key in the final merged schema. The original constraints on the
attributes from each of the two schemas will also be copied to the merged schema to prevent any information loss.

4. Difference in field names/entities: entities or attributes across the two schemas may pertain to the same concept but use different names and terminologies. In the example property graphs, the terms *lname* and *lastname* for *Student* node use different terms for the same concept.

The literature offers numerous solutions for identifying and resolving such conflicts using lexicon, ontology, or string algorithms [103, 42, 54, 45].

The final integrated schema addressing the four heterogeneity scenarios above is obtained using the algorithm described in the next section.

**8.6 Algorithm**

The core algorithm can be summarized as follows. Start with one input graph as the base graph. Merge the second into the graph based on a likelihood match for each new node against the base graph. The match between two nodes is determined based on the four solutions described in Section 6, algorithms from the literature and a user-defined threshold value. If there is no good match, the node is not merged but added as a new node to the integrated graph. Figures 11-13 presents our algorithm as three main modules - *determineNodeTypesForMergedSchema*, *mergeNodes*, and *mergeRelationships*.

The input schemas modelled using property graphs (*G_1* and *G_2*) are represented using a four-element tuple. The four elements are sets of nodes (*N*), edges (*E*), node-labels (*NL*), and edge-labels (*EL*) in the graph. The notation *NL(G)* and *NL_1(G_1)*
Figure 8.11: Module for unifying the node-types from input schemas

refers to node-labels in the property graph $G$ and $G_1$ respectively. The output, merged schema is represented as $(N, E, NL, EL)$ where each of the set elements is the union of the corresponding elements from the input graphs. The set of nodes $N$ in the merged schema is the union of $N_1(G_1)$ and $N_2(G_2)$. The merged schema is initialized as empty.

The algorithm consists of three main parts: (a) capturing all unique node types from all the input schemas into the final integrated schema, (b) union of nodes $N_1$ and $N_2$, and (c) union of relationships $E_1$ and $E_2$. 
The algorithm proceeds by first identifying the unique node-types across all the input schemas. Each node-type can be considered as an artifact/entity holding a certain set of properties modelled using key-value pairs. Figure 8.11 shows this module. In our example for schema integration here, we use node labels to capture the node type. The module (Figure 8.11) copies all the labels from the first input schema to the integrated schema (Step 1.1). Step 1.2 then iterates over each of the labels in the second schema,
$G_2$ to compare it against all the labels in the merged schema so far. If a match is found that meets a threshold value, then the node is merged with the matched node. The mapping between the matched node from the partial merged schema and the new node is also stored (Step 1.2.2).

The algorithm uses the `isMatch` function which takes four arguments: (a) the node, $l$, to be searched, (b) set of node labels $NL(G)$ in the merged schema, and (c) a threshold value in the range 0 and 1 for matching, and (d) a boolean argument to signify the type of return value. If the fourth argument is true (Step 1.2.1), it returns the matching node, and if false, a boolean value is returned. This return boolean value signifies if the label $l$ exists in the set of partial merged schema labels or not. If no match is found indicating a new node-type, then the label is added to the set of labels of merged schema (else block in step 1.2.2). The notation $NL(G)$ represents set of node-labels for merged schema graph $G$. However, if the node label already exists, then the mapping is stored in a data structure `map_labels` (Step 1.2.2). This function `isMatch` can be customized by applying different schema matching algorithms from the literature.

To understand the rationale behind storing this mapping, refer to our sample input schemas in Figure 8.10. We have nodes (`Assistant Professor` and `Lisa`) labelled `Faculty` and `Faculty member`. The labels are representing the same entity but using different terms. The final integrated schema consolidates them into a single label `Faculty`. This mapping information is now important to merge nodes of type `Faculty member` in the `uc_medicine` into nodes with type `Faculty` in the merged schema.

The next steps involve adding nodes and relationships in the final integrated schema. Figures 8.12 and 8.13 show the modules addressing this functionality.
After identifying the node-types for the integrated graph, the algorithm iterates through every node in the input graphs. It compares node-type (label) with the set of labels in the integrated schema (Step 2.1.1). If an exact match is not found (for example, *Faculty* and *Faculty Member* node-types in Figure 8.10), the closest mapping is found between the current node’s label and the set of labels in the integrated schema (step 2.1.1.a in the else block). A new node is then created with the mapped node-type. The notation $N(G)$ refers to set of nodes in the merged property graph $G$. Similarly, the algorithm iterates through each node in the next input graph. If the current node’s label matches with one of the node’s labels in the partial integrated schema, the functions *mapAttributesForEntity* and *addNonOverlappingAttributes* are invoked.

The final module involves creates edges between nodes in the partial integrated schema. Figure 8.13 shows this module. The source and target nodes for each relationship in the input graphs is read(Steps 3.1.1 and 3.1.2) and the corresponding nodes

---

Figure 8.13: Module for unifying relationships from input schemas

```plaintext
3. function mergeRelationships(G1, G2)
  3.1 for every relationship $r$ in $G1$ and $G2$
     3.1.1 Get source node as source
     3.1.2 Get target node as target
     3.1.3 Find the corresponding nodes for source and target nodes in the set of nodes in the merged schema
       mapped_source = map_node(source, G(N))
       mapped_target = map_node(target, G(N))
     3.1.4 Create a relationship $R$ in the merged schema with
       source_node(R) = mapped_source, and
       target_node(R) = mapped_target, and
       properties_{ms}(R) = properties(r)
     3.1.5 Add(G(E), r)
  end for
end function
```
in the partial integrated schema are identified (Step 3.1.3). The relationship is then created between the nodes resulting in the final integrated schema (Step 3.1.5).

Figure 8.14 shows the integrated schema obtained using our algorithm for the sample input property graphs (Figure 8.10). Some points to note are as follows.

1. The figure shows the integrated schema for the Student, Faculty, Course and ResearchCredits node types and one relationship has_advisor. Note the surrogate
key custom_id in the Student label. The native primary keys of the individual schemas uc_ceas and uc_medicine are also preserved. Further, note that the schema uc_medicine originally employed the term FacultyMember instead of Faculty. Using string matching algorithms, we consolidated these two labels into one as Faculty.

2. The participation and cardinality constraints for has_advisor is 0 and 1, respectively, in the final integrated schema. The original min/max constraints in the native schemas were (0,1) and (1,1), respectively.

The framework provided here merges two property graphs, which can be mapped results from heterogeneous source models. The basic features we address for matching nodes and merging them, for example, can be extended with more sophisticated and powerful techniques from the literature. We provide a modular proof-of-concept for graph merging.

8.7 Evaluation

In this section, we discuss two case studies to evaluate the effectiveness and coverage of our schema integration algorithm [51, 112]. The first case study refers to a schema integration example by Petermann et al. [98, 97]. The authors provide data models for two heterogeneous systems (enterprise resource planning and customer issue tracking) of a food trading company, and they further employ the models to demonstrate the effectiveness of their graph-based data integration approach [98, 97]. We adopt their data sources to test the effectiveness of our approach and compare our integrated schema with their result [98].
For the second case study, we use the schemas modelled by Batini et al. [23]. Through these case study we discuss the features covered by our integrated schema. Our focus is on providing a framework to illustrate schema integration over property graphs that can be further extended and optimized.

### 8.7.1 Example 1

We consider the FoodBroker data source presented by Petermann et al. [97]. The model captures two schemas as shown in Figure 8.15. The result obtained through

---

**Figure 8.15:** **FoodBroker** – Enterprise Resource Planning and Customer Issue Tracking schemas [97]
Figure 8.16: Integrated schema obtained through our approach (Example 1)

our algorithm (Figure 8.16) results in a schema similar to theirs [97]. The Employee and User nodes are merged along with their attributes into one node Employee. Similarly, Customer and Client entities from the two schemas are consolidated into the Customer entity in our integrated schema. In order to highlight schema integration, we constrained sample data in our graph to one instance for each node type. The cardinality constraints (Figure 8.15) such as one SalesInvoice can be created for multiple instances of SalesOrder are captured using properties on the relationships in the
Figure 8.17: *Book* and *Publication* schemas [23]

graph. Once the interschema relationships are determined, the properties of the similar entities are merged.

The added advantage offered by our approach is its ability to handle model diversity. If the native input schemas are in heterogeneous models, the labels of the nodes in their corresponding property graphs (Section 3) can be used to capture the data source or model. This allows the schema administrator to preserve native model information while gaining the benefits of collective information as well that is obtained from the integrated schema.
8.7.2 Example 2

In this example, we consider schema examples modelled by Batini et al. [23]. The input schemas, *Book* and *Publication*, are shown in Figure 8.17. The integrated schema obtained through our algorithm is shown in Figure 8.18. Our integrated schema creates the same entities (*Publisher, Book, University, and Topics*). The main features of our schema are as follows:

1. We also capture the the participation and cardinality constraints in our integrated schema (Figure 8.18). The *Book* schema (Figure 16) originally shows a
1:1 relationship between the entities Book and Topics while Publication schema (Figure 16) exhibits a 1:m relationship between similar entities Publication and Keywords. In our integrated schema, we resolve this conflict in the cardinalities by modeling the relationship as a 1:m. Batini et al. [23] present and discuss the final integrated schema using a conceptual model (ER). Our approach, based on property graphs, addresses the integration challenge from a graph perspective.

2. We observe that our integrated schema created an additional attribute Publisher in the Book entity. At this point, our algorithm does not capture the schema conflicts that can arise when some information is modelled as an attribute in one schema and as a relationship in another schema. The information about a book publisher is modelled as a relationship in the Book schema and an attribute in the Publication schema (Figure 8.17). Batini et al. [23] identify and cover this conflict. Our framework can be extended to address this kind of schema heterogeneity.

3. Batini et al. [23] also show Book entity as a subset of the Publication entity, considering that publication can also include journals in addition to books. Our work does not yet address identifying and modelling subset relationships. Again, extensions based on the solutions provided in the literature can be incorporated into our framework.

There are numerous opportunities for extending our work to incorporate modules that address a wide variety of heterogeneous features. The main contribution of our work is in providing a framework that employs property graphs as the representation
model. Using graph databases allows addressing schema and data integration using one modeling paradigm.

### 8.8 Related work

Data integration and exchange has received significant research attention by both academia and industry practitioners for more than two decades. One of the frequent approaches is defining an intermediate, canonical model that can capture the commonalities and differences of individual, heterogeneous models, thus providing a uniform representation [? , 38, 20, 30, 21, 39, 98]. Relational, XML, and RDF represent some examples of data models that have been considered for this purpose. We identify two key points that may be raised toward our choice of graph model and discuss each of them below. A discussion of these points highlights the rationale and novelty of our solution.

- **Semantic web technologies facilitate integration by establishing links**

Our idea of employing a property graph model comes from recognizing the data management revolution brought on by big data. In terms of databases, a new class of storage models have emerged called NoSQL databases and graph databases represent one of the categories of the NoSQL family. In this context, we speculate that it would be useful to have frameworks that would allow transformation of different data formats into a model that is amenable to the big data management challenges and our approach represents an effort in this direction. We recognize the immense potential offered by the semantic web research community towards facilitating integration [62, 70, 69, 34]. RDF model based on subject-object-predicate framework represents the de-facto standard in the linked data and semantic web community and it already leverages a graph model. However, the choice about selecting one model over the other also depends on
the problem and domain at hand [36, 114, 62]. Our vision is to offer an interoperable and integration framework in such a way that it not only facilitates integration of heterogeneous modeling concepts in a flexible and extensible manner, but is also native concept-preserving and aligns with the NoSQL family of data models. Our graph model, as a property graph addresses these requirements.

Furthermore, Bouhali et al. [36] have highlighted potential performance gains in executing large-scale queries over NoSQL graph databases as compared to RDF engines. RDF data needs to be loaded into a SPARQL engine for efficient query performance and authors cite that while contributions have been made towards optimizing query execution in SPARQL but there still remains scope for further improvement in terms of matching up with the performance offered by graph databases for some large-scale queries. Vasilyeva et al. [114] have also compared a graph model with an RDF store and they conclude that since RDF stores both data and metadata using the same format, it requires RDFS and OWL to distinguish the schema from the actual data.

• What is the advantage of graph based approach over dominant and successful models such as the relational model?

In comparison to relational databases, data modeling using graphs offers performance gains in processing interconnected data. This is achieved by avoiding the join operation required in the relational model [?, 118]. Furthermore, the relational model requires a schema to be defined whereas graph models are flexible [91]. Our work towards schema mapping (Section 3) closely aligns with Buohali et al. [36]. Buohali et al. [36] consider translation from RDF to property graphs only. Our application has a broader scope. We build upon their work and extend the scope by making the approach flexible to allow incorporation of additional models.
In terms of schema integration over graph databases, Petermann et al. [98, 97] present a system for graph integration and analytics. The system is based on a property graph model and provides three types of graphs: unified metadata graph (UMG), integrated instance graph (IIG) and business transaction graph (BTG). For generating the metadata graph, their approach extracts the schema of objects to translate it into the property graph model. While our contribution on schema integration is closely related to the focus of their work (generating metadata graph for integration), we also include preservation of native model concepts while handling schema mapping. In case of the need for integration over schemas originally expressed in heterogeneous models, our mapping approach (Section 3) supports annotation of nodes with labels that define the native data model of the schema elements.

The motivation for employing graph based approach also comes from the fact that graph databases belong to the family of NoSQL data stores. Thus, our framework and algorithm for schema mapping and integration over property graphs lay a foundation for integration of schema-based and schema-less data stores.

8.9 Conclusion and future work

We advocate the idea of employing graph databases as a means of bridging the gap between schema-based and schema-less data stores. Our initial results for schema mapping present a proof-of-concept by illustrating transformation of relational and RDF schemas as the first step. We believe that our approach lays a foundation for addressing the variety aspect of big data and bringing traditional data into a big data environment. The second contribution of our work lies in presenting a schema merging algorithm over property graphs. In this chapter, we present a proof-of-concept to
illustrate the proposed algorithm. Our approach offers a framework that can be further optimized and it is flexible to incorporate additional schema integration scenarios.

We have translated some traditional models to a property graph. We envision extending our work by incorporating additional data stores and illustrating integration. Once we have that achieved that we can incorporate an evaluation study of the transformation process to address the efficiency of the approach. A performance study of querying an integrated graph schema versus disconnected original native schemas is another research direction. The idea of reverse engineering the graph model to obtain the schemas in the original models can also be useful [36, ?] to leverage tools from the native data environments.
Chapter 9: Conclusion and Future Work

In our work, we have made three contributions towards the use of graph databases in addressing schema management challenges: mapping, integration and evolution.

In the context of schema evolution in a warehouse environment, we developed a provenance-based approach for providing an assessment of the impact of schema level changes to the related schemas and mappings in a data warehouse environment. The approach is a unified solution to the problem of schema evolution in warehouses because of its ability to handle changes at multiple artifacts under one framework. Our work focuses on identifies the impact syntactically. The approach can be extended to incorporate algorithms that can determine semantic and performance-based impact as well. A change in the column values (units) representing currency information would be harder to detect syntactically unless some information beyond data type for the column is provided. This corresponds to the impact assessment of a semantic level change. Similarly, the impact of making a schema change and having an approach which can identify its impact on the query plans is an example of performance-based impact assessment. Furthermore, identifying impact of compound schema changes would be another research direction to pursue.

For schema mapping and interoperability, we advocate the idea of employing graph databases as a means of bridging the gap between schema-based and schema-less data
stores. Our initial results for schema mapping present a proof-of-concept by illustrating transformation of relational and RDF schemas as the first step. We believe that our approach lays a foundation for addressing the variety aspect of big data and bringing traditional data into a big data environment. Our approach offers a framework that can be further optimized and it is extensible to incorporate additional data stores. The idea of reverse engineering the graph model to obtain the schemas in the original models can also be useful to leverage tools from the native data environments.

Next, we have presented a schema merging algorithm over property graphs. In this paper, we present a proof-of-concept to illustrate the proposed algorithm. Our approach offers a framework that can be further optimized and it is extensible to incorporate additional schema integration scenarios. We have translated some traditional models to a property graph. We envision extending our work by incorporating additional data stores and illustrating integration. A performance study of querying an integrated graph schema versus disconnected original native schemas is another research direction.
Appendix A: Input artifacts

This appendix includes the schema and instance generation scripts developed in MySQL for each of the four input schemas in our work, SQL queries and ETL workflows used in the case study in Chapter 6.

A.1 Input schemas - Operational, Reconciled and Warehouse Schema

This section focuses on the schema and instance generation scripts for each of the four input schemas in our work.

A.1.1 Schema generation script for operational schema - *sakila_op1*

```
-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';

-- Schema mydb

-- Schema sakila_op1

-- Schema sakila_op1
```
CREATE SCHEMA IF NOT EXISTS 'sakila_op1' DEFAULT CHARACTER SET utf8 ;
USE 'sakila_op1' ;

-- Table 'sakila_op1'. 'actor'
CREATE TABLE IF NOT EXISTS 'sakila_op1'. 'actor' ( 
    'actor_id' SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
    'fname' VARCHAR(45) NOT NULL,
    'lname' VARCHAR(45) NOT NULL,
    'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY ('actor_id'),
    INDEX 'idx_actor_last_name' ('lname' ASC)
) ENGINE = InnoDB
AUTO_INCREMENT = 4
DEFAULT CHARACTER SET = utf8;

-- Table 'sakila_op1'. 'address'
CREATE TABLE IF NOT EXISTS 'sakila_op1'. 'address' ( 
    'address_id' SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
    'street_address' VARCHAR(50) NOT NULL,
    'city' VARCHAR(50) NOT NULL,
    'country' VARCHAR(25) NOT NULL,
    'postal_code' VARCHAR(10) NOT NULL,
    'phone' VARCHAR(20) NOT NULL,
    'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY ('address_id')
) ENGINE = InnoDB
AUTO_INCREMENT = 8
DEFAULT CHARACTER SET = utf8;

-- Table 'sakila_op1'. 'category'
CREATE TABLE IF NOT EXISTS 'sakila_op1'. 'category' ( 
    'category_id' TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,
    'name' VARCHAR(25) NOT NULL,
    'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
)
ON UPDATE CURRENT_TIMESTAMP,
PRIMARY KEY ('category_id'))
ENGINE = InnoDB
AUTO_INCREMENT = 15
DEFAULT CHARACTER SET = utf8;

-- Table `sakila_op1`.`staff`
CREATE TABLE IF NOT EXISTS `sakila_op1`.`staff`
(
    `staff_id` TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,
    `first_name` VARCHAR(45) NOT NULL,
    `last_name` VARCHAR(45) NOT NULL,
    `address_id` SMALLINT(5) UNSIGNED NOT NULL,
    `picture` BLOB NULL DEFAULT NULL,
    `email` VARCHAR(50) NULL DEFAULT NULL,
    `store_id` TINYINT(3) UNSIGNED NOT NULL,
    `active` TINYINT(1) NOT NULL DEFAULT '1',
    `username` VARCHAR(16) NOT NULL,
    `password` VARCHAR(40) CHARACTER SET 'utf8' COLLATE 'utf8_bin' NULL DEFAULT NULL,
    `last_update` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
ON UPDATE CURRENT_TIMESTAMP,
PRIMARY KEY (`staff_id`),
INDEX `idx_fk_store_id` (`store_id` ASC),
INDEX `idx_fk_address_id` (`address_id` ASC),
CONSTRAINT `fk_staff_address`
FOREIGN KEY (`address_id`)
REFERENCES `sakila_op1`.`address` (`address_id`)
ON UPDATE CASCADE,
CONSTRAINT `fk_staff_store`
FOREIGN KEY (`store_id`)
REFERENCES `sakila_op1`.`store` (`store_id`)
ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 3
DEFAULT CHARACTER SET = utf8;

-- Table `sakila_op1`.`store`
CREATE TABLE IF NOT EXISTS `sakila_op1`.`store`
(
    `store_id` TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,
    `store_id` TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,
CREATE TABLE IF NOT EXISTS `sakila_op1`.`customer` (  
    `customer_id` SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,  
    `store_id` TINYINT(3) UNSIGNED NOT NULL,  
    `first_name` VARCHAR(45) NOT NULL,  
    `last_name` VARCHAR(45) NOT NULL,  
    `email` VARCHAR(50) NULL DEFAULT NULL,  
    `address_id` SMALLINT(5) UNSIGNED NOT NULL,  
    `active` TINYINT(1) NOT NULL DEFAULT '1',  
    `create_date` DATETIME NOT NULL,  
    `last_update` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP  
    ON UPDATE CURRENT_TIMESTAMP,  
    PRIMARY KEY (`customer_id`),  
    INDEX `idx_fk_store_id` (`store_id` ASC),  
    INDEX `idx_fk_address_id` (`address_id` ASC),  
    INDEX `idx_last_name` (`last_name` ASC),  
    CONSTRAINT `fk_customer_address`  
    FOREIGN KEY (`address_id`)  
    REFERENCES `sakila_op1`.`address` (`address_id`)  
    ON UPDATE CASCADE,  
    CONSTRAINT `fk_customer_store`  
    FOREIGN KEY (`store_id`)  
    REFERENCES `sakila_op1`.`store` (`store_id`)  
    ON UPDATE CASCADE)
```sql
ON UPDATE CASCADE
ENGINE = InnoDB
AUTO_INCREMENT = 4
DEFAULT CHARACTER SET = utf8;

-- Table 'sakila_op1'. 'language'
CREATE TABLE IF NOT EXISTS 'sakila_op1'. 'language' (  
  'language_id' TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,
  'name' CHAR(20) NOT NULL,
  'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP  
  ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY ('language_id'))
ENGINE = InnoDB
AUTO_INCREMENT = 3
DEFAULT CHARACTER SET = utf8;

-- Table 'sakila_op1'. 'film'
CREATE TABLE IF NOT EXISTS 'sakila_op1'. 'film' (  
  'film_id' SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
  'title' VARCHAR(255) NOT NULL,
  'description' TEXT NULL DEFAULT NULL,
  'release_year' YEAR NULL DEFAULT NULL,
  'language_id' TINYINT(3) UNSIGNED NOT NULL,
  'original_language_id' TINYINT(3) UNSIGNED NULL DEFAULT NULL,
  'rental_duration' TINYINT(3) UNSIGNED NOT NULL DEFAULT '3',
  'rental_rate' DECIMAL(4,2) NOT NULL DEFAULT '4.99',
  'length' SMALLINT(5) UNSIGNED NULL DEFAULT NULL,
  'replacement_cost' DECIMAL(5,2) NOT NULL DEFAULT '19.99',
  'rating' ENUM('G','PG','PG-13','R','NC-17') NULL DEFAULT 'G',
  'special_features' SET('Trailers','Commentaries','Deleted Scenes','Behind the Scenes')  
  NULL DEFAULT NULL,
  'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP  
  ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY ('film_id'),
  INDEX 'idx_title' ('title' ASC),
  INDEX 'idx_fk_language_id' ('language_id' ASC),
  INDEX 'idx_fk_original_language_id' ('original_language_id' ASC),
  CONSTRAINT 'fk_film_language'  
  FOREIGN KEY ('language_id')
);```
REFERENCES 'sakila_op1'.'language' ('language_id')
ON UPDATE CASCADE,
CONSTRAINT 'fk_film_language_original'
FOREIGN KEY ('original_language_id')
REFERENCES 'sakila_op1'.'language' ('language_id')
ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 968
DEFAULT CHARACTER SET = utf8;

-- Table 'sakila_op1'.'film_actor'
CREATE TABLE IF NOT EXISTS 'sakila_op1'.'film_actor' (
  'actor_id' SMALLINT(5) UNSIGNED NOT NULL,
  'film_id' SMALLINT(5) UNSIGNED NOT NULL,
  'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
ON UPDATE CURRENT_TIMESTAMP,
PRIMARY KEY ('actor_id', 'film_id'),
INDEX 'idx_fk_film_id' ('film_id' ASC),
CONSTRAINT 'fk_film_actor_actor'
FOREIGN KEY ('actor_id')
REFERENCES 'sakila_op1'.'actor' ('actor_id')
ON UPDATE CASCADE,
CONSTRAINT 'fk_film_actor_film'
FOREIGN KEY ('film_id')
REFERENCES 'sakila_op1'.'film' ('film_id')
ON UPDATE CASCADE)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;

-- Table 'sakila_op1'.'film_category'
CREATE TABLE IF NOT EXISTS 'sakila_op1'.'film_category' (
  'film_id' SMALLINT(5) UNSIGNED NOT NULL,
  'category_id' TINYINT(3) UNSIGNED NOT NULL,
  'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
ON UPDATE CURRENT_TIMESTAMP,
PRIMARY KEY ('film_id', 'category_id'),
INDEX 'fk_film_category' ('category_id' ASC),
CONSTRAINT 'fk_film_category_category'
FOREIGN KEY ('category_id')
REFERENCES `sakila_op1`.`category` (`category_id`)
ON UPDATE CASCADE,
CONSTRAINT `fk_film_category_film`
FOREIGN KEY (`film_id`)
REFERENCES `sakila_op1`.`film` (`film_id`)
ON UPDATE CASCADE)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;

-- -----------------------------------------------------
-- Table `sakila_op1`.`inventory`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `sakila_op1`.`inventory` (
`inventory_id` MEDIUMINT(8) UNSIGNED NOT NULL AUTO_INCREMENT,
`film_id` SMALLINT(5) UNSIGNED NOT NULL,
`store_id` TINYINT(3) UNSIGNED NOT NULL,
`last_update` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
ON UPDATE CURRENT_TIMESTAMP,
PRIMARY KEY (`inventory_id`),
INDEX `idx_fk_film_id` (`film_id` ASC),
INDEX `idx_store_id_film_id` (`store_id` ASC, `film_id` ASC),
CONSTRAINT `fk_inventory_film`
FOREIGN KEY (`film_id`)
REFERENCES `sakila_op1`.`film` (`film_id`)
ON UPDATE CASCADE,
CONSTRAINT `fk_inventory_store`
FOREIGN KEY (`store_id`)
REFERENCES `sakila_op1`.`store` (`store_id`)
ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 4423
DEFAULT CHARACTER SET = utf8;

-- -----------------------------------------------------
-- Table `sakila_op1`.`rental`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `sakila_op1`.`rental` (
`rental_id` INT(11) NOT NULL AUTO_INCREMENT,
`rental_date` DATETIME NOT NULL,
`inventory_id` MEDIUMINT(8) UNSIGNED NOT NULL,
`customer_id` SMALLINT(5) UNSIGNED NOT NULL,
`return_date` DATETIME NULL DEFAULT NULL,
`staff_id` TINYINT(3) UNSIGNED NOT NULL,
CREATE TABLE IF NOT EXISTS 'sakila_op1'.'payment' (
    'payment_id' SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
    'customer_id' SMALLINT(5) UNSIGNED NOT NULL,
    'staff_id' TINYINT(3) UNSIGNED NOT NULL,
    'rental_id' INT(11) NULL DEFAULT NULL,
    'amount' DECIMAL(5,2) NOT NULL,
    'payment_date' DATETIME NOT NULL,
    'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY ('payment_id'),
    INDEX 'idx_fk_staff_id' ('staff_id' ASC),
    INDEX 'idx_fk_customer_id' ('customer_id' ASC),
    INDEX 'fk_payment_rental' ('rental_id' ASC),
    CONSTRAINT 'fk_payment_customer'
    FOREIGN KEY ('customer_id')
    REFERENCES 'sakila_op1'.'customer' ('customer_id')
    ON UPDATE CASCADE,
    CONSTRAINT 'fk_payment_rental'
    FOREIGN KEY ('rental_id')
    REFERENCES 'sakila_op1'.'rental' ('rental_id')
    ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 11257
DEFAULT CHARACTER SET = utf8;
FOREIGN KEY ('rental_id')
    REFERENCES 'sakila_op1'. 'rental' ('rental_id')
    ON DELETE SET NULL
    ON UPDATE CASCADE,
CONSTRAINT 'fk_payment_staff'
    FOREIGN KEY ('staff_id')
    REFERENCES 'sakila_op1'. 'staff' ('staff_id')
    ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 54
DEFAULT CHARACTER SET = utf8;

SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;

A.1.2 Schema generation script for operational schema - sakila_op2

-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';

-- -----------------------------------------------------
-- Schema mydb
-- -----------------------------------------------------
-- -----------------------------------------------------
-- Schema sakila_op2
-- -----------------------------------------------------

CREATE SCHEMA IF NOT EXISTS 'sakila_op2' DEFAULT CHARACTER SET utf8 ;
USE 'sakila_op2' ;

-- -----------------------------------------------------
-- Table 'sakila_op2'. 'actor'
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS 'sakila_op2'.'actor' (
    'actor_id' SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
    'fname' VARCHAR(45) NOT NULL,
    'lname' VARCHAR(45) NOT NULL,
    'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY ('actor_id'),
    INDEX 'idx_actor_last_name' ('lname' ASC))
ENGINE = InnoDB
AUTO_INCREMENT = 33
DEFAULT CHARACTER SET = utf8;

-- Table 'sakila_op2'.'address'
CREATE TABLE IF NOT EXISTS 'sakila_op2'.'address' (
    'address_id' SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
    'address' VARCHAR(50) NOT NULL,
    'city' VARCHAR(50) NOT NULL,
    'country' VARCHAR(25) NOT NULL,
    'postal_code' VARCHAR(10) NULL DEFAULT NULL,
    'phone' VARCHAR(20) NOT NULL,
    'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY ('address_id'))
ENGINE = InnoDB
AUTO_INCREMENT = 11
DEFAULT CHARACTER SET = utf8;

-- Table 'sakila_op2'.'category'
CREATE TABLE IF NOT EXISTS 'sakila_op2'.'category' (
    'category_id' TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,
    'name' VARCHAR(25) NOT NULL,
    'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY ('category_id'))
ENGINE = InnoDB
AUTO_INCREMENT = 17
DEFAULT CHARACTER SET = utf8;
CREATE TABLE IF NOT EXISTS `sakila_op2`.`customer` (  
    `customer_id` SMALLINT(5)_UNSIGNED NOT NULL AUTO_INCREMENT,  
    `first_name` VARCHAR(45) NOT NULL,  
    `last_name` VARCHAR(45) NOT NULL,  
    `email` VARCHAR(50) NULL DEFAULT NULL,  
    `address_id` SMALLINT(5) UNSIGNED NOT NULL,  
    `active` TINYINT(1) NOT NULL DEFAULT '1',  
    `create_date` DATETIME NOT NULL,  
    `last_update` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP  
    ON UPDATE CURRENT_TIMESTAMP,  
    PRIMARY KEY (`customer_id`),  
    INDEX `idx_fk_address_id` (`address_id` ASC),  
    INDEX `idx_last_name` (`last_name` ASC),  
    CONSTRAINT `fk_customer_address`  
    FOREIGN KEY (`address_id`)  
    REFERENCES `sakila_op2`.`address` (`address_id`)  
    ON UPDATE CASCADE)  
ENGINE = InnoDB  
AUTO_INCREMENT = 7  
DEFAULT CHARACTER SET = utf8;

CREATE TABLE IF NOT EXISTS `sakila_op2`.`film` (  
    `film_id` SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,  
    `title` VARCHAR(255) NOT NULL,  
    `description` TEXT NULL DEFAULT NULL,  
    `release_year` YEAR NULL DEFAULT NULL,  
    `language` VARCHAR(25) NOT NULL,  
    `rental_duration` TINYINT(3) UNSIGNED NOT NULL DEFAULT '3',  
    `rental_rate` DECIMAL(4,2) NOT NULL DEFAULT '4.99',  
    `length` SMALLINT(5) UNSIGNED NOT NULL DEFAULT NULL,  
    `replacement_cost` DECIMAL(5,2) NOT NULL DEFAULT '19.99',  
    `rating` ENUM('G','PG','PG-13','R','NC-17') NULL DEFAULT 'G',  
    `special_features` SET('Trailers','Commentaries','Deleted Scenes','Behind the Scenes') NULL DEFAULT NULL,  
    `last_update` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP  
    ON UPDATE CURRENT_TIMESTAMP,  
    PRIMARY KEY (`film_id`),  
    INDEX `idx_title` (`title` ASC))
```sql
ENGINE = InnoDB
AUTO_INCREMENT = 974
DEFAULT CHARACTER SET = utf8;

-- Table 'sakila_op2'.film_actor
CREATE TABLE IF NOT EXISTS 'sakila_op2'.film_actor (  
  actor_id SMALLINT(5) UNSIGNED NOT NULL,
  film_id SMALLINT(5) UNSIGNED NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP  
  ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (actor_id, film_id),
  INDEX idx_fk_film_id (film_id ASC),
  CONSTRAINT fk_film_actor_actor  
    FOREIGN KEY (actor_id)
    REFERENCES sakila_op2.actor (actor_id)  
    ON UPDATE CASCADE,
  CONSTRAINT fk_film_actor_film  
    FOREIGN KEY (film_id)
    REFERENCES sakila_op2.film (film_id)  
    ON UPDATE CASCADE)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;

-- Table 'sakila_op2'.film_category
CREATE TABLE IF NOT EXISTS 'sakila_op2'.film_category (  
  film_id SMALLINT(5) UNSIGNED NOT NULL,
  category_id TINYINT(3) UNSIGNED NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP  
  ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (film_id, category_id),
  INDEX fk_film_category_category (category_id ASC),
  CONSTRAINT fk_film_category_actor  
    FOREIGN KEY (category_id)
    REFERENCES sakila_op2.category (category_id)  
    ON UPDATE CASCADE,
  CONSTRAINT fk_film_category_film  
    FOREIGN KEY (film_id)
    REFERENCES sakila_op2.film (film_id)  
    ON UPDATE CASCADE)
```
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;

-- -----------------------------
-- Table 'sakila_op2'. 'staff'
-- -----------------------------
CREATE TABLE IF NOT EXISTS 'sakila_op2'. 'staff' (  'staff_id' TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,  'first_name' VARCHAR(45) NOT NULL,  'last_name' VARCHAR(45) NOT NULL,  'address_id' SMALLINT(5) UNSIGNED NOT NULL,  'picture' BLOB NULL DEFAULT NULL,  'email' VARCHAR(50) NULL DEFAULT NULL,  'active' TINYINT(1) NOT NULL DEFAULT '1',  'username' VARCHAR(16) NOT NULL,  'password' VARCHAR(40) CHARACTER SET utf8 COLLATE utf8_bin NULL DEFAULT NULL,  'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,  PRIMARY KEY ('staff_id'),  INDEX 'idx_fk_address_id' ('address_id' ASC),  CONSTRAINT 'fk_staff_address' FOREIGN KEY ('address_id') REFERENCES 'sakila_op2'. 'address' ('address_id') ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 3
DEFAULT CHARACTER SET = utf8;

-- -----------------------------
-- Table 'sakila_op2'. 'rental'
-- -----------------------------
CREATE TABLE IF NOT EXISTS 'sakila_op2'. 'rental' (  'rental_id' INT(11) NOT NULL AUTO_INCREMENT,  'rental_date' DATETIME NOT NULL,  'film_id' SMALLINT(5) UNSIGNED NOT NULL,  'customer_id' SMALLINT(5) UNSIGNED NOT NULL,  'return_date' DATETIME NULL DEFAULT NULL,  'staff_id' TINYINT(3) UNSIGNED NOT NULL,  'last_update' TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,  PRIMARY KEY ('rental_id'),  INDEX 'idx_fk_customer_id' ('customer_id' ASC),
INDEX `idx_fk_staff_id` ("staff_id" ASC),
INDEX `idx_fk_film_id` ("film_id" ASC),
CONSTRAINT 'fk_rental_customer'
    FOREIGN KEY ("customer_id")
    REFERENCES sakila_op2."customer" ("customer_id")
    ON UPDATE CASCADE,
CONSTRAINT 'fk_rental_film'
    FOREIGN KEY ("film_id")
    REFERENCES sakila_op2."film" ("film_id")
    ON UPDATE CASCADE,
CONSTRAINT 'fk_rental_staff'
    FOREIGN KEY ("staff_id")
    REFERENCES sakila_op2."staff" ("staff_id")
    ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 13808
DEFAULT CHARACTER SET = utf8;

-- ----------------------------
-- Table 'sakila_op2'.payment'
-- ----------------------------
CREATE TABLE IF NOT EXISTS sakila_op2.payment (  
    "payment_id" SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,  
    "customer_id" SMALLINT(5) UNSIGNED NOT NULL,  
    "staff_id" TINYINT(3) UNSIGNED NOT NULL,  
    "rental_id" INT(11) NULL DEFAULT NULL,  
    "amount" DECIMAL(5,2) NOT NULL,  
    "payment_date" DATETIME NOT NULL,  
    "last_update" TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP  
    ON UPDATE CURRENT_TIMESTAMP,  
    PRIMARY KEY ("payment_id"),  
    INDEX `idx_fk_staff_id` ("staff_id" ASC),  
    INDEX `idx_fk_customer_id` ("customer_id" ASC),  
    INDEX `fk_payment_rental` ("rental_id" ASC),  
    CONSTRAINT 'fk_payment_customer'
        FOREIGN KEY ("customer_id")
        REFERENCES sakila_op2."customer" ("customer_id")
        ON UPDATE CASCADE,
    CONSTRAINT 'fk_payment_rental'
        FOREIGN KEY ("rental_id")
        REFERENCES sakila_op2."rental" ("rental_id")
        ON DELETE SET NULL  
        ON UPDATE CASCADE,
    CONSTRAINT 'fk_payment_staff'
FOREIGN KEY ('staff_id')
REFERENCES 'sakila_op2'.staff ('staff_id')
ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 150
DEFAULT CHARACTER SET = utf8;

SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;

A.1.3 Schema generation script for reconciled schema - sakila_ids

-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';

DROP SCHEMA IF EXISTS 'sakila_ids' ;

CREATE SCHEMA IF NOT EXISTS 'sakila_ids' DEFAULT CHARACTER SET utf8 ;
USE 'sakila_ids' ;

CREATE TABLE IF NOT EXISTS 'sakila_ids'.actor
(
    'actor_id' SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
    'first_name' VARCHAR(45) NOT NULL,
    'last_name' VARCHAR(45) NOT NULL,
    'last_update' DATETIME NULL DEFAULT NULL,
    'ds_actor_id' SMALLINT(5) UNSIGNED NOT NULL,

    PRIMARY KEY ('actor_id')
) ENGINE = InnoDB
AUTO_INCREMENT = 150
DEFAULT CHARACTER SET = utf8;

SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
CREATE TABLE IF NOT EXISTS 'sakila_ids'.'country' (
    'country_id' SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
    'country' VARCHAR(50) NOT NULL,
    'last_update' DATETIME NULL DEFAULT NULL,
    PRIMARY KEY ('country_id'))
ENGINE = InnoDB
AUTO_INCREMENT = 15
DEFAULT CHARACTER SET = utf8;

CREATE TABLE IF NOT EXISTS 'sakila_ids'.'city' (
    'city_id' SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
    'city' VARCHAR(50) NOT NULL,
    'country_id' SMALLINT(5) UNSIGNED NOT NULL,
    'last_update' DATETIME NULL DEFAULT NULL,
    PRIMARY KEY ('city_id'),
    INDEX 'idx_fk_country_id' ('country_id' ASC),
    CONSTRAINT 'fk_city_country'
    FOREIGN KEY ('country_id')
    REFERENCES 'sakila_ids'.'country' ('country_id')
    ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 17
DEFAULT CHARACTER SET = utf8;

CREATE TABLE IF NOT EXISTS 'sakila_ids'.'address' (
'address' VARCHAR(50) NOT NULL,
'address2' VARCHAR(50) NULL DEFAULT NULL,
'district' VARCHAR(20) NOT NULL,
'city_id' SMALLINT(5) UNSIGNED NOT NULL,
'postcode' VARCHAR(10) NULL DEFAULT NULL,
'phone' VARCHAR(20) NOT NULL,
'last_update' DATETIME NULL DEFAULT NULL,
'ds_address_id' TINYINT(3) UNSIGNED NOT NULL DEFAULT '0',
'ds_name' VARCHAR(10) NULL DEFAULT NULL,
PRIMARY KEY ('address_id'),
INDEX 'idx_fk_city_id' ('city_id' ASC),
CONSTRAINT 'fk_address_city'
FOREIGN KEY ('city_id')
REFERENCES 'sakila_ids'.city ('city_id')
ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 50
DEFAULT CHARACTER SET = utf8;

-- -------------------------------
-- Table 'sakila_ids'.category
-- -------------------------------
CREATE TABLE IF NOT EXISTS 'sakila_ids'.category (  
'category_id' TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,  
'name' VARCHAR(25) NOT NULL,  
'last_update' DATETIME NULL DEFAULT NULL,  
'ds_category_id' TINYINT(3) UNSIGNED NOT NULL DEFAULT '0',  
'ds_name' VARCHAR(10) NULL DEFAULT NULL,  
PRIMARY KEY ('category_id'))
ENGINE = InnoDB
AUTO_INCREMENT = 13
DEFAULT CHARACTER SET = utf8;

-- -------------------------------
-- Table 'sakila_ids'.staff
-- -------------------------------
CREATE TABLE IF NOT EXISTS 'sakila_ids'.staff (  
'staff_id' TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,  
'first_name' VARCHAR(45) NOT NULL,  
'last_name' VARCHAR(45) NOT NULL,  
'address_id' SMALLINT(5) UNSIGNED NOT NULL,  
picture' BLOB NULL DEFAULT NULL,  
'email' VARCHAR(50) NULL DEFAULT NULL,
CREATE TABLE IF NOT EXISTS `sakila_ids`.`store` (  
    `store_id` TINYINT(3) UNSIGNED NOT NULL AUTO_INCREMENT,  
    `manager_staff_id` TINYINT(3) UNSIGNED NOT NULL,  
    `address_id` SMALLINT(5) UNSIGNED NOT NULL,  
    `last_update` DATETIME NULL DEFAULT NULL,  
    `ds_store_id` TINYINT(3) UNSIGNED NOT NULL,  
    `ds_name` VARCHAR(10) NULL DEFAULT NULL,  
    PRIMARY KEY (`store_id`),  
    UNIQUE INDEX `idx_unique_manager` (`manager_staff_id` ASC),  
    INDEX `idx_fk_address_id` (`address_id` ASC),  
    CONSTRAINT `fk_store_address`  
      FOREIGN KEY (`address_id`)  
      REFERENCES `sakila_ids`.`address` (`address_id`)  
      ON UPDATE CASCADE,  
    CONSTRAINT `fk_store_staff`  
      FOREIGN KEY (`manager_staff_id`)  
      REFERENCES `sakila_ids`.`staff` (`staff_id`)  
      ON UPDATE CASCADE)  
ENGINE = InnoDB  
AUTO_INCREMENT = 37  
DEFAULT CHARACTER SET = utf8;
CREATE TABLE IF NOT EXISTS `sakila_ids`.`film` (
    `film_id` SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
    `title` VARCHAR(255) NOT NULL,
    `description` TEXT NULL DEFAULT NULL,
    `release_year` YEAR NULL DEFAULT NULL,
    `language_id` TINYINT(3) UNSIGNED NOT NULL,
    `original_language_id` TINYINT(3) UNSIGNED NULL DEFAULT NULL,
    `rental_duration` TINYINT(3) UNSIGNED NOT NULL DEFAULT '3',
    `rental_rate` DECIMAL(4,2) NOT NULL DEFAULT '4.99',
    `length` SMALLINT(5) UNSIGNED NULL DEFAULT NULL,
    `replacement_cost` DECIMAL(5,2) NOT NULL DEFAULT '19.99',
    `rating` ENUM('G','PG','PG-13','R','NC-17') NULL DEFAULT 'G',
    `special_features` SET('Trailers','Commentaries','Deleted Scenes','Behind the Scenes') NULL DEFAULT NULL,
    `last_update` DATETIME NULL DEFAULT NULL,
    `ds_film_id` SMALLINT(5) UNSIGNED NOT NULL,
    `ds_name` VARCHAR(10) NULL DEFAULT NULL,
    PRIMARY KEY (`film_id`),
    INDEX `idx_title` (`title` ASC),
    INDEX `idx_fk_language_id` (`language_id` ASC),
    INDEX `idx_fk_original_language_id` (`original_language_id` ASC),
    CONSTRAINT `fk_film_language` FOREIGN KEY (`language_id`)
        REFERENCES `sakila_ids`.`language` (`language_id`) ON UPDATE CASCADE,
    CONSTRAINT `fk_film_language_original` FOREIGN KEY (`original_language_id`)
        REFERENCES `sakila_ids`.`language` (`language_id`) ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 24
DEFAULT CHARACTER SET = utf8;
CREATE TABLE IF NOT EXISTS `sakila_ids`.`film_actor` (  
    `actor_id` SMALLINT(5) UNSIGNED NOT NULL,  
    `film_id` SMALLINT(5) UNSIGNED NOT NULL,  
    `last_update` DATETIME NULL DEFAULT NULL,  
    `ds_film_id` SMALLINT(5) UNSIGNED NOT NULL,  
    `ds_actor_id` SMALLINT(5) UNSIGNED NOT NULL,  
    `ds_name` VARCHAR(10) NULL DEFAULT NULL,  
    PRIMARY KEY (`actor_id`, `film_id`),  
    INDEX `idx_fk_film_id` (`film_id` ASC),  
    CONSTRAINT `fk_film_actor_actor`  
      FOREIGN KEY (`actor_id`)  
      REFERENCES `sakila_ids`.`actor` (`actor_id`)  
      ON UPDATE CASCADE,  
    CONSTRAINT `fk_film_actor_film`  
      FOREIGN KEY (`film_id`)  
      REFERENCES `sakila_ids`.`film` (`film_id`)  
      ON UPDATE CASCADE)  
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;

CREATE TABLE IF NOT EXISTS `sakila_ids`.`film_category` (  
    `film_id` SMALLINT(5) UNSIGNED NOT NULL,  
    `category_id` TINYINT(3) UNSIGNED NOT NULL,  
    `last_update` DATETIME NULL DEFAULT NULL,  
    `ds_film_id` SMALLINT(5) UNSIGNED NOT NULL,  
    `ds_category_id` TINYINT(3) UNSIGNED NOT NULL,  
    `ds_name` VARCHAR(10) NULL DEFAULT NULL,  
    PRIMARY KEY (`film_id`, `category_id`),  
    INDEX `fk_film_category_category` (`category_id` ASC),  
    CONSTRAINT `fk_film_category_category`  
      FOREIGN KEY (`category_id`)  
      REFERENCES `sakila_ids`.`category` (`category_id`)  
      ON UPDATE CASCADE,  
    CONSTRAINT `fk_film_category_film`  
      FOREIGN KEY (`film_id`)  
      REFERENCES `sakila_ids`.`film` (`film_id`)  
      ON UPDATE CASCADE)  
ENGINE = InnoDB
```
CREATE TABLE IF NOT EXISTS `sakila_ids`.`film_text`  
    (`film_id` SMALLINT(5) NOT NULL,
     `title` VARCHAR(255) NOT NULL,
     `description` TEXT NULL DEFAULT NULL,
     PRIMARY KEY (`film_id`),
     FULLTEXT INDEX `idx_title_description` (`title` ASC, `description` ASC))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;
```

```
CREATE TABLE IF NOT EXISTS `sakila_ids`.`inventory`  
    (`inventory_id` MEDIUMINT(8) UNSIGNED NOT NULL AUTO_INCREMENT,
     `film_id` SMALLINT(5) UNSIGNED NOT NULL,
     `store_id` TINYINT(3) UNSIGNED NOT NULL,
     `last_update` DATETIME NULL DEFAULT NULL,
     `ds_inventory_id` MEDIUMINT(8) UNSIGNED NOT NULL,
     `ds_name` VARCHAR(10) NULL DEFAULT NULL,
     PRIMARY KEY (`inventory_id`),
     INDEX `idx_fk_film_id` (`film_id` ASC),
     INDEX `idx_store_id_film_id` (`store_id` ASC, `film_id` ASC),
     CONSTRAINT `fk_inventory_film`
         FOREIGN KEY (`film_id`)  
         REFERENCES `sakila_ids`.`film` (`film_id`)  
         ON UPDATE CASCADE,
     CONSTRAINT `fk_inventory_store`
         FOREIGN KEY (`store_id`)  
         REFERENCES `sakila_ids`.`store` (`store_id`)  
         ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 6
DEFAULT CHARACTER SET = utf8;
```

```
CREATE TABLE IF NOT EXISTS `sakila_ids`.`rental`  
```
```
CREATE TABLE IF NOT EXISTS 'sakila_ids'.'rental' (  
'rental_id' INT(11) NOT NULL AUTO_INCREMENT,
'rental_date' DATETIME NOT NULL,
'inventory_id' MEDIUMINT(8) UNSIGNED NULL DEFAULT NULL,
'film_id' SMALLINT(5) UNSIGNED NULL DEFAULT NULL,
'customer_id' SMALLINT(5) UNSIGNED NOT NULL,
'return_date' DATETIME NULL DEFAULT NULL,
'staff_id' TINYINT(3) UNSIGNED NOT NULL,
'last_update' DATETIME NULL DEFAULT NULL,
'ds_rental_id' INT(10) UNSIGNED NOT NULL,
'ds_name' VARCHAR(10) NULL DEFAULT NULL,
PRIMARY KEY ('rental_id'),
UNIQUE INDEX 'rental_date' ('rental_date' ASC, 'inventory_id' ASC, 'customer_id' ASC),
INDEX 'idx_fk_inventory_id' ('inventory_id' ASC),
INDEX 'idx_fk_customer_id' ('customer_id' ASC),
INDEX 'idx_fk_staff_id' ('staff_id' ASC),
INDEX 'fk_rental_film' ('film_id' ASC),
CONSTRAINT 'fk_rental_customer'
    FOREIGN KEY ('customer_id')
    REFERENCES 'sakila_ids'.'customer' ('customer_id')
    ON UPDATE CASCADE,
CONSTRAINT 'fk_rental_film'
    FOREIGN KEY ('film_id')
    REFERENCES 'sakila_ids'.'film' ('film_id')
    ON UPDATE CASCADE,
CONSTRAINT 'fk_rental_inventory'
    FOREIGN KEY ('inventory_id')
    REFERENCES 'sakila_ids'.'inventory' ('inventory_id')
    ON UPDATE CASCADE,
CONSTRAINT 'fk_rental_staff'
    FOREIGN KEY ('staff_id')
    REFERENCES 'sakila_ids'.'staff' ('staff_id')
    ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 11
DEFAULT CHARACTER SET = utf8;

-- -----------------------------------------------------
-- Table 'sakila_ids'.'payment'
-- -----------------------------------------------------

CREATE TABLE IF NOT EXISTS 'sakila_ids'.'payment' (  
'payment_id' SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
'customer_id' SMALLINT(5) UNSIGNED NOT NULL,
'staff_id' TINYINT(3) UNSIGNED NOT NULL,
```sql
'payment_id' INT(11) NULL DEFAULT NULL,
'amount' DECIMAL(5,2) NOT NULL,
'payment_date' DATETIME NOT NULL,
'last_update' DATETIME NULL DEFAULT NULL,
'ds_payment_id' SMALLINT(5) UNSIGNED NOT NULL,
'ds_name' VARCHAR(10) NULL DEFAULT NULL,
PRIMARY KEY ('payment_id'),
INDEX 'idx_fk_staff_id' ('staff_id' ASC),
INDEX 'idx_fk_customer_id' ('customer_id' ASC),
INDEX 'fk_payment_rental' ('rental_id' ASC),
CONSTRAINT 'fk_payment_customer'
  FOREIGN KEY ('customer_id')
  REFERENCES 'sakila_ids'.customer ('customer_id')
  ON UPDATE CASCADE,
CONSTRAINT 'fk_payment_rental'
  FOREIGN KEY ('rental_id')
  REFERENCES 'sakila_ids'.rental ('rental_id')
  ON DELETE SET NULL
  ON UPDATE CASCADE,
CONSTRAINT 'fk_payment_staff'
  FOREIGN KEY ('staff_id')
  REFERENCES 'sakila_ids'.staff ('staff_id')
  ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 11
DEFAULT CHARACTER SET = utf8;

SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;

A.1.4 Schema generation script for warehouse schema - sakila_dwh

-- MySQL Script generated by MySQL Workbench
-- 12/19/16 10:12:40
-- Model: New Model Version: 1.0
-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
```
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';

-- Schema mydb
-- -----------------------------------------------------
-- Schema sakila_dwh
-- -----------------------------------------------------

CREATE SCHEMA IF NOT EXISTS 'sakila_dwh' DEFAULT CHARACTER SET latin1;
USE 'sakila_dwh';

-- Table 'sakila_dwh'.'dim_actor'
-- -----------------------------------------------------
DROP TABLE IF EXISTS 'sakila_dwh'.'dim_actor' ;
CREATE TABLE IF NOT EXISTS 'sakila_dwh'.'dim_actor' (
    'actor_key' INT(10) NOT NULL AUTO_INCREMENT,
    'actor_last_update' DATETIME NOT NULL,
    'actor_last_name' VARCHAR(45) NOT NULL,
    'actor_first_name' VARCHAR(45) NOT NULL,
    'actor_id' INT(11) NOT NULL,
    PRIMARY KEY ('actor_key'))
    ENGINE = InnoDB
    AUTO_INCREMENT = 201
    DEFAULT CHARACTER SET = latin1;

-- Table 'sakila_dwh'.'dim_customer'
-- -----------------------------------------------------
DROP TABLE IF EXISTS 'sakila_dwh'.'dim_customer' ;
CREATE TABLE IF NOT EXISTS 'sakila_dwh'.'dim_customer' (
    'customer_key' INT(8) NOT NULL AUTO_INCREMENT,
    'customer_last_update' DATETIME NOT NULL DEFAULT '1970-01-01 00:00:00',
    'customer_id' INT(8) NULL DEFAULT NULL,
    'customer_first_name' VARCHAR(45) NOT NULL DEFAULT NULL,
    'customer_last_name' VARCHAR(45) NOT NULL DEFAULT NULL,
    'customer_email' VARCHAR(50) NULL DEFAULT NULL,
    'customer_active' CHAR(3) NULL DEFAULT NULL,
-- Table 'sakila_dwh'. 'dim_date'

DROP TABLE IF EXISTS 'sakila_dwh'. 'dim_date';

CREATE TABLE IF NOT EXISTS 'sakila_dwh'. 'dim_date' (
    'date_key' INT(8) NOT NULL,
    'date_value' DATE NOT NULL,
    'date_short' CHAR(12) NOT NULL,
    'date_medium' CHAR(16) NOT NULL,
    'date_long' CHAR(24) NOT NULL,
    'date_full' CHAR(32) NOT NULL,
    'day_in_year' SMALLINT(5) NOT NULL,
    'day_in_month' TINYINT(3) NOT NULL,
    'is_first_day_in_month' CHAR(10) NOT NULL,
    'is_last_day_in_month' CHAR(10) NOT NULL,
    'day_abbreviation' CHAR(3) NOT NULL,
    'day_name' CHAR(12) NOT NULL,
    'week_in_year' TINYINT(3) NOT NULL,
    'week_in_month' TINYINT(3) NOT NULL,
    'is_first_day_in_week' CHAR(10) NOT NULL,
    'is_last_day_in_week' CHAR(10) NOT NULL,
    'month_number' TINYINT(3) NOT NULL,
    'month_abbreviation' CHAR(3) NOT NULL,
    'month_name' CHAR(12) NOT NULL,
    'year2' CHAR(2) NOT NULL,
    'year4' SMALLINT(5) NOT NULL,
    'quarter_name' CHAR(2) NOT NULL,
'quarter_number' TINYINT(3) NOT NULL,
'year_quarter' CHAR(7) NOT NULL,
'year_month_number' CHAR(7) NOT NULL,
'year_month_abbreviation' CHAR(8) NOT NULL,
PRIMARY KEY ('date_key'),
UNIQUE INDEX 'date' USING BTREE ('date_value' ASC),
UNIQUE INDEX 'date_value' ('date_value' ASC))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;

-- -----------------------------------------------------
-- Table 'sakila_dwh'. 'dim_film'
-- -----------------------------------------------------
DROP TABLE IF EXISTS 'sakila_dwh'. 'dim_film';

CREATE TABLE IF NOT EXISTS 'sakila_dwh'. 'dim_film' (  
  'film_key' INT(8) NOT NULL AUTO_INCREMENT,  
  'film_last_update' DATETIME NOT NULL,  
  'film_title' VARCHAR(64) NOT NULL,  
  'film_description' TEXT NOT NULL,  
  'film_release_year' SMALLINT(5) NOT NULL,  
  'film_language' VARCHAR(20) NOT NULL,  
  'film_original_language' VARCHAR(20) NOT NULL,  
  'film_rental_duration' TINYINT(3) NULL DEFAULT NULL,  
  'film_rental_rate' DECIMAL(4,2) NULL DEFAULT NULL,  
  'film_duration' INT(8) NULL DEFAULT NULL,  
  'film_replacement_cost' DECIMAL(5,2) NULL DEFAULT NULL,  
  'film_rating_code' CHAR(5) NULL DEFAULT NULL,  
  'film_rating_text' VARCHAR(30) NULL DEFAULT NULL,  
  'film_has_trailers' CHAR(4) NULL DEFAULT NULL,  
  'film_has_commentaries' CHAR(4) NULL DEFAULT NULL,  
  'film_has_deleted_scenes' CHAR(4) NULL DEFAULT NULL,  
  'film_has_behind_the_scenes' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_action' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_animation' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_children' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_classics' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_comedy' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_documentary' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_drama' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_family' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_foreign' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_games' CHAR(4) NULL DEFAULT NULL,  
  'film_in_category_horror' CHAR(4) NULL DEFAULT NULL, 

215


CREATE TABLE IF NOT EXISTS 'sakila_dwh'.'dim_film_actor_bridge' (  
  'film_key' INT(8) NOT NULL,  
  'actor_key' INT(10) NOT NULL,  
  'actor_weighting_factor' DECIMAL(3,2) NOT NULL,  
  PRIMARY KEY ('film_key', 'actor_key'),  
  INDEX 'dim_actor_dim_film_actor_bridge_fk' ('actor_key' ASC),  
  CONSTRAINT 'fk_bridge_actor'  
    FOREIGN KEY ('actor_key')  
    REFERENCES 'sakila_dwh'.'dim_actor' ('actor_key'),  
  CONSTRAINT 'fk_bridge_film'  
    FOREIGN KEY ('film_key')  
    REFERENCES 'sakila_dwh'.'dim_film' ('film_key'))

CREATE TABLE IF NOT EXISTS 'sakila_dwh'.'dim_staff' (  
  'staff_key' INT(8) NOT NULL AUTO_INCREMENT,  
  'staff_last_update' DATETIME NOT NULL DEFAULT '1970-01-01 00:00:00',  
  'staff_first_name' VARCHAR(45) NULL DEFAULT NULL,  
  'staff_last_name' VARCHAR(45) NULL DEFAULT NULL,  
  'staff_id' INT(8) NULL DEFAULT NULL,  
  'staff_store_id' INT(8) NULL DEFAULT NULL,


-- Table `sakila_dwh`.`dim_store`
-- -------------------------------

DROP TABLE IF EXISTS `sakila_dwh`.`dim_store` ;

CREATE TABLE IF NOT EXISTS `sakila_dwh`.`dim_store` (  
    `store_key` INT(8) NOT NULL AUTO_INCREMENT,  
    `store_last_update` DATETIME NOT NULL DEFAULT '1970-01-01 00:00:00',  
    `store_id` INT(8) NULL DEFAULT NULL,  
    `store_address` VARCHAR(64) NULL DEFAULT NULL,  
    `store_district` VARCHAR(20) NULL DEFAULT NULL,  
    `store_postal_code` VARCHAR(10) NULL DEFAULT NULL,  
    `store_phone_number` VARCHAR(20) NULL DEFAULT NULL,  
    `store_city` VARCHAR(50) NULL DEFAULT NULL,  
    `store_country` VARCHAR(50) NULL DEFAULT NULL,  
    `store_manager_staff_id` INT(8) NULL DEFAULT NULL,  
    `store_manager_first_name` VARCHAR(45) NULL DEFAULT NULL,  
    `store_manager_last_name` VARCHAR(45) NULL DEFAULT NULL,  
    `store_version_number` SMALLINT(5) NULL DEFAULT NULL,  
    `store_valid_from` DATE NULL DEFAULT NULL,  
    `store_valid_through` DATE NULL DEFAULT NULL,  
PRIMARY KEY (`store_key`),  
INDEX `store_id` USING BTREE (`store_id` ASC))
ENGINE = InnoDB
AUTO_INCREMENT = 4
DEFAULT CHARACTER SET = latin1;

-- Table `sakila_dwh`.`dim_time`
-- -------------------------------

DROP TABLE IF EXISTS `sakila_dwh`.`dim_time` ;

CREATE TABLE IF NOT EXISTS `sakila_dwh`.`dim_time` (  
    `store_last_update` DATETIME NOT NULL DEFAULT '1970-01-01 00:00:00',  
    `store_id` INT(8) NULL DEFAULT NULL,  
    `store_address` VARCHAR(64) NULL DEFAULT NULL,  
    `store_district` VARCHAR(20) NULL DEFAULT NULL,  
    `store_postal_code` VARCHAR(10) NULL DEFAULT NULL,  
    `store_phone_number` VARCHAR(20) NULL DEFAULT NULL,  
    `store_city` VARCHAR(50) NULL DEFAULT NULL,  
    `store_country` VARCHAR(50) NULL DEFAULT NULL,  
    `store_manager_staff_id` INT(8) NULL DEFAULT NULL,  
    `store_manager_first_name` VARCHAR(45) NULL DEFAULT NULL,  
    `store_manager_last_name` VARCHAR(45) NULL DEFAULT NULL,  
    `store_version_number` SMALLINT(5) NULL DEFAULT NULL,  
    `store_valid_from` DATE NULL DEFAULT NULL,  
    `store_valid_through` DATE NULL DEFAULT NULL,  
PRIMARY KEY (`store_key`),  
INDEX `store_id` USING BTREE (`store_id` ASC))
ENGINE = InnoDB
AUTO_INCREMENT = 4
DEFAULT CHARACTER SET = latin1;
```
'time_key' INT(8) NOT NULL,
'time_value' TIME NOT NULL,
'hours24' TINYINT(3) NOT NULL,
'hours12' TINYINT(3) NULL DEFAULT NULL,
'minutes' TINYINT(3) NULL DEFAULT NULL,
'seconds' TINYINT(3) NULL DEFAULT NULL,
'am_pm' CHAR(3) NULL DEFAULT NULL,
PRIMARY KEY ('time_key'),
UNIQUE INDEX 'time_value' ('time_value' ASC))
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;

-- -----------------------------------------------------
-- Table 'sakila_dwh'.fact_rental
-- -----------------------------------------------------
DROP TABLE IF EXISTS 'sakila_dwh'.fact_rental ;

CREATE TABLE IF NOT EXISTS 'sakila_dwh'.fact_rental (  
'customer_key' INT(8) NOT NULL,
'staff_key' INT(8) NOT NULL,
'film_key' INT(8) NOT NULL,
'store_key' INT(8) NOT NULL,
'rental_date_key' INT(8) NOT NULL,
'return_date_key' INT(10) NOT NULL,
'rental_time_key' INT(8) NOT NULL,
'count_returns' INT(10) NOT NULL,
'count_rentals' INT(8) NOT NULL,
'rental_duration' INT(11) NULL DEFAULT NULL,
'rental_last_update' DATETIME NULL DEFAULT NULL,
'rental_id' INT(11) NULL DEFAULT NULL,
INDEX 'dim_store_fact_rental_fk' ('store_key' ASC),
INDEX 'dim_staff_fact_rental_fk' ('staff_key' ASC),
INDEX 'dim_time_fact_rental_fk' ('rental_time_key' ASC),
INDEX 'dim_film_fact_rental_fk' ('film_key' ASC),
INDEX 'dim_date_fact_rental_fk' ('return_date_key' ASC),
INDEX 'dim_customer_fact_rental_fk' ('customer_key' ASC),
INDEX 'fk_return_date_key' ('return_date_key' ASC),
CONSTRAINT 'fk_customer'  
FOREIGN KEY ('customer_key')  
REFERENCES 'sakila_dwh'.dim_customer ('customer_key'),
CONSTRAINT 'fk_film'  
FOREIGN KEY ('film_key')  
REFERENCES 'sakila_dwh'.dim_film ('film_key')
ON DELETE NO ACTION
```
ON UPDATE NO ACTION,
CONSTRAINT 'fk_rental_date_key'
  FOREIGN KEY ('rental_date_key')
  REFERENCES 'sakila_dwh'.'dim_date' ('date_key')
  ON DELETE NO ACTION
  ON UPDATE NO ACTION,
CONSTRAINT 'fk_rental_time_key'
  FOREIGN KEY ('rental_time_key')
  REFERENCES 'sakila_dwh'.'dim_time' ('time_key'),
CONSTRAINT 'fk_return_date_key'
  FOREIGN KEY ('return_date_key')
  REFERENCES 'sakila_dwh'.'dim_date' ('date_key')
  ON DELETE NO ACTION
  ON UPDATE NO ACTION,
CONSTRAINT 'fk_staff'
  FOREIGN KEY ('staff_key')
  REFERENCES 'sakila_dwh'.'dim_staff' ('staff_key')
  ON DELETE NO ACTION
  ON UPDATE NO ACTION,
CONSTRAINT 'fk_store'
  FOREIGN KEY ('store_key')
  REFERENCES 'sakila_dwh'.'dim_store' ('store_key')
  ON DELETE NO ACTION
  ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1
COMMENT = 'Switched storage engine from MyISAM to InnoDB to allow foreign key constraints';

SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;

A.1.5 Instance generation script for operational schemas - sakila_op1 and sakila_op2

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';

-- -------------------------------------
-- Insert data into operational schemas actor table
-- -------------------------------------
insert into sakila_op1.actor
    select actor_id, fname, lname, last_update from sakila.actor
    where actor_id in (1,2,3);

insert into sakila_op2.actor
    select actor_id, fname, lname, last_update from sakila.actor
    where actor_id in (32,28,12);

-- -------------------------------------
-- Insert data into sakila_op1.language table
-- -------------------------------------
insert into sakila_op1.language
    select * from sakila.language where language_id <= 2;

-- -------------------------------------
-- Insert data into film table
-- -------------------------------------
insert into sakila_op1.film
    select * from sakila.film where film_id in (1,555,967, 447, 663);

insert into sakila_op2.film
    select film_id, title, description, release_year, 'English', rental_duration,
    rental_rate, length, replacement_cost, rating, special_features, last_update from sakila.film
    where film_id in (585, 973, 513);

-- -------------------------------------
-- Insert data into film_actor table
-- -------------------------------------
insert into sakila_op1.film_actor
    select * from sakila.film_actor where film_id in (1,555,967, 447, 663)
    and actor_id in (1,2,3);

insert into sakila_op2.film_actor
    select * from sakila.film_actor where film_id in (585, 973, 513)
    and actor_id in (32,28,12);

-- -------------------------------------
-- Insert data into category table
-- -------------------------------------
insert into sakila_op1.category
    select * from sakila.category where category_id in (6,5,14);

insert into sakila_op2.category
    select * from sakila.category where category_id in (6,5,14);
select * from sakila.category where category_id in (7, 6, 16);

-- Insert data into film_category table
-- --------------------------------------------
insert into sakila_op1.film_category
select * from sakila.film_category where category_id in (6, 5, 14)
and film_id in (1, 555, 967);

insert into sakila_op2.film_category
select * from sakila.film_category where category_id in (7, 6, 16)
and film_id in (585, 973, 513);

-- Insert data into inventory table
-- --------------------------------------------
insert into sakila_op1.inventory
select * from sakila.inventory where inventory_id in (5, 2525, 4422, 3021, 2060);

-- Insert data into store table
-- --------------------------------------------
insert into sakila_op1.store
select * from sakila.store;

-- Insert data into address table
-- --------------------------------------------
insert into sakila_op1.address
select address_id, address as street_address,
(select city from sakila.city where city_id=
(select city_id from sakila.address where address_id=o.address_id)) as city,
(select country from sakila.country where country_id in
(select country_id from sakila.city where city_id in
(select city_id from sakila.address where address_id=o.address_id))) as country,
postal_code, phone, last_update
from sakila.address o where address_id in (1, 2, 3, 4, 5, 6, 7);

insert into sakila_op2.address
select address_id, address as street_address,
(select city from sakila.city where city_id=
(select city_id from sakila.address where address_id=o.address_id)) as city,
(select country from sakila.country where country_id in
(select country_id from sakila.city where city_id in
(select country_id from sakila.city where city_id in
(select city_id from sakila.address where address_id=o.address_id))) as country,
postal_code, phone, last_update
from sakila.address o where address_id in (1,2,3,4, 8,9,10);

--  ---------------------------------------------------------------
-- Insert data into staff table
--  ---------------------------------------------------------------

insert into sakila_op1.staff
select * from sakila.staff;

insert into sakila_op2.staff
select staff_id, first_name, last_name, address_id, picture, email,
active, username, password, last_update from sakila.staff;

--  ---------------------------------------------------------------
-- Insert data into customer table
--  ---------------------------------------------------------------

insert into sakila_op1.customer
select * from sakila.customer where address_id in (1,2,3,4,5,6,7);

insert into sakila_op2.customer
select customer_id, first_name, last_name, email, address_id, active, create_date,
last_update from sakila.customer where address_id in (1,2,3,4, 8,9,10);

--  ---------------------------------------------------------------
-- Insert data into rental table
--  ---------------------------------------------------------------

insert into sakila_op1.rental
select * from sakila.rental where rental_id in (76, 11256);

insert into sakila_op2.rental
select rental_id, rental_date, 0 as film_id, customer_id, return_date, staff_id,
last_update from sakila.rental where rental_id in (1297, 13807,1575);
update sakila_op2.rental set film_id=513 where rental_id=1297;
update sakila_op2.rental set film_id=585 where rental_id=1575;
update sakila_op2.rental set film_id=973 where rental_id=13807;

--  ---------------------------------------------------------------
-- Insert data into payment table
--  ---------------------------------------------------------------
insert into sakila_op1.payment
select * from sakila.payment where rental_id in (76, 11256);

insert into sakila_op2.payment
select * from sakila.payment where rental_id in (1297, 13807, 1575);

SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;

A.2 ETL workflows

This section focuses on the set of ETL workflows between (a) each of the operational schemas and the reconciled schema, and (b) reconciled and the warehouse schema. While the workflows between the integrated and warehouse schema are provided by Casters et. al [43], the transformations between the operational and reconciled schemas were developed as a part of our work.

A.2.1 ETL workflows between the operational and reconciled schema

Chapter 6 presenting our case study covered four transformations, load_actor, load_language, load_address, load_film. We include the rest of 12 transformations in this section here.

![Figure A.1: ETL transformation - load_film_actor](image-url)
Figure A.2: ETL transformation - load_store

Figure A.3: ETL transformation - load_staff
Figure A.4: ETL transformation - load_rental

Figure A.5: ETL transformation - load_payment

Figure A.6: ETL transformation - load_inventory

Figure A.7: ETL transformation - load_film_category
Figure A.8: ETL transformation - aux_staff_store

Figure A.9: ETL transformation - load_customer
Figure A.10: ETL transformation - load_country

Figure A.11: ETL transformation - load_category

Figure A.12: ETL transformation - load_city
A.2.2 ETL workflows between the reconciled and warehouse schema

Chapter 6 presented a brief discussion of each of the transformations. It consists of a total of nine workflows which are included in this section.

Figure A.13: ETL transformation - load_dim_actor [43]
Figure A.14: ETL transformation - load_dim_customer [43]
Figure A.15: ETL transformation - load_dim_film [43]
Figure A.16: ETL transformation - load_dim_date [43]
Figure A.17: ETL transformation - load_dim_time [43]
Figure A.18: ETL transformation - load_dim_store [43]
Figure A.19: ETL transformation - load_dim_staff [43]
Figure A.20: ETL transformation - load_fact_rental [43]

Figure A.21: ETL transformation - load_fetch_address [43]
A.3 End user queries (MySQL)

This section focuses on the set of twelve SQL queries considered in our case study. The set offers variety in terms of the use of clauses, columns embedded within functions, join operations, aggregate functions and the queries cover different schemas.

```sql
{Q1}:
use sakila_ids;
SELECT rental_id FROM rental
WHERE inventory_id = 10
AND customer_id = 3
AND return_date IS NULL;

{Q2}:
use sakila_ids;
SELECT CONCAT(customer.last_name, ', ', customer.first_name) AS customer,
address.phone, film.title
FROM rental INNER JOIN customer ON rental.customer_id = customer.customer_id
INNER JOIN address ON customer.address_id = address.address_id
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id
WHERE rental.return_date IS NULL
AND rental_date + INTERVAL film.rental_duration DAY < CURRENT_DATE()
LIMIT 5;

{Q3}:
use sakila_op2;
SELECT actor.fname, actor.lname
FROM actor where actor.lname LIKE '%MAN%' ORDER BY actor.fname;

{Q4}:
use sakila_op1;
SELECT customer_id, first_name, last_name from customer
where customer_id NOT IN (select customer_id from rental)

{Q5}:
use sakila_ids;
SELECT film.film_id, title from film
where film.film_id in
(select film_category.film_id from film_category where
film_category.category_id in
(select category.category_id from category where
name='Documentary')
and film.language_id=
(select language.language_id from language where
236
name='English' LIMIT 1);

\textbf{Q6}:
use sakila_ids;
SELECT cu.customer_id AS ID, CONCAT(cu.first_name, _utf8' ', cu.last_name) AS name,
a.address AS address, a.postal_code AS 'zip code',
a.phone AS phone, city.city AS city, country.country AS country,
IF(cu.active, _utf8'active', _utf8'') AS notes, cu.store_id AS SID
FROM customer AS cu JOIN address AS a
ON cu.address_id = a.address_id JOIN city ON a.city_id = city.city_id
JOIN country ON city.country_id = country.country_id;

\textbf{Q7}:
use sakila_ids;
SELECT film.film_id AS FID, film.title AS title, film.description AS description,
category.name AS category, film.rental_rate AS price,
film.length AS length, film.rating AS rating,
GROUP_CONCAT(CONCAT(actor.first_name, _utf8' ', actor.last_name) SEPARATOR ', ') AS actors
FROM category LEFT JOIN film_category
ON category.category_id = film_category.category_id
LEFT JOIN film ON film_category.film_id = film.film_id
JOIN film_actor ON film.film_id = film_actor.film_id
JOIN actor ON film_actor.actor_id = actor.actor_id
GROUP BY film.film_id, category.name;

\textbf{Q8}:
use sakila_ids;
SELECT film.film_id AS FID, film.title AS title, film.description AS description,
category.name AS category, film.rental_rate AS price,
film.length AS length, film.rating AS rating,
GROUP_CONCAT(CONCAT(UCASE(SUBSTR(actor.first_name,1,1)),
LCASE(SUBSTR(actor.first_name,2,LENGTH(actor.first_name))),_utf8' ',
CONCAT(UCASE(SUBSTR(actor.last_name,1,1)),
LCASE(SUBSTR(actor.last_name,2,LENGTH(actor.last_name))))))) SEPARATOR ', ') AS actors
FROM category LEFT JOIN film_category ON category.category_id = film_category.category_id
LEFT JOIN film ON film_category.film_id = film.film_id
JOIN film_actor ON film.film_id = film_actor.film_id
JOIN actor ON film_actor.actor_id = actor.actor_id
GROUP BY film.film_id, category.name;

\textbf{Q9}:
use sakila_ids;
SELECT
c.name AS category, SUM(p.amount)
FROM payment AS p
INNER JOIN rental AS r ON p.rental_id = r.rental_id
INNER JOIN inventory AS i ON r.inventory_id = i.inventory_id
INNER JOIN film AS f ON i.film_id = f.film_id
INNER JOIN film_category AS fc ON f.film_id = fc.film_id
INNER JOIN category AS c ON fc.category_id = c.category_id
GROUP BY c.name
ORDER BY SUM(p.amount) DESC;

\textbf{Q10}:
use sakila_ids;
SELECT film.film_id AS FID, film.title AS title, film.description AS description,
category.name AS category, film.rental_rate AS price, film.length AS length, film.rating AS rating,
CONCAT(UCASE(SUBSTR(actor.first_name,1,1)),LCASE(SUBSTR(actor.first_name,2,LENGTH(actor.first_name))),_utf8' ',
CONCAT(UCASE(SUBSTR(actor.last_name,1,1)),LCASE(SUBSTR(actor.last_name,2,LENGTH(actor.last_name)))))) SEPARATOR ', ') AS actors
FROM category LEFT JOIN film_category ON category.category_id = film_category.category_id
LEFT JOIN film
ON film_category.film_id = film.film_id
JOIN film_actor ON film.film_id = film_actor.film_id
JOIN actor ON film_actor.actor_id = actor.actor_id
GROUP BY film.film_id, category.name;

\textbf{Q11}:
use sakila_dwh;
select sum(count_rentals), customer_first_name, customer_last_name, customer_email
from fact_rental f join dim_customer c on f.customer_key=c.customer_key
group by c.customer_key having sum(count_rentals)> 10
order by sum(count_rentals);

\textbf{Q12}:
use sakila_dwh;
select count(count_rentals), date_medium from fact_rental f join dim_date d
on f.rental_date_key = d.date_key
join dim_film film on film.film_key=f.film_key
where film.in_category_foreign='Yes'
group by year4, month_number
order by year4, month_number
Appendix B: Java code for Chapter 7 (User Interface and Impact Assessment)

This appendix contains the complete Java code that we developed for implementing the user interface (Chapter 7) and schema integration (Chapter 8).

The code we have consists of two parts: (1) frontend code written in JSP (Java Servlet Pages) and Bootstrap CSS, and (2) code that handles interaction with database and handles the user’s request and response, written using Java Servlets. The first part is the user presentation layer and the filenames have extensions .jsp. The second part of the code consists of .java files.

The overall code structure can be understood as shown in Figure B.1.

B.1 Front end code

This section presents code for the four files - index.jsp, etl.jsp, query.jsp, lineage.jsp. These files provide user interface to the four main functionalities of our work. These functionalities include conversion of schemas, ETL workflows, queries to graph format and allowing user to specify artifacts for schema change and observing their impact.
B.1.1 index.jsp

This section shows the code for the first index.jsp file. This file launches a user view in the form of a webpage which consists of a simple textbox, a dropdown and a button. The user specifies the name of the schema that needs to be transformed to a Neo4j graph in the textbox, selects the schema type (operational, reconciled, warehouse) from the dropdown. Clicking the submit button, an action is triggered which calls a method in another file, SchemaServlet, described later in this chapter.

```html
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1" session="true"%>
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Index page</title>
    <link href="bootstrap.min.css" rel="stylesheet">
    <script type="text/javascript">
```

Figure B.1: A high-level overview of the frontend code files
Impact Assessment of Schema Evolution in a Data Warehouse Environment

Step 1: Load and view artifacts

Transform a given schema to a neo4j graph

A data warehouse environment is formed of multiple artifacts. Please select the artifact type from one of the following in the list: (a) Operational schema, (b) Reconciled schema, (c) ETL workflows, (d) Data warehouse and (e) Queries.

This step allows transformation of relational schemas into a Neo4j graph.

<form class="form-inline" method="post" action="SchemaServlet">
  <div class="form-group">
    <label for="schematype">Schema type:</label>
    <select class="form-control" id="schematype" name="schematype">
      <option value="operational" selected>Operational</option>
      <option value="reconciled">Reconciled</option>
      <option value="warehouse">Data Warehouse</option>
    </select>
    <label for="txt_schemaname">Schema name</label>
    <input type="text" class="form-control" name="txt_schemaname">
  </div>
  <br />
  <button type="submit" class="btn btn-default" name="btn_convertToGraph">Convert to Neo4j graph</button>
  <button type="submit" class="btn btn-default" name="btn_view">View the graph</button>
  <button type="submit" class="btn btn-default" name="btn_next">Next</button>
  <div style="color: #FF0000;">${errorMessage}</div>
  <div style="color: 'blue';">${successMessage}</div>
  <textarea rows="10" cols="100" name="txt_output" id="txt_output">${graphQuery}</textarea>
</form>
B.1.2 etl.jsp

This section shows the code for the first etl.jsp file. This file launches a user view in the form of a webpage which consists of a simple textbox, a dropdown and four possible actions in the form of buttons. The user specifies the name of the folder which contains the XML representations of the ETL workflows. The dropdown allows the user to specify whether the transformations map the operational schemas to the reconciled schema or if they are the mappings between the reconciled schema to the warehouse schema. The interface offers four possible actions that the user can perform - (1) Convert the XML representations to CSV, (2) Generate the Neo4j graph by reading the CSV files, (3) View the graph, or (4) Move to the next web page. Clicking any of the buttons, an action is triggered which calls a method in another file, ETLServlet, described later in this appendix.

```html
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Index page</title>
<link href="bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div style="text-align: center">
<h1>Impact Assessment of Schema Evolution in a Data Warehouse Environment</h1>
<hr><br>
<h2>Step 1: Load and view artifacts</h2>
<i>A data warehouse environment is formed of multiple artifacts. Please select the artifact type from one of the following in the list: (a) Operational schema, (b) Reconciled schema, (c) ETL workflows, (d) Data warehouse schema, and (e) Queries. </i>
<br>
This step allows transformation of ETL workflows into a Neo4j graph.
```
B.1.3 queries.jsp

This file provides interface where user can convert SQL queries to Neo4j graph. The process starts by having the user specify the name of the folder which contains the XML representations of the SQL queries. This XML representation is obtained using third-party utility called GSQLParser. First the user generates a CSV formatted data for the given XML representation. Then, the Neo4j graph is generated by reading the csv file.
Impact Assessment of Schema Evolution in a Data Warehouse Environment

Step 1: Load and view artifacts

Transform SQL queries to a neo4j graph

A data warehouse environment is formed of multiple artifacts. Please select the artifact type from one of the following in the list:
(a) Operational schema, (b) Reconciled schema, (c) ETL workflows, (d) Data warehouse schema, and (e) Queries. This step allows transformation of SQL queries expressed over each of the schemas into a Neo4j graph.

Specify the folder location containing XML files:

Convert from XML to CSV
Generate Neo4j graph
View the graph
Next
B.1.4 lineage.jsp

This file offers the interface where the user can specify schema change on any artifact. The code discussed later in this appendix, LineageServlet.java describes how the user is presented with the results depending on the specified artifact.

```html
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html lang="en">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Index page</title>
        <link href="bootstrap.min.css" rel="stylesheet">
        <script type="text/javascript">
            function toggle() {
                if (document.getElementById("artifacttype").value == "schema") {
                    document.getElementById("txt_relationname").disabled=true;
                    document.getElementById("txt_attributename").disabled=true;
                    document.getElementById("txt_schemaname").disabled='';
                } else if (document.getElementById("artifacttype").value == "relation") {
                    document.getElementById("txt_relationname").disabled='';
                    document.getElementById("txt_attributename").disabled=true;
                    document.getElementById("txt_schemaname").disabled='';
                } else if (document.getElementById("artifacttype").value == "attribute") {
                    document.getElementById("txt_relationname").disabled='';
                    document.getElementById("txt_attributename").disabled='';
                    document.getElementById("txt_schemaname").disabled='';
                }
            }
        </script>
    </head>

    <body>
        <div style="text-align: center">
            <h1>Impact Assessment of Schema Evolution in a Data Warehouse Environment</h1>
            <hr>
        </div>
    </body>
```
<h2>Provenance - Understand relationships between various artifacts</h2>

A data warehouse environment is formed of multiple artifacts. Please select the artifact type from one of the following in the list: (a) Operational schema, (b) Reconciled schema, (c) ETL workflows, (d) Data warehouse schema, and (e) Queries. 

This step allows querying about connections of a particular artifact.

<form class="form-inline" method="post" action="LineageServlet">
  <div class="form-group">
    <label for="artifacttype">Artifact type:</label>
    <select class="form-control" id="artifacttype" name="artifacttype"
      onChange="toggle()">
      <option value="schema" selected>Schema</option>
      <option value="relation">Relation</option>
      <option value="attribute">Attribute</option>
    </select>
  </div>
  <br>
  <label for="txt_schemaname">Schema name</label>
  <input type="text" class="form-control" name="txt_schemaname" id="txt_schemaname"
    size=20>
  <br>
  <label for="txt_relationname">Relation name</label>
  <input type="text" class="form-control" name="txt_relationname" id="txt_relationname"
    size=20 disabled=true>
  <br>
  <label for="txt_attributename">Attribute name</label>
  <input type="text" class="form-control" name="txt_attributename" id="txt_attributename"
    size=20 disabled=true>
  <br>
  <button type="submit" class="btn btn-default" name="btn_sources">Determine its sources</button>
  <button type="submit" class="btn btn-default" name="btn_impact">Determine its impact</button>
  <div style="color: #FF0000;">${errorMessage}</div>
  <div style="color: 'blue';">${successMessage}</div>
  <label><h1>Run the following command in Neo4j</h1></label>
  <textarea rows="20" cols="100" name="txt_output" id="txt_output">${graphQuery}</textarea>
</form>
B.2 Java servlet code for handling user requests - servlet files

This section shows the code for handling any requests received from any of the four user interfaces (index.jsp, etl.jsp, queries.jsp, lineage.jsp. We present the code in four subsections as follows.

1. SchemaServlet, ETLServlet, QueryServlet, LineageServlet for handling requests received from index.jsp, etl.jsp, queries.jsp, lineage.jsp.

2. SchemaGraphController, ETLGraphController, QueryGraphController for sending commands to Neo4j from within Java to generate graph for schemas, ETL workflows and queries respectively.

3. XMLParser, QueryXMLParser for managing the parsing of XML files corresponding to ETL workflows and queries respectively.

4. Additional helper files

B.2.1 SchemaServlet.java

```java
import java.io.File;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
```
/**
 * Servlet implementation class SchemaServlet
 */
public class SchemaServlet extends HttpServlet {
private static final long serialVersionUID = 1L;

public SchemaServlet() {
super();
}

protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
}

protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {

HttpSession session = request.getSession();
if (session != null) {
session.invalidate();
}
String schema_name = request.getParameter("txt_schemaname");
if (request.getParameter("btn_convertToGraph") != null) {
//clear the textarea containing Cypher query from previous run
session.setAttribute("graphQuery",null);

String arch_layer = request.getParameter("schematype");
System.out.println(schema_name + "," + arch_layer);
// Prerequisite: Points to a location where a folder for each schema would exist
String csvFilesDirPath = "C:\\Users\\usplib\\workspace\\"
+ "ImpactAssessmentProject\\csvfiles\\"
+ schema_name + "\\";
File csvFilesDir = new File(csvFilesDirPath);
if (!csvFilesDir.exists()) {
    session.setAttribute("errorMessage",
"Please export the schema information as CSV files first and try again\n");
} else {
    SchemaGraphController graph = new SchemaGraphController();
    if (SchemaGraphController.getDbConnection() == null) {
        session.setAttribute("errorMessage",
"Could not connect to Neo4j. Make sure that the database is running");
    } else {
    
}
graph.generateGraph(schema_name, arch_layer, "file:///" + csvFilesDirPath);
if (graph.getResultStatus() == null) {
    session.setAttribute("successMessage",
    "Schema successfully transformed to the graph format.");
} else {
    session.setAttribute("errorMessage",
    "Following errors occurred while generating graph:
    + graph.getResultStatus());
}
}
}
response.sendRedirect("index.jsp");

if (request.getParameter("btn_view") != null) {
    String neo4j_query="MATCH (s:Schema)-[r]->(rel:Relation)-[ra]-(a:Attribute) where "+ "s.name='"+session.getAttribute("schema_name")+"'
    + " WITH s, r, rel, ra, a "
    + "OPTIONAL MATCH (a)-[fk]->(fk_a:Attribute) RETURN s, r, rel, ra, a";
    String neo4j_view_query = "Click here to access the graph: http://localhost:7474/
    + "\nExecute the following query:\n    + neo4j_query;
    session.setAttribute("graphQuery", neo4j_view_query);
    response.sendRedirect("index.jsp");
}
if (request.getParameter("btn_next") != null) {
    if (session != null) {
        session.invalidate();
    }
    response.sendRedirect(request.getContextPath() + "/etl.jsp");
}

B.2.2 ETLServlet.java

import java.io.File;
import java.io.IOException;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 * Servlet implementation class IndexPageForwarderServlet
 */
public class ETLServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    String csv_path=null;
    String xmlLocation=null;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public ETLServlet() {
        super();
        // TODO Auto-generated constructor stub
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session=request.getSession();
        // When user clicks 'Convert XML to CSV' button
        if (request.getParameter("btn_convertToCsv") != null){
            /* This is the folder containing XML files
            * Currently files are located in the following two locations:
            * C:\Users\usplib\Desktop\DissertationCode\ETL files sakila_ids\XML
            * C:\Users\usplib\Desktop\DissertationCode\ETL ids-dwh\XML
            */
            String folder_path = request.getParameter("txt_folder");
        }
    }

    HttpSession session=request.getSession();

    // When user clicks 'Convert XML to CSV' button
    if (request.getParameter("btn_convertToCsv") != null){
        /* This is the folder containing XML files
         * Currently files are located in the following two locations:
         * C:\Users\usplib\Desktop\DissertationCode\ETL files sakila_ids\XML
         * C:\Users\usplib\Desktop\DissertationCode\ETL ids-dwh\XML
         */
        String folder_path = request.getParameter("txt_folder");
    }
String etl_layer = request.getParameter("etlLayer");
xmLocation=folder_path;

if(etl_layer.equals("op-ids")){
   //This is the location where csv files will be generated
   csv_path="C:\\Users\usplib\workspace\ImpactAssessmentProject\\csvfiles\\op-ids\\";
   deleteCSVFiles("csvfiles\\op-ids\\");
   generateCsvData(csv_path);
}
else {
   csv_path="C:\\Users\usplib\workspace\\ImpactAssessmentProject\\csvfiles\\ids-dwh\\";
   deleteCSVFiles("csvfiles\\ids-dwh\\");
   generateCsvData(csv_path);
}
generateDimensionLookupUpdateStepCsvData(csv_path+"dimlookupupdate_steps.csv");
session.setAttribute("successMessage","Operation completed successfully");
response.sendRedirect("etl.jsp");
}

// When user clicks 'Convert CSV to Graph' button
if (request.getParameter("btn_convertToGraph") != null){
   session.removeAttribute("successMessage");
   ETLGraphController graph=new ETLGraphController();
   String neo4j_csvPath=new String("file:///").concat(csv_path.replaceAll("\\\","\\\\\\\\";
   graph.generateETLStepNodes(neo4j_csvPath);
   graph.generateRelationships(neo4j_csvPath);
   session.setAttribute("successMessage","Operation completed successfully");
   response.sendRedirect("etl.jsp");
}

// When user clicks 'Next' button
if (request.getParameter("btn_next") != null){
   if(session!=null){
      session.invalidate();
   }
   response.sendRedirect("query.jsp");
}

public void deleteCSVFiles(String foldername){

   File folder = new File("C:\\Users\usplib\workspace\\ImpactAssessmentProject\\"+
   foldername);
   File[] listOffiles = folder.listFiles();
   System.out.println("Number of files to be deleted:"+listOfFiles.length);

   251
for (int i = 0; i < listOfFiles.length; i++) {
    if (listOfFiles[i].isFile()) {
        listOfFiles[i].delete();
    }
}

public void generateCsvData(String csv_path){
    generateOutputStepsCsvData(csv_path+"output_steps.csv");
    generateInputStepsCsvData(csv_path+"input_steps.csv");
    generateDBLookupStepCsvData(csv_path+"dblookup_steps.csv");
    generateDBJoinStepCsvData(csv_path+"dbjoin_steps.csv");
    generateDimensionLookupUpdateStepCsvData(csv_path+"dimlookupupdate_steps.csv");
    generateCombinationLookupUpdateStepCsvData(csv_path+"comblookupupdate_steps.csv");
}

public void generateOutputStepsCsvData(String csvPath){
    XMLParser xmlparser=new XMLParser();
    File folder = new File(xmlLocation);
    File[] listOfFiles = folder.listFiles();
    for (int i = 0; i < listOfFiles.length; i++) {
        if (listOfFiles[i].isFile()) {
            List<OutputStep> outputSteps=
                xmlparser.parseXMLForOutputSteps(xmlLocation+"\\"+listOfFiles[i].getName());
            ETLExcelExporter.generateOutputStepsCsvFile(csvPath, outputSteps);
        }
    }
}

public void generateInputStepsCsvData(String csvPath){
    XMLParser xmlparser=new XMLParser();
    File folder = new File(xmlLocation);
    File[] listOfFiles = folder.listFiles();
    for (int i = 0; i < listOfFiles.length; i++) {
        if (listOfFiles[i].isFile()) {
            //One transformation can be reading from multiple data sources
            List<TableInputStep> inputSteps=
                xmlparser.parseXMLForInputSteps(xmlLocation+"\\"+listOfFiles[i].getName());
            ETLExcelExporter.generateInputStepsCsvFile(csvPath, inputSteps);
        }
    }
}
public void generateDBLookupStepCsvData(String csvPath){

XMLParser xmlparser=new XMLParser();
File folder = new File(xmlLocation);
File[] listOfFiles = folder.listFiles();
for (int i = 0; i < listOfFiles.length; i++) {
    if (listOfFiles[i].isFile()) {
        //One transformation can be reading from multiple data sources
        List<DBLookupStep> dblookupsteps = 
xlparser.parseXMLForDBLookupSteps(xmlLocation+"\\"+
                    listOfFiles[i].getName());
ETLExcelExporter.generateDBLookupCsvFile(csvPath, dblookupsteps);
    }
}
}

public void generateDimensionLookupUpdateStepCsvData(String csvPath){

XMLParser xmlparser=new XMLParser();
File folder = new File(xmlLocation);
File[] listOfFiles = folder.listFiles();
for (int i = 0; i < listOfFiles.length; i++) {
    if (listOfFiles[i].isFile()) {
        List<DimLookupUpdateStep> dimlookupupdatesteps = 
xlparser.parseXMLForDimLookupUpdateSteps(xmlLocation+"\\"+
                    listOfFiles[i].getName());
ETLExcelExporter.generateDimLookupUpdateCsvFile(csvPath, dimlookupupdatesteps);
    }
}
}

public void generateCombinationLookupUpdateStepCsvData(String csvPath){

XMLParser xmlparser=new XMLParser();
File folder = new File(xmlLocation);
File[] listOfFiles = folder.listFiles();
for (int i = 0; i < listOfFiles.length; i++) {
    if (listOfFiles[i].isFile()) {
        System.out.println("**********File " + listOfFiles[i].getName());
        List<CombinationLookupUpdateStep> combinationlookupupdatesteps = 
xlparser.parseXMLForCombinationLookupUpdateSteps(xmlLocation+"\\"+
                    listOfFiles[i].getName());
ETLExcelExporter.generateCombinationLookupUpdateCsvFile(csvPath, combinationlookupupdatesteps);
    }
}
}
public void generateDBJoinStepCsvData(String csvPath) {

    XMLParser xmlParser = new XMLParser();
    File folder = new File(xmlLocation);
    File[] listOfFiles = folder.listFiles();
    for (int i = 0; i < listOfFiles.length; i++) {
        if (listOfFiles[i].isFile()) {
            System.out.println("**********File " + listOfFiles[i].getName());
            // One transformation can be reading from multiple data sources
            List<DBJoinStep> dbjoinsteps =
                xmlParser.parseXMLForDBJoinSteps(xmlLocation + "\\" +
                listOfFiles[i].getName());
            ETLExcelExporter.generateDBJoinCsvFile(csvPath, dbjoinsteps);
        }
    }
}

B.2.3 QueryServlet.java

```java
import java.io.File;
import java.io.IOException;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 */
```
* Servlet implementation class QueryServlet
*/
public class QueryServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    String csv_path=null;
    String neo4j_read_path=null;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public QueryServlet() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    // TODO Auto-generated method stub
}

    /**
     * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        HttpSession session=request.getSession();
        if (request.getParameter("btn_convertToCsv")!=null){
            String src_folder = request.getParameter("txt_folder");
            csv_path="C:\Users\usplib\workspace\ImpactAssessmentProject\csvfiles\queries.csv";
            generateCSVFile(src_folder,csv_path);
            session.setAttribute("successMessage", "CSV file generated successfully");
            response.sendRedirect("query.jsp");
        }

        if (request.getParameter("btn_convertToGraph") != null){
            csv_path="C:\Users\usplib\workspace\ImpactAssessmentProject\csvfiles\queries.csv";
            neo4j_read_path=new String("file://").concat(csv_path.replaceAll("\\",\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\...
response.sendRedirect("query.jsp");
}
// When user clicks 'Next' button
if (request.getParameter("btn_next") != null)
{
if (session != null)
{
    session.invalidate();
}
response.sendRedirect("lineage.jsp");
}

// When the user clicks 'View the graph' button
if (request.getParameter("btn_view") != null)
{
    String neo4j_view_query = "Click here to access the graph: http://localhost:7474/
    + "\n    + "MATCH (n:Attribute)-[r]->(q:Query) RETURN n, r, q";
    session.setAttribute("graphQuery", neo4j_view_query);
    response.sendRedirect("query.jsp");
}

public void generateCSVFile(String src_folder, String target)
{
    deleteCSVFileIfExists();
    QueryXMLParser xmlparser = new QueryXMLParser();
    List<Query> queryObjs = xmlparser.parseXML(src_folder + "\queries-xml.xml");
    QueryExcelExporter.generateCsvFile(target, queryObjs);
}

public void deleteCSVFileIfExists()
{
    File f = new File("C:\\Users\\usplib\\workspace\\ImpactAssessmentProject\\" + "csvfiles\\queries.csv");
    if (f.isFile())
        f.delete();
}

B.2.4 LineageServlet.java
import java.io.IOException;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import java.io.IOException;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 * Servlet implementation class LineageServlet
 */
public class LineageServlet extends HttpServlet {
private static final long serialVersionUID = 1L;
public LineageServlet() {
    super();
}

protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
}

/**
 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
 * response)
 */

protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
String neo4j_query = null;
HttpSession session = request.getSession();
StringBuffer lblheader=new StringBuffer("Run the following command in Neo4j
");
}
String schema_name = request.getParameter("txt_schemaname");
String relation_name = request.getParameter("txt_relationname");
String attribute_name = request.getParameter("txt_attributename");
String artifact_type = request.getParameter("artifacttype");

if (request.getParameter("btn_sources") != null) {
    String path = null;
    String query = null;
    ServletContext context = getServletContext();
    switch (artifact_type) {
        case "schema":
            if (isOperationalSchema(schema_name)) {
                query = "This is an operational schema. " + "ETL transformations are not responsible for populating it.";
            } else if (isReconciledSchema(schema_name)) {
                path = context.getRealPath("cypher-ids-schema-src.txt");
                query = SourceAnalysis.getCypherQuery(schema_name, path);
            } else if (isWarehouseSchema(schema_name)) {
                path = context.getRealPath("cypher-dwh-schema-src.txt");
                query = SourceAnalysis.getCypherQuery(schema_name, path);
            }
            break;
        case "relation":
            if (isOperationalSchema(schema_name)) {
                query = "MATCH (a:Attribute {isKey:'YES'})-[r]->(t:Attribute {schema:'" + schema_name + ", relation:'" + relation_name + "} RETURN a,r";
            }
            if (isReconciledSchema(schema_name)) {
                query = "MATCH (a:Attribute)-[r*1..3]->(t:Attribute {schema:'" + schema_name + ", relation:'" + relation_name + "} where a.schema='sakila_op1' OR a.schema='sakila_op2' RETURN a,r";
            }
            if (isWarehouseSchema(schema_name)) {
query = "MATCH (a:Attribute)-[r*1..6]->(t:Attribute {schema:\"" + schema_name + "\", relation:\"" + relation_name + "\"}) where a.schema='sakila_op1' OR a.schema='sakila_op2' RETURN a,r";
}

break;

switch (request.getParameter("btn_type") != null) {
    case "attribute":
        if (isOperationalSchema(schema_name)) {
            query = "MATCH (a:Attribute {isKey:'YES'})-[r]->(t:Attribute {name:\"" + attribute_name + "\", schema:\"" + schema_name + "\", relation:\"" + relation_name + "\"}) RETURN a,r";
        }
        if (isReconciledSchema(schema_name)) {
            query = "MATCH (a:Attribute)-[r*1..3]->(t:Attribute {name:\"" + attribute_name + "\", schema:\"" + schema_name + "\", relation:\"" + relation_name + "\"}) where a.schema='sakila_op1' OR a.schema='sakila_op2' RETURN a,r";
        }
        if (isWarehouseSchema(schema_name)) {
            query = "MATCH (a:Attribute)-[r*1..6]->(t:Attribute {name:\"" + attribute_name + "\", schema:\"" + schema_name + "\", relation:\"" + relation_name + "\"}) where a.schema='sakila_op1' OR a.schema='sakila_op2' RETURN a,r";
        }
        break;
        default:
            break;
}

session.setAttribute("graphQuery", query);
response.sendRedirect("lineage.jsp");

else if (request.getParameter("btn_impact") != null) {

259
String path = null;
String query = null;
ServletContext context = getServletContext();
StringBuffer query_from_file;
int schema_startIndex, schema_endIndex, relation_startIndex,
relation_endIndex, attribute_startIndex, attribute_endIndex;
switch (artifact_type) {
    case "schema":
        if (isOperationalSchema(schema_name))
            path = context.getRealPath("cypher-opr-schema-impact.txt");
        else if (isReconciledSchema(schema_name))
            path = context.getRealPath("cypher-ids-schema-impact.txt");
        else if (isWarehouseSchema(schema_name))
            path = context.getRealPath("cypher-dwh-schema-impact.txt");

        query_from_file= new StringBuffer(SourceAnalysisAttribute.getCypherQuery
        (schema_name,path));
        schema_startIndex=query_from_file.indexOf("<name of the schema>");
        schema_endIndex=schema_startIndex + 20;
        query_from_file = (query_from_file.replace(schema_startIndex,
        schema_endIndex, schema_name));
        query=query_from_file.toString();
        break;

    case "relation":
        if (isOperationalSchema(schema_name))
            path = context.getRealPath("cypher-opr-relation-impact.txt");

        if (isReconciledSchema(schema_name))
            path = context.getRealPath("cypher-ids-relation-impact.txt");

        if (isWarehouseSchema(schema_name))
            path = context.getRealPath("cypher-dwh-relation-impact.txt");

        query_from_file= new StringBuffer(SourceAnalysisAttribute.getCypherQuery
        (schema_name,path));
        relation_startIndex=query_from_file.indexOf("<name of the relation>");
        relation_endIndex=relation_startIndex + 22;
        query_from_file = (query_from_file.replace(relation_startIndex,
        relation_endIndex, relation_name));

        schema_startIndex=query_from_file.indexOf("<name of the schema>");
        schema_endIndex=schema_startIndex + 20;

        break;
query_from_file = (query_from_file.replace(schema_startIndex, schema_endIndex, schema_name));

query=query_from_file.toString();
break;

case "attribute":
    if (isOperationalSchema(schema_name)) {
        path = context.getRealPath("cypher-opr-attribute-impact.txt");
    } else if (isReconciledSchema(schema_name)) {
        path = context
            .getRealPath("cypher-ids-attribute-impact.txt");
    } else if (isWarehouseSchema(schema_name)) {
        path = context
            .getRealPath("cypher-dwh-attribute-impact.txt");
    }

    query_from_file= new StringBuffer(SourceAnalysisAttribute.getCypherQuery
        (schema_name,path));
    relation_startIndex=query_from_file.indexOf("<name of the relation>");
    relation_endIndex=relation_startIndex + 22;
    query_from_file = (query_from_file.replace(relation_startIndex, relation_endIndex, relation_name));

    schema_startIndex=query_from_file.indexOf("<name of the schema>");
    schema_endIndex=schema_startIndex + 20;
    query_from_file = (query_from_file.replace(schema_startIndex, schema_endIndex, schema_name));

    attribute_startIndex=query_from_file.indexOf("<name of the attribute>"); attribute_endIndex=attribute_startIndex + 23;
    query_from_file = (query_from_file.replace(attribute_startIndex, attribute_endIndex, attribute_name));

    query=query_from_file.toString();
break;

default:
    break;
}

if (relation_name!=null & attribute_name!=null)
B.3 Java code for generating CSV files

This section presents the code for two specific files - (1) ETLExcelExporter, (2) QueryExcelExporter. The code in this section is responsible for writing data to the CSV files.

B.3.1 ETLExcelExporter.java

```java
import java.io.FileWriter;
```
import java.io.File;
import java.io.IOException;
import java.util.List;

public class ETLExcelExporter {

public static void generateOutputStepsCsvFile(String sFileName, List<OutputStep> steps) {
    try {
        boolean fileExists = false;
        System.out.println("Exporting to Excel");
        File f = new File(sFileName);
        if (f.exists())
            fileExists = true;

        FileWriter writer = new FileWriter(sFileName, true);
        if (!fileExists) {
            writer.append("TransformationName");
            writer.append(",");
            writer.append("StepName");
            writer.append(",");
            writer.append("StepType");
            writer.append(",");
            writer.append("DatabaseName");
            writer.append(",");
            writer.append("TableName");
            writer.append(",");
            writer.append("AttributeName");
            writer.append("
");  
        }

        for (OutputStep stepObj : steps) {
            System.out.println("ExcelExporter:" + stepObj);

            for (int index = 0; index < stepObj.getAttributes().size(); index++) {
                writer.append(stepObj.getTransName());
                writer.append(",");
                writer.append(stepObj.getStepName());
                writer.append(",");
                writer.append(stepObj.getStepType());
                writer.append(",");
                writer.append(stepObj.getDbName());
                writer.append(",");
                writer.append(stepObj.getTableName());
                writer.append(",");
                writer.append(stepObj.getAttributeName());
                writer.append("
");  
            }
        }
    }
}
writer.append(stepObj.getTableName());
writer.append(',');
writer.append(stepObj.getAttributes().get(index));
writer.append('
');
}
}
writer.flush();
writer.close();
} catch (IOException e) {
e.printStackTrace();
}

public static void generateInputStepsCsvFile(String sFileName, List<TableInputStep> steps) {
try {
boolean fileExists = false;
System.out.println("Exporting to Excel");
File f = new File(sFileName);
if (f.exists())
fileExists = true;

FileWriter writer = new FileWriter(sFileName, true);
if (!fileExists) {
writer.append("TransformationName");
writer.append(',');
writer.append("StepName");
writer.append(',');
writer.append("DatabaseName");
writer.append(',');
writer.append("SQL");
writer.append('
');
}
for (TableInputStep stepObj : steps) {
// System.out.println("ExcelExporter:"+stepObj);
writer.append(stepObj.getTransName());
writer.append(',');
writer.append(stepObj.getStepName());
writer.append(',');
writer.append(stepObj.getStepName());
writer.append(',');
writer.append(stepObj.getDbName());
writer.append(',,');

writer.append(stepObj.getSql() .replaceAll("[	
\r\s]", " "));
writer.append('
');

}

writer.flush();
writer.close();
} catch (IOException e) {
e.printStackTrace();
}

public static void generateDBJoinCsvFile(String sFileName, List<DBJoinStep> steps) {
try {
boolean fileExists = false;
System.out.println("Exporting to Excel");
File f = new File(sFileName);
if (f.exists()) {
fileExists = true;
}

FileWriter writer = new FileWriter(sFileName, true);
if (!fileExists) {

writer.append("TransformationName");
writer.append(',');
writer.append("StepName");
writer.append(',');
writer.append("StepType");
writer.append(',');
writer.append("DatabaseName");
writer.append(',');
writer.append("SQL");

writer.append('
');
}

for (DBJoinStep stepObj : steps) {

265
System.out.println("ExcelExporter:"+stepObj);

writer.append(stepObj.getTransName());
writer.append(",");
writer.append(stepObj.getStepName());
writer.append(",");
writer.append("DBJoin");
writer.append(",");

writer.append(stepObj.getDbName());
writer.append(",");

writer.append(stepObj.getSql().replaceAll(\[\t\n\r\s\], " ");
writer.append('
');

writer.flush();
writer.close();
} catch (IOException e) {
    e.printStackTrace();
}

public static void generateDBLookupCsvFile(String sFileName,
List<DBLookupStep> steps) {
    try {
        boolean fileExists = false;
        System.out.println("Exporting to Excel");
        File f = new File(sFileName);
        if (f.exists())
            fileExists = true;

        FileWriter writer = new FileWriter(sFileName, true);
        if (!fileExists) {
            writer.append("TransformationName");
            writer.append(",");
            writer.append("StepName");
            writer.append(",");
            writer.append("StepType");
            writer.append(",");
            writer.append("DatabaseName");
            writer.append(",");

        }

        writer.append("TransformationName");
        writer.append(",");
        writer.append("StepName");
        writer.append(",");
        writer.append("StepType");
        writer.append(",");
        writer.append("DatabaseName");
        writer.append(",");

    } catch (IOException e) {
        e.printStackTrace();
    }
}
for (DBLookupStep stepObj : steps) {
    System.out.println("ExcelExporter:" + stepObj);
    for (int index = 0; index < stepObj.getAttributes().size(); index++) {
        writer.append(stepObj.getTransName());
        writer.append(',');
        writer.append(stepObj.getStepName());
        writer.append(',');
        writer.append("DBLookup");
        writer.append(',');
        writer.append(stepObj.getDbName());
        writer.append(',');
        writer.append(stepObj.getTableName());
        writer.append(',');
        writer.append(stepObj.getAttributes().get(index));
        writer.append('
');
    }
}
writer.flush();
writer.close();
} catch (IOException e) {
    e.printStackTrace();
}
}

public static void generateDimLookupUpdateCsvFile(String sFileName, List<DimLookupUpdateStep> steps) {
    try {
        boolean fileExists = false;
        System.out.println("Exporting to Excel");
        File f = new File(sFileName);
        if (f.exists())
            fileExists = true;

        FileWriter writer = new FileWriter(sFileName, true);
        if (!fileExists) {
            writer.append("TableName");
            writer.append(',');
            writer.append("AttributeName");
            writer.append('
');
        }

        for (DBLookupStep stepObj : steps) {
            System.out.println("ExcelExporter:" + stepObj);
            for (int index = 0; index < stepObj.getAttributes().size(); index++) {
                writer.append(stepObj.getTransName());
                writer.append(',');
                writer.append(stepObj.getStepName());
                writer.append(',');
                writer.append("DBLookup");
                writer.append(',');
                writer.append(stepObj.getDbName());
                writer.append(',');
                writer.append(stepObj.getTableName());
                writer.append(',');
                writer.append(stepObj.getAttributes().get(index));
                writer.append('
');
            }
        }
        writer.flush();
        writer.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
writer.append("TransformationName");
writer.append(',');
writer.append("StepName");
writer.append(',');
writer.append("StepType");
writer.append(',');
writer.append("DatabaseName");
writer.append(',');
writer.append("TableName");
writer.append(',');
writer.append("AttributeName");
writer.append(',');
writer.append("isUpdateStep");
writer.append('
');
}

for (DimLookupUpdateStep stepObj : steps) {
System.out.println("\tExcelExporter:" + stepObj);

for (int index = 0; index < stepObj.getAttributes().size(); index++) {

    writer.append(stepObj.getTransName());
    writer.append(',');
    writer.append(stepObj.getStepName());
    writer.append(',');
    writer.append("DimensionLookup");
    writer.append(',');
    writer.append(stepObj.getDbName());
    writer.append(',');
    writer.append(stepObj.getTableName());
    writer.append(',');
    writer.append(stepObj.getAttributes().get(index));
    writer.append(',');
    if (stepObj.isUpdateStep() == true)
        writer.append("Update");
    else
        writer.append("Lookup");
    writer.append('
');
}
}
writer.flush();
writer.close();
public static void generateCombLookupUpdateCsvFile(String sFileName, List<CombinationLookupUpdateStep> steps) {
    try {
        boolean fileExists = false;
        System.out.println("Exporting to Excel");
        File f = new File(sFileName);
        if (f.exists())
            fileExists = true;

        FileWriter writer = new FileWriter(sFileName, true);
        if (!fileExists) {
            writer.append("TransformationName");
            writer.append(',');
            writer.append("StepName");
            writer.append(',');
            writer.append("StepType");
            writer.append(',');
            writer.append("DatabaseName");
            writer.append(',');
            writer.append("TableName");
            writer.append(',');
            writer.append("AttributeName");
            writer.append('n');
        }

        for (CombinationLookupUpdateStep stepObj : steps) {
            System.out.println("\tExcelExporter:" + stepObj);

            for (int index = 0; index < stepObj.getAttributes().size(); index++) {
                writer.append(stepObj.getTransName());
                writer.append(',');
                writer.append(stepObj.getStepName());
                writer.append(',');
                writer.append(stepObj.getStepType());
                writer.append(',');
                writer.append(stepObj.getDbName());
                writer.append(',');
                writer.append(stepObj.getTableName());
                writer.append(',');
                writer.append(stepObj.getAttributeName());
                writer.append(',');
                writer.append(stepObj.getTableName());
            }
        }
    }

    catch (IOException e) {
        e.printStackTrace();
    }
}

writer.append(',,
writer.append(stepObj.getAttributes().get(index));
writer.append('n

}

writer.flush();
writer.close();
} catch (IOException e) {
e.printStackTrace();
}
}

B.3.2  QueryExcelExporter.java

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;

public class QueryExcelExporter {

public static void generateCsvFile(String sFileName, List<Query> queryObjs) {
try {
boolean fileExists = false;
System.out.println("Exporting to Excel");
File f = new File(sFileName);
if (f.exists()) {
System.out.println("File exists\n" + f.getAbsolutePath());
fileExists = true;
} else {
System.out.println("File does not exist \n");
}

FileWriter writer = new FileWriter(sFileName, true);
if (!fileExists) {

writer.append("QueryLabel",
writer.append(",
writer.append("AttributeName",

writer.append(','
writer.append('n

}
writer.append(',');
writer.append("RelationName");
writer.append(',');
writer.append("DatabaseName");
writer.append(' 
');
}

for (Query q : queryObjs) {
System.out.println("ExcelExporter:" + q);

String queryTitle = q.getQueryLabel();
writer.append(queryTitle.substring(queryTitle.lastIndexOf('\\') + 1,
queryTitle.lastIndexOf('.')));
writer.append(',');
writer.append(q.getAttribute().toLowerCase());
writer.append(',');
writer.append(q.getRelationName().toLowerCase());
writer.append(',');
writer.append(q.getDbName().toLowerCase());
writer.append(' 
');
}

writer.flush();
writer.close();
}

B.4 Java code for generating Neo4j graph

This section presents the code for three specific files - (1) SchemaGraphController, (2) ETLGraphController, and (3) QueryGraphController. Each of these code files read the csv files and use Neo4j API within Java to submit Cypher queries to the Neo4j interface.
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.Transaction;
import org.neo4j.jdbc.Driver;
import org.neo4j.rest.graphdb.RestAPI;

public class SchemaGraphController {

static Connection dbConnection;
String resultStatus = null;

public static Connection getDbConnection() {
return dbConnection;
}

public static void setDbConnection(Connection dbConnection) {
SchemaGraphController.dbConnection = dbConnection;
}

public String getResultStatus() {
return resultStatus;
}

public void setResultStatus(String resultStatus) {
this.resultStatus = resultStatus;
}

public SchemaGraphController() {
try {
Class.forName("org.neo4j.jdbc.Driver");
final Driver driver = new Driver(); // org.neo4j.jdbc.Driver
final Properties props = new Properties();
props.put("user", "neo4j");
props.put("password", "neo4jdb675");
}
String url = "jdbc:neo4j://localhost:7474/";
dbConnection = driver.connect(url, props);

} catch (ClassNotFoundException ex) {
System.out
.println("Class not found exception in GraphController class constructor");
} catch (SQLException ex) {
System.out
.println("SQLException in GraphController class constructor");
} catch (Exception ex) {
System.out
.println("Exception in GraphController class constructor");
}

public void generateGraph(String dbname, String architectureLayer,
String path) {

System.out.println("Generating graph for database:" + dbname
+ " in the " + architectureLayer
+ "\n********************************************************************************n");

generateNodes(path, dbname, architectureLayer);
if (getResultStatus() == null)
generateEdges(path, dbname);

}

public void generateNodes(String path, String dbname,
String architectureLayer) {
generateSchemaNodes(path, dbname, architectureLayer);
if (getResultStatus() == null)
generateRelationNodes(path);
if (getResultStatus() == null)
generateAttributeNodes(path);

}

public void generateEdges(String path, String dbname) {
generateSchemaRelationRelationships(path);
if (getResultStatus() == null)
generateRelationAttributeRelationships(path);
if (getResultStatus() == null)
generateEntityintegrityRelationships(path);
if (getResultStatus() == null)
generateReferentialIntegrityRelationships(path, dbname);
}

public void generateSchemaNodes(String excelfilePath, String dbname, String architectureLayer) {
    try {
        Statement stmt = dbConnection.createStatement();
        System.out.println("********Generating schema nodes**********");
        ResultSet rs = stmt.executeQuery("MERGE(m:Schema{name:'" + dbname + "',type:'" + architectureLayer + ":})");
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
        resultStatus = "Error occurred in generating schema node\n";
    }
}

public void generateRelationNodes(String excelfilePath) {
    try {
        Statement stmt = dbConnection.createStatement();
        System.out.println("********Generating relation nodes**********");
        ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM " + excelfilePath + "Relations.csv" AS line
            "MERGE (r:Relation{name:line.table_name, schema:line.schema_name})");
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
        resultStatus = "Error occurred in generating relation nodes\n";
    }
}

public void generateAttributeNodes(String excelfilePath) {
    try {
        Statement stmt = dbConnection.createStatement();
        System.out.println("********Generating attribute nodes**********");
        ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM " + excelfilePath + "Attributes.csv" AS line
            "MERGE (a:Attribute{name: line.column_name, datatype: line.data_type, "
            + "relation:line.table_name, schema: line.table_schema, isKey: line.is_Key})");
    } catch (Exception ex) {
System.out.println(ex.getMessage());
resultStatus = "Error occurred in generating attribute node\n";
}
}

public void generateEntityintegrityRelationships(String excelfilePath) {
try {
Statement stmt = dbConnection.createStatement();
{
System.out.println("Generating entity integrity relationships...");
ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM " + excelfilePath + " Attributes.csv" AS line " + "MATCH (src:Attribute{name: line.column_name, datatype: line.data_type, " + "relation:line.table_name, schema: line.table_schema, isKey: line.is_Key})" + "MATCH (to:Relation {name: line.table_name, schema: line.table_schema})" + " where src.relation=to.name and src.schema=to.schema and src.isKey='YES' " + " MERGE (src)-[:Impacts {type:'PrimaryKey'}]->(to);";
}
} catch (Exception ex) {
System.out.println(ex.getMessage());
resultStatus = "Error occurred in generating primary key relationships\n";
}
}

public void generateRelationAttributeRelationships(String excelfilePath) {
try {
Statement stmt = dbConnection.createStatement();
{
System.out.println("********Generating relation-attribute relationships********");
ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM " + excelfilePath + " Attributes.csv" AS line " + "MATCH (to:Attribute{name: line.column_name, datatype: line.data_type, " + "relation:line.table_name, schema: line.table_schema, isKey: line.is_Key})" + "MATCH (src:Relation {name: line.table_name, schema: line.table_schema})" + " where src.name=to.relation and src.schema=to.schema " + " MERGE (src)-[:Impacts]->(to);";
}
public void generateReferentialIntegrityRelationships(String excelfilePath, String dbname) {
    try {
        Statement stmt = dbConnection.createStatement();
        System.out.println("********Generating foreign key relationships**********");
        ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM ". "ReferentialIntegrityConstraints.csv" AS line "+ "MATCH (to:Attribute{name: line.attr_name, relation: line.relation_name, "+ "schema: "+ dbname+ "}))"+
                " MATCH (src:Attribute {name: line.referenced_column_name, "+ "relation: line.referenced_relation_name, schema:'"+
                + dbname+ "}))"+
                " MERGE (src)-[:Impacts {type:'ForeignKey'}]->(to)";
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
        resultStatus = "Error occurred in generating foreign key relationships\n";
    }
}

public void generateSchemaRelationRelationships(String excelfilePath) {
    try {
        Statement stmt = dbConnection.createStatement();
        System.out.println("********Generating relation-schema relationships**********");
        ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM ". "Relations.csv" AS line MATCH (src:Schema {name: line.schema_name}) "+
                "MATCH (to:Relation {name: line.table_name, schema:line.schema_name})"+
                " where src.name=to.schema MERGE (src)-[:Impacts]->(to)";
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
        resultStatus = "Error occurred in generating relation-schema relationships\n";
    }
}
B.4.2 ETLGraphController.java

```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Transaction;
import org.neo4j.jdbc.Driver;

public class ETLGraphController {
    static Connection con;
    boolean resultETLSteps = false;

    public ETLGraphController() {
        try {
            Class.forName("org.neo4j.jdbc.Driver");
            final Driver driver = new Driver(); // org.neo4j.jdbc.Driver
            final Properties props = new Properties();
            props.put("user", "neo4j");
            props.put("password", "neo4jdb675");
            String url = "jdbc:neo4j://localhost:7474/";
            con = driver.connect(url, props);
        } catch (ClassNotFoundException ex) {
            System.out.println("Class not found exception in GraphController class constructor");
        } catch (SQLException ex) {
            System.out.println("SQLException in GraphController class constructor");
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
            resultStatus = "Error occurred in generating schema-relation relationships\n";
        }
    }
}
```
public void generateETLStepNodes(String csvDataSource) {
    generateInputStepNodes(csvDataSource, etl_layer);
    generateOutputStepNodes(csvDataSource, etl_layer);
    generateDBLookupStepNodes(csvDataSource, etl_layer);
    generateDBJoinStepNodes(csvDataSource, etl_layer);
    generateDimLookupUpdateStepNodes(csvDataSource, etl_layer);
    generateCombinationLookupUpdateStepNodes(csvDataSource, etl_layer);
}

public void generateRelationships(String csvDataSource) {
    generateReconciledToETLDependencyRelationships(csvDataSource);
    generateETLToDWHDependencyRelationships(csvDataSource);
    generateDBLookupDependencyRelationships(csvDataSource);
    generateDBJoinDependencyRelationships(csvDataSource);
    generateDimLookupUpdateDependencyRelationships(csvDataSource);
    generateCombinationLookupUpdateDependencyRelationships(csvDataSource);
    generateInputToOutputStepsRelationships(csvDataSource);
    generateRelationToETLStepRelationships(csvDataSource);
    generateETLStepToRelationRelationships(csvDataSource);
    generateSchemaToETLStepRelationships(csvDataSource);
    generateETLStepToSchemaRelationships(csvDataSource);
}

public void generateInputStepNodes(String csvDataSource, String etl_layer) {
    try {
        Statement stmt = con.createStatement();
        {
            ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM " + csvDataSource + "input_steps.csv" AS line "+ csvDataSource + "" + "input_steps.csv" AS line " + "MERGE (st:Step{name: line.StepName, transname: line.TransformationName," + "steptype:"+ "TableInput"," + "layer:"+etl_layer+"})|");
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
    public void generateOutputStepNodes(String csvDataSource) {
        try {
            
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
Statement stmt = con.createStatement();
{

ResultSet rs = stmt
.executeQuery("LOAD CSV WITH HEADERS FROM \
+ csvDataSource
+ "output_steps.csv" AS line 
+ "MERGE (st:Step{name: line.StepName, transname: line.TransformationName, 
+ "steptype: line.StepType, layer:'"+etl_layer+"'})");

}
} catch (Exception ex) {
System.out.println(ex.getMessage());
}

public void generateDBLookupStepNodes(String csvDataSource) {
try {
Statement stmt = con.createStatement();
ResultSet rs = stmt
.executeQuery("LOAD CSV WITH HEADERS FROM \
+ csvDataSource
+ "dblookup_steps.csv" AS line 
+ "MERGE (st:Step{name: line.StepName, transname: line.TransformationName, 
+ "steptype: line.StepType, layer:'"+etl_layer+"'})");

} catch (Exception ex) {
System.out.println(ex.getMessage());
}

}

public void generateDBJoinStepNodes(String csvDataSource) {
try {
Statement stmt = con.createStatement();
ResultSet rs = stmt
.executeQuery("LOAD CSV WITH HEADERS FROM \
+ csvDataSource
+ "dbjoin_steps.csv" AS line 
+ "MERGE (st:Step{name: line.StepName, transname: line.TransformationName, 
+ "steptype: line.StepType, layer:'"+etl_layer+"'})");

} catch (Exception ex) {
System.out.println(ex.getMessage());
}
public void generateDimLookupUpdateStepNodes(String csvDataSource) {
    try {
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM " + csvDataSource + ", dimlookupupdate_steps.csv" AS line "+ "MERGE (st:Step{name: line.StepName, transname: line.TransformationName, "stepname: line.StepName, transname: line.TransformationName, " + "steptype: line.StepType, update_lookup:line.isUpdateStep, layer:\"+etl_layer+\\}])");
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}

public void generateCombinationLookupUpdateStepNodes(String csvDataSource) {
    try {
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM " + csvDataSource + ", comblookupupdate_steps.csv" AS line "+ "MERGE (st:Step{name: line.StepName, transname: line.TransformationName, "stepname: line.StepName, transname: line.TransformationName, " + "steptype: line.StepType, layer:\"+etl_layer+\\}])");
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}

public void generateDimLookupUpdateDependencyRelationships(String csvDataSource) {
    try {
        Statement stmt = con.createStatement();
        { 
            ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM " + csvDataSource + ", dimlookupupdate_steps.csv" AS line ")
+
+
+
+
+

" MATCH (a:Attribute{name: (line.AttributeName), schema:(line.DatabaseName), "
"relation: line.TableName})"
" MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "
"steptype: line.StepType}) WHERE st.update_lookup=\’Update\’"
" MERGE (st)-[:Impacts]->(a)");

ResultSet rs2 = stmt
.executeQuery("LOAD CSV WITH HEADERS FROM \""
+ csvDataSource
+ "dimlookupupdate_steps.csv\" AS line "
+ "MATCH (a:Attribute{name: (line.AttributeName), schema:(line.DatabaseName), "
+ "relation: line.TableName})"
+ "MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "
+ "steptype: line.StepType}) WHERE st.update_lookup=\’Lookup\’"
+ " MERGE (a)-[:Impacts]->(st)");
}
} catch (Exception ex) {
System.out.println(ex.getMessage());
}
}
public void generateCombinationLookupUpdateDependencyRelationships(
String csvDataSource) {
try {
Statement stmt = con.createStatement();
{
ResultSet rs = stmt
.executeQuery("LOAD CSV WITH HEADERS FROM \""
+ csvDataSource
+ "comblookupupdate_steps.csv\" AS line "
+ " MATCH (a:Attribute{name: (line.AttributeName), schema:(line.DatabaseName), "
+ "relation: line.TableName})"
+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "
+ "steptype: line.StepType})"
+ " MERGE (st)-[:Impacts]->(a)");
}
} catch (Exception ex) {
System.out.println(ex.getMessage());
}
}
public void generateETLToDWHDependencyRelationships(String csvDataSource) {
try {
Statement stmt = con.createStatement();

281


ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM ".csvDataSource
+ csvDataSource
+ "output_steps.csv" AS line "+ " MATCH (a:Attribute{name: (line.AttributeName), schema:(line.DatabaseName), "
+ "relation: line.TableName})" + " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "
+ "steptype: line.StepType})"
+ " MERGE (st)-[:Impacts]->(a);"
} catch (Exception ex) {
System.out.println(ex.getMessage());
}

public void generateDBLookupDependency Relationships(String csvDataSource) {
try {
Statement stmt = con.createStatement();
{
ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM ".csvDataSource
+ csvDataSource
+ "dblookup_steps.csv" AS line "+ " MATCH (a:Attribute{name: (line.AttributeName), schema:(line.DatabaseName), "
+ "relation: line.TableName})"
+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "
+ "steptype: line.StepType})"
+ " MERGE (a)-[:Impacts]->(st);"
} catch (Exception ex) {
System.out.println(ex.getMessage());
}

public void generateDBJoinDependency Relationships(String csvDataSource) {
try {
Statement stmt = con.createStatement();
{
ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM ".csvDataSource
+ csvDataSource

public void generateReconciledToETLDependencyRelationships(String csvDataSource) {
    try {
        Statement stmt = con.createStatement();
        { 
            ResultSet rs = stmt.executeUpdate("LOAD CSV WITH HEADERS FROM " "+ csvDataSource "+ "dbjoin_steps.csv\" AS line \" "+ " MATCH (a:Attribute{name: line.AttributeName, schema:(line.DatabaseName), " "+ "relation: line.TableName})" "+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, " "+ "steptype: line.StepType})" "+ " MERGE (a)-[:Impacts]->(st)"; 
        } 
        catch (Exception ex) { 
            System.out.println(ex.getMessage()); 
        } 
    } 
}

public void generateInputToOutputStepsRelationships(String csvDataSource) {
    try {
        Statement stmt = con.createStatement();
        { 
            ResultSet rs = stmt.executeUpdate("MATCH (input:Step) where input.steptype='" "+ "TableInput\" OR input.steptype='" "+ "DBLookup\" OR input.steptype='" 
+ " TableInput\" + ")" "+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, " "+ "steptype:)':" "+ "TableInput\" + ")" "+ " MERGE (a)-[:Impacts]->(st)"; 
        } 
        catch (Exception ex) { 
            System.out.println(ex.getMessage()); 
        } 
    } 
}
```java
public void generateRelationToETLStepRelationships(String csvDataSource) {
    try {
        Statement stmt = con.createStatement();
        ResultSet rs = stmt
                .executeQuery("LOAD CSV WITH HEADERS FROM "
                        + csvDataSource
                        + "input_steps.csv" AS line 
                        + " MATCH (a:Relation{name: line.TableName, schema:line.DatabaseName})
                        + " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, steptype: 'TableInput'})
                        + " MERGE (a)-[:Impacts]->(st)\n    ");
        ResultSet rs2 = stmt
                .executeQuery("LOAD CSV WITH HEADERS FROM "
                        + csvDataSource
                        + "dblookup_steps.csv" AS line 
                        + " MATCH (a:Relation{name: line.TableName, schema:line.DatabaseName})
                        + " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, steptype: 'TableInput'})
                        + " MERGE (a)-[:Impacts]->(st)\n    ");
        ResultSet rs3 = stmt
                .executeQuery("LOAD CSV WITH HEADERS FROM "
                        + csvDataSource
                        + "dimlookupupdate_steps.csv" AS line 
                        + " MATCH (a:Relation{name: line.TableName, schema:line.DatabaseName})
                        + " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, steptype: line.StepType})")
                        + " MERGE (a)-[:Impacts]->(st)\n    ");
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}
```
MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "stepetype: line.StepType}) WHERE st.update_lookup='Lookup'
MERGE (a)-[:Impacts]->(st);

ResultSet rs4 = stmt
.executeQuery("LOAD CSV WITH HEADERS FROM \
+ csvDataSource
+ "dbjoin_steps.csv" AS line "+ " MATCH (a:Relation{name: (line.TableName), schema:(line.DatabaseName)})"
+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "stepetype: line.StepType})"
+ " MERGE (a)-[:Impacts]->(st)";
}
catch (Exception ex) {
System.out.println(ex.getMessage());
}

public void generateETLStepToRelationRelationships(
String csvDataSource) {
try {
Statement stmt = con.createStatement();
{
ResultSet rs = stmt
.executeQuery("LOAD CSV WITH HEADERS FROM \
+ csvDataSource
+ "output_steps.csv" AS line "+ " MATCH (a:Relation{name: line.TableName, schema:line.DatabaseName})"
+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "stepetype: line.StepType})"
+ " MERGE (st)-[:Impacts]->(a)";

ResultSet rs2 = stmt
.executeQuery("LOAD CSV WITH HEADERS FROM \
+ csvDataSource
+ "dimlookupupdate_steps.csv" AS line "+ " MATCH (a:Relation{name:line.TableName, schema:line.DatabaseName})"
+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "stepetype: line.StepType}) WHERE st.update_lookup='Update'
+ " MERGE (st)-[:Impacts]->(a)";

ResultSet rs3 = stmt
.executeQuery("LOAD CSV WITH HEADERS FROM \
"
+ csvDataSource
+ "comblookupupdate_steps.csv" AS line
+ " MATCH (a:Relation{name: (line.TableName), schema:(line.DatabaseName)})"
+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "
+ "steptype: line.StepType})"
+ " MERGE (st)-[:Impacts]->(a)"
+ 
+ try {
+ catch (Exception ex) {
+ System.out.println(ex.getMessage());
+ }
+
+ public void generateSchemaToETLStepRelationships(
+ String csvDataSource) {
+ try {
+ Statement stmt = con.createStatement();
+ {
+ ResultSet rs = stmt
+ .executeQuery("LOAD CSV WITH HEADERS FROM \
+ + csvDataSource
+ + "input_steps.csv" AS line 
+ + " MATCH (a:Schema{name:line.DatabaseName})"
+ + " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, steptype:\""
+ + "TableInput\" + \\"})"
+ + " MERGE (a)-[:Impacts]->(st)"");
+
+ ResultSet rs2 = stmt
+ .executeQuery("LOAD CSV WITH HEADERS FROM \
+ + csvDataSource
+ + "dblookup_steps.csv" AS line 
+ + " MATCH (a:Schema{name:line.DatabaseName})"
+ + " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "
+ + "steptype: line.StepType})"
+ + " MERGE (a)-[:Impacts]->(st)"");
+
+ ResultSet rs3 = stmt
+ .executeQuery("LOAD CSV WITH HEADERS FROM \
+ + csvDataSource
+ + "dimlookupupdate_steps.csv" AS line 
+ + " MATCH (a:Schema{name:line.DatabaseName})"
+ + " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, "
+ + "steptype: line.StepType}) WHERE st.update_lookup=\'Lookup\'"
+ + " MERGE (a)-[:Impacts]->(st)"");
ResultSet rs4 = stmt.executeQuery("LOAD CSV WITH HEADERS FROM \\
+ csvDataSource \\
+ "djoin_steps.csv" AS line \\
+ " MATCH (a:Schema{name:line.DatabaseName})" \\
+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, " \\
+ "steptype: line.StepType})" \\
+ " MERGE (a)-[:Impacts]->(st)"");

} catch (Exception ex) {
    System.out.println(ex.getMessage());
}

public void generateETLStepToSchemaRelationships(String csvDataSource) {
    try {
        System.out.println(csvDataSource);
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM \\
+ csvDataSource \\
+ "output_steps.csv" AS line \\
+ " MATCH (a:Schema{name:line.DatabaseName})" \\
+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, " \\
+ "steptype: line.StepType})" \\
+ " MERGE (st)-[:Impacts]->(a)"");

        ResultSet rs2 = stmt.executeQuery("LOAD CSV WITH HEADERS FROM \\
+ csvDataSource \\
+ "dimlookupupdate_steps.csv" AS line \\
+ " MATCH (a:Schema{name:line.DatabaseName})" \\
+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, " \\
+ "steptype: line.StepType}) WHERE st.update_lookup='Update'" \\
+ " MERGE (st)-[:Impacts]->(a)"");

        ResultSet rs3 = stmt.executeQuery("LOAD CSV WITH HEADERS FROM \\
+ csvDataSource \\
+ "comblookupupdate_steps.csv" AS line \\
+ " MATCH (a:Schema{name:line.DatabaseName})" \\
+ " MATCH (st:Step {name: line.StepName, transname: line.TransformationName, " \\
+ "steptype: line.StepType})" WHERE st.update_lookup='Update'" \\
+ " MERGE (st)-[:Impacts]->(a)"");
} catch (Exception ex) {
    System.out.println(ex.getMessage());
}

B.4.3 QueryGraphController.java

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import org.neo4j.jdbc.Driver;

public class QueryGraphController {

    static Connection con;

    public QueryGraphController() {
        try {
            Class.forName("org.neo4j.jdbc.Driver");
            final Driver driver = new Driver(); //org.neo4j.jdbc.Driver
            final Properties props = new Properties();
            props.put("user", "neo4j");
            props.put("password", "neo4jdb675");
            String url="jdbc:neo4j://localhost:7474/";
            con = driver.connect(url, props);
        }
        catch(ClassNotFoundException ex) {
            System.out.println("Class not found exception in GraphController class constructor");
        }
        catch(SQLException ex) {
            System.out.println("SQLException in GraphController class constructor");
        }
        catch(Exception ex) {
            System.out.println("Exception in GraphController class constructor");
        }
    }

}
public void generateQueryNodesAndDependencies(String csvDataSource) {

generateQueryNodes(csvDataSource);
generateQuerySchemaDependencyRelationships(csvDataSource);
}

public void generateQueryNodes(String csvDataSource) {
    try {
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM "+csvDataSource+" AS line "+
 + "MERGE (q:Query{name: line.QueryLabel})");
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}

public void generateQuerySchemaDependencyRelationships(String csvDataSource) {
    try {
        Statement stmt = con.createStatement();
        {
            ResultSet rs = stmt.executeQuery("LOAD CSV WITH HEADERS FROM "+csvDataSource+" AS line "+
 + "MATCH (q:Query{name: line.QueryLabel})" +
 + " MATCH (a:Attribute {name: (line.AttributeName), relation: (line.RelationName), schema: (line.DatabaseName)})" +
 + " MERGE (a)-[:Impacts]->(q)";
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}

B.5 XML parsers for ETL workflows and queries

This section presents code that parses the XML representations of ETL workflows and SQL queries. ETL workflows are parsed to retrieve information about different
step types and the database-related data. The XML representation of SQL queries are parsed to identify the dependencies between each query and the attributes.

B.5.1 XMLParser.java

```java
import java.io.IOException;
import java.util.List;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class XMLParser {

    public static Document dom;

    public List<OutputStep> parseXMLForOutputSteps(String xmlFilePath){
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(true);
        DocumentBuilder builder;
        Document doc = null;
        OutputStep outputStep=null;
        List<OutputStep> outputsteps=new ArrayList<OutputStep>();
        try {
            builder = factory.newDocumentBuilder();
            doc = builder.parse(xmlFilePath);
            XPathFactory xpathFactory = XPathFactory.newInstance();
            XPath xpath = xpathFactory.newXPath();
            /* One transformation is composed of several steps */
            List<String> listOfAllSteps=getStepNames(xpath,doc);
            for(String step:listOfAllSteps){
```
int stepType=getStepType(xpath, doc, step);
switch(stepType){
case 0: outputStep=
   InsertUpdateStepParser.parseXMLForInsertUpdateStep(doc, xpath, step);
   outputsteps.add(outputStep);
   break;
case 1: outputStep=
   TableOutputStepParser.parseXMLForTableOutputStep(doc, xpath, step);
   outputsteps.add(outputStep);
   break;
default: break;
}
}
}

} catch (ParserConfigurationException | SAXException | IOException e) {
   e.printStackTrace();
}

return outputsteps;
}

public List<TableInputStep> parseXMLForInputSteps(String xmlFilePath){

    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    DocumentBuilder builder;
    Document doc = null;
    String steptype="TableInput";
    TableInputStep tableinputStep=null;
    List<TableInputStep> inputsteps=new ArrayList<TableInputStep>();
    try {
        builder = factory.newDocumentBuilder();
        doc = builder.parse(xmlFilePath);
        XPathFactory xpathFactory = XPathFactory.newInstance();
        XPath xpath = xpathFactory.newXPath();
        /* One transformation is composed of several Table Input steps */
        List<String> listOfAllStepNames=getStepNamesByType(xpath,doc,steptype);

        for(String stepname:listOfAllStepNames){
            tableinputStep=TableInputStepParser.parseXMLByStepName(doc, xpath, stepname);
            inputsteps.add(tableinputStep);
        }
    } catch (ParserConfigurationException | SAXException | IOException e) {
        e.printStackTrace();
    }

    return inputsteps;
}
return inputsteps;
}

private List<String> getStepNamesByType(XPath xpath, Document doc,String steptype){
List<String> stepNames=new ArrayList<String>();

try {
    XPathExpression expr =
    xpath.compile("/transformation/step[type='"+steptype+'"]/name/text()");

    NodeList nodes = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
    for (int i = 0; i < nodes.getLength(); i++)
        stepNames.add(nodes.item(i).getNodeValue());
}

return stepNames;
}

public List<DBLookupStep> parseXMLForDBLookupSteps(String xmlFilePath){

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
DocumentBuilder builder;
Document doc = null;
String steptype="DBLookup";
DBLookupStep dblookupstep=null;
List<DBLookupStep> dblookupsteps=new ArrayList<DBLookupStep>();
try {
    builder = factory.newDocumentBuilder();
    doc = builder.parse(xmlFilePath);
    XPathFactory xpathFactory = XPathFactory.newInstance();
    XPath xpath = xpathFactory.newXPath();
    /* One transformation is composed of several Database lookup steps */
    List<String> listOfAllStepNames=getStepNamesByType(xpath,doc,steptype);

    for(String stepname:listOfAllStepNames){
        dblookupstep=DBLookupStepParser.parseXMLByStepName(doc, xpath, stepname);
        dblookupsteps.add(dblookupstep);
    }
}
catch (ParserConfigurationException | SAXException | IOException e) {

public List<DBJoinStep> parseXMLForDBJoinSteps(String xmlFilePath)
{
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    DocumentBuilder builder;
    Document doc = null;
    String steptype="DBJoin";
    DBJoinStep dbjoinstep=null;
    List<DBJoinStep> dbjoinsteps=new ArrayList<DBJoinStep>();
    try {
        builder = factory.newDocumentBuilder();
        doc = builder.parse(xmlFilePath);
        XPathFactory xpathFactory = XPathFactory.newInstance();
        XPath xpath = xpathFactory.newXPath();
        /* One transformation is composed of several Table Input steps */
        List<String> listOfAllStepNames=getStepNamesByType(xpath,doc,steptype);
        for(String stepname:listOfAllStepNames){
            dbjoinstep=DBJoinStepParser.parseXMLByStepName(doc, xpath, stepname);
            dbjoinsteps.add(dbjoinstep);
        }
    } catch (ParserConfigurationException | SAXException | IOException e) {
        e.printStackTrace();
    }
    return dbjoinsteps;
}

public List<DimLookupUpdateStep> parseXMLForDimLookupUpdateSteps(String xmlFilePath){
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    DocumentBuilder builder;
    Document doc = null;
    String steptype="DimensionLookup";
    DimLookupUpdateStep dimlookupstep=null;
    List<DimLookupUpdateStep> dimlookupsteps=new ArrayList<DimLookupUpdateStep>();
    try {
        builder = factory.newDocumentBuilder();
        doc = builder.parse(xmlFilePath);
        XPathFactory xpathFactory = XPathFactory.newInstance();
        XPath xpath = xpathFactory.newXPath();
        /* One transformation is composed of several Table Input steps */
        List<String> listOfAllStepNames=getStepNamesByType(xpath,doc,steptype);
        for(String stepname:listOfAllStepNames){
            dimlookupstep=DimLookupUpdateStepParser.parseXMLByStepName(doc, xpath, stepname);
            dimlookupsteps.add(dimlookupstep);
        }
    } catch (ParserConfigurationException | SAXException | IOException e) {
        e.printStackTrace();
    }
    return dimlookupsteps;
}
List<DimLookupUpdateStep> dimlookupupdatesteps =
new ArrayList<DimLookupUpdateStep>();
try {
    builder = factory.newDocumentBuilder();
doc = builder.parse(xmlFilePath);
    XPathFactory xpathFactory = XPathFactory.newInstance();
    XPath xpath = xpathFactory.newXPath();
    /* One transformation is composed of several Table Input steps */
    List<String> listOfAllStepNames = getStepNamesByType(xpath, doc, steptype);
    for (String stepname : listOfAllStepNames) {
        dimlookupstep = DimensionLookupStepParser.parseXMLForDimensionLookupStep(
            doc, xpath, stepname);
        // A Dimension Lookup/Update step can be a read-only step (Lookup) or
        // write step (Update)
        dimlookupupdatesteps.add(dimlookupstep);
    }
} catch (ParserConfigurationException | SAXException | IOException e) {
    e.printStackTrace();
}
return dimlookupupdatesteps;
}

public List<CombinationLookupUpdateStep> parseXMLForCombinationLookupUpdateSteps(String xmlFilePath) {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    DocumentBuilder builder;
    Document doc = null;
    String steptype = "CombinationLookup";
    CombinationLookupUpdateStep comblookupstep = null;
    List<CombinationLookupUpdateStep> comblookupupdatesteps =
    new ArrayList<CombinationLookupUpdateStep>();
    try {
        builder = factory.newDocumentBuilder();
doc = builder.parse(xmlFilePath);
    XPathFactory xpathFactory = XPathFactory.newInstance();
    XPath xpath = xpathFactory.newXPath();
    /* One transformation is composed of several Table Input steps */
    List<String> listOfAllStepNames = getStepNamesByType(xpath, doc, steptype);
    for (String stepname : listOfAllStepNames) {
        comblookupstep = CombinationLookupStepParser.parseXMLForCombinationLookupStep(
            doc, xpath, stepname);
        // Combination Lookup/Update step can be a read-only step (Lookup) or
        // write step (Update)
        comblookupupdatesteps.add(comblookupstep);
    }
} catch (ParserConfigurationException | SAXException | IOException e) {
    e.printStackTrace();
}
return comblookupupdatesteps;
}
(doc, xpath, stepname);
comblookupupdatesteps.add(comblookupstep);
}

} catch (ParserConfigurationException | SAXException | IOException e) {
e.printStackTrace();
}

return comblookupupdatesteps;

private int getStepType(XPath xpath, Document doc, String stepname) {
int stepType=-1;
String cleanStepName=stepname.replace('"', "");

try {

XPathExpression expr =
    xpath.compile("/transformation/step[name='"+cleanStepName+"']/type/text()"); /*
    String str = (String) expr.evaluate(doc, XPathConstants.STRING);
    if(str.equals("InsertUpdate")) stepType=0;
    else if(str.equals("TableOutput")) stepType=1;

} catch (XPathExpressionException e) {
e.printStackTrace();
}

return stepType;

}

private List<String> getStepNames(XPath xpath, Document doc){
List<String> stepNames=new ArrayList<String>();
try {

XPathExpression expr =
    xpath.compile("/transformation/step/name/text()");

NodeList nodes = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
for (int i = 0; i < nodes.getLength(); i++)
    stepNames.add(nodes.item(i).getNodeValue());

} catch (XPathExpressionException e) {
e.printStackTrace();
}
B.5.2 TableInputStepParser.java

import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import org.w3c.dom.Document;

public class TableInputStepParser {
    public static TableInputStep parseXMLByStepName(Document doc, XPath xpath, String stepname) {
        TableInputStep stepObject = null;
        String transname = getTransformationName(doc, xpath);
        String dbname = getDatabaseName(doc, xpath, stepname);
        String sql = getSQL(doc, xpath, stepname);
        stepObject = new TableInputStep(transname, stepname, dbname, sql);
        return stepObject;
    }

    private static String getDatabaseName(Document doc, XPath xpath, String stepname) {
        String connectionname = null;
        String dbname = null;
        try {
            // create XPathExpression object
            XPathExpression expr = xpath.compile("/transformation/step[name='" + stepname + "]/connection/text()");
            connectionname = (String) expr.evaluate(doc, XPathConstants.STRING);

            XPathExpression dbexpr = xpath.compile("/transformation/connection[name='" + connectionname + "]/database/text()");
            dbname = (String) dbexpr.evaluate(doc, XPathConstants.STRING);
        } catch (XPathExpressionException e) {
            e.printStackTrace();
        }
        return dbname;
    }
}
private static String getSQL(Document doc, XPath xpath, String stepname) {
    String sql = null;
    try {
        // create XPathExpression object
        XPathExpression expr = xpath.compile("/transformation/step[name='" + stepname + "']/sql/text()")
        sql = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return sql;
}

private static String getTransformationName(Document doc, XPath xpath) {
    String name = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/info/name/text()")
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}

B.5.3 TableOutputStepParser.java

import java.util.ArrayList;
import java.util.List;

import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;

public class TableOutputStepParser {

    public static OutputStep parseXMLForTableOutputStep(Document doc, XPath xpath, String stepname) {
        OutputStep stepObject = null;

        String transname = getTransformationName(doc, xpath);

        String dbname = getDatabaseName(doc, xpath, stepname);
        String tableName = getTableName(doc, xpath, stepname);

        List<String> fieldnames = getDatabaseFieldNames(doc, xpath, stepname);

        stepObject = new OutputStep(transname, stepname, "TableOutput", dbname, tableName, fieldnames);

        return stepObject;
    }

    private static String getDatabaseName(Document doc, XPath xpath, String stepname) {
        String connectionname = null;
        String dbname = null;
        try {
            // create XPathExpression object
            XPathExpression expr = xpath.compile("/transformation/step[name='" + stepname + "]/connection/text()");
            connectionname = (String) expr.evaluate(doc, XPathConstants.STRING);

            XPathExpression dbexpr = xpath.compile("/transformation/connection[name='" + connectionname + "]/database/text()");
            dbname = (String) dbexpr.evaluate(doc, XPathConstants.STRING);

        } catch (XPathExpressionException e) {
            e.printStackTrace();
        }
        return dbname;
    }

    private static List<String> getDatabaseFieldNames(Document doc, XPath xpath, String stepname) {
        return null;
    }

}
XPath xpath, String stepname) {
    List<String> list = new ArrayList<>();
    try {
        // create XPathExpression object
        XPathExpression expr = xpath.compile("/transformation/step[name='" + stepname + "']/fields/field/column_name/text()");
        // evaluate expression result on XML document
        NodeList nodes = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
        for (int i = 0; i < nodes.getLength(); i++)
            list.add(nodes.item(i).getNodeValue());
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return list;
}

private static String getTransformationName(Document doc, XPath xpath) {
    String name = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/info/name/text()");
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}

private static String getTableName(Document doc, XPath xpath, String stepname) {
    String name = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/step[name='" + stepname + "']/table/text()");
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}
B.5.4 InsertUpdateStepParser.java

```java
import java.util.ArrayList;
import java.util.List;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;

public class InsertUpdateStepParser {
    public static OutputStep parseXMLForInsertUpdateStep(Document doc, XPath xpath, String step) {
        OutputStep stepObject = null;
        String transname = getTransformationName(doc, xpath);
        String dbname = getDatabaseName(doc, xpath, step);
        String tableName = getTableName(doc, xpath, step);
        List<String> fieldnames = getDatabaseFieldNames(doc, xpath, step);
        stepObject = new OutputStep(transname, step, "Insert/Update", dbname, tableName, fieldnames);
        return stepObject;
    }

    private static String getDatabaseName(Document doc, XPath xpath, String stepname) {
        String connectionname = null;
        String dbname = null;
        try {
            XPathExpression expr = xpath.compile("/transformation/step[name='" + stepname + "]/connection/text()".getTransformationName(doc, xpath));
            connectionname = (String) expr.evaluate(doc, XPathConstants.STRING);
            XPathExpression dbexpr = xpath.compile("/transformation/connection[name='" + connectionname + "]/database/text()".getDatabaseName(doc, xpath, step));
        }
    }

    private static String getDatabaseName(Document doc, XPath xpath, String stepname) {
        String connectionname = null;
        String dbname = null;
        try {
            XPathExpression expr = xpath.compile("/transformation/step[name='" + stepname + "]/connection/text()".getTransformationName(doc, xpath));
            connectionname = (String) expr.evaluate(doc, XPathConstants.STRING);
            XPathExpression dbexpr = xpath.compile("/transformation/connection[name='" + connectionname + "]/database/text()".getDatabaseName(doc, xpath, step));
        }
```
dbname = (String) dbexpr.evaluate(doc, XPathConstants.STRING);

} catch (XPathExpressionException e) {
    e.printStackTrace();
}
return dbname;

private static List<String> getDatabaseFieldNames(Document doc, XPath xpath, String stepname) {
    List<String> list = new ArrayList<String>();
    try {
        // create XPathExpression object
        XPathExpression expr = xpath.compile("/transformation/step[name='" + stepname + "]/lookup/value/name/text()");
        // evaluate expression result on XML document
        NodeList nodes = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
        for (int i = 0; i < nodes.getLength(); i++)
            list.add(nodes.item(i).getNodeValue());
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return list;
}

private static String getTransformationName(Document doc, XPath xpath) {
    String name = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/info/name/text()");
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}

private static String getTableName(Document doc, XPath xpath, String stepname) {
    String tablename = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/step[name='" + stepname + "]/lookup/table/text()");
tablename = (String) expr.evaluate(doc, XPathConstants.STRING);
} catch (XPathExpressionException e) {
    e.printStackTrace();
}

return tablename;
}

B.5.5 DimensionLookupStepParser.java

import java.util.ArrayList;
import java.util.List;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;

public class DimensionLookupStepParser {

    public static DimLookupUpdateStep parseXMLForDimensionLookupStep(
        Document doc, XPath xpath, String stepName) {
        DimLookupUpdateStep stepObject = null;

        String transname = getTransformationName(doc, xpath);
        String dbname = getDatabaseName(doc, xpath, stepName);
        String tableName = getTableName(doc, xpath, stepName);
        List<String> fieldnames = getDatabaseFieldNames(doc, xpath, stepName);
        boolean isUpdateStep = getStepUpdateStatus(doc, xpath, stepName);

        stepObject = new DimLookupUpdateStep(transname, stepName,
            "Dimension Lookup", dbname, tableName, fieldnames, isUpdateStep);

        return stepObject;
    }

    private static boolean getStepUpdateStatus(Document doc, XPath xpath,
        String stepname) {

        String transname = getTransformationName(doc, xpath);
        String dbname = getDatabaseName(doc, xpath, stepName);
        String tableName = getTableName(doc, xpath, stepName);

        List<String> fieldnames = getDatabaseFieldNames(doc, xpath, stepName);
        boolean isUpdateStep = getStepUpdateStatus(doc, xpath, stepName);

        stepObject = new DimLookupUpdateStep(transname, stepName,
            "Dimension Lookup", dbname, tableName, fieldnames, isUpdateStep);

        return stepObject;
    }

    private static boolean getStepUpdateStatus(Document doc, XPath xpath,
        String stepname) {

    }
boolean updateStatus = false;
try {
    XPathExpression expr = xpath.compile("/transformation/step[name=\"\
    + stepname + "\"]/update/text()\");
    String updateStatus_str = (String) expr.evaluate(doc,
    XPathConstants.STRING);
    updateStatus = (updateStatus_str.equals("Y") ? true : false);
    System.out.println("It is an update step:" + updateStatus);
} catch (XPathExpressionException e) {
    e.printStackTrace();
}
return updateStatus;
}

private static String getDatabaseName(Document doc, XPath xpath,
    String stepname) {
    String connectionname = null;
    String dbname = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/step[name=\"\
        + stepname + "\"]/connection/text()\");
        connectionname = (String) expr.evaluate(doc, XPathConstants.STRING);
        XPathExpression dbexpr = xpath
            .compile("/transformation/connection[name='\"\
                + connectionname + "\"]/database/text()\");
        dbname = (String) dbexpr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return dbname;
}

private static List<String> getDatabaseFieldNames(Document doc,
    XPath xpath, String stepname) {
    List<String> list = new ArrayList<>();
    try {
        XPathExpression expr_lookupkey_field = xpath
            .compile("/transformation/step[name=\"\
                + stepname + "\"]/fields/key/lookup/text()\");
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return list;
}
XPathExpression expr_daterange_start = xpath
.compile("/transformation/step[name='" + stepname + '"]/fields/date/from/text()");

XPathExpression expr_daterange_end = xpath
.compile("/transformation/step[name='" + stepname + '"]/fields/date/to/text()");

XPathExpression expr_version_field = xpath
.compile("/transformation/step[name='" + stepname + '"]/fields/return/version/text()");

XPathExpression expr_surrogate_key = xpath
.compile("/transformation/step[name='" + stepname + '"]/fields/return/name/text()");

XPathExpression expr_updateOrInsertFields = xpath
.compile("/transformation/step[name='" + stepname + '"]/fields/field/lookup/text()");

// evaluate expression result on XML document

NodeList nodes = (NodeList) expr_updateOrInsertFields.evaluate(doc, XPathConstants.NODESET);
for (int i = 0; i < nodes.getLength(); i++)
list.add(nodes.item(i).getNodeValue());

String lookupkey_field = (String) expr_lookupkey_field.evaluate(doc, XPathConstants.STRING);
String daterange_start_field = (String) expr_daterange_start.evaluate(doc, XPathConstants.STRING);
String daterange_end_field = (String) expr_daterange_end.evaluate(doc, XPathConstants.STRING);
String version_field = (String) expr_version_field.evaluate(doc, XPathConstants.STRING);
String surrogatekey_field = (String) expr_surrogate_key.evaluate(doc, XPathConstants.STRING);

// for (int i = 0; i < nodes.getLength(); i++)
// list.add(nodes.item(i).getNodeValue());
if (lookupkey_field != null && !lookupkey_field.isEmpty())
list.add(lookupkey_field);
if (daterange_start_field != null && !daterange_start_field.isEmpty())
list.add(daterange_start_field);
if (daterange_end_field != null && !daterange_end_field.isEmpty())
list.add(daterange_end_field);
if (version_field != null && !version_field.isEmpty())
list.add(version_field);
if (surrogatekey_field != null && !surrogatekey_field.isEmpty())
list.add(surrogatekey_field);

} catch (XPathExpressionException e) {
private static String getTransformationName(Document doc, XPath xpath) {
    String name = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/info/name/text()")
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}

private static String getTableName(Document doc, XPath xpath, String stepname) {
    String name = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/step[name=" + stepname + "]/table/text()")
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}

private static String getStepName(Document doc, XPath xpath) {
    String name = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/step[type='DimensionLookup']/name/text()")
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}
B.5.6 CombinationLookupStepParser.java

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;

public class CombinationLookupStepParser {

    public static CombinationLookupUpdateStep parseXMLForCombinationLookupStep(
            Document doc, XPath xpath, String stepName) {
        CombinationLookupUpdateStep stepObject = null;
        String transname = getTransformationName(doc, xpath);
        String dbname = getDatabaseName(doc, xpath, stepName);
        String tableName = getTableName(doc, xpath, stepName);
        List<String> fieldnames = getDatabaseFieldNames(doc, xpath, stepName);
        stepObject = new CombinationLookupUpdateStep(transname, stepName,
                "CombinationLookup", dbname, tableName, fieldnames);
        return stepObject;
    }

    private static String getDatabaseName(Document doc, XPath xpath,
            String stepname) {
        String connectionname = null;
        String dbname = null;
        try {
            // create XPathExpression object
            XPathExpression expr = xpath.compile("/transformation/step[name=\"" + stepname + "]/connection/text()");
            connectionname = (String) expr.evaluate(doc, XPathConstants.STRING);

            XPathExpression dbexpr = xpath.compile("/transformation/connection[name=\"" + connectionname + "]/database/text()");
            dbname = (String) dbexpr.evaluate(doc, XPathConstants.STRING);
        } catch (XPathExpressionException e) {
            e.printStackTrace();
        }
        return dbname;
    }

    // Other methods...
}
```
dbname = (String) dbexpr.evaluate(doc, XPathConstants.STRING);

} catch (XPathExpressionException e) {
    e.printStackTrace();
} return dbname;

private static List<String> getDatabaseFieldNames(Document doc, XPath xpath, String stepname) {
    List<String> list = new ArrayList<>();
    try {
        XPathExpression expr_surrogate_key = xpath
            .compile("/transformation/step[name='" + stepname + "]/fields/return/name/text()");
        XPathExpression expr_dbfields = xpath
            .compile("/transformation/step[name='" + stepname + "]/fields/key/lookup/text()");
        // evaluate expression result on XML document
        NodeList nodes = (NodeList) expr_dbfields.evaluate(doc, XPathConstants.NODESET);
        for (int i = 0; i < nodes.getLength(); i++)
            list.add(nodes.item(i).getNodeValue());
        String surrogatekey_field = (String) expr_surrogate_key.evaluate(doc, XPathConstants.STRING);
        if (surrogatekey_field != null && !surrogatekey_field.isEmpty())
            list.add(surrogatekey_field);
    }

    String surrogatekey_field = (String) expr_surrogate_key.evaluate(doc, XPathConstants.STRING);

    if (surrogatekey_field != null && !surrogatekey_field.isEmpty())
        list.add(surrogatekey_field);

    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return list;
}

private static String getTransformationName(Document doc, XPath xpath) {
    String name = null;
    try {
        XPathExpression expr = xpath
            .compile("/transformation/info/name/text()");
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    }

    String surrogatekey_field = (String) expr_surrogate_key.evaluate(doc, XPathConstants.STRING);

    if (surrogatekey_field != null && !surrogatekey_field.isEmpty())
        list.add(surrogatekey_field);

    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return list;
}
private static String getTableName(Document doc, XPath xpath, String stepname) {
    String name = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/step[name='" + stepname + "]/table/text()");
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}

private static String getStepName(Document doc, XPath xpath) {
    String name = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/step[type='DimensionLookup']/name/text()");
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}

B.5.7 DBJoinStepParser.java

import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import org.w3c.dom.Document;
public class DBJoinStepParser {

public static DBJoinStep parseXMLByStepName(Document doc, XPath xpath, String stepname) {

DBJoinStep stepObject = null;
String transname = getTransformationName(doc, xpath);
String dbname = getDatabaseName(doc, xpath, stepname);
String sql = getSQL(doc, xpath, stepname);
stepObject = new DBJoinStep(transname, stepname, dbname, sql);
return stepObject;
}

private static String getDatabaseName(Document doc, XPath xpath, String stepname) {

String connectionname = null;
String dbname = null;
try {
    XPathExpression expr = xpath.compile("/transformation/step[name='"+stepname+"']/
                             + "connection/text()");
    connectionname = (String) expr.evaluate(doc, XPathConstants.STRING);

    XPathExpression dbexpr = xpath.compile("/transformation/
    + "connection[name='"+connectionname+"']/database/text()"");
    dbname = (String) dbexpr.evaluate(doc, XPathConstants.STRING);
}
catch (XPathExpressionException e) {
    e.printStackTrace();
}
return dbname;
}

private static String getSQL(Document doc, XPath xpath, String stepname) {

String sql = null;
try {
    XPathExpression expr = xpath.compile("/transformation/step[name='"+stepname+"']/
                             + "sql/text()";
    sql = (String) expr.evaluate(doc, XPathConstants.STRING);
}
catch (XPathExpressionException e) {
    e.printStackTrace();
}

}
private static String getTransformationName(Document doc, XPath xpath) {
    String name = null;
    try {
        XPathExpression expr =
            xpath.compile("/transformation/info/name/text()") ;
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}

B.5.8 DBLookupStepParser.java

import java.util.ArrayList;
import java.util.List;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;

public class DBLookupStepParser {

    public static DBLookupStep parseXMLByStepName(Document doc, XPath xpath,
    String stepname) {

        DBLookupStep stepObject = null;
        String transname = getTransformationName(doc, xpath);
        String dbname = getDatabaseName(doc, xpath, stepname);
        String tablename = getTableName(doc, xpath, stepname);
List<String> fieldnames = getDatabaseFieldNames(doc, xpath, stepname);

stepObject = new DBLookupStep(transname, stepname, dbname, tablename, fieldnames);

return stepObject;
}

private static List<String> getDatabaseFieldNames(Document doc, XPath xpath, String stepname) {
List<String> list = new ArrayList<String>();
try {

// create XPathExpression object
XPathExpression expr = xpath.compile("/transformation/step[name=" + stepname + "]/lookup/key/field/text()");
// evaluate expression result on XML document
NodeList nodes = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
for (int i = 0; i < nodes.getLength(); i++)
list.add(nodes.item(i).getNodeValue());

XPathExpression return_expr = xpath.compile("/transformation/step[name=" + stepname + "]/lookup/value/name/text()");
// evaluate expression result on XML document
NodeList return_nodes = (NodeList) return_expr.evaluate(doc, XPathConstants.NODESET);
for (int i = 0; i < return_nodes.getLength(); i++)
list.add(return_nodes.item(i).getNodeValue());
} catch (XPathExpressionException e) {
e.printStackTrace();
}
return list;
}

private static String getDatabaseName(Document doc, XPath xpath, String stepname) {
String connectionname = null;
String dbname = null;

try {

// create XPathExpression object
XPathExpression expr = xpath.compile("/transformation/step[name=" + stepname + "]/connection/text()");
connectionname = (String) expr.evaluate(doc, XPathConstants.STRING);

XPathExpression dbexpr = xpath.compile("/transformation/connection[name='"+connectionname+"']/database/text()");
dbname = (String) dbexpr.evaluate(doc, XPathConstants.STRING);

} catch (XPathExpressionException e) {
e.printStackTrace();
}
return dbname;

private static String getTableName(Document doc, XPath xpath, String stepname) {
String tablename = null;
try {
    // create XPathExpression object
    XPathExpression expr = xpath.compile("/transformation/step[name='"+stepname+"']/lookup/table/text()");
    tablename = (String) expr.evaluate(doc, XPathConstants.STRING);
}
    catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return tablename;
}

private static String getTransformationName(Document doc, XPath xpath) {
    String name = null;
    try {
        XPathExpression expr = xpath.compile("/transformation/info/name/text()");
        name = (String) expr.evaluate(doc, XPathConstants.STRING);
    }
    catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return name;
}
public class QueryXMLParser {

    public List<Query> parseXML(String xmlFilePath) {
        try {
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            factory.setNamespaceAware(true);
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document doc = builder.parse(xmlFilePath);
            XPathFactory xpathFactory = XPathFactory.newInstance();
            XPath xpath = xpathFactory.newXPath();
            List<String> listOfAllFileNames = getFileNamesByType(xpath, doc);
            for (String f : listOfAllFileNames) {
                System.out.println(f);
                queryObjs.addAll(buildQueryObject(xpath, doc, f));
            }
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
        return queryObjs;
    }

    private List<String> getFileNamesByType(XPath xpath, Document doc) {
        // Implementation
    }

    private List<Query> buildQueryObject(XPath xpath, Document doc, String fileName) {
        // Implementation
    }
}
private List<Query> buildQueryObject(XPath xpath, Document doc, String filename){
    List<String> queryInfoList=new ArrayList<String>();
    List<Query> listoOfObjects=new ArrayList<Query>();
    try {
        String attrExpr="/columnImpactResult/file[@name = "+filename+"']/targetColumn/sourceColumn/@name";
        String relExpr="/columnImpactResult/file[@name = "+filename+"']/targetColumn/sourceColumn/@tableName";
        String dbExpr="/columnImpactResult/file[@name = "+filename+"']/targetColumn/sourceColumn/@tableOwner";
        XPathExpression expr = xpath.compile(attrExpr);
        XPathExpression expr2 = xpath.compile(relExpr);
        XPathExpression db_expr = xpath.compile(dbExpr);
        NodeList attr_nodes = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
        NodeList rel_nodes = (NodeList) expr2.evaluate(doc, XPathConstants.NODESET);
        NodeList db_nodes = (NodeList) db_expr.evaluate(doc, XPathConstants.NODESET);
        for (int i = 0; i < attr_nodes.getLength(); i++){
            String attributeName=attr_nodes.item(i).getNodeValue();
            String relName=rel_nodes.item(i).getNodeValue();
            String dbName=db_nodes.item(i).getNodeValue();
            String queryObjStr=filename+","+dbName+","+relName+","+attributeName;
            if(!queryInfoList.contains(queryObjStr)){
                queryInfoList.add(queryObjStr);
                Query obj=new Query();
                obj.setQueryLabel(filename);
                obj.setAttribute(attributeName);
                obj.setRelationName(relName);
                obj.setDbName(dbName);
                listoOfObjects.add(obj);
            }
        }
    }catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return listoOfObjects;
}

private List<String> getFileNamesByType(XPath xpath, Document doc){
    List<String> fileNames=new ArrayList<String>();
    try {
        XPathExpression expr = xpath.compile("/columnImpactResult/file/@name");
        NodeList nodes = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
        for (int i = 0; i < nodes.getLength(); i++){
            String completeFileName=nodes.item(i).getNodeValue();
            fileNames.add(completeFileName);
        }
    }catch (XPathExpressionException e) {
        e.printStackTrace();
    }
    return fileNames;
}
fileNames.add(completeFileName);
}
} catch (XPathExpressionException e) {
    e.printStackTrace();
}    
    return fileNames;
}
}

B.6  Java objects for various artifacts

B.6.1  TableInputStep.java

public class TableInputStep {

    private String transName;
    private String stepName;
    private String dbName;
    private String sql;

    @Override
    public String toString() {
        return "TableInputStep [transName=" + transName + ", stepName=" + stepName + ", dbName=" + dbName + ", sql=" + sql + "]";
    }

    public TableInputStep(String transName, String stepName, String dbName, String sql) {
        super();
        this.transName = transName;
        this.stepName = stepName;
        this.dbName = dbName;
        this.sql = sql;
    }

    public String getTransName() {
        return transName;
    }

    public void setTransName(String transName) {
        this.transName = transName;
    }

    public String getStepName() {

B.6.2 OutputStep.java

import java.util.ArrayList;
import java.util.List;

public class OutputStep {
    private String transName;
    private String stepName;
    private String stepType;
    private String dbName;
    private String tableName;
    private List<String> attributes=new ArrayList<String>();

    public OutputStep(String transName, String stepName, String stepType,
    String dbName, String tableName, List<String> attributes) {
        super();
        this.transName = transName;
        this.stepName = stepName;
        this.stepType = stepType;
        this.dbName = dbName;
        this.tableName = tableName;
        this.attributes = attributes;
    }
}
public String getTransName() {
    return transName;
}

public void setTransName(String transName) {
    this.transName = transName;
}

public String getStepName() {
    return stepName;
}

public void setStepName(String stepName) {
    this.stepName = stepName;
}

public String getStepType() {
    return stepType;
}

public void setStepType(String stepType) {
    this.stepType = stepType;
}

public String getDbName() {
    return dbName;
}

public void setDbName(String dbName) {
    this.dbName = dbName;
}

public String getTableName() {
    return tableName;
}

public void setTableName(String tableName) {
    this.tableName = tableName;
}

public List<String> getAttributes() {
    return attributes;
}

public void setAttributes(List<String> attributes) {
    this.attributes = attributes;
}

@Override
public String toString() {
    return "OutputStep [transName=" + transName + ", stepName=" + stepName + ", stepType=" + stepType + ", dbName=" + dbName + ", tableName=" + tableName + ", attributes=" + attributes + "]";
}
B.6.3 DimLookupUpdateStep.java

```java
import java.util.ArrayList;
import java.util.List;

public class DimLookupUpdateStep {

    private String transName;
    private String stepName;
    private String dbName;
    private String tableName;
    private List<String> attributes = new ArrayList<String>();
    private String stepType;
    private boolean isUpdateStep;

    @Override
    public String toString() {
        return "DimLookupUpdateStep [transName=" + transName + ", stepName=" + stepName + ", dbName=" + dbName + ", tableName=" + tableName + ", attributes=" + attributes + "]";
    }

    public boolean isUpdateStep() {
        return isUpdateStep;
    }

    public void setUpdateStep(boolean isUpdateStep) {
        this.isUpdateStep = isUpdateStep;
    }

    public String getTransName() {
        return transName;
    }

    public void setTransName(String transName) {
        this.transName = transName;
    }

    public String getStepName() {
        return stepName;
    }

    public void setStepName(String stepName) {
    }
}
```
public void setStepName(String stepName) {
    this.stepName = stepName;
}

public String getDbName() {
    return dbName;
}

public void setDbName(String dbName) {
    this.dbName = dbName;
}

public String getTableName() {
    return tableName;
}

public void setTableName(String tableName) {
    this.tableName = tableName;
}

public DimLookupUpdateStep(String transName, String stepName, String stepType, String dbName, String tableName, List<String> attributes, boolean isUpdateStep) {
    super();
    this.transName = transName;
    this.stepName = stepName;
    this.stepType = stepType;
    this.dbName = dbName;
    this.tableName = tableName;
    this.attributes = attributes;
    this.isUpdateStep = isUpdateStep;
}

public List<String> getAttributes() {
    return attributes;
}

public void setAttributes(List<String> attributes) {
    this.attributes = attributes;
}

}
public class DBLookupStep {

    private String transName;
    private String stepName;
    private String dbName;
    private String tableName;
    private List<String> attributes = new ArrayList<String>();

    @Override
    public String toString() {
        return "DBLookupStep [transName=" + transName + ", stepName=" + stepName + ", dbName=" + dbName + ", tableName=" + tableName + ", attributes=" + attributes + "]";
    }

    public String getTransName() {
        return transName;
    }

    public void setTransName(String transName) {
        this.transName = transName;
    }

    public String getStepName() {
        return stepName;
    }

    public void setStepName(String stepName) {
        this.stepName = stepName;
    }

    public String getDbName() {
        return dbName;
    }

    public void setDbName(String dbName) {
        this.dbName = dbName;
    }

    public String getTableName() {
        return tableName;
    }

    public void setTableName(String tableName) {
        this.tableName = tableName;
    }

}
B.6.5  DBJoinStep.java

public class DBJoinStep {

    private String transName;
    private String stepName;
    private String dbName;
    private String sql;

    public String getSql() {
        return sql;
    }

    public void setSql(String sql) {
        this.sql = sql;
    }

    @Override

public String toString() {
    return "DBJoinStep [transName=" + transName + ", stepName=" + stepName + ", dbName=" + dbName + ", sql=" + sql + "]";
}

global void setTransName(String transName) {
    this.transName = transName;
}

global void setStepName(String stepName) {
    this.stepName = stepName;
}

global void setDbName(String dbName) {
    this.dbName = dbName;
}

public DBJoinStep(String transName, String stepName, String dbName, String sql) {
    super();
    this.transName = transName;
    this.stepName = stepName;
    this.dbName = dbName;
    this.sql = sql;
}

B.6.6 CombinationLookupUpdateStep.java

import java.util.ArrayList;
import java.util.List;
public class CombinationLookupUpdateStep {

    private String transName;
    private String stepName;
    private String dbName;
    private String tableName;
    private List<String> attributes = new ArrayList<String>();
    private String stepType;

    @Override
    public String toString() {
        return "CombinationLookupUpdateStep [transName=" + transName
    + ", stepName=" + stepName + ", dbName=" + dbName
    + ", tableName=" + tableName + ", attributes=" + attributes
    + "]";
    }

    public String getTransName() {
        return transName;
    }

    public void setTransName(String transName) {
        this.transName = transName;
    }

    public String getStepName() {
        return stepName;
    }

    public void setStepName(String stepName) {
        this.stepName = stepName;
    }

    public String getDbName() {
        return dbName;
    }

    public void setDbName(String dbName) {
        this.dbName = dbName;
    }

    public String getTableName() {
        return tableName;
    }

    public String setTableName(String tableName) {
        this.tableName = tableName;
    }

    public String getAttributes() {
        return attributes.toString();
    }

    public void setAttributes(List<String> attributes) {
        this.attributes = attributes;
    }

    public String getStepType() {
        return stepType;
    }

    public void setStepType(String stepType) {
        this.stepType = stepType;
    }
}
public void setTableName(String tableName) {
    this.tableName = tableName;
}

public CombinationLookupUpdateStep(String transName, String stepName,
        String stepType, String dbName, String tableName,
        List<String> attributes) {
    super();
    this.transName = transName;
    this.stepName = stepName;
    this.stepType = stepType;
    this.dbName = dbName;
    this.tableName = tableName;
    this.attributes = attributes;
}

public List<String> getAttributes() {
    return attributes;
}

public void setAttributes(List<String> attributes) {
    this.attributes = attributes;
}

public String getStepType() {
    return stepType;
}

public void setStepType(String stepType) {
    this.stepType = stepType;
}

}

B.6.7 Query.java

public class Query {

    private String queryLabel;
    private String attribute;
    private String relationName;
    private String dbName;

public String getQueryLabel() {
    return queryLabel;
}

public void setQueryLabel(String queryLabel) {
    this.queryLabel = queryLabel;
}

public String getAttribute() {
    return attribute;
}

public void setAttribute(String attribute) {
    this.attribute = attribute;
}

public String getRelationName() {
    return relationName;
}

public void setRelationName(String relationName) {
    this.relationName = relationName;
}

public String getDbName() {
    return dbName;
}

public void setDbName(String dbName) {
    this.dbName = dbName;
}

@Override
public String toString() {
    return "Query [queryLabel=" + queryLabel + ", attribute=" + attribute + ", relationName=" + relationName + ", dbName=" + dbName + "]";
}

public Query(String queryLabel, String attribute, String relationName, String dbName) {
    super();
    this.queryLabel = queryLabel;
    this.attribute = attribute;
    this.relationName = relationName;
    this.dbName = dbName;
}

public Query() {
    super();
}
Appendix C: Evaluation Results

This section shows the screenshot highlighting the impact assessment graph for the different artifacts considered in the case study. Each of the results is accompanied by the corresponding Cypher query that was created and executed in Neo4j to determine the above impact.

C.1 Drop attribute from operational schema

`sakila_op1.staff.first_name`

Consider the case of dropping an attribute `first_name` from the relation `staff` in the operational schema, `sakila_op1`. Figure C.1 shows the presented impact.

Figure C.2 presents the corresponding query that was created and executed in Neo4j to determine the above impact.

C.2 Rename attribute from operational schema

`sakila_op1.customer.active`

Consider the case of dropping an attribute `active` from the relation `customer` in the operational schema, `sakila_op1`. Figure C.3 shows the presented impact.

Figure C.4 presents the corresponding query that was created and executed in Neo4j to determine the above impact.
C.3  Drop attribute from reconciled schema

*sakila_ids.film.title*

Consider the case of changing the datatype for the attribute *title* from the relation *film* in the reconciled schema, *sakila_ids*. Figure C.5 shows the query and the presented impact.

Figure C.6 presents the corresponding query that was created and executed in Neo4j to determine the above impact.

C.4  Rename attribute from reconciled schema

*sakila_ids.category.name*

Consider the case of renaming an attribute *name* from the relation *category* in the reconciled schema, *sakila_ids*. Figure C.7 shows the query and the presented impact.
Figure C.8 presents the corresponding query that was created and executed in Neo4j to determine the above impact.

C.5 Change datatype for attribute from reconciled schema

sakila_ids.actor.first_name

Consider the case of dropping an attribute first_name from the relation actor in the reconciled schema, sakila_ids. Figure C.9 shows the query and the presented impact.

Figure C.10 presents the corresponding query that was created and executed in Neo4j to determine the above impact.
Figure C.2: Cypher query for impact of sakila_op1.staff.first_name

1. MATCH (a:Attribute{name:'first_name', relation:'staff', schema:'sakila_op1'})
2. WITH a
3. OPTIONAL MATCH (a)-[r1]->(distance1_pk_fk) where r1.type='PrimaryKey' OR r1.type='ForeignKey'
4. WITH a, r1, distance1_pk_fk
5. OPTIONAL MATCH (a)-[rq]->(query Query)
6. WITH a, r1, distance1_pk_fk, rq, query
7. OPTIONAL MATCH (a)-[r2]->(distance1_step:Step)
8. WITH a, r1, distance1_pk_fk, r2, distance1_step, rq, query
9. OPTIONAL MATCH (distance1_step)-[r3]->(ids_gateway_step:Step)
10. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, rq, query
11. OPTIONAL MATCH (ids_gateway_step)-[r4]->(ids_attributes:Attribute)
12. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, rq, query
13. OPTIONAL MATCH (ids_attributes)-[r5]->(ids_pk_fk) where r5.type='PrimaryKey' OR r5.type='ForeignKey'
14. WITH a, r1, distance1_pk fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_fk, rq, query
15. OPTIONAL MATCH (ids_attributes)-[rq2]->(query_ids:Query)
   WITH a, r1, distance1_pk fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_fk, rq, query, rq2, query_ids
16. OPTIONAL MATCH (ids_attributes)-[r6]->(dwh_input_step:Step {layer:'ids-dwh'})
17. WITH a, r1, distance1_pk fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_fk, r6, dwh_input_step, rq, query, rq2, query_ids
18. OPTIONAL MATCH (dwh_input_step)-[r7]->(dwh_output_step:Step {layer:'ids-dwh'})
19. WITH a, r1, distance1_pk fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_fk, r6, dwh_input_step, r7, dwh_output_step, rq, query, rq2, query_ids
20. OPTIONAL MATCH (dwh_output_step)-[r8]->(dwh_attr:Attribute {schema:'sakila_dwh'})
21. WITH a, r1, distance1_pk fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_fk, r6, dwh_input_step, r7, dwh_output_step, r8, dwh_attr, rq, query, rq2, query_ids
22. OPTIONAL MATCH (dwh_attr)-[r9]->(dwh_query:Query)
23. RETURN a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, r5, ids_pk_fk, r6, dwh_input_step, r7, dwh_output_step, r8, dwh_attr, r9, dwh_query, rq, query, rq2, query_ids
Figure C.3: The complete path illustrating the artifacts impacted by drop attribute operation on *sakila_op1.customer.active*
1. MATCH (a:Attribute{name='active', relation:'customer', schema:'sakila_op1'})
2. WITH a
3. OPTIONAL MATCH (a)-[r1]->(distance1_pk_fk) where r1.type='PrimaryKey' OR r1.type='ForeignKey'
4. WITH a, r1, distance1_pk_fk
5. OPTIONAL MATCH (a)[-rq]->(query Query)
6. WITH a, r1, distance1_pk_fk, rq, query
7. MATCH (a)[-r2]->(distance1_step:Step)
8. WITH a, r1, distance1_pk_fk, r2, distance1_step, rq, query
9. MATCH (distance1_step)[-r3]->(ids_gateway_step:Step)
10. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, rq, query
11. MATCH (ids_gateway_step)[-r4]->(ids_attributes:Attribute)
12. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes, rq, query
13. OPTIONAL MATCH (ids_attributes)[-r5]->(ids_pk_fk) where r5.type='PrimaryKey' OR r5.type='ForeignKey'
14. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes,r5, ids_pk_fk, rq, query
15. OPTIONAL MATCH (ids_attributes)[-r6]->(query_ids:Query)
16. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes,r5, ids_pk_fk, rq, query, rq2, query_ids
17. OPTIONAL MATCH (ids_attributes)[-r7]->(dwh_input_step:Step {layer:'ids-dwh'})
18. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes,r5, ids_pk_fk, r6, dwh_input_step, rq, query, rq2, query_ids
19. dwh_input_step, rq, query, rq2, query_ids
20. OPTIONAL MATCH (dwh_input_step)[-r8]->(dwh_output_step:Step {layer:'ids-dwh'})
21. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes,r5, ids_pk_fk, r6, dwh_input_step, r7, dwh_output_step, rq, query, rq2, query_ids
22. OPTIONAL MATCH (dwh_output_step)[-r9]->(dwh_attr:Attribute {schema:'sakila_dwh'})
23. WITH a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes,r5, ids_pk_fk, r6, dwh_input_step, r7, dwh_output_step, r8, dwh_attr, rq, query, rq2, query_ids
24. OPTIONAL MATCH (dwh_attr)[-r10]->(dwh_query Query)
25. RETURN a, r1, distance1_pk_fk, r2, distance1_step, r3, ids_gateway_step, r4, ids_attributes,r5, ids_pk_fk, r6, dwh_input_step, r7, dwh_output_step, r8, dwh_attr, r9, dwh_query, rq, query, rq2, query_ids

Figure C.4: Cypher query for impact of sakila_op1.customer.active
Figure C.5: The complete path illustrating the artifacts impacted by attribute, *sakila_ids.film.title*

```
1. MATCH (a:Attribute {name:'title', relation: 'film', schema:'sakila_ids'})
2. WITH a
3. OPTIONAL MATCH (a)-[r1]->(distance1_pk_fk) where r1.type='PrimaryKey' OR
   r1.type='ForeignKey'
4. WITH a, r1, distance1_pk_fk
5. OPTIONAL MATCH (a)-[rq]-(query:Query)
6. WITH a, r1, distance1_pk_fk, rq, query
7. OPTIONAL MATCH (t:Step {layer:'oprids'})-[r2]-(a)
8. WITH a, r1, distance1_pk_fk, t, r2, rq, query
9. OPTIONAL MATCH (a)-[r9:1..2]-(dwh_steps:Step {layer:'ids-dwh'})
10. WITH a, r1, distance1_pk_fk, r3, dwh_steps, r2, rq, query
11. OPTIONAL MATCH (dwh_steps)-[r4]-(dwh_attr:Attribute {schema:'sakila_dwh'})
12. WITH a, r1, distance1_pk_fk, r2, dwh_steps, r3, dwh_attr, r4, rq, query
13. OPTIONAL MATCH (dwh_attr)-[r5]-(dwh_query:Query)
14. RETURN a, r1, distance1_pk_fk, r2, dwh_steps, r3, dwh_attr, dwh_query, r5, r4, rq, query
```

Figure C.6: Cypher query for impact of *sakila_ids.film.title*
Figure C.7: The complete path illustrating the artifacts impacted by sakila_ids.category.name

1. MATCH (a:Attribute {name: 'name', relation: 'category', schema:'sakila_ids'})
2. WITH a
3. OPTIONAL MATCH (a)-[r1]->(distance1_pk_fk) where r1.type='PrimaryKey' OR r1.type='ForeignKey'
4. WITH a, r1, distance1_pk_fk
5. OPTIONAL MATCH (a)-[rq]->(query:Query)
6. WITH a, r1, distance1_pk_fk, rq, query
7. OPTIONAL MATCH (t:Step {layer:'opr-ids'})-[r2]->(a)
8. WITH a, r1, distance1_pk_fk, t, r2, rq, query
9. OPTIONAL MATCH (a)-[r3*1..2]->(dwh_steps:Step {layer:'ids-dwh'})
10. WITH a, r1, distance1_pk_fk, r3, dwh_steps, r2, rq, query
11. OPTIONAL MATCH (dwh_steps)-[r4]->(dwh_attr:Attribute {schema:'sakila_dwh'})
12. WITH a, r1, distance1_pk_fk, r2, dwh_steps, r3, dwh_attr, r4, rq, query
13. OPTIONAL MATCH (dwh_attr)-[r5]->(dwh_query:Query)
14. RETURN a, r1, distance1_pk_fk, r2, dwh_steps, r3, dwh_attr, dwh_query, r5, r4, rq, query

Figure C.8: Cypher query for sakila_ids.category.name
Figure C.9: The complete path illustrating the artifacts impacted by *sakila_ids.actor.first_name*

1. MATCH (a:Attribute{name: 'name', relation: 'category', schema:'sakila_ids'})
2. WITH a
3. OPTIONAL MATCH (a)-[r1]->(distance1.pk fk) where r1.type='PrimaryKey' OR r1.type='ForeignKey'
4. WITH a, r1, distance1.pk fk
5. OPTIONAL MATCH (a)-[rq]->(query:Query)
6. WITH a, r1, distance1.pk fk, rq, query
7. OPTIONAL MATCH (t:Step {layer:'opr-ids'})-[r2]->(a)
8. WITH a, r1, distance1.pk fk, t, r2, rq, query
9. OPTIONAL MATCH (a)-[r3^1..2]->(dwh_steps:Step {layer:'ids-dwh'})
10. WITH a, r1, distance1.pk fk, r3, dwh_steps, r2, rq, query
11. OPTIONAL MATCH (dwh_steps)-[r4]->(dwh_attr:Attribute{schema:'sakila_dwh'})
12. WITH a, r1, distance1.pk fk, r2, dwh_steps, r3, dwh_attr, r4, rq, query
13. OPTIONAL MATCH (dwh_attr)[r5]->(dwh_query:Query)
14. RETURN a, r1, distance1.pk fk, r2, dwh_steps, r3, dwh_attr, dwh_query, r5, r4, rq, query

Figure C.10: Cypher query for *sakila_ids.actor.first_name*
C.6 Drop attribute from the warehouse schema

\textit{sakila.dwh.dim.date.day.abbreviation}

Consider the case of dropping an attribute \textit{day.abbreviation} from the relation \textit{dim.date} in the warehouse schema, \textit{sakila.dwh}. Figure C.11 shows the presented impact.

Figure C.12 presents the corresponding query that was created and executed in Neo4j to determine the above impact.

C.7 Rename attribute from the warehouse schema

\textit{sakila.dwh.dim.film.film.in.category.foreign}

Consider the case of renaming an attribute \textit{film.in.category.foreign} from the relation \textit{dim.film} in the warehouse schema, \textit{sakila.dwh}. Figure C.13 shows the query and the presented impact.

Figure C.14 presents the corresponding query that was created and executed in Neo4j to determine the above impact.
1. MATCH (a:Attribute {name: 'day_abbreviation', relation: 'dim_date', schema:'sakila_dwh'})
2. WITH a
3. OPTIONAL MATCH (a)-[r1]->(distance1_pk_fk) where r1.type='PrimaryKey' OR r1.type='ForeignKey'
4. WITH a, r1, distance1_pk_fk
5. OPTIONAL MATCH (a)-[rq]->(query:Query)
6. WITH a, r1, distance1_pk_fk, rq, query
7. OPTIONAL MATCH (dwh_output:Step {layer:'ids-dwh'})-[r2]->(a)
8. WITH a, r1, distance1_pk_fk, r2, rq, query, dwh_output
9. OPTIONAL MATCH (a)-[r3]->(dwh_input:Step {layer:'ids-dwh'})
10. RETURN a, r1, distance1_pk_fk, r2, r3, rq, query, dwh_output, dwh_input

Figure C.12: Cypher query for sakila_dwh.dim_date.day_abbreviation

C.8 Change datatype for an attribute in the warehouse schema sakila_dwh.dim_customer.customer_key

Consider the case of dropping an attribute customer_key from the relation dim_customer in the warehouse schema, sakila_dwh. Figure C.15 shows the query and the presented impact.

Figure C.16 presents the corresponding query that was created and executed in Neo4j to determine the above impact.

C.9 Rename relation in the operational schema sakila_op2.film_category

Consider the case of dropping a relation film_category from the operational schema sakila_op2. Figure C.17 shows the query and the presented impact.

Figure C.18 presents the corresponding query that was created and executed in Neo4j to determine the above impact.
C.10 Drop relation in the reconciled schema \textit{sakila_ids.film_category}

Consider the case of dropping relation \textit{film_category} from the reconciled schema \textit{sakila_ids}. Figure C.19 shows the query and the presented impact.

Figure C.20 presents the corresponding query that was created and executed in Neo4j to determine the above impact.

C.11 Drop relation in the reconciled schema \textit{sakila_ids.store}

Consider the case of dropping an attribute \textit{store} from the reconciled schema \textit{sakila_ids}. Figure C.21 shows the query and the presented impact.

Figure C.22 presents the corresponding query that was created and executed in Neo4j to determine the above impact.
1. MATCH (a:Attribute{name: 'film_in_category_foreign', relation: 'dim_film', schema:'sakila_dwh'})
2. WITH a
3. OPTIONAL MATCH (a)-[r1]-(distance1_pk_fk) where r1.type='PrimaryKey' OR r1.type='ForeignKey'
4. WITH a, r1, distance1_pk_fk
5. OPTIONAL MATCH (a)-[rq]-(query:Query)
6. WITH a, r1, distance1_pk_fk, rq, query
7. OPTIONAL MATCH (dwh_output:Step {layer:'ids-dwh'})-[r2]-(a)
8. WITH a, r1, distance1_pk_fk, r2, rq, query, dwh_output
9. OPTIONAL MATCH (a)-[r3]-(dwh_input:Step {layer:'ids-dwh'})
10. RETURN a, r1, distance1_pk_fk, r2, r3, query, dwh_output, dwh_input

Figure C.14: Cypher query for sakila_dwh.dim_film.film_in_category_foreign

C.12 Rename relation in the warehouse schema sakila_dwh.dim_store

Consider the case of renaming the relation dim_store from the warehouse schema sakila_dwh. Figure C.23 shows the query and the presented impact.

Figure C.24 presents the corresponding query that was created and executed in Neo4j to determine the above impact.

C.13 Evolving reconciled schema sakila_ids

Consider the case of evolving the schema sakila_ids. The operation will have an impact on the queries expressed over the schema, ETL transformations between sakila_op1 and reconciled schema sakila_ids that load data into the relations of sakila_ids. Furthermore, some transformations between the reconciled and warehouse schema will also break. Figure C.25 shows the presented impact.
Figure C.15: The complete path illustrating the artifacts impacted by drop attribute `sakila_dwh.dim_customer.customer_key` operation

Figure C.26 presents the corresponding query that was created and executed in Neo4j to determine the above impact.

C.14 **Evolving warehouse schema** *sakila_dwh*

Consider the case of evolving the schema, *sakila_dwh*. The operation will have an impact on the queries expressed over the schema, ETL transformations between *sakila_ids* and warehouse schema *sakila_dwh*. Figure C.27 shows the presented impact.

Figure C.28 presents the corresponding query that was created and executed in Neo4j to determine the above impact.
Figure C.16: Cypher query for *sakila_dwh.dim_customer.customer_key*

```
1. MATCH (a:Attribute {name: 'customer_key', relation: 'dim_customer',
    schema: 'sakila_dwh'})
2. WITH a
3. OPTIONAL MATCH (a)-[r1]->(distance1_pk_fk) where r1.type='PrimaryKey' OR
   r1.type='ForeignKey'
4. WITH a, r1, distance1_pk_fk
5. OPTIONAL MATCH (a)-[rq]->(query:Query)
6. WITH a, r1, distance1_pk_fk, rq, query
7. OPTIONAL MATCH (dwh_output:Step {layer:'ids-dwh'})-[r2]->(a)
8. WITH a, r1, distance1_pk_fk, r2, rq, query, dwh_output
9. OPTIONAL MATCH (a)-[r3]->(dwh_input:Step {layer:'ids-dwh'})
10. RETURN a, r1, distance1_pk_fk, r2, r3, rq, query, dwh_output, dwh_input
```

Figure C.17: The complete path illustrating the artifacts impacted by relation, *sakila_op2.film_category*
MATCH (rel:Relation{name: 'film_category', schema: 'sakila_op2'})
WITH rel
MATCH (rel)-[r1]->(s:Step{layer: 'opr-ids'})
WITH rel, r1, s
OPTIONAL MATCH (rel)-[r2]->(q:Query)
WITH rel, r1, s, r2, q
OPTIONAL MATCH (s)-[r3]->(s2)
WITH rel, r1, s, r2, q, r3, s2
OPTIONAL MATCH (s2)-[r4]->(ids_relation:Relation)
WITH rel, r1, s, r2, q, r3, s2, r4, ids_relation
OPTIONAL MATCH (ids_relation)-[rq2]->(query_ids:Query)
WITH rel, r1, s, r2, q, r3, s2, r4, ids_relation, rq2, query_ids
OPTIONAL MATCH (ids_relation)-[r6]->(dwh_input_step:Step{layer: 'ids-dwh'})
WITH rel, r1, s, r2, q, r3, s2, r4, ids_relation, rq2, query_ids, r6, dwh_input_step
OPTIONAL MATCH (dwh_input_step)-[r7]->(dwh_output_step:Step{layer: 'ids-dwh'})
WITH rel, r1, s, r2, q, r3, s2, r4, ids_relation, rq2, query_ids, r6, dwh_input_step, r7, dwh_output_step
OPTIONAL MATCH (dwh_output_step)-[r8]->(dwh_rel:Relation{schema: 'sakila_dwh'})
WITH rel, r1, s, r2, q, r3, s2, r4, ids_relation, rq2, query_ids, r6, dwh_input_step, r7, dwh_output_step, r8, dwh_rel
OPTIONAL MATCH (dwh_rel)-[r9]->(dwh_query:Query)
WITH rel, r1, s, r2, q, r3, s2, r4, ids_relation, rq2, query_ids, r6, dwh_input_step, r7, dwh_output_step, r8, dwh_rel, r9, dwh_query
OPTIONAL MATCH (rel)-[attr]->(a:Attribute)-[fk]->(a2:Attribute)
RETURN rel, r1, s, r2, q, r3, s2, r4, ids_relation, rq2, query_ids, r6, dwh_input_step, r7, dwh_output_step, r8, dwh_rel, r9, dwh_query, attr, a, fk, a2

Figure C.18: Cypher query for the impact of rename relation operation on sakila_op2.film_category
Figure C.19: The complete path illustrating the artifacts impacted by sakila_ids.film_category

1. MATCH (rel:Relation{name: 'film_category', schema: 'sakila_ids'})
2. WITH rel
3. OPTIONAL MATCH (opidsstep:Step{layer:'opr-ids'})-[sa]->(rel)
4. WITH rel, opidsstep, sa
5. OPTIONAL MATCH (rel)-[r1]->(s:Step{layer:'ids-dwh'})
6. WITH rel, opidsstep, sa, r1, s
7. OPTIONAL MATCH (rel)-[r2]->(q:Query)
8. WITH rel, opidsstep, sa, r1, r2, q
9. OPTIONAL MATCH (s)-[r3]->(s2)
10. WITH rel, opidsstep, sa, r1, s, r2, q, r3, s2
11. OPTIONAL MATCH (s2)-[r4]->(dwh_rel:Relation)
12. WITH rel, opidsstep, sa, r1, s, r2, q, r3, s2, r4, dwh_rel
13. OPTIONAL MATCH (dwh_rel)-[r9]->(dwh_query:Query)
14. WITH rel, opidsstep, sa, r1, s, r2, q, r3, s2, r4, dwh_rel, r9, dwh_query
15. OPTIONAL MATCH (rel)-[attr]->(a:Attribute)-[fk]->(a2:Attribute) where fk.type='ForeignKey'
16. RETURN rel, opidsstep, sa, r1, s, r2, q, r3, s2, r4, dwh_rel, r9, dwh_query, attr, a,fk,a2

Figure C.20: Cypher query for sakila_ids.film_category
Figure C.21: The complete path illustrating the artifacts impacted by `sakila_ids.store`

1. MATCH (rel:Relation{name: 'store', schema: 'sakila_ids'})
2. WITH rel
3. OPTIONAL MATCH (opidsstep:Step{layer:'opr-ids'}-[sa]->(rel)
4. WITH rel, opidsstep, sa
5. OPTIONAL MATCH (rel)-[r1]->(s:Step{layer:'ids-dwh'})
6. WITH rel, opidsstep, sa, r1, s
7. OPTIONAL MATCH (rel)-[r2]->(q:Query)
8. WITH rel, opidsstep, sa, r1, s, r2, q
9. OPTIONAL MATCH (s)-[r3]->(s2)
10. WITH rel, opidsstep, sa, r1, s, r2, q, r3, s2
11. OPTIONAL MATCH (s2)-[r4]->(dwh_re:Relation)
12. WITH rel, opidsstep, sa, r1, s, r2, q, r3, s2, r4, dwh_re
13. OPTIONAL MATCH (dwh_re)-[r9]->(dwh_query Query)
14. WITH rel, opidsstep, sa, r1, s, r2, q, r3, s2, r4, dwh_re, r9, dwh_query
15. OPTIONAL MATCH (rel)-[attr]->(a:Attribute)-[fk]->(a2:Attribute) where fk.type='ForeignKey'
16. RETURN rel, opidsstep, sa, r1, s, r2, q, r3, s2, r4, dwh_re, r9, dwh_query, attr, a,fk,a2

Figure C.22: Cypher query for `sakila_ids.store`
Figure C.23: The complete path illustrating the artifacts impacted by drop attribute, *sakila_dwh.dim_store* operation

1. MATCH (rel:Relation{name: 'dim_store', schema: 'sakila_dwh'})
2. WITH rel
3. OPTIONAL MATCH (idsdwhstep:Step{layer:'ids-dwh'})-[sa]->(rel)
4. WITH rel, sa, idsdwhstep
5. OPTIONAL MATCH (rel)-[r9]->(dwh_query:Query)
6. WITH rel, sa, idsdwhstep, r9, dwh_query
7. OPTIONAL MATCH (rel)-[attr]->(a:Attribute)-[fk]->(a2:Attribute) where fk type='ForeignKey'
8. RETURN rel, sa, idsdwhstep, r9, dwh_query, attr, a, fk, a2

Figure C.24: Cypher query for *sakila_dwh.dim_store*
Figure C.25: The complete path illustrating the artifacts impacted by evolving the reconciled schema, *sakila_ids*

```
1. MATCH (sch:Schema{name:'sakila_ids'})
2. WITH sch
3. OPTIONAL MATCH (sch)-[r2]->(q Query)
4. WITH sch, r2, q
5. OPTIONAL MATCH (opridstep:Step[layer:'opr-ids'])-[rpr1]->(sch)
   WITH sch, r2, q, opr1, opridstep
6. OPTIONAL MATCH (sch)-[r4]->(idsdwhstep:Step[layer:'ids-dwh'])
7. WITH sch, r2, q, opr1, opridstep, r4, idsdwhstep
8. OPTIONAL MATCH (idsdwhstep)-[dwhetl]->(idsdwhstep2:Step[layer:'ids-dwh'])
9. WITH sch, r2, q, opr1, opridstep, r4, idsdwhstep, dwhetl, idsdwhstep2
10. OPTIONAL MATCH (idsdwhstep2)-[r6]->(dwh_schema:Schema)
11. WITH sch, r2, q, opr1, opridstep, r4, idsdwhstep, dwhetl, idsdwhstep2, r6, dwh_schema
12. OPTIONAL MATCH (dwh_schema)-[r7]->(dwh_query:Query)
13. RETURN sch, r2, q, opr1, opridstep, r4, idsdwhstep, dwhetl, idsdwhstep2, r6, dwh_schema, r7, dwh_query
```

Figure C.26: Cypher query for impact of evolving *sakila_ids*
Figure C.27: The complete path illustrating the artifacts impacted by evolving the warehouse schema, sakila_dwh

MATCH (sch:Schema{name:'sakila_dwh'})
WITH sch
OPTIONAL MATCH (sch)-[r2]->(q:Query)
WITH sch, r2, q
OPTIONAL MATCH (idsdwhstep:Step{layer:'ids-dwh'})-[dwhetl]->(sch)
WITH sch, r2, q, idsdwhstep, dwhetl
RETURN sch, r2, q, idsdwhstep, dwhetl

Figure C.28: Cypher query for impact of evolving sakila_dwh
Appendix D: Pre-requisites and manual steps for software tool

D.1 Prerequisites for transforming schemas to a graph representation: index.jsp

The user interface for converting input schemas (sakila_op1, sakila_op2, sakila_ids, and sakila_dwh) to a graph is captured by the front-end program: index.jsp. This section lists the requirements needed before executing this endpoint.

1. For each schema that is converted to a graph representation, the current project must contain a sub-folder with the same name as the schema. This sub-folder should contain three CSV files: Relations.csv, Attributes.csv, and ReferentialIntegrityConstraints.csv.

D.2 Prerequisites and manual steps for transforming ETL workflows to a graph representation: etl.jsp

The user interface for converting ETL workflows to a graph is captured by the front-end program: etl.jsp. This section lists the requirements that are needed before executing this endpoint. We also list the Cypher queries that are executed manually to complete the transformation.
1. Each of the workflows in Pentaho are exported in XML format using the native export feature available within pentaho. The XML files are saved in a folder and the complete path of this folder is specified on the user interface etl.jsp. Figure D.1 shows an example.

![Figure D.1: Sample user input for user interface etl.jsp](image)

2. On clicking the button *Convert from XML to CSV*, a set of CSV files are generated. Depending on the selection in the dropdown menu in the user interface (Operational-Reconciled, Reconciled-Warehouse), the files are generated in the folders *op-ids*, or *ids-dwh* respectively. The files *input_steps.csv*, and *dbjoin_steps* are edited. These files correspond to the information about ETL steps of type *TableInput*, and *Database Join* and these steps contain select-from SQL queries. The attribute names from each of the SQL queries are extracted and captured in the column *AttributeName* in the CSV files. Some SQL queries are also listed in the select star notation (select *). Thus the list of attributes from the particular
relation in the query are manually added to the CSV file. Figure D.2 shows an sample of the generated CSV file *input_steps.csv*.

![Figure D.2: Limitation of the generated CSV data for the user interface etl.jsp](image)

Figure D.2: Limitation of the generated CSV data for the user interface *etl.jsp*

Figure D.3 shows an sample of the CSV file *input_steps.csv* that is edited with the extracted attribute related information.

![Figure D.3: The revised CSV file *input_steps.csv* after performing edits](image)

Figure D.3: The revised CSV file *input_steps.csv* after performing edits

3. Extra relationships are generated for a few transformations. The reason is related to the design of the workflows and the Java application code corresponding to it. In the generated Neo4j graph, for each ETL transformation, an ETL step that reads data from a database is connected to an ETL step of one of the
following types `TableOutput`, `Insert Update`, `DimensionLookupUpdate`, `Combina-
tionLookupUpdate`. For the last two step types, they are modelled as output steps
depending on whether they are performing lookups or updates.

The reason for spurious relationships is related to the design of ETL workflows
and how our code in Java handles the mapping of ETL transformations to a graph.

Figure D.4 shows an example of an ETL transformation. In this transformation,
there are two steps `Insert Update` and `Insert Update 2` which are representing the
output steps. The correct mapping of the given transformation to a graph maps
each of the steps in the top row to the output step `Insert Update` and each of the
steps in the bottom row to the step `Insert Update 2`.

However, when the graph is generated through the tool, each of the input steps
in the transformation is mapped to both of the output steps `Insert Update` and
`Insert Update 2`. The reason for the generation of this incorrect mapping is that
we relate an ETL step of type `TableInput`, `Database Lookup` and `Database Join`
to any step type that is of type `Insert/Update`, `TableOutput`, `DimensionLookup`
or `CombinationLookup`. Figure D.5 shows the transformation of the `load_rental`

Figure D.4: An ETL transformation containing two parallel workflows
workflow to Neo4j graph using our tool. The graph shows how each of the input steps are connected to both of the output steps.

![Graph showing ETL workflow transformation to Neo4j](image)

**Figure D.5:** Transformation of ETL workflow *load_rental* to Neo4j using our tool

Following Cypher queries are executed manually in the Neo4j browser to address this limitation and delete extra relationships.

```cypher
MATCH (s:Step)-[r] -> (s2:Step {name:'Insert / Update 2'})
where s.transname='load_rental'
and (s.name='ds1.rental' OR s.name='Lookup customer_id'
OR s.name='Lookup inventory id' OR s.name='Lookup staff id')
DELETE r

MATCH (s:Step)-[r]->(s2:Step {name:'Insert / Update'})
where s.transname='load_rental' and
(s.name='ds2 rental' OR s.name='Lookup customer_id 2'
OR s.name='Lookup film id' OR s.name='Lookup staff id 2')
DELETE r
```

351
Similarly, extra relationships are generated for the following transformations as well: *load_language*, *load_film*, *load_dim_film*, and *load_fact_rental*. Following Cypher queries are executed to delete the extra relationships.

- Delete extra relationships in the transformation *load_language*

```
MATCH (s:Step)-[r]->(s2:Step {name:'Insert / Update 2'})
where s.transname='load\_language'
and (s.name='Extract language from ds1' OR s.name='max\_language\_last\_update')
DELETE r
```

```
MATCH (s:Step)-[r]->(s2:Step {name:'Insert / Update'})
where s.transname='load\_language'
and (s.name='Extract language from ds2')
DELETE r
```

-- Delete extra relationships from the transformation *load_film*

```
MATCH (s:Step)-[r]->(s2:Step {name:'Insert / Update 2'})
where s.transname='load\_film' and (s.name='Read ds1.film')
DELETE r
```

```
MATCH (s:Step)-[r]->(s2:Step{name:'Insert / Update'})
where s.transname='load\_film' AND s.name='Read ds2.film'
DELETE r
```

```
MATCH (s:Step)-[r]->(s2:Step {name:'Insert / Update 2'})
where s.transname='load\_film'
and (s.name='Lookup language\_id' OR s.name='Lookup original\_language\_id')
DELETE r
```

```
MATCH (s:Step)-[r]->(s2:Step {name:'Insert / Update'})
where s.transname='load\_film' and (s.name='Lookup language\_id 2')
DELETE r
```

-- Delete extra relationships from *load_dim_film*

```
MATCH (s:Step)-[r]->(s2:Step {name:'dim\_film\_actor\_bridge'})
where s.transname='load\_dim\_film' and
(s.name='Lookup Category' OR s.name='Lookup Original Language'
 OR s.name='max\_dim\_film\_last\_update' OR s.name='Film'
 OR s.name='Lookup Language' OR s.name='Get film categories')
DELETE r
```

MATCH (s:Step)-[r]->(s2:Step {name:'Load dim_film'})
where s.transname='load_dim_film'
and (s.name='Get film\_actor ' OR s.name='Lookup dim\_actor' )
DELETE r

-- Delete extra relationships from load_fact_rental
MATCH (s)-[r]->(s2:Step) where s.transname='load_fact_rental'
and (s2.name='Lookup dim staff key'
OR s2.name='Lookup dim customer key' OR s2.name='Lookup dim store key')
DELETE r

D.3 Prerequisites and manual steps for transforming SQL queries to a graph representation: query.jsp

This section presents the required files needed for the successful execution of the functionality captured in the user interface query.jsp.

1. The SQL queries are converted to XML format using the GSQLParser utility.
   The generated output in XML format is saved in a file queries-xml.xml. In the user interface query.jsp as shown in Figure D.6, specify the folder location which contains the file queries-xml.xml.

2. Few queries are not parsed correctly by GSQLParser. These queries contain nested function calls and they are missed during the transformation. For example, consider the following expression in a SQL query:
   \textit{CONCAT(UCASE(SUBSTR(actor.last\_name,1,1))}. GSQLParser does not identify the attribute \textit{actor.last\_name} in this case.

   Thus, the CSV file that is generated through the \textit{Convert from XML to CSV} functionality in the user interface is manually edited. The attributes embedded within the nested function calls are added manually to the CSV file.
Figure D.6: Specifying the folder location in the user interface query.jsp
Bibliography


