Interpreting Relational Databases in the RDF Domain *

Alexandre Bertails
World Wide Web
Consortium/MIT
32 Vassar Street
Cambridge, MA 02140
bertails@w3.org

Eric Prud'hommeaux
World Wide Web
Consortium/MIT
32 Vassar Street
Cambridge, MA 02140
eric@w3.org

ABSTRACT

The W3C's "Direct Mapping of Relational Data to RDF" defines a simple, practical and intuitive interpretation of SQL database tables as RDF graphs. This document specifies the formal data models for RDB (Relational DataBase) and RDF and defines a denotational semantics of RDB in the RDF domain. We show how this mapping treats all of the important features of SQL tables, like cardinality and NULLs, and yields an RDF graph which preserves the relational information.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Relational databases; D.3.1 [**Formal Definitions and Theory**]: Semantics

General Terms

Algorithms, Performance, Standardization, Theory

Keywords

RDF, Semantic Web, relational database, SQL, denotational semantics

1. INTRODUCTION

Motivations to map relational data to RDF abound[20]: science, technology and business need to integrate more and more diverse data sources; the far majority of data that we expect machines to interpret are in relational databases[16]. As RDF, this data can interact with the large amounts of interesting (e.g. scientific) data already mechanically transformed by the Linked Data [17][8] community; etc. RDF's web foundations and distributed extensibility model offer the database community a simple and effective mechanism to contribute to a growing continuum of information.

The semantics of a query over an RDF graph are specified directly in the SPARQL Specification [21], while the semantics for SQL data are expressed with respect to queries and constraints in SQL specifications [4]. The W3C RDB2RDF Working Group has begun work on a *Direct Mapping* [5], a normative representation of SQL data as RDF graphs. The product of mapping a given database is called the *Direct Graph* for that database. This document provides a formal representation of the Direct Mapping as an RDF semantics, describes the features and motivations of the design choices and illustrates practical and theoretic applications.

Defining a simple mapping from SQL databases to RDF graphs effectively expresses that data in the Semantic Web, where it can be used in several ways: it can be queried directly using SPARQL, it can be suplemented with schema information (e.g. RDFS, OWL, SKOS) and used for A-box inference, it can be transformed via e.g. SPARQL CONSTRUCT, N3 rules, RIF to graphs which represent the domain information in popular schema, and perhaps most importantly, it can provide the necessary definition to map SPARQL queries to SQL queries, eliminating the latency costs of warehousing. All of these uses require no further definition or standardization as the dependent languages (SPARQL, OWL, etc.) are fully defined with respect to input RDF graphs.

2. OVERVIEW

While we frequently refer to "relational" data, the far majority of that data is defined and accessed via the *Structured Query Language* [4]; this motivated the RDB2RDF WG to define the Direct Graph for SQL Base Tables and Views.

Consider an example contact database:

People

ID	fname	addr
7	Bob	18
8	Sue	NULL

Addrs

ID	city	state
18	Cambridge	MA

with a foreign key from `People.addr` to `Addrs.ID` .

Empirically, we see a database as a set of *table*s in a given *schema*. The schema for each table defines a set of *columns* (attributes), each of some datatype. The *foreign key*s link lists of columns in a table to another list of columns in a table, creating a relational graph.

*This document was created as input to the *W3C RDB2RDF Working Group* at `http://www.w3.org/2001/sw/rdb2rdf/`

K-CAP'11, June 26–29, 2011, Banff, Alberta, Canada.

The Direct Mapping for each row in the database produces a set of RDF triples with a shared subject – the row identifier – and with predicates and objects conveying both the database values:

```
<People/ID=7> <People#ID> 7 .
<People/ID=7> <People#fname> "Bob" .
<People/ID=8> <People#ID> 8 .
<People/ID=8> <People#fname> "Sue" .
<Addrs/ID=18> <Addrs#ID> 18 .
<Addrs/ID=18> <Addrs#city> "Cambridge" .
<Addrs/ID=18> <Addrs#state> "MA" .
```

and the relational graph:

```
<People/ID=7> <People#addr> <Addrs/ID=18> .
```

In order to accurately model the SQL table input, we examine the SQL Framework specification [3] §4.3 Tables ¶1:

```
A table has an ordered collection of one or
more columns and an unordered collection of
zero or more rows. Each column has a name
and a data type. Each row has, for each
column, exactly one value in the data type
of that column.
```

Combining this with the popular Relational Model [12][2], we see a natural model for a table as a Header holding the column names and datatypes and a body of tuples with values for those columns:

$$\begin{aligned} Table &::= (Header, List(CandidateKey), \\ &\quad Set(ForeignKey), Body) \\ Header &::= List((ColumnName, Datatype)) \\ Body &::= MultiSet(Row) \\ Row &::= List(CellValue) \end{aligned}$$

However, the Direct Mapping defines a mapping only for base tables and views, as defined in ¶7:

```
No two columns of a base table or a viewed
table can have the same name. Derived tables,
other than viewed tables, may contain more
than one column with the same name.
```

which means that column names are unique, which gives us partial functions from ColumnName to Datatype and from ColumnName to CellValue:

$$\begin{aligned} header &: ColumnName \to Datatype \\ row &: ColumnName \to CellValue \end{aligned}$$

The relational model's notion of a `candidate key` is called a `unique` in SQL, defined in §4.6.6.3 Table constraints:

```
A unique constraint specifies one or more
columns of the table as unique columns. A
unique constraint is satisfied if and only
if no two rows in a table have the same
non-null values in the unique columns.
```

Null values may appear in a candidate key so long as that set of column values is still unique between tuples. A primary key however, may have no NULLs, so the tuple nodes in the Direct Mapping will never need to encode a NULL value. Note that the mapping from candidate key values to tuple nodes may include NULL values.

The row identifier for tuples in tables with no primary key is a blank node assigned to that tuple. If the relational graph may include references to such tuples, it will be via some (non-primary) candidate key on the table. In this example, the Protein database, both the `uniprot` and `ebi` columns form unique keys, but neither is elevated to primary key:

Protein

uniprot	ebi	name
P04637	366083	P53
Q00987	389668	MDM2

EBIact

target	actor	action
366083	389668	antagonist

The list of EBI's cataloged interactions, `EBIact`, happens to use EBI's protein identifiers. The `EBIact target` and `actor` columns are foreign keys to `Proteins.ebi`, and the primary key of `EBIact` is (`target` and `actor`). Because these columns reference to tuples in a table with no primary key, they must use for an object the same blank node used as the row identifier for the referenced tuple.

```
_:a <Protein#uniprot> "P04637" .
_:a <Protein#ebi> 366083 .
_:a <Protein#name> "P53" .
_:b <Protein#uniprot> "Q00987"  .
_:b <Protein#ebi> 389668 .
_:b <Protein#name> "MDM2" .
<EBIact/target=366083,actor=389668>
              <EBIact#target> _:a .
<EBIact/target=366083,actor=389668>
              <EBIact#actor> _:b .
<EBIact/target=366083,actor=389668>
              <EBIact#action> "antagonist" .
```

Tables with no candidate keys at all may have duplicates, but then SQL can't express foreign keys to them. Thus, we see that the SQL data is a graph and not a multi-graph (which would be awkward to map to an RDF graph).

There is some controversy within the RDB2RDF Working Group over whether the Direct Mapping should include special handling for `link table`s (AKA many-to-many tables). Rows in tables with exactly two foreign keys which subsume all of the attributes in the table can be represented as triples with the subject being the target of the first foreign key and the object being the target of the second. While this is an arguably more intuitive expression of the domain data frequently encoded in link tables, the rule itself is necessarily brittle with respect to monotonic changes to the schema. For instance, the addition of a timestamp attribute, which implies no changes to the domain interpretation of any row, would radically change the RDF representation, breaking any queries or rules referencing link table properties.

3. ABSTRACT DATA MODELS

The RDB and RDF *Abstract Data Types* (ADTs) make use of the commonly defined ADTs *Set*, *List* and *MultiSet*, used here as type constructors. For example, $Set(A)$ denotes

the type for the sets of elements of type A. We assume that they come with their common operations, such as the function $size : Set \to Int$.

We follow a type-as-specification approach, thus the ADTs are actually dependent types. For example, $\{s : Set(A) \,|\, size(s) \leq 1\}$ is a subtype of $Set(A)$ with at most one element, often called the option type.

3.1 RDB Abstract Data Type

$$
\begin{array}{rcl}
Database & ::= & Set(Table) \\
Table & ::= & (TableName, Header, List(CandidateKey), \\
 & & \quad Set(ForeignKey), Body) \\
Header & ::= & Set((ColumnName, Datatype)) \\
Body & ::= & MultiSet(Row) \\
Row & ::= & Set((ColumnName, CellValue)) \\
CellValue & ::= & LexicalValue \,|\, NULL \\
ForeignKey & ::= & (List(ColumnName), Table, CandidateKey) \\
CandidateKey & ::= & List(ColumnName) \\
Datatype & ::= & Int \,|\, Float \,|\, Date \,|\, \ldots \\
TableName & ::= & String \\
ColumnName & ::= & String
\end{array}
$$

3.2 RDB accessor functions

$$
\begin{array}{rcl}
tablename & : & Table \to TableName \\
header & : & Table \to Header \\
candidateKeys & : & Table \to List(CandidateKey) \\
primaryKey & : & Table \to \{s : Set(CandidateKey) \,|\, size(s) \leq 1\} \\
foreignKeys & : & Table \to Set(ForeignKey) \\
unary & : & ForeignKey \to Boolean \\
lexicals & : & Table \to Set(\{c : ColumnName \,|\, !\ unary(c)\}) \\
body & : & Table \to Body \\
datatype & : & \{h : Header\} \\
 & & \quad \to \{c : ColumnName \,|\, \exists\ d, (c, d) \in h\} \\
 & & \quad \to \{d : Datatype \,|\, (c, d) \in h\} \\
table & : & \{r : Row\} \to \{t : Table \,|\, r \in t\} \\
value & : & \{r : Row\} \to \{c : ColumnName \,|\, c \in r\} \\
 & & \quad \to CellValue \\
dereference & : & \{r : Row\} \\
 & & \quad \to \{fk : ForeignKey \\
 & & \qquad\quad | \, fk \in foreignKeys(table(r))\} \\
 & & \quad \to \{targetRow : Row \\
 & & \qquad\quad | \, P(r, fk, targetRow)\}
\end{array}
$$

In $dereference$, P is a predicate binding the row r to the row $targetRow$, referenced by the foreign key fk. The values for the foreign key columns in r equal the values of the referenced columns in the returned row:

$$
\begin{array}{l}
P(r, fk, targetRow) = \\
\quad \textbf{let}\ (columnNames, targetTable, ck) = fk\ \textbf{in} \\
\quad targetRow \in body(targetTable) \\
\quad \textbf{and}\ \forall c_i^{fk} \in columnNames, \forall c_j^{ck} \in ck, \\
\qquad \forall (c_k^r, v_k^r) \in r, \forall (c_l^{target}, v_l^{target}) \in targetRow, \\
\qquad i = j \to c_i^{fk} = c_k^r \to c_j^{ck} = c_l^{target} \to v_k^r = v_l^{target}
\end{array}
$$

3.3 RDF Abstract Data Type

$$
\begin{array}{rcl}
Graph & ::= & Set(Triple) \\
Triple & ::= & (Subject, Predicate, Object) \\
Subject & ::= & IRI \,|\, BlankNode \\
Predicate & ::= & IRI \\
Object & ::= & IRI \,|\, BlankNode \,|\, Literal \\
BlankNode & ::= & RDF\ blank\ node \\
Literal & ::= & PlainLiteral \,|\, TypedLiteral \\
PlainLiteral & ::= & lexicalForm \\
 & & \,|\, (lexicalForm, langageTag) \\
TypedLiteral & ::= & (lexicalForm, IRI) \\
IRI & ::= & RDF\ URI\ ^1\ [18] \\
lexicalForm & ::= & Unicode\ string\ ^2\ [18]
\end{array}
$$

We don't need to provide accessors as we are simply constructing RDF graphs and their components. In order to stay simple, we'll use a Turtle [7] like syntax for injecting elements in these types.

So armed, we set forth to define the Direct Mapping.

4. DIRECT MAPPING

Inhabitants of RDB 3.1 *are denoted* by mathematical objects living in the RDF domain 3.1. We call this *denotational semantics* of RDB the *Direct Mapping*.

Most of the functions are higher-order functions, relying on a function ϕ mapping a row to an RDF node. We assume that this function maps any Tuple to a unique row IRI. ϕ is formally defined by the following axioms:

$$
\begin{array}{l}
\forall\ db : Database, \forall\ r : Row, \qquad (1) \\
\quad r \in db \to primaryKey(table(r)) \neq \emptyset \to \phi(r)\ \textbf{is}\ an\ IRI \\
\forall\ db : Database, \forall\ r : Row, \qquad (2) \\
\quad r \in db \to primaryKey(table(r)) = \emptyset \to \phi(r)\ \textbf{is}\ a\ BlankNode
\end{array}
$$

4.1 Denotational semantics

The Direct Mapping is defined by induction on the structure of RDB. Thus it is defined for *any SQL database* and its execution is guarantied to terminate.

The entry point is $[\![\ ]\!]^{\phi}_{database}$. Note that not all the functions need to be parameterized by ϕ.

[1] subsequently restricted by SPARQL to exclude space (`#x20`)

[2] `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`

$$
\begin{array}{rcl}
[\![\ ]\!]^{\phi}_{database} & : & Database \to Graph \\
[\![db]\!]^{\phi}_{database} & = & \{triple \,|\, triple \in [\![t]\!]^{\phi}_{table} \,|\, t \in db\} \\
[\![\ ]\!]^{\phi}_{table} & : & Table \to Set(Triple) \\
[\![t]\!]^{\phi}_{table} & = & \{triple \,|\, triple \in [\![r]\!]^{\phi}_{row} \,|\, r \in body(t)\} \\
[\![\ ]\!]^{\phi}_{row} & : & Row \to Set(Triple) \\
[\![r]\!]^{\phi}_{row} & = & \textbf{let } s = \phi(r) \textbf{ in} \\
 & & \{(s,p,o) \,|\, (p,o) \in [\![r, fk]\!]^{\phi}_{ref} \\
 & & \qquad |\, noNulls(r, fk) \\
 & & \qquad |\, fk \in references(t)\} \\
 & \cup & \{(s,p,o) \,|\, (p,o) \in [\![r, c]\!]_{lex} \\
 & & \qquad |\, value(r,c) \neq NULL \,|\, c \in lexicals(t)\} \\
 & \cup & \{(s, rdf{:}type, ue(tablename(table(r))))\} \\
[\![\ ,\ ]\!]^{\phi}_{ref} & : & (Row, ForeignKey) \to (Predicate, Object) \\
[\![r, fk]\!]^{\phi}_{ref} & = & \textbf{let } p = [\![table(r), fk]\!]_{col} \textbf{ in} \\
 & & \textbf{let } targetRow = dereference(r, fk) \textbf{ in} \\
 & & \textbf{let } o = \phi(targetRow) \textbf{ in} \\
 & & (p, o) \\
[\![\ ,\ ]\!]_{lex} & : & (Row, Column) \to \\
 & & \to \{s : Set((Predicate, Object)) \,|\, size(s) \leq 1\} \\
[\![r, c]\!]_{lex} & = & \textbf{let } p = [\![table(r), c]\!]_{col} \textbf{ in} \\
 & & \textbf{let } v = value(r, c) \textbf{ in} \\
 & & \textbf{let } d = datatype(header(table(r))(c)) \textbf{ in} \\
 & & \textbf{if } v \textbf{ is } NULL \textbf{ then } \emptyset \\
 & & \textbf{else if } d \textbf{ is } String \textbf{ then } \{(p, v)\} \\
 & & \quad \textbf{else let } datatype\_iri = [\![d]\!]_{datatype} \textbf{ in} \\
 & & \quad \{(p, (v, datatype\_iri))\} \\
[\![\ ,\ ]\!]_{col} & : & (Row, List(Column)) \to IRI \\
[\![r, c*]\!]_{col} & = & ue(tablename(table(r))) + \text{`\#`} + ue(c_0) \\
 & & \qquad + \text{`,`} + \cdots + \text{`,`} + ue(c_{n-1}) \\
[\![\ ]\!]_{datatype} & : & Datatype \to IRI \\
[\![d]\!]_{datatype} & = & \textbf{if } d \textbf{ is } Int \textbf{ then } xsd{:}integer \\
 & & \textbf{else if } d \textbf{ is } Float \textbf{ then } xsd{:}float \\
 & & \textbf{else if } d \textbf{ is } Date \textbf{ then } xsd{:}date \\
 & & \ldots \\
ue & : & String \to String \\
ue(s) & = & \texttt{URL-encoding} \; of \; s \; per \; [22] \\
noNulls & : & Row \to ForeignKey \to Boolean \\
noNulls(r, fk) & = & \textbf{let } (columnNames, \_, \_) = fk \textbf{ in} \\
 & & \forall c \in columnNames, value(r, c) \neq NULL
\end{array}
$$

Note that strict URL-encoding is not necessary. It is sufficient to encode the potential border characters [/,=#].

5. HONORING SQL SEMANTICS

The Direct Mapping attempts to emulate the utility of the source database. The most direct way to measure this is to compare the data one could extract with SQL queries over the relational data vs. SPARQL queries over the direct graph.

5.1 Row identifier

Another design goal is that the direct graph be compatible with the principles of Linked Data[9]. In particular, row identifiers for addressable rows are IRIs, e.g. `<EBIact/target=366083,actor=389668>`, while row identifiers for which the database offers no repeatable access are blank nodes, e.g. `_:b`.

5.2 Foreign Keys

As stated in 2, *foreign key*s create a relational graph. The Direct Mapping encodes this graph as $reference$ triples, where the predicate name is a product of the table name and the column names forming the foreign key. A peculiar wrinkle of this is that the predicate name for unary foreign keys (foreign keys composed of only one column) would conflict with the name of the *scalar* triple for that column. In the case of unary foreign keys, the *scalar* triple is not formed. As an example, the unary foreign key from *People.addr* to *Addrs.ID* produced a triple:

```
<People/ID=7> <People#addr> <Addrs/ID=18>
```

but no triple:

```
<People/ID=7> <People#addr> 18
```

$Reference$ triples are not generated for foreign keys on rows with any $NULL$s for the attributes in the foreign key. This reflects the SQL behavior for the common idiom for enforcing foreign key constraints in SQL queries. In SQL, $NULL \neq NULL$ so a constraint testing the attributes in an nary foreign key from referrer s to referent t $s.attr_0 \neq t.attr_0 AND ... s.attr_i \neq t.attr_i$ will be false if any of the values are $NULL$. This is emulated by the $noNulls$ constraint on $references$ in $[\![r]\!]^{\phi}_{row}$.

5.3 Cardinality

SQL allows a table to contain multiple rows with exactly the same column values. In fact, this is a frequent outcome of projecting away columns which keep rows distinct, which can happen in a defined view. SQL maintains a strict accounting for these rows, preserving their cardinality in both query results and aggregate functions.

Recall that for tables with no primary key, the Direct Mapping assigns each row a blank node. Any indistinct rows in the table become distinct subjects for a set of triples. Thus, SPARQL queries over the direct graph will carry the same cardinality as would SQL queries over the source table.

5.4 NULL attributes

RDF itself has no real model for absent information; a lack of domain information generally translates to a lack of assertions. When some, but not all, information is known about a record in an SQL database, the tabular nature requires an encoding for absent information. The popular convention is to use the `NULL` value for this. The Direct Mapping addresses this case by creating *scalar* triples only for non-null cell values, and creating $reference$ triples only when the referenced tuple exists. Recall in the first table that *Sue* had a NULL *addr*; which corresponded to no record in the *Addrs* table. Consequentially, the direct graph for this table had no arc of the form:

```
<People/ID=8> <People#addr> <Addrs/ID=??>
```

6. "RULE" TRANSFORMATIONS

The schema and node labels of the direct graph are entirely subordinate to the relational schema and data. This is counter to the Semantic Web practice of sharing schemas and identifiers where possible. A variety of standards, e.g. RIF and SPARQL CONSTRUCT, define functions mapping one RDF graph to another. Given our earlier direct graph, a given SPARQL CONSTRUCT:

```
@prefix ebi: <http://purl.uniprot.org/intact/> .
@prefix core: <http://purl.uniprot.org/core/> .
@prefix uni: <http://purl.uniprot.org/uniprot/> .

CONSTRUCT {
  _:i a core:Interaction .
  _:i core:participant ?e1 .
  ?e1 rdfs:label ?l1 .
  ?e1 owl:sameAs ?u1 .
  _:i core:participant ?e2 .
  ?e2 rdfs:label ?l2 .
  ?e2 owl:sameAs ?u2 .
}
WHERE {
  ?i  <EBIact#target> ?target .
  ?i  <EBIact#actor> ?actor .
  ?target <Protein#uniprot> ?ulab1  .
  ?target <Protein#ebi> ?elab1 .
  ?target <Protein#name> "P53" .
  BIND (IRI(CONCAT(uni:, ?ulab1)) AS ?u1)
  BIND (IRI(CONCAT(ebi:, ?elab1)) AS ?e1)
  ?actor <Protein#uniprot> ?ulab2  .
  ?actor <Protein#ebi> ?elab2 .
  ?actor <Protein#name> "MDM2" .
  BIND (IRI(CONCAT(uni:, ?ulab2)) AS ?u2)
  BIND (IRI(CONCAT(ebi:, ?elab2)) AS ?e2)
}
```

will produce an RDF graph consistent in names and structure with the Uniprot RDF graph about the same resources:

```
@prefix ebi: <http://purl.uniprot.org/intact/> .
@prefix core: <http://purl.uniprot.org/core/> .
@prefix uni: <http://purl.uniprot.org/uniprot/> .
_:i139         a core:Interaction .
_:i139         core:participant ebi:EBI-366083 .
ebi:EBI-366083 rdfs:label "TP53" .
ebi:EBI-366083 owl:sameAs uni:P04637 .
_:i139         core:participant ebi:EBI-389668 .
ebi:EBI-389668 rdfs:label "MDM2" .
ebi:EBI-389668 owl:sameAs uni:Q00987 .
```

Combining the Direct Mapping with reversible graph transformation functions create virtual, Semantic Web-friendly graphs, like the Uniprot graph above. Of course, the magic here is in the reversibility of the transformation functions. How easy will it be to turn a query over the above graph into an SQL query over the input tables?

7. COMPILATION TO SQL

The Direct Mapping can be realized many ways, but of course an SQL implementation is of particular interest. The direct graph is an RDF graph, but the behavior of that graph can be approximated with an SQL view. The view may, of course, be either materialized or virtual. Translating SPARQL 1.0 queries to SQL queries over a triple table has been well-studied [13][11][14], so we illustrate here only the creation of the view and the required built-in functions. The Direct Mapping manipulates table and column names so a general implementation requires second order logic. Here we briefly describe the process for mapping a given database schema to an SQL view of RDF-like assertions; the minutia are captured in `http://www.w3.org/2011/02/DM-KCAP/SQL` .

SQL schemas which fully capture the semantics of an RDF graph are cumbersome so we'll use a schema which captures all of the RDF graph expressivity and count on the Direct Mapping view to respect the constraints (such as, the subject of a triple is exclusively either an IRI or a blank node):

```
CREATE TABLE triples (
  sIRI CHAR(40), sBNode CHAR(8),
  pIRI CHAR(40),
  oIRI CHAR(40), oBNode CHAR(8), oLexical CHAR(40),
             oDatatype CHAR(40), oLangTag CHAR(10),
  UNIQUE (sIRI, sBNode, pIRI, oIRI,
          oBNode, oLexical, oDatatype, oLangTag)
)
```

The only subtlety to interpreting this table as RDF comes in recognizing any object without a datatype to be an RDF simple literal [18].

A major caveat of the SQL compilation is that we must have an identifier unique to each row. This ensures the cardinality behavior described in 5.3. As row identifiers are not in the SQL language, we must modify (or copy) each table which has no primary key to institute our own row identifier (here borrowing the vendor-specific reserved word "rownum" already defined in Oracle to uniquely identify rows):

```
ALTER TABLE Protein
  ADD rownum INT UNSIGNED NOT NULL
        AUTO_INCREMENT PRIMARY KEY
```

The Direct Mapping URL-encodes (the UE function in the Direct Mapping) all SQL values and names (table names, column names). In order to be confident of proper encoding in the face of arbitrary data, the database needs a UE function [3]:

```
CREATE FUNCTION urlencode (s VARCHAR(4096))
    RETURNS VARCHAR(4096) ...
```

The subject, predicate and object of the triples in the direct graph are IRIs, blank nodes or RDF Literals, as determined by $\mathcal{E}[\![\ ]\!]^{\phi}_{row}$. For tables with IRI subjects, the subject will be calculated by concatenating the table name, with the ","-separated list of the column names:

```
CONCAT(UE("EBIact"), "/",
       UE("target"), "=", UE(target),",",
       UE("actor"), "=", UE(actor)) AS sIRI
```

Blank node subjects will require a name unique to that table and the rownum.

```
CONCAT("_:", UE("Protein"), UE(Protein.rownum))
 AS sBNode
```

Predicates are trivially expressed as an IRI:

```
CONCAT(UE("EBIact"), "#", UE("target"))
 AS pIRI
```

Objects generated from foreign keys must join against the referenced table in order to calculate the row identifier. In this case, the row identifier is a blank node:

[3] `http://snippets.dzone.com/posts/show/7746` provides one which works for MySQL and possibly other databases

```
CONCAT("_:", UE("Protein"), UE(Protein.rownum))
 AS oBNode ... FROM EBIact
 INNER JOIN Protein ON EBIact.target=Protein.ebi
```

Literal objects will either be simple literals:

```
Protein.uniprot AS oLexical,
 NULL AS oDatatype, NULL AS oLangTag
```

or have a datatype. Since oLexical is a `CHAR(40)` type and Protein.ebi is an integer, we must ensure that we inject its lexical form via the appropriate cast:

```
CAST(Protein.ebi AS CHAR(40)) AS oLexical,
 "xsd:integer" as oDatatype, NULL AS oLangTag
```

Given this set of tools, we can generate type triples for each table, as well as triples for each column and each foreign key in those tables. This will be in the form `CREATE VIEW RDF AS` plus a `UNION` of the `SELECT`s which exhaustively cover $\mathcal{E}[\![\ ]\!]^{\phi}_{row}$:

```
CREATE VIEW RDF AS
      SELECT table1 type assertion
UNION SELECT table1 foreign key 1
UNION SELECT table1 foreign key N
UNION SELECT table1 column 1
UNION SELECT table1 column N
UNION SELECT tableN type assertion
...
```

Taking the $r = EBIact$ from earlier, $(\phi(r), rdf : type, UE(relation(t)))$ is realized as:

```
SELECT
 CONCAT(UE("EBIact"), "/",
        UE("target"), "=", UE(target),",",
        UE("actor"), "=", UE(actor)) AS sIRI
 NULL AS sBNode,
 "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" AS pIRI,
 UE("EBIact") AS oIRI, NULL AS oBNode, NULL AS oLexical,
 NULL AS oDatatype, NULL AS oLangTag
 FROM EBIact
```

Eliding the copious NULL columns required to line up the disjoints, in $\{(\phi(r), p, o) \,|\, (p, o) \in \mathcal{E}[\![\texttt{r, fk}]\!]^{\phi}_{ref} \,|\, fk \in references(t)\}$, the function $references(t)$ ranges over *target* and *actor*:

```
SELECT CONCAT(UE("EBIact"), ...) ... AS sIRI, ...
 CONCAT(UE("EBIact"), "#", UE("target")) AS pIRI,
 ...
 CONCAT("_:", UE("Protein"), UE(Protein.rownum)) AS oBNode
 ...
 FROM EBIact INNER JOIN Protein ON EBIact.target=Protein.ebi
 WHERE
UNION
SELECT CONCAT(UE("EBIact"), ...) ... AS sIRI, ...
 CONCAT(UE("EBIact"), "#", UE("actor")) AS pIRI,
 ...
 CONCAT("_:", UE("Protein"), UE(Protein.rownum)) AS oBNode
 ...
 FROM EBIact INNER JOIN Protein ON EBIact.actor=Protein.ebi
```

In $\{(\phi(r), p, o) \,|\, (p, o) \in \mathcal{E}[\![\texttt{r, a}]\!]_{lex} \,|\, a \in lexicals(t)\}$, $lexicals(t)$ is the set $\{action\}$. In order to be consistent with the NULLs behavior described in 5.4, we need a constraint that the triple is generated only if the value is non-null:

```
SELECT CONCAT(UE("EBIact"), ...) ... AS sIRI, ...
  CONCAT(UE("EBIact"), "#", UE("action")) AS pIRI,
  ... action AS oLexical, ...
  FROM EBIact WHERE action IS NOT NULL
```

By construction, this SQL view respects the constraints required to interpret this table as an RDF graph, e.g. that an object be either an IRI, blank node, or literal with an optional datatype or language tag.

8. STATE OF THE ART

There are many tools which map relational databases to RDF, some by materializing RDF graphs as documents or in a database, some by creating a virtual view, queries upon which are transformed to SQL queries over the relational database. They vary in the degree of curation of the relational data, ranging from conveying just the values of the attributes in each row to representing an RDF graph using graph patterns and identifiers consistent with other Semantic Web representations of the domain data. An example of the less-curated end of the spectrum is SquirrelRDF [23], which is designed to represent each row in an SQL database as a common subject node with a set of properties with literal values. (SquirrelRDF can also express LDAP hierarchies or IMAP email repositories [15] as RDF.) SquirrelRDF on its own does not capture the relational graph; any queries will need to explicate the connections between tables by including constraints on coincident attributes (much as SQL joins are constrained, e.g. `EBIact.actor=Protein.ebi`).

D2R [10] is probably the most popular RDB to RDF mapping tool. The D2R mapping language is an RDF declaration of the mappings from rows in relational tables to RDF graphs (plus the required JDBC connection parameters). This mapping is focused around `ClassMap`s, which denote the lexical structure and type of row identifiers, and `PropertyBridge`s, which specify the creation of a row identifier's properties. D2R's default mapping includes the link table behavior described in the Overview. (The R2RML specification in progress has an expressivity similar to D2R.) D2R tools can use this mapping language to create a materialized view (dump an RDF graph), or to translate SPARQL queries to SQL queries.

Virtuoso RDF Views [1] can also map SPARQL queries to SQL queries, though the configuration language, a mixture of DDL and SPARQL, is somewhat harder to summarize. SPASQL [19] compiles SPARQL queries directly to an execution plan on a MySQL server, bypassing SQL. Triplify [6] maps URI patterns to SQL queries, with the aim of providing a Linked Data [9] endpoint. See the Related Work section of [6] for an excellent summary of different mapping approaches.

All of these tools attempt to find a sweet point between simplicity and utility. Simplicity enables confident interpretation, both for users and for architects adding more functionality to a semantic tool chain. Utility is served by communicating enough of the relational structure to appeal to users and give them an intuitive graph which is either useful in its own right, or transformable into a useful graph. A good example of this trade-off is the special treatment for link tables.

The Direct Mapping produces an RDF graph, to be manipulated further to aptly reflect domain knowledge within the Semantic Web. SquirrelRDF and D2R both provide analogous default representations of relational databases; SquirrelRDF without the relational graph. As far as the authors know, this paper is the first attempt to define a formal semantics for a mapping from relational data to RDF, allowing implementers to write complete code, and provid-

ing a grounding for query optimizers and other researchers to demonstrate e.g SPARQL-to-SQL query rewriters faithful to the Direct Mapping semantics.

9. CONCLUSION

The Direct Mapping is a simple step which places relational data into the Semantic Web. This definition may be used in many ways, ranging from serializing a direct graph for possible further manipulation by RDF tools, to defining queryable virtual graphs. While the exact details (e.g. punctuation) of the Direct Mapping are being hammered out by the RDB2RDF Working Group, we have shown the utility of this approach; breaking the representation of non-RDF domain data into a domain-agnostic "direct" representation, leaving the domain-specific manipulation to RDF tools. A functional Scala implementation[4] of the Direct Mapping has been used to validate[5] the RDB2RDF Working Group's tests. This important specification lays the groundwork for existing and nascent graph transformation tools to draw valuable relational data into the Semantic Web.

10. ACKNOWLEDGMENTS

The RDB2RDF Working Group, in particular, Juan Sequeda and Marcelo Arenas who have pursued a Datalog definition of the Direct Mapping. Helena Deus, for geeking through effective use cases.

11. REFERENCES

[1] Mapping sql data to rdf. http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSSQL2RDF.

[2] Wikipedia article: Relational model. http://en.wikipedia.org/wiki/Relational_model.

[3] Iso/iec 9075 part 1: Sql/framework. 2006.

[4] Iso/iec 9075 part 2: Sql/foundation. 2006.

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

[6] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller. Triplify – light-weight linked data publication from relational databases. *WWW2009*.

[7] D. Beckett and T. Berners-Lee. Turtle - terse rdf triple language. 2008. http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/.

[8] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Towards a mashup to build bioinformatics knowledge systems. *Journal of Biomedical Informatics*, 41:706–716, 2008.

[9] T. Berners-Lee. Linked data - design issues. 2010. http://www.w3.org/DesignIssues/LinkedData.

[10] C. Bizer and A. Seaborne. D2rq - treating non-rdf databases as virtual rdf graphs. *ISWC2004 (posters)*, November 2004.

[11] A. Chebotko, S. Lu, H. M. Jamil, and F. Fotouhi. Semantics preserving sparql-to-sql query translation for optional graph patterns. Technical report, 2006.

[12] E. F. Codd. The relational model for database management. 1990.

[13] R. Cyganiak. A relational algebra for sparql. Technical report, 2005.

[14] B. Elliott1, E. Cheng, C. Thomas-Ogbuji, and Z. M. Ozsoyoglu. A complete translation from sparql into efficient sql.

[15] D. Eynard, J. Recker, and C. Sayers. An imap plugin for squirrelrdf. Technical report. http://www.hpl.hp.com/techreports/2007/HPL-2007-161.pdf.

[16] B. He, M. Patel, Z. Zhang, and K. Chang. Accessing the deep web. *Communications of the ACM*, 50(5):94–101, 2007.

[17] A. Jentzsch, J. Zhao, O. Hassanzadeh, K.-H. Cheung, M. Samwald, and B. Andersson. Linking open drug data. 2009.

[18] G. Klyne and J. J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[19] E. Prud'hommeaux. Spasql: Sparql support in mysql. http://www.w3.org/2005/05/22-SPARQL-MySQL/XTech.

[20] E. Prud'hommeaux, M. Hausenblas, S. Auer, L. Feigenbaum, D. Miranker, A. Fogarolli, and J. Sequeda. Use cases and requirements for mapping relational databases to rdf. 2010. http://www.w3.org/TR/2010/WD-rdb2rdf-ucr-20100608/.

[21] E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf. 2008. http://www.biopax.org/release/biopax-level3-documentation.pdf.

[22] D. Raggett, A. L. Hors, and I. Jacobs. Html 4.01 specification. 1999. http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1.

[23] D. Steer. Squirrelrdf. 2006. http://jena.sourceforge.net/SquirrelRDF/.

[4] http://www.w3.org/2011/02/DM-KCAP/code

[5] http://www.w3.org/2011/02/DM-KCAP/srvc