Converting Relational to Graph Databases

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

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

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

ABSTRACT

Graph Database Management Systems provide an effective and efficient solution to data storage in current scenarios where data are more and more connected, graph models are widely used, and systems need to scale to large data sets. In this framework, the conversion of the persistent layer of an application from a relational to a graph data store can be convenient but it is usually an hard task for database administrators. In this paper we propose a methodology to convert a relational to a graph database by exploiting the schema and the constraints of the source. The approach supports the translation of conjunctive SQL queries over the source into graph traversal operations over the target. We provide experimental results that show the feasibility of our solution and the efficiency of query answering over the target database.

1. INTRODUCTION

There are several application domains in which the data have a natural representation as a graph. This happens for instance in the Semantic Web, in social and computer networks, and in geographic applications. In these contexts, relational systems are usually unsuitable to store data since they hardly capture their inherent graph structure. Moreover, and more importantly, graph traversals over highly connected data require complex join operations, which can make typical operations on this kind of data inefficient and applications hard to scale. For these reasons, a new brand category of data stores, called GDBMSs (Graph Database Management Systems), is emerging. In GDBMSs data are natively stored as graphs and queries are expressed in terms of graph traversal operations. This allows applications to scale to very large graph-based data sets. In addition, since GDBMSs do not rely on a rigid schema, they provide a more flexible solution in scenarios where the organization of data evolves rapidly. In this framework, the migration of the persistent layer of an application from a relational to a graph-based storage system can be very beneficial. This task can be however very hard for software engineers and a tool supporting this activity, possibly in an automatic way, is clearly essential. Actually, there already exists solutions to this problem [3, 11], but they usually refer to specific target data models, such as RDF. Moreover, they usually follow a naive approach in which, basically, tuples are mapped to nodes and foreign keys to edges, but this approach does not take into account the query load and can make graph traversals expensive. Last, but not least, none of them consider the problem of mapping queries over the source into efficient queries over the target. Yet, this is fundamental to reduce the impact on the logic layer of the application and to provide, if needed, a relational view over the target.

In this paper we propose a comprehensive approach to the automatic migration of databases from relational to graph storage systems. Specifically, our technique converts a relational database **r** into a graph database **g** and maps any conjunctive query over **r** into a graph query over **g**. The translation takes advantage of the integrity constraints defined over the source and try to minimize the number of accesses needed to answer queries over the target. Intuitively, this is done by storing in the same node data that likely occur together in query results. We refer to a general graph data model and a generic query language for graph databases: this makes the approach independent of the specific GDBMSs chosen as a target. In order to test the feasibility of our approach, we have developed a complete system for converting relational to graph databases that implements the above described technique. A number of experiments over available data stores have shown that there is no loss of data in translation, and that queries over the source are translated into efficient queries over the target.

The rest of the paper is organized as follows. Section 6 discusses related works. In Section 2 we introduce some preliminary notions that are used in Section 3 and in Section 4 to illustrate the data and the query mapping technique, respectively. Finally, Section 5 discusses some experimental results and Section 7 sketches conclusions and future works.

2. PRELIMINARIES

A graph data model for relational databases. As usual, we assume that: (i) a relational database schema $\mathbf{R}$ is a set of relation schemas $R_1(X_1), \ldots, R_n(X_n)$, where R_i is the name of the $i-$th relation and X_i is the set of its attributes, and (ii) a relational database $\mathbf{r}$ over $\mathbf{R}$ is a set of relations $r_1, \ldots, r_n$ over $R_1(X_1), \ldots, R_n(X_n)$, respectively, where r_i is a set of tuples over $R_i(X_i)$. In the following, we will underline the attributes of a relation that belong

Proceedings of the First International Workshop on Graph Data Management Experience and Systems (GRADES 2013), June 23, 2013 - New York, NY, USA.

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

Tag (TG)

	tuser	tcomment
t_7	u02	c01

Blog (BG)

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

Comment (CT)

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

Figure 1: An example of relational database

to its primary key and we will denote by $R_i.A \xrightarrow{fk} R_j.B$ a foreign key between the attribute A of a relation R_i and the attribute B of a relation R_j[1]. A relational schema $\mathbf{R}$ can be naturally represented in terms of a graph by considering the keys and the foreign keys of $\mathbf{R}$. This representation will be used in first step of the conversion of a relational into a graph database and is defined as follows.

Definition 1 (Relational Schema Graph). *Given a relational schema $\mathbf{R}$, the relational schema graph $\mathbf{RG}$ for $\mathbf{R}$ is a directed graph $\langle N, E\rangle$ such that: (i) there is a node $A \in N$ for each attribute A of a relation in $\mathbf{R}$ and (ii) 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 $\mathbf{R}$ and A_j is a non-key attribute of R, (b) A_i, A_j belong to a key of a relation R in $\mathbf{R}$, (c) A_i, A_j belong to R_i and R_j respectively and there is a foreign key between $R_i.A_i$ and $R_j.A_j$.*

For instance, let us consider the relational database $\mathbf{R}$ for a social application in Figure 1. Note that this is a typical application scenario for which relational DBMS are considered not suited [9]. It involves the following foreign keys: FR.fuser $\xrightarrow{fk}$ US.uid, FR.fblog $\xrightarrow{fk}$ BG.bid, BG.admin $\xrightarrow{fk}$ US.uid, CT.cblog $\xrightarrow{fk}$ BG.bid, CT.cuser $\xrightarrow{fk}$ US.uid, TG.tuser $\xrightarrow{fk}$ US.uid and TG.tcomment $\xrightarrow{fk}$ CT.cid. Then, the relational schema graph for $\mathbf{R}$ is depicted in Figure 2. We say that a *hub* in a graph is a node having more than one incoming edges, a *source* is a node without incoming edges, and a *sink* is a node without outcoming edges. For instance, in the graph in Figure 2 FR.fuser is a source, CT.date is a sink, and US.uid is a hub. In a relational schema graph we focus our attention on *full schema paths*, i.e., paths from a source node to a sink node. This is because, in relational schema graphs, they represent logical relationships between concepts of the database and for this reason they correspond to natural way to join the tables of the database for answering queries. Referring to Figure 2, we have the full schema paths shown in Figure 3.

Graph Databases. Recently, graph database models are receiving a new interest with the diffusion of GDBMSs. Unfortunately, due to diversity of the various systems and of the lack of theoretical studies on them, 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 on the size of data. This makes a GDBMS very efficient to compute local analysis on graph-based data and makes it suitable in scenarios where data size increases rapidly. Secondly, a GDBMS stores data by means of a multigraph , usually called *property graph* [12], where every node and every edge is associated with 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.

[1]Note that, in this paper, we only consider foreign keys over single attributes. Foreign key over multiple attributes can be managed by means of references to tuple identifiers.

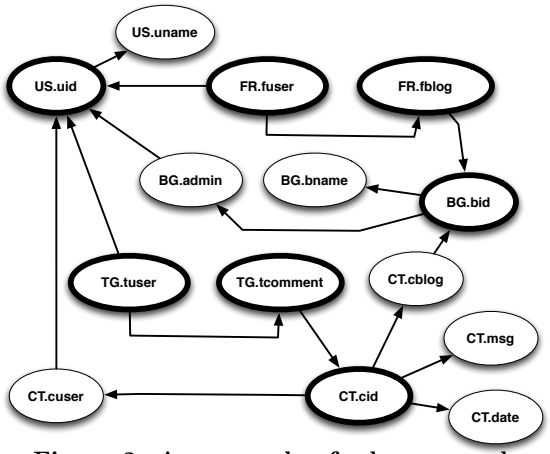

Figure 2: An example of schema graph

sp_1 : FR.fuser → US.uid → US.uname.
sp_2 : FR.fuser → FR.fblog → BG.bid → BG.bname.
sp_3 : FR.fuser → FR.fblog → BG.bid → BG.admin → US.uid → US.uname.
sp_4 : TG.tuser → US.uid → US.uname.
sp_5 : TG.tuser → TG.tcomment → CT.cid → CT.msg.
sp_6 : TG.tuser → TG.tcomment → CT.cid → CT.date.
sp_7 : TG.tuser → TG.tcomment → CT.cid → CT.cblog → BG.bid → BG.bname.
sp_8 : TG.tuser → TG.tcomment → CT.cid → CT.cuser → US.uid → US.uname.
sp_9 : TG.tuser → TG.tcomment → CT.cid → CT.cblog → BG.bid → BG.admin → US.uid → US.uname.

Figure 3: An example of full schema paths

Definition 2 (Graph Database). *A graph database is a multigraph $\mathbf{g} = (N, E)$ where every node $n \in N$ is associated with a set of pairs $\langle key, value\rangle$ and every edge $e \in E$ is associated with a label.*

An example of graph database is reported in Figure 4: it represents a portion of the relational database in Figure 1. Note that a tuple t of 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.

Graph Query Languages. The various proposals of query languages for graph data models [14] can be clas-

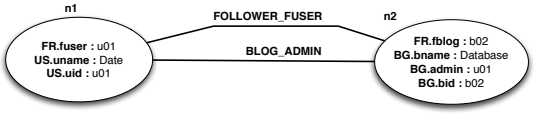

Figure 4: An example of property graph

sified into two main categories. The former includes languages, such as SPARQL and Cypher , in which queries are expressed as a graphs and query evaluation relies on graph matching between the query and the database. The limitation of this approach is that graph matching is very expensive on large databases [4]. The latter category includes languages that rely on expressions denoting paths of the database. Among them, we mention Gremlin , XPath, and XQuery. 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 that adopts an XQuery-like syntax. Expressions of this language are based on *path expressions* in which, as usual, square parentheses denote conditions on nodes and the slash character (/) denotes the relationship between a node n and an edge incoming to or outcoming from n. 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. DATA CONVERSION

This section describes our method for converting a relational database $\mathbf{r}$ into a graph database $\mathbf{g}$. Usually, existing GDBMSs provide ad-hoc importers implementing a naive approach that creates a node n for each tuple t over a schema $R(X)$ occurring in $\mathbf{r}$, such that n has a property $\langle A, t[A]\rangle$ for each attribute $A \in X$. Moreover, two nodes n_1 and n_2 for a pair of tuples t_1 and t_2 are connected in $\mathbf{g}$ if t_1 and t_2 are joined. Conversely, in our approach we try to aggregate values of different tuples in the same node to speed-up traversal operations over $\mathbf{g}$. The basic idea is to try to store in the same node of $\mathbf{g}$ data values that are likely to be retrieved together in the evaluation of queries. Intuitively, these values are those that belong to *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 $t_1[A] = t_2[B]$. Referring to Figure 1, t_{11} and t_8 are joinable tuples, since $\mathsf{CT.cblog} \xrightarrow{fk} \mathsf{BG.bid}$ and $t_{11}[\mathsf{cblog}] = t_8[\mathsf{bid}]$. However, by just aggregating together joinable tuples we could run the risk to accumulate a lot of data in each node, which is not appropriate for graph databases. Therefore, we consider a data aggregation strategy based on a more restrictive property, which we call *unifiability*. First, we need to introduce a preliminary notion. We say that an attribute A_i of a relation R is `n2n` if: (i) A_i belongs to the key $K = \{A_1, \ldots, A_k\}$ of R and (ii) for each A_j of K there exists a foreign key constraints $R.A_j \xrightarrow{fk} R'.B$ for some relation R' in $\mathbf{r}$ different from R. Intuitively, a set of `n2n` attributes of a relation implement a many-to-many relationship between entities. Referring again to Figure 1, FR.fuser and FR.fblog are `n2n`. Then we say that two data values v_1 and v_2 are unifiable in a relational database $\mathbf{r}$ if one of the following holds: (i) there is a tuple t of a relation R in $\mathbf{r}$ such that: $t[A] = v_1$, $t[B] = v_2$, and A and B are not `n2n`, (ii) there is a pair of joinable tuples t_1 and t_2 of relations R_1 and R_2 respectively

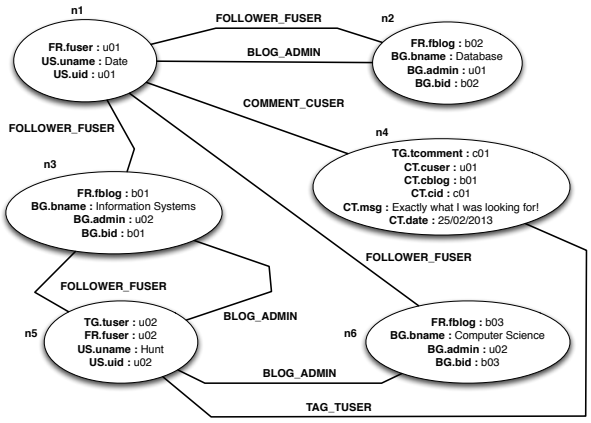

Figure 5: An example of graph database

in $\mathbf{r}$ such that: $t_1[A] = v_1$, $t_2[B] = v_2$, and A is `n2n`, and (iii) there is a pair of joinable tuples t_1 and t_2 of relations R_1 and R_2 respectively in $\mathbf{r}$ such that: $t_1[A] = v_1$, $t_2[B] = v_2$, A and B are not `n2n`, and there is other no tuple t_3 in $\mathbf{r}$ that is joinable with t_2.

While this notion seems quite intricate, we show that it 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. Indeed, our technique aims at identifying and aggregating efficiently unifiable data by exploiting schema and constraints of the source relational database. Let us consider the relational database in Figure 1. In this case, data is aggregated in six nodes, as shown in Figure 5. For instance the node labeled by $n1$ aggregates data values occurring in t_1, t_3, t_4 and t_5. Similarly the node labeled by n_2 involves data from t_9 and t_4, while n_3 aggregates data values from t_8 and t_6. In this paper, the data conversion process takes into account only the schema of $\mathbf{r}$. Of course, it could be taken into account a set of "frequent" queries over $\mathbf{r}$. This is subject of future work.

More in detail, given the relation database $\mathbf{r}$ with the schema $\mathbf{R}$, and the set SP of all full schema paths in the relational schema graph $\mathbf{RG}$ for $\mathbf{R}$, we generate a graph database $\mathbf{g} = (N, E)$ from $\mathbf{r}$ as shown in Algorithm 1. Our procedure iterates on the elements of SP; in each iteration, a schema path $sp = A_1 \rightarrow \ldots \rightarrow A_k$ is analyzed from the source A_1 to the sink A_k. Let us remind that each A_i of sp corresponds to an attribute in $\mathbf{r}$. The set of data values associated to A_i in the tuples of $\mathbf{r}$ is the active domain of A_i: we will use a primitive getAll($\mathbf{r}$,A_i) that given the relational database $\mathbf{r}$ and an attribute A_i returns all the values v associated to A_i in $\mathbf{r}$. The set of elements $\{\langle A_i, v_j\rangle | v_j \in \mathsf{getAll}(\mathbf{r}, A_i)\}$ is the set of properties to associate to the nodes of $\mathbf{g}$. In our procedure, when we include all the active domain of an attribute A_i in the nodes of $\mathbf{g}$, we say that A_i is *visited*, i.e. A_i is inserted in a set VS of visited attributes. Therefore, the analysis of a schema path (i.e. performed by cond(sp, A_i, VS)) can encounter five cases.

case 1. The current attribute A_i to analyze is a source, i.e. A_1, and both A_i and the following attribute A_{i+1}, i.e. A_2, are not visited. In this case we are at the beginning of the migration, and we are creating new nodes from scratch: the function NewNode is responsible of this task. For instance, referring to Figure 3, our procedure analyzes sp_1 for first; A_i is FR.fuser while A_{i+1} is US.uid. Since A_i is a source and A_{i+1} is not visited, we encounter the **case 1**. For each

data value in the domain of FR.fuser, that is $\{u01, u02\}$, we generate a new node to insert in the set N of $\mathbf{g}$: n_1 and n_5. Then we include the properties $\langle$FR.fuser, $u01\rangle$ and $\langle$FR.fuser, $u02\rangle$ in n_1 and n_5, respectively. At the end, the attribute FR.fuser will be included in VS.

Algorithm 1: Create a graph database **g**

```
   Input  : A relational database r, a set SP of full schema paths
   Output: A graph database g
1  VS ← ∅;
2  g ← (∅, ∅);
3  foreach sp ∈ SP do
4      foreach A_i ∈ sp do
5          switch cond(sp, A_i, VS) do
6              case 1  NewNode(A_i, r, g);
7              case 2  NewProperty(A_i, r, g);
8              case 3  NewProperty(A_i, sp, r, g);
9              case 4  NewNodeEdge(A_i, sp, r, g);
10             case 5  NewEdge(A_i, sp, r, g);
11         VS ← VS ∪ {A_i};
12     return g;
```

case 2. The current attribute A_i to analyze is a source, i.e. A_1, A_i is not visited but the following attribute A_{i+1}, i.e. A_2, is visited. In this case there is a foreign key constraint between A_i and A_{i+1}, i.e. $A_i \rightarrow A_{i+1}$. Since A_{i+1} is visited, we have a node $n \in N$ with the property $\langle A_{i+1}, v\rangle$ where $v \in$ getAll(**r**, A_i). Therefore for each $v \in$ getAll(**r**, A_i) we have to retrieve a node $n \in N$ (i.e. the label l associated to n) and to insert a new property $\langle A_i, v\rangle$ in n, as performed by the function NewProperty taking as input A_i, **r**, and **g**. For instance, when we start to analyze sp_4 (i.e., sp_1, sp_2 and sp_3 were analyzed), we have $A_i =$ TG.tuser and $A_{i+1} =$ US.uid. TG.tuser is a source and not visited while US.uid is visited, since encountered in both sp_1 and sp_3. Therefore we have the **case 2**: getAll(**r**, TG.tuser) is $\{u02\}$ and $n5$ is the node with the property $\langle$US.uid, $u02\rangle$. Finally we insert the new property $\langle$TG.tuser, $u02\rangle$ in $n5$.

case 3. In this case the current attribute A_i is not visited and is not a source neither an hub or a `n2n` node. Therefore we have to iterate on all nodes n generated or updated by analyzing A_{i-1}. In each node n where there was inserted a property $\langle A_{i-1}, v_1\rangle$, we have to insert also a property $\langle A_i, v_2\rangle$ as shown in Case 3: we call the function NewProperty taking as input A_i, sp, **r**, and **g**. More in detail we have to understand if A_i and A_{i-1} are in the same relation (i.e. we are in the same tuple) or not (i.e. we are following a foreign key). In the former we have to extract the data value v_2 from the same tuple containing v_1 (line 5) otherwise v_2 is v_1 (line 6). We use the function getTable to retrieve the relation R in **r** containing a given attribute a (lines 3-4). Finally, we insert the new property (by calling the function INS) in the node n to which is associated the label label(n), coming from the iteration on the attribute A_{i-1}. For instance iterating on sp_1, when A_i is US.uname and A_{i-1} is US.uid we have the **case 3**: we iterate on the nodes $n1$ and $n5$ containing the properties $\langle$US.uid, $u01\rangle$ and $\langle$US.uid, $u02\rangle$, respectively. Since US.uname and US.uid are in the same relation User (US), we extract from US the values associated to US.uname in the tuples t_1 and t_2, referring to Figure 1. Then we insert the properties $\langle$US.uname, $Date\rangle$ and $\langle$US.uname, $Hunt\rangle$ in $n1$ and $n5$, respectively.

case 4. The current attribute A_i is not visited and it is an hub or a `n2n` node in **g**. As in **case 3**, we have to iterate on all nodes n generated or updated by analyzing A_{i-1}. Differently from **case 3**, for each data value in the domain of A_i we generate a new node with label l_i and we insert the property $\langle A_i, v\rangle$ in the node. Then we link the node with label l_j generated or updated analyzing A_{i-1} to the node with label l_i just generated. Given the attribute A_{i-1} and the relation R which A_{i-1} belongs to, the label l_e assigned to the new edge is built by the concatenation of R and A_{i-1}. This task is performed by the function NewNodeEdge. Let us consider the schema path sp_2 and the attribute FR.fblog as current attribute A_i to analyze. It is not visited and a `n2n` node in **g**. In the previous iteration, the analysis of FR.fuser (i.e. A_{i-1}) updated the node with label $n1$. In the current iteration, we have to generate three new nodes, i.e. with labels $n2$, $n3$ and $n6$, and to include the properties $\langle$FR.fblog, $b12\rangle$, $\langle$FR.fblog, $b02\rangle$, $\langle$FR.fblog, $b03\rangle$, repsectively, since getAll(**r**, FR.fblog) is $\{b01, b02, b03\}$. Finally given the label l_e equal to FOLLOWER_FUSER, i.e. FR.fuser belongs to the relation Follower, we generate the edges with label l_e between $n1$ and $n2$, $n1$ and $n3$, $n1$ and $n6$.

Case 3: NewProperty(A_i, sp, **r**, **g**)

```
1  A_{i-1} ← sp[i − 1];
2  foreach node n in g such that n contains a property
   ⟨A_{i-1}, v_1⟩ do
3      R_1 ← getTable(r, A_{i-1});
4      R_2 ← getTable(r, A_i);
5      if R_1 = R_2 then v_2 ← π_{A_i} σ_{A_{i-1}=v_1}(R_1);
6      else v_2 ← v_1;
7      INS(g, label(n), A_i, v_2);
```

Case 5: NewEdge(A_i, sp, **r**, **g**)

```
1  A_{i-1} ← sp[i − 1];
2  foreach node n in g such that n contains a property
   ⟨A_{i-1}, v_1⟩ do
3      R_1 ← getTable(r, A_{i-1}); R_2 ← getTable(r, A_i);
4      if R_1 = R_2 then V ← π_{A_i} σ_{A_{i-1}=v_1}(R_1);
5      else V ← {v_1};
6      foreach v ∈ V do
7          l_i ← getNode(g, A_i, v);
8          if l_i ≠ NIL then l_e ← build(r, A_{i-1});
           newEdge(g, l_j, l_i, l_e);
```

case 5. The last case occurs when we are analyzing the last schema paths and in particular the last attributes in a schema path. In this case we link two nodes generated in the previous iterations. The current attribute A_i is (i) not visited and `n2n` or (ii) visited and an hub. Moreover there exists a node in **g** with a property $\langle A_i, v\rangle$, and the attribute A_{i-1} is not a source. As shown in Case 3, our procedure iterates on the nodes with label l_j built or updated analyzing A_{i-1} and retrieves the node with label l_i to link it with the node with label l_j. We have to discern if A_{i-1} and A_i are in the same relation or not. Given R_1 and R_2 the relations which A_{i-1} and A_i belong to, respectively, if R_1 and R_2 are the same then A_{i-1} and A_i are in the same tuple and we extract all data values V associated to A_i in the tuple (line 4). Otherwise we are considering a foreign key constraint between A_{i-1} and A_i: V is $\{v1\}$ (line 5), where $v1$ is the value in the property $\langle A_{i-1}, v1\rangle$ included in the node with label l_j. Finally for each data value v in V we retrieve the node with label l_i including the property $\langle A_i, v\rangle$ and, if it exists, we link the node with label l_j to the node with label l_i (lines 6-8). Let us consider the schema path sp_3 and US.uid as current attribute A_i. Since in the previous iteration the procedure analyzed sp_1, US.uid

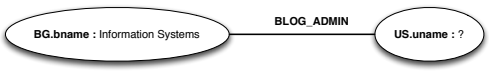

Figure 6: The query template for Q'.

is visited now; moreover US.uid is an hub and the previous attribute BG.admin is not a source. We have the **case 5**: since the nodes with labels $n1$ and $n2$ contain the properties $\langle$US.uid, $u01\rangle$ and $\langle$BG.admin, $u01\rangle$, respectively, a new edge with label BLOG_ADMIN is built between that nodes (i.e. similarly between the nodes with labels $n3$ and $n5$).

4. QUERY TRANSLATION

Our mechanism for translating conjunctive (that is, select-join-projection) queries, expressed in SQL, into path traversal operations over the graph database exploits the schema of the source relational. For the sake of simplicity, we consider an intermediate step in which we map the SQL query in a graph-based internal structure, that we call *query template* (QT for short). Basically, a QT denotes all the sub-graphs of the target graph database that include the result of the query. A QT is then translated into a path traversal query (see Section 2). Given a query Q the construction of a QT proceeds as follows.

1. We built a minimal set SP of full schema paths such that for each join condition $R_i.A_i = R_j.A_j$ occurring in Q, an edge $(R_i.A_i, R_j.A_j)$ is contained in at least one sp in SP;

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

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

4. QT is the graph database obtained by applying the data conversion procedure illustrated in Section 3 over SP and $\mathbf{r_Q}$.

We explain our technique by the following query example Q'.

```
select   US.uname
from     User US, Tag TG, Blog BG, Comment CT
where    (BG.bid = CT.cblog) and(CT.cid = TG.tcomment) and
         (TG.tuser = US.uid) and(BG.bname = 'Inf. Systems')
```

On the relational database of Figure 1, Q' selects all the users that have left a comment on the *Information Systems* blog. As said above, referring to Figure 3, (1) a minimal set of full schema paths that contain all the join conditions of Q' is $\mathsf{SP}_1 = \{sp_4, sp_7\}$ is built. (2) Since from the selection condition *(BG.bname = 'Information Systems')* the attribute `BG.bname` is already occurring in sp_7 we do not have to include more paths in SP_1. (3) From the selection condition *(BG.bname = 'Information Systems')* and the attribute `US.uname` of the SELECT clause, we build $\mathbf{r_{Q'}}$ = {BLOG(bname), USER(uname)}, where BLOG(bname) contains one tuple with the data value *Information Systems* and USER(uname) contains one tuple with the special symbol ?, respectively, as instance. (4) From SP_1 and $\mathbf{r_{Q'}}$, we obtain the query template QT' shown in Figure 6. It is straightforward to map QT' into a XQuery-like path traversal expression QPT' as follows.

```
for     $x in /[BG.bname='Informative Systems'],
        $y in $x/BLOG_ADMIN/*
return  $y/US.uname
```

We start from the node with the property $\langle$ BG.bname, *Information Systems* $\rangle$. Moreover, in the condition we express the fact that this node reaches another node through the link BLOG_ADMIN. Finally, from these nodes we return the values of the property with key US.uname (i.e. in our example we have only Hunt).

5. EXPERIMENTAL RESULTS

We have developed the techniques described in this paper in a Java system called R2G. Experiments were conducted on a dual core 2.66GHz Intel Xeon, running Linux RedHat, with 4 GB of memory and a 2-disk 1Tbyte striped RAID array. We considered real datasets with different sizes (i.e. number of tuples). In particular we used MONDIAL (17.115 tuples and 28 relations) and two ideal counterpoints (due to the larger size), IMDB (1.673.074 tuples in 6 relations) and WIKIPEDIA (200.000 tuples in 6 relations), as described in [6]. The authors in [6] defined a benchmark of 50 keyword search queries for each dataset. We used the tool in [7] to generate SQL queries from the keyword-based queries defined in [6].

Dataset	Neo4J	OrientDB	R2G_N	R2G_O
MONDIAL	7.4 sec	5.3 sec	13.9 sec	9.3 sec
WIKIPEDIA	70.7 sec	66.5 sec	161.5 sec	148.7 sec
IMDB	8.1 min	10.2 min	16.2 min	22.1 min

Table 1: Performance of translations from r to g

R2G has been embedded and tested in two different GDBMSs: Neo4J and OrientDB. In the following we denote with R2G_N and R2G_O the implementations of R2G in Neo4J and OrientDB, respectively. First of all we evaluate data loading, that is time to produce a graph database starting from a SQL dump. We compared R2G against native data importers of Neo4J and OrientDB, that use a naive approach to import a SQL dump, that is one node for each tuple and one edge for each foreign key reference. In our transformation process we query directly the RDBMS to build schema graph and compute schema paths and then to extract data values. For our purposes we used PostgreSQL 9.1 (denoted as RDB). Table 1 shows the performance of this task. Neo4J and OrientDB importers perform better than our system, i.e. about two times better. This is due to the fact that R2G has to process the schema information of relational database (i.e. the schema graph) while the competitor systems directly import data values from the SQL dump. Then we evaluated the performance of query execution. For each dataset, we grouped the queries in five sets (i.e. ten queries per set): each set is homogeneous with respect to the complexity of the queries (e.g., number of keywords, number of results and so on). For instance referring to IMDB, the first set (i.e. Q1-Q10) searches information about the actors (providing the name as input), while the second set (i.e. Q11-Q20) seeks information about movies (providing the title as input). The other sets combine actors, movie and characters. For each set, we ran the queries ten times and measured the average response time. We performed *cold-cache* experiments (i.e. by dropping all file-system caches before restarting the various systems and

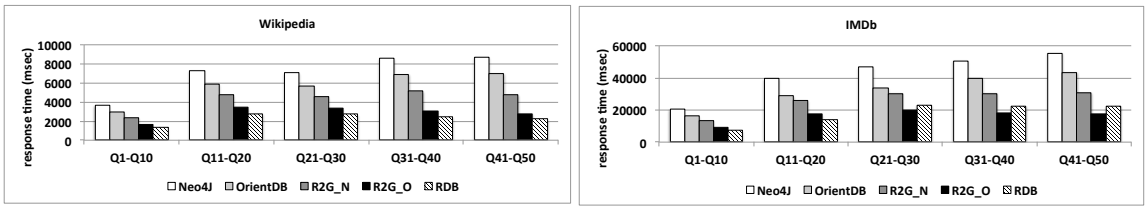

Figure 7: Performance on databases

running the queries) and *warm-cache* experiments (i.e. without dropping the caches). Figure 7 shows the performance for cold-cache experiments. Due to space constraints, in the figure we report times only on IMDb and Wikipedia, since their much larger size poses more challenges. In particular we show also times in the relational database (i.e. RDB) as global time reference, not for a direct comparison with relational DBMS. Our system performs consistently better for most of the queries, significantly outperforming the others in some cases (e.g., sets Q21-Q30 or Q31-Q40). We highlight how our data mapping procedure allows OrientDB to perform better than RDB in IMDb (having a more complex schema). This is due to our strategy reducing the space overhead and consequently the time complexity of the overall process w.r.t. the competitors that spend much time traversing a large number of nodes. Warm-cache experiments follow a similar trend.

6. RELATED WORKS

The need to convert relational data into graph modeled data [1] emerged particularly with the advent of Linked Open Data (LOD) [8] since many organizations needed to make available their information, usually stored in relational databases, on the Web using RDF. For this reason, several solutions have been proposed to support the translation of relational data into RDF. Some of them focus on mapping the source schema into an ontology [5, 10, 13] and rely on a naive transformation technique in which every relational attribute becomes an RDF predicate and every relational values becomes an RDF literal. Other approaches, such as R_2O [11] and D2RQ [3], are based on a declarative language that allows the specification of the map between relational data and RDF. As shown in [8], they all provide rather specific solutions and do not fulfill all the requirements identified by the RDB2RDF (http://www.w3.org/TR/2012/CR-rdb-direct-mapping-20120223/) Working Group of the W3C. Inspired by draft methods defined by the W3C, the authors in [13] provide a formal solution where relational databases are directly mapped to RDF and OWL trying to preserve the semantics of information in the transformation. All of those proposals focus on mapping relational databases to Semantic Web stores, a problem that is more specific than converting relational to general, graph databases, which is our concern. On the other hand, some approaches have been proposed to the general problem of database translation between different data models (e.g., [2]) but, to the best of our knowledge, there is no work that tackles specifically the problem of migrating data and queries from a relational to a graph database management system. Actually, existing GDBMSs are usually equipped with facilities for importing data from a relational database, but they all rely on naive techniques in which, basically, each tuple is mapped to a node and foreign keys are mapped to edges. This approach however does not fully exploit the capabilities of GDBMSs to represent graph-shaped the information. Moreover, there is no support to query translation in these systems. Finally, it should be mentioned that some works have done on the problem of translating SPARQL queries to SQL to support a relational implementation of RDF databases [13]. But, this is different from the problem addressed in this paper.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented an approach to migrate automatically data and queries from relational to graph databases. The translation makes use of the integrity constraints defined over the source to suitably build a target database in which the number of accesses needed to answer queries is reduced. We have also developed a system that implements the translation technique to show the feasibility of our approach and the efficiency of query answering. In future works we intend to refine the technique proposed in this paper to obtain a more compact target database.

8. REFERENCES

[1] R. Angles and C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), 2008.
[2] P. Atzeni, P. Cappellari, R. Torlone, P. A. Bernstein, and G. Gianforme. Model-independent schema translation. *VLDB J.*, 17(6):1347–1370, 2008.
[3] C. Bizer. D2r map - a database to rdf mapping language. In *WWW (Posters)*, 2003.
[4] C. Bizer and A. Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
[5] F. Cerbah. Learning highly structured semantic repositories from relational databases:. In *ESWC*, pages 777–781, 2008.
[6] J. Coffman and A. C. Weaver. A framework for evaluating database keyword search strategies. In *CIKM*, pages 729–738, 2010.
[7] S. B. et al. Keyword search over relational databases: a metadata approach. In *SIGMOD*, pages 565–576, 2011.
[8] M. Hert, G. Reif, and H. C. Gall. A comparison of rdb-to-rdf mapping languages. In *I-SEMANTICS*, pages 25–32, 2011.
[9] F. Holzschuher and R. Peinl. Performance of graph query languages - comparison of cypher, gremlin and native access in neo4j. In *EDBT/ICDT Workshops*, pages 195–204, 2013.
[10] W. Hu and Y. Qu. Discovering simple mappings between relational database schemas and ontologies. In *ISWC/ASWC*, pages 225–238, 2007.
[11] J. B. Rodríguez and A. Gómez-Pérez. Upgrading relational legacy data to the semantic web. In *WWW*, pages 1069–1070, 2006.
[12] M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.
[13] J. Sequeda, M. Arenas, and D. P. Miranker. On directly mapping relational databases to rdf and owl. In *WWW*, pages 649–658, 2012.
[14] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.