R2G: a Tool for Migrating Relations to Graphs

Roberto De Virgilio
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
dvr@dia.uniroma3.it

Antonio Maccioni
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
maccioni@dia.uniroma3.it

Riccardo Torlone
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
torlone@dia.uniroma3.it

ABSTRACT

We present **R2G**, a tool for the automatic migration of databases from a relational to a Graph Database Management System (GDBMS). GDBMSs provide a flexible and efficient solution to the management of graph-based data (e.g., social and semantic Web data) and, in this context, the conversion of the persistent layer of an application from a relational to a graph format can be very beneficial. **R2G** provides a thorough solution to this problem with a minimal impact to the application layer: it transforms a relational database **r** into a graph database **g** and any conjunctive query over **r** into a graph query over **g**. Constraints defined over **r** are suitably used in the translation to minimize the number of data access required by graph queries. The approach refers to an abstract notion of graph database and this allows **R2G** to map relational database into different GDBMSs. The demonstration of **R2G** allows the direct comparison of the relational and the graph approaches to data management.

1. INTRODUCTION

Graphs provide a natural representation of data in several application domains such as computer networks, biology, geomatics and, more recently, social networks and the Semantic Web. The need of these kind of applications to manage highly-connected data in an effective and efficient way has originated a brand new category of storage systems, usually called graph database management systems (GDBMS). In such systems data are stored natively in graph structures and queries are defined in terms of graph traversals. While in RDBMSs relationships between data in different tables are represented by means of values that appear in tuples, in GDBMS data are stored in nodes and references are represented explicitly by means of edges between nodes. This allows GDBMSs to scale more naturally to large sets of graph-based data as they do not require expensive join operations. In addition, since GDBMSs do not rely on a rigid schema, they are more flexible in situations where the schema evolves rapidly.

In this framework, the migration of databases from a relational to a graph-based storage system can be beneficial for applications but clearly, an automatic support to this process is essential. Actually, many tools and techniques have been developed for this purpose, in particular, for migrating relational databases to RDF [7]. However, the various approaches provide ad-hoc solutions, as they usually consider a specific target system or data model and rely on rather naive techniques (e.g., tuples are simply mapped to nodes and foreign keys to edges) that do not fully exploit the features of the target systems and do not consider the query load. Moreover, and more important, they disregard the important issue of query mapping: how to translate queries over the source into queries over the target. This is clearly fundamental to suitably adapt the application layer to the new persistent layer in an effective and efficient way.

We provide a contribution to this problem by presenting **R2G**, a tool for the automatic migration of databases from a relational to a graph storage system that converts a relational database **r** into a graph database **g** and any conjunctive query over **r** into a traversal of **g**. The translation takes advantage of the constraints defined over the source for minimizing the number of accesses needed to answer queries over the target. Intuitively, this is done by storing in the same node of the target the data that are likely to occur together in query results. Another important issue is that our approach relies on an abstract notion of graph database and this allows **R2G** to store the target graph database into different GDBMSs.

The demonstration of **R2G** aims at illustrating and discussing with the audience the following issues:

- how a relational database can be transformed into a graph database in which the entities and relationships of the source application are naturally represented in terms of nodes and edges;
- how conjunctive queries on the source database can be translated into graph traversal operations over the target database;
- how the number of data accesses needed to answer queries over the target can be minimized by considering the constraints defined over the source;
- how the migration can be made system-independent by using an abstract notion of graph databases that is suitable for different GDBMSs;
- advantages and limitations of the relational and of the graph approach to data management.

The rest of the presentation of **R2G** is organized as follows. In Section 2 we present the graph model adopted in our approach while in Section 3 we provide an overview of the system. In Section 4 we illustrate an outline of the demonstration of our tool and finally, in Section 5, we draw some conclusions. More details on the techniques implemented in **R2G** can be found in [3].

User (US)

	uid	uname
t_1	u01	Date
t_2	u02	Hunt

Follower (FR)

	fuser	fblog
t_3	u01	b01
t_4	u01	b02
t_5	u01	b03
t_6	u02	b01

Blog (BG)

	bid	bname	admin
t_8	b01	Information Systems	u02
t_9	b02	Database	u01
t_{10}	b03	Computer Science	u02

Tag (TG)

	tuser	tcomment
t_7	u02	c01

Comment (CT)

	cid	cblog	cuser	msg	date
t_{11}	c01	b01	u01	This is what I was looking for!	25/02/2013

Figure 1: A relational database.

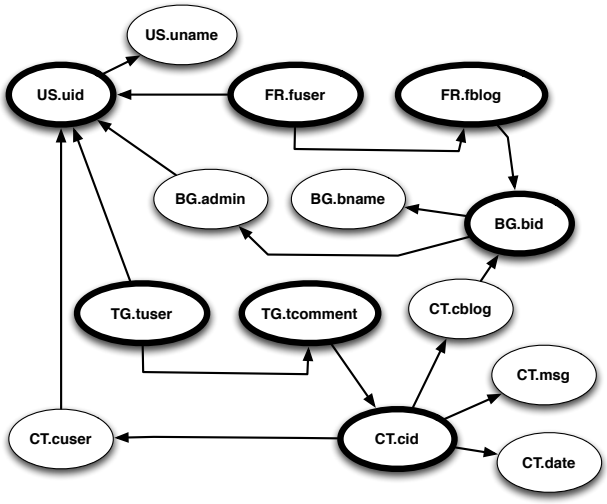

Figure 2: The schema graph for the database in Figure 1.

2. PRELIMINARIES

2.1 Schema graphs and full schema paths

A relational schema can be naturally represented in terms of a graph by considering the constraints defined on it. This representation is used in first step of the conversion of a relational into a graph database and is defined as follows.

Given a relational schema $\mathcal{R}$, the *schema graph* **RG** for $\mathcal{R}$ is a directed graph $\langle N, E\rangle$ such that:

- there is a node $A \in N$ for each attribute A of a relation in $\mathcal{R}$,
- there is an edge $(A_i, A_j) \in E$ if one of the following holds: (a) A_i belongs to a key of a relation R in $\mathcal{R}$ and A_j is a non-key attribute of R, (b) A_i, A_j belong to a key of a relation R in $\mathcal{R}$, (c) A_i, A_j belong to R_i and R_j respectively and there is a foreign key between A_i and A_j.

Let us consider for instance the relational database $\mathcal{R}$ for a social application in Figure 1 in which keys are underlined and foreign key can be easily inferred. The schema graph for $\mathcal{R}$ is depicted in Figure 2.

The transformation technique adopted by **R2G** consider the *full schema paths* of a schema graph, that is, the paths from a source node to a sink node. This is because they represent logical relationships between concepts of the database and for this reason they correspond to natural ways to join the tables of the database for answering queries. A full schema path of the graph in Figure 2 is, for instance, the following:

$$\text{TG.tuser} \rightarrow \text{TG.tcomment} \rightarrow \text{CT.cid} \rightarrow \text{CT.msg}$$

2.2 Graph Databases

Recently, graph database models are receiving a new interest with the spread of GDBMSs. Unfortunately, due to diversity of the various systems, there is no accepted definition of data model for GDBMSs and of the features provided by them. However, almost all the existing systems exhibit three main characteristics.

First of all, at physical level, a graph database satisfies the so called *index-free adjacency* property: each node stores information about its neighbors only and no global index of the connections between nodes exists. As a result, the traversal of an edge is basically independent of the size of data. This allows the efficient computation of local analyses on graph-based data and makes GDBMSs well-suited in scenarios where the size of data increases rapidly.

Secondly, a GDBMS stores data in a multigraph (a graph where two nodes can be connected by more than one edge), usually called *property graph* [6], where every node and every edge has associated a set of key-value pairs, called properties. We consider here a simplified version of a property graph where only nodes have properties, which represent actual data, while edges have just labels that represent relationships between data in nodes.

We then simply define a *graph database* as a labeled multigraph $\mathbf{g} = (N, E)$ where every node $n \in N$ is associated with a set of pairs $\langle$*key, value*$\rangle$. An example of graph database is reported in Figure 3. Note that a tuple t over a relation schema $R(X)$ is represented here by set of pairs $\langle A, t[A]\rangle$, where $A \in X$ and $t[A]$ is the restriction of t on A.

The third feature common to GDBMSs is the fact that data is queried using path traversal operations expressed in some graph-based query language, as discussed next.

2.3 Graph Query Languages

The various proposals of query languages for graph data models [8] can be classified into two main categories. The former includes languages, such as SPARQL and Cypher [5], in which queries are expressed as a graph and query evaluation relies on graph matching between the query and the database. A limitation of this approach is that graph matching is expensive on large databases [1]. The latter category includes languages that rely on expressions denoting paths of the database. Among them, we mention Gremlin [5], XPath, and XQuery [4]. These languages, usually called *traversal query languages*, are more suitable for an efficient implementation.

For the sake of generality, in this paper we consider an abstract traversal query language with an XQuery-like syntax. Expressions of this language are based on *path expressions* in which, as usual, square parentheses denote conditions and the slash character (/) denotes the adjacency between nodes and edges. We will also make use of variables, which range over paths and are denoted by the prefix $, of the `for` construct, to iterate over path expressions, and of the `return` construct, to specify the values to return as output.

3. SYSTEM OVERVIEW

3.1 Architecture of the tool

R2G has four main components: (1) the Metadata Analyzer (MA), which inspects schema and constraints of the source relational database and builds the corresponding schema graph, (2) the Data Mapper (DM), which generates the data mapping, (3) the Query Mapper (QM), which is responsible for translating queries, and (4) the Graph Manager (GM), which actually migrates the data and executes queries over the target database by using the mappings generated by the DM and the QM. Both the DM and the QM take advantage of the output of MA.

R2G has been developed in Java and uses the Tinkerpop[1] framework to implement the abstract interface over graph databases. Specifically, the GM component of **R2G** relies on Tinkerpop Blueprints for data management and on Tinkerpop Gremlin for query processing. PostgreSQL and JDBC have been used as source RDBMS and relational API, while Neo4J[2] and OrientDB[3] have been used as target GDBMSs.

The rest of this section sketches the mapping techniques used by the Data and the Query Mapper, respectively. More details on our approach can be found in [3].

3.2 Data Mapping

Existing GDBMSs usually provide ad-hoc relational importers based on a naive approach that generates a node for each tuple occurring in the source database and an edge for each pair of *joinable* tuples, that is, tuples t_1 and t_2 over R_1 and R_2 respectively such that there is a foreign key constraint between $R_1.A$ and $R_2.B$ and $t_1[A] = t_2[B]$. Conversely, in our approach we aggregate values of different tuples in the same node to speed-up traversal operations over the target. The basic idea is to store in the same node data values that are likely to be retrieved together in the evaluation of queries. Intuitively, these values are those that belong to joinable tuple. However, by just aggregating together joinable tuples we could run the risk to accumulate a lot of data in each node.

Therefore, we consider a data aggregation strategy based on a more restrictive property, which we call *unifiability*. In the definition that follows, a multi-key is a key of a relation composed by a set of attributes each of which is a foreign key for a different relation. An example of multi-key is the set of attributes {tuser,tcomment} of relation Tag in Figure 1. Basically, a multi-key implements a many-to-many relationship.

Two data values $v_1 = t_1[A]$ and $v_2 = t_2[B]$ are *unifiable* in a relational database **r** if one of the following holds: (i) $t_1 = t_2$ and both A and B do not belong to a multi-key; (ii) t_1 and t_2 are joinable and A belongs to a multi-key; (iii) t_1 and t_2 are joinable, A and B do not belong to a multi-key and there is no other tuple t_3 that is joinable with t_2.

This notion guarantees a balanced distribution of data among the nodes of the target graph database and an efficient evaluation of queries over the target that correspond to joins over the source. In Figure 3 it is shown the graph database obtained by aggregating in the nodes the unifiable values occurring in the relational database of Figure 1. In this graph, an edge between two nodes n_1 and n_2 with label R_A denotes that values occurring in n_1 and n_2 are related by a foreign key constraint from the attribute A of relation R. For instance, in the graph of Figure 3 there is an edge labeled

[1] http://tinkerpop.com/
[2] http://neo4j.org/
[3] http://www.orientdb.org/orient-db.htm

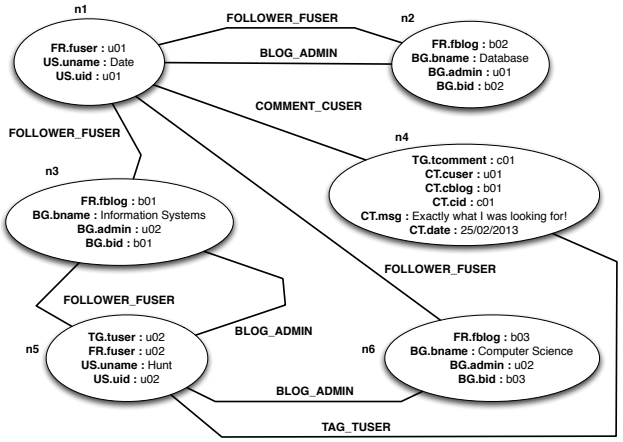

Figure 3: The graph database generated by R2G from the relational database in Figure 1.

BLOG_ADMIN between n_3 and n_5 because of the value u02 of attribute ADMIN in relation BLOG.

In [3] we illustrated a technique for identifying and aggregating efficiently unifiable values. This is done by following the full schema paths of the schema graph for the source relational database.

3.3 Query Mapping

R2G is able to translate *conjunctive* SQL queries (select-projection-join operation) into path traversal operations over the graph database. The translation technique first generates a graph-based structure, called *query template* (QT for short), that denotes the sub-graphs of the target graph database that include the result of the query. A query template is then translated into a path traversal query.

We show the generation of a query template by means of an example. Let us consider the following SQL query Q on top of the relational database of Figure 1.

```
SELECT US.uname
FROM User US, Blog BG
WHERE (BG.admin = US.uid) AND
      (BG.bname = 'Inf. Systems')
```

The query retrieves the names of the users that have commented the *Inf. Systems* blog. The construction of the corresponding query template considers the schema graph **sg** in Figure 2 and proceeds as follows.

1. We identify a minimal set SP of full schema paths in **sg** such that for each join condition $R_i.A_i = R_j.A_j$ occurring in Q, there is an edge $(R_i.A_i, R_j.A_j)$ in at least one sp in SP. Note that this set forms a connected graph since Q is a conjunctive query. Referring to our example, the minimal set we obtain is the following:

   sp_1 : TG.tuser → US.uid → US.uname.
   sp_2 : TG.tuser → TG.tcomment → CT.cid → CT.cblog

2. If there is an attribute in a selection condition $R.A = c$ that does not occur in any full schema path in SP, another full schema path sp that includes both A and an attribute in a full schema path sp' in SP is added to SP;

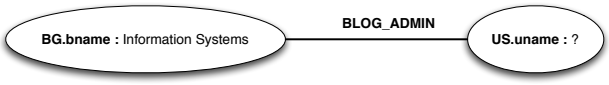

Figure 4: The query template for Q.

3. We build a relational database $\mathbf{r_Q}$ made of: (i) a set of tables $R(A)$ having as instance c for each selection condition $R.A = c$, and (ii) a set of tables $R(A)$ having as instance the special symbol ? for each attribute $R.A$ in the SELECT clause of Q;

4. The final query template is then obtained by applying the data conversion procedure to SP and $\mathbf{r_Q}$, cited in Section 3.2 and described in [3] . We obtain in our example the query template in Figure 4. The label $\langle$US.uname, ?$\rangle$ indicates that the value associated to US.uname is the goal of the query. By matching this query template against the graph database in Figure 3 it is easy to see that the result of the query is Hunt.

The query pattern we obtain is then translated into a graph traversal language expression. By adopting the syntax of the abstract query language mentioned in Section 2, we obtain the following expression:

```
FOR $x in /[BG.bname='Inf. Systems']
    $y in $x/BLOG_ADMIN/*
RETURN $y/US.uname
```

This expression can be finally computed over the target graph database with a graph traversal operation expressed in the specific language adopted by the target GDBMS.

4. DEMONSTRATION OUTLINE

In the demonstration of **R2G** we will use both synthetic and real data sets of different size. In particular we consider MONDIAL (17.115 tuples and 28 relations), ACADEMIC[4] (4.200 tuples) and two ideal counterpoints (due to the larger size): IMDB and WIKIPEDIA (200.000 tuples). You can find more details about these data sets in [2]. The demonstration includes four scenarios that aim at showing all the capabilities of the system in different use cases.

Scenario A. The first scenario shows how the data migration takes place with all the above data sets. The user can become familiar with **R2G** and monitor how different relational databases are transformed into graph databases with our system. The user will also exploit a graphical tool[5] to navigate the content of the graph database and will be able to monitor the status of **R2G** through an interactive front-end.

Scenario B. The second scenario aims at illustrating how the query mapping is performed. We will provide a set of SQL queries for each data set; the user can monitor, step-by-step, how the SQL statements are translated by **R2G** into path traversal expressions. Here, we will use Gremlin as query language. For example, the query Q discussed in Section 3.3 is translated into the following expression:

```
g.V.filter{it.BGbname=='Inf. Systems'}.
  outE.filter{it.label=='BLOG_ADMIN'}.
  inV.'USuname'
```

At the end, the user can perform the same query on both the relational and graph DBMSs and compare the outcomes. The interactive front-end will support the user in this task.

Scenario C. With respect to **Scenario B**, the third scenario allows users to freely submit SQL queries to the system. In response, the user can obtain the result of the query over Neo4J and OrientDB or, alternatively, the translation of the SQL query into a Gremlin expression.

Scenario D. This scenario involves alternative approaches adopted by the import facilities of Neo4J and OrientDB. This serves to illustrate how our techniques outperform the naive solution of the competitors. Since Neo4J and OrientDB do not provide any query mapping facility, we cannot provide any comparison in **Scenario B** and in **Scenario C**, but only in **Scenario A**.

5. CONCLUSION

In this paper we have illustrated **R2G**, a tool for migrating databases from a relational to a Graph Database Management System (GDBMS). **R2G** relies on a general representation of relational data in terms of a graph and provides methods and techniques that, taking into account the integrity constraints defined on the source, map data and queries to the target system in an effective and efficient way. **R2G** provides a support for database reengineering in application domains where the adoption of a GDBMS is convenient and enables, in those domains, the comparison of the relational and the graph-based technology to manage and query data.

6. REFERENCES

[1] C. Bizer and A. Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[2] J. Coffman and A. C. Weaver. A framework for evaluating database keyword search strategies. In *the 19th ACM Conference on Information and Knowledge Management (CIKM)*, pages 729–738, 2010.

[3] R. De Virgilio, A. Maccioni, and R. Torlone. Converting relational to graph databases. In *the first International Workshop on Graph Data Management Experiences and Systems (GRADES), co-located with SIGMOD/PODS*, page 1, 2013.

[4] J. Hidders and J. Paredaens. Xpath/xquery. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 3659–3665. Springer US, 2009.

[5] F. Holzschuher and R. Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *the Second International Workshop on Querying Graph Structured Data (GraphQ), co-located with EDBT/ICDT*, pages 195–204, 2013.

[6] M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.

[7] S. S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. T. Jr, S. Auer, J. Sequeda, and A. Ezzat. A survey of current approaches for mapping of relational databases to rdf. *W3C*. http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf, 2009.

[8] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[4] A synthetic data set based on the schema in Figure 1

[5] https://github.com/neo4j-contrib/neoclipse