Abstract. A key goal of the Semantic Web is to expose the vast quantities of data currently held in relational databases so as to create a “Giant Global Graph”, expressed as RDF. However, simple algorithms for automatically mapping relational data to RDF are wasteful and inefficient. I argue that the mapping can be significantly improved by using a schema design tailored to RDF, and the smaller graph that results will promote powerful query mechanisms like faceted search.

The research reported builds a substantial RDF graph from a real data archive in the cultural heritage domain. I will discuss specific characteristics of this domain, and propose a generic RDB2RDF mapping for such data, based on the simple “People, Places, Things and Events” approach frequently used in heritage data management. This mapping results in a graph of 20 million triples, which represents a ten-fold reduction in size over the standard approach.

Key words: RDF, relational database, RDB2RDF, mapping algorithm, cultural heritage

1 Introduction

By far the most popular way of storing structured data is the relational database. Vast quantities of carefully curated information, collected over centuries sometimes, are held in this way. Each of the world’s cultural archives is a jewel in its own right but realising their potential, by matching them and stringing them together over Web, is very hard. The databases are held in proprietary RDBMS software on servers behind firewalls and, if the data is exposed on the Web at all, it is generally presented in a format determined by the data owners, through a specially designed interface on their own website. Each database typically has its own schema design and, even if it were possible for a web-based software agent to access it without being stopped by security measures, detailed knowledge of the schema is needed for meaningful querying. This means in practice that the data curators must themselves provide any gateways to their holdings that exist.

In other domains, this is often a desirable state of affairs. Commercial organisations do not particularly want outside users burrowing into their financial
databases, for example. In the cultural heritage domain however, the curators
generally do want to make their holdings as widely available as possible. Cultural
information has often been gathered in “silos”: the museums curate information
about physical objects whilst other archives keep data about the historical sites
where they were found, yet others have holdings about the lives of the people
who used the objects, and so on. The ideal would be to hold this information in
such a way that web agents could query across the distributed silos and present
what they found in a way that suited their particular user. We are still a long
way from this goal, but one of the aims of the Semantic Web (see [1, 2]) is to
address the problem by making data linking as straightforward as HTML makes
document linking.

This paper describes the conversion of the archive data of the Royal Commis-
sion on the Ancient and Historical Monuments of Scotland\(^1\) (RCAHMS) from
relational database (RDB) format to a graph of RDF triples, that will be access-
sible to Semantic Web tools. This is one step in a larger project, named \textit{tether}\(^2\)
whose goal is to combine the RDB data with binary relations extracted from
free text documents, and with specialised domain ontologies [3, 4], and so allow
SPARQL querying across the whole unified dataset and across any related ones
dealt with through the same pipeline (see [5]).

There are many systems available or in development for RDB to RDF trans-
lation, and examples are described in Sect. 2. I will argue that these are often
wasteful, cluttering the RDF graph with unnecessary triples, and that the stan-
dard approaches can severely limit the future usefulness of the generated RDF.
In Sect. 3 the typical characteristics of cultural heritage data are described, as
they have an impact on the RDB2RDF conversion. The core of the paper is Sect.
4, which lists issues to be dealt with in RDB2RDF conversion and proposes an
RDF schema design that requires manual intervention, but which is generic for
this domain and has characteristics relevant to any domain. Section 5 explains
how this extra initial work pays for itself in terms of greater query power over the
final graph. Finally, Sect. 6 draws some overall conclusions about the RDB2RDF
process.

2 Basic RDB to RDF Conversion

There is considerable interest in RDB2RDF conversion at present, and the W3C
has recently set up an Incubator Group to work on it.\(^3\) The basic process is
straightforward and is illustrated in Fig. 1, which shows a much simplified ver-
sion of the RCAHMS database \texttt{SITE} table. Each database tuple (or table row)
becomes a cluster of triples grouped around a bnode (RDF blank node) whose
type (implemented as an \texttt{rdf:type} property) is a class (\texttt{rdfs:Class}) corre-
responding to the table name. Hence this conversion is sometimes called “Table
to Class” transformation.

\(^1\) \url{http://www.rcahms.gov.uk/}

\(^2\) “Tether” is a dialect word for three, used in the north of England for counting sheep.

\(^3\) \url{http://www.w3.org/2005/Incubator/rdb2rdf/}
Fig. 1. Translation of a relational table tuple to RDF, using bnodes.

The use of bnodes is an immediately contentious issue. The method illustrated in Fig. 1 is often known as “duck typing”: the bnode has the properties of a site (viz. siteNo, name, parish and classification) so let’s call it a site. For reasons explained below, I advocate using direct typing instead, by moving the database primary key attribute (siteNo in Fig. 1) into the position of the bnode, with a URI such as :siteid#site1, and putting it in a new class named siteid. Thus it is not the database table which corresponds to an RDF class, but the table’s primary key.

We should note that, already, we may have compromised a fully automatic procedure. The primary key has to be identified, which requires knowledge of the relational schema. The key may be specified as part of the table definition, and hence accessible to software, but this is not mandatory. (It is not so specified for most tables in the RCAHMS dataset for example, as they were created many years ago, when SQL was less comprehensive.) If no suitable primary key exists, as is perfectly possible (in the limit, the entire tuple may constitute the unique key), then one may be generated for the RDF graph. Recognising that this is necessary also requires schema knowledge.

Using identified nodes instead of bnodes, the basic process is that each RDF triple consisting of subject node, predicate (or property) arc and object node:

... is derived from the database as follows:

— where the “attribute” is a URI derived directly from the database field or column name (sometimes called “Column as Predicate” mapping), and the “value”

4 I am following the usual convention of abbreviating URIs through prefixes, so :siteid#site1 represents “(http://www.ltg.ed.ac.uk/tether/siteid#site1)”.

<table>
<thead>
<tr>
<th>siteNo</th>
<th>name</th>
<th>parish</th>
<th>classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dirleton Castle</td>
<td>Dirleton</td>
<td>defence</td>
</tr>
<tr>
<td>2</td>
<td>Dirleton Cottage</td>
<td>Dirleton</td>
<td>residential</td>
</tr>
<tr>
<td>3</td>
<td>Drem Airfield</td>
<td>Dirleton</td>
<td>military</td>
</tr>
<tr>
<td>4</td>
<td>Jamie’s Neuk</td>
<td>Dirleton</td>
<td>military</td>
</tr>
</tbody>
</table>
is the content of the field and is represented as a literal value in this basic translation procedure. The term “row_key” is used here to avoid confusion with “rowid”, which is used in most relational databases for the hidden identifier used by the data dictionary, and not for the primary key chosen by the database designer.

Taking one of the triples from Fig. 1, we have:

\[
\text{siteid#site1 :name “Dirleton Castle”}
\]

It is worth stressing the qualitative difference between :name and “Dirleton Castle”. However the predicate URI is derived, whether by the “Column as Predicate” method or as described in Sect. 4 below, it is metadata and its precise form is ultimately at the whim of a schema designer. The object node however, whether a literal as here or a URI as suggested below, contains genuine content data that must be preserved.

The procedure outlined corresponds to the approach described by Berners-Lee [6], where database attributes are translated to RDF property arcs with full provenance information about their database source.

There is a growing number of automatic conversion tools that work on these principles, such as D2RQ, Dartgrid [7], Dan Connolly’s dbview program, R2O [8] and Triplify, with more or less scope for customisation by the user in each case. The mapping to RDF does not necessarily have to be instantiated. Instead a virtual graph may be constructed, saving space and ensuring data currency, at the expense of increased processing cost at query time. SquirrelRDF and R2D2 (a sister project of D2RQ) are examples of tools that construct a graph “view” of a relational database, in effect allowing SPARQL queries to be run instead of SQL ones. D2RQ and Virtuoso give the user the choice of an instantiated graph or a virtual one.

One of the key issues is how to generate suitable URIs for the RDF “resources”. The subject and property of an RDF triple must both be resources with URLs, whilst the object may be either a URI or a literal string. If the object is a literal, as in the basic translation process described above, it automatically becomes a “leaf” node at the edge of the RDF graph, as it cannot be a subject node without a URI. (See [10, 11, 12] for full treatment of RDF syntax.) Generation of URIs is a vexed question that I will return to in Sect. 4.2 below. In the examples looked at so far, the tacit assumption is that the RDB remains the “master” version of the data, and the RDF is a derived copy. In the long-term we should surely be planning for RDF datasets that don’t need to carry their life history in this way, any more than an RDB designer expects to record in every field information about the flat files, spreadsheets, hierarchical databases or manuscript notebooks, that held each piece of data before it was moved to a relational database.

5 http://www4.wiwiss.fu-berlin.de/bizer/D2RQ/
6 http://dig.csail.mit.edu/2006/dbview/dbview.py
7 http://triplify.org/About
8 http://jena.sourceforge.net/SquirrelRDF/
9 http://aksw.informatik.uni-leipzig.de/Projects/R2D2
3 Cultural Heritage Data

The data used for this work is a complete snapshot of a real (and therefore constantly changing) archive, belonging to RCAHMS.\footnote{The data has been made available by RCAHMS for research purposes; please contact the author if you are interested in using it. It is in various formats, the source being an Oracle database. In particular, a corpus of 1500 documents taken from “long” database fields has been annotated for named entities and for binary relations.} The relational database consists of about 250,000 records describing historical sites in Scotland, linked to around 1.5 million records of associated archive material such as text documents, photographs, maps, site drawings, bibliographic material and so forth. The collection has been assembled over ten decades (2008 marks RCAHMS’ first centenary) by field survey and research. As recording methods have changed over this period, and as resources have been scarcer at some times than others, the detailed format varies across the archive. Many database tables are sparsely populated as there are very few mandatory fields, field content is not uniform and is generally not controlled. Typographical errors are fairly common, partly because of lack of data entry controls, and partly because the bulk of the data was captured by Optical Character Recognition, almost twenty years ago. As is common in this domain, there are a great many text “notes” fields, of varying length up to “long” fields that are limited only by physical machine constraints.

As well as oddities introduced by the OCR step, there is a sprinkling of spurious non-printable characters throughout the data fields. These may result from bugs in data entry software and probably also from the varied life the electronic data has led, being moved from platform to platform as technology changes. Twenty years ago electronic archives of this size, requiring multi-user access, had to be held on big central mainframes. (Nowadays of course it fits comfortably on a laptop.) The RCAHMS data has moved from an IBM EBCDIC machine to an ASCII one and thence to a UTF-8 character set. To take a particular example, the newline character has sometimes been \texttt{LF} (linefeed, hex OA), sometimes \texttt{CR} (carriage return, hex OD) and sometimes the two in combination. Platform conversion did not always deal properly with differences like these, and unwanted hex characters crop up infrequently but unpredictably throughout the data. Of course, character errors like these are a great deal easier to fix than data content gaps where fields simply don’t contain data that is wanted for querying.

Although RCAHMS has worked steadily to upgrade and clean the data, manual correction is a mammoth task for a poorly resourced body, so finding automatic methods is a priority. The work described in this paper is part of a larger project to explore the use of NLP and Semantic Web tools for standardising and promoting access to cultural data. From experience of several similar archives, the RCAHMS data is very typical of the domain. Most such organisations are faced with major data reorganisation projects nowadays, to meet high expectations for Web-based access.

These factors all impinge on the RDB2RDF process, but there are other important characteristics of RDB data that are not specific to this domain.
way entity relationships are designed, the way tables are joined, the way data is put into Normal Forms — these are all tailored for relational databases and do not necessarily translate into efficient RDF graphs.

When translating a database to RDF one needs to check whether all of it is actually needed. Of the 257 tables in the RCAHMS schema, some are backwaters not currently maintained, some are part of the data administration framework, and some are frankly the sort of detritus that tends to accumulate in a twenty year old database. Only 24 tables were selected as the basis for the tether design and, from these, just 9 classes were created at the top of the RDF ontology schema: for sites, archive items, named collections, bibliographic items, bibliographic references, people, organisations, agent roles, and linear features. All but the last mentioned are used throughout the heritage domain and collectively the set provides comprehensive coverage of the basic data needs of that domain. Figure 2 shows the relationships between these classes.

4 The Need for RDF Schema Design

Having outlined the basic mechanism for RDB2RDF conversion, and described the nature of the real-world data being dealt with, I will now list each of the points on which the tether conversion process differs from that given above. The RDF schema design, or upper ontology, for tether is available at [http://synapse.inf.ed.ac.uk/~kate/tetherdb_edited.nt](http://synapse.inf.ed.ac.uk/~kate/tetherdb_edited.nt), in NTriples format. The system’s RDB2RDF translation software is in Java and is also available on request.
4.1 Dealing with Relational Joins

Where two RDB tables are in a many-to-many relationship, a new table must be introduced between them. Figure 3 shows an example, based on a simplified version of the RCAHMS site and archive entities. Each site will typically have many archive items associated with it and conversely a particular archive item such as a map may reference many separate sites, so the relationship is many-to-many. The SITE-ARC table in Fig. 3 resolves the relationships by indicating which sites are linked to which archive items and vice versa.

The first thing to note about SITE-ARC is that it uses a concatenated primary key: the combination of the two foreign keys, siteNo and arcNo. This is perfectly good RDB design, but for RDF we save a lot of trouble (see Sect. 4.7) by taking the time to generate a surrogate primary key for the table. Let us call it siteArcNo. The graph that the standard procedure then generates is shown in Fig. 4, on the left hand side. (Strict URI prefixing has been omitted, for simplicity.)

It can be seen that the five siteArc nodes are redundant. The resources they represent have no properties of their own, and the node is merely an intermediary between a site and an arc node — impossible to do without in the RDB but not needed in the RDF. The right hand side of Fig. 4 shows the same graph with the redundant nodes pruned out. A new siteArc predicate has been introduced which, strictly speaking, is bi-directional. However, there seems no reason in a practical implementation for not simply picking the direction, and in tether these key-to-key links between database entities always point in the direction their names suggest, e.g. in a siteArc triple the subject is always a site and the object is always an arc (or, strictly, siteid and arcid). The convention adopted is that the arc always points from foreign key to primary key or, equivalently, from the “many” end to the “one” end (see Fig. 2).
Clearly, if the intermediary RDB table is not there merely to resolve the many-to-many link but has attributes of its own, then the table cannot be pruned out and we have a pair of one-to-many links instead. The REF class shown in Fig. 2 corresponds approximately to a table in the RDB that resolves the many-to-many join between sites and bibliographic items. In this case the reference entity has local attributes such as page numbers, so it must be retained. Deciding this automatically from the RDB data dictionary may be impossible.

One-to-many joins are treated similarly. If foreign key fields are treated like the other table fields, they will produce redundant arcs that can be dropped. Figure 4 demonstrates this for many-to-many, but the same is true for one-to-many joins. There is no need to generate a triple pointing at a foreign key value as, when the child table is processed, a triple will automatically be generated pointing at each of its parent records. Alternatively one could always create the triple from parent to child if preferred; the point is that one does not need two of them, pointing both ways between the same two nodes.

For the RCAHMS database eliminating redundant key-to-key links saves almost 2.8 million triples.

4.2 URI naming

There is plenty of guidance on how to produce “Cool URIs” (see [13], [14], [15], [16]), i.e. ones which are persistent, actually point to something, and are reasonably easy to read.
The manner of dealing with “# URIs”, or HTML fragment identifiers, is too large a subject to handle fully here. In brief, tether is designed so that the upper schema (all of the rdfs:Class and rdfs:SubClassOf hierarchy) can be served via standard HTML pages or as RDF through content negotiation and 303 redirection, following the W3C guidance given in [15].

A fundamental point to make about URIs in RDF is that they must be distinct when they refer to distinct resources but there should not be a plethora of different URIs for the same resource. This point is often side-stepped in RDB2RDF work, where every single database table cell may be given a unique URI. Yet much of the benefit of using RDF is completely thrown away if we lose the chance to link subgraphs together on shared nodes.

It is very easy to generate unique URIs, but much less easy to decide whether two URIs actually indicate the same thing. Is my “http://www.ltg.ed.ac.uk/tether/loc/place#Edinburgh” referring to the same thing as your “http://www.geonames.org/2650225/edinburgh.html”? Well, possibly, but it’s easier to tell at the design stage than later. Designers often say that rdfs:seeAlso or owl:sameAs links can easily be added later, but this is rarely true.

A good RDF schema should be as simple as it can be whilst keeping sufficient expressive power. In tether the URIs used are not primarily intended to tie each data item to its position in the RDB, but instead to give it a canonical representation so that two data items that refer to the same thing will get the same URI. To take an example: for historical reasons the RCAHMS site table has five different columns for administrative area names: parish, region, district, county and council. A data value of “Edinburgh” in any of these fields will translate to a URI of “http://www.ltg.ed.ac.uk/tether/loc/place#Edinburgh”. This URI is then typed as “http://www.ltg.ed.ac.uk/tether/loc/place#parish”, or whichever is the appropriate class. This results in multiple class membership if the same value also occurs in the council field, say.

4.3 Avoiding Duplication

In the standard RDB2RDF mechanisms the URI of each property arc embodies metadata about the RDB source, such as “http://www.acme.com/mycat/schem1/empdb/ems/shoe”. The source node of this triple will also probably identify the cell it comes from, say “http://www.acme.com/mycat/schem1/empdb/ems/rowid=123;col=shoe”. The source node may also have an rdf:type property pointing to a class such as “http://www.acme.com/mycat/schem1/empdb/ems/”. If the object node is not a literal we will have to repeat much of this metadata.

12 I leave aside philosophical debate about whether an HTML text document can ever be an RDF resource. The point at issue is more basic than an argument about reference grounding: are we talking about Edinburgh, Scotland or one of the many others, or about a person with the unusual name of “Edinburgh”?

13 The class hierarchy is encoded in the URIs, to promote RESTful (Representational State Transfer — see [17, 18]) web service access.

14 The example is from [6], this data item being about an employee’s shoe size.
again at that end of the triple. The data bloat is staggering, and the duplication is poor practice from a maintenance point of view. We have a lot of work to do if a zealous DBA insists that the `emps` table should really be named `emp`, say.

My argument is that the graph should be as compact as possible, and node and property URIs should not share each other’s functions. Data items belong in instance nodes, named with a view to distinguishing ones that are different and allowing serendipitous links to be found between ones referring to the same thing. Each needs a type property to establish it as one of a class (or of several classes), and this is where identifying metadata from the RDB belongs: specifying what kind of thing the object node refers to. The predicate does not need to repeat this information.

4.4 Nouns or Verbs? Properties or Classes?

It is sometimes helpful to think of the RDF triple as “subject–verb–object”, so that the graph arcs become a set of verbs and the nodes are nouns. Thinking in terms of “things” connected by action words makes it more natural to argue, as just done above, that the right place for RDB metadata describing the type of thing in a node, is a class definition. What the property arc should represent is a verb phrase expressing which of the allowed relationships in our graph is present. The design of the predicate set is covered in Sect. 4.10, but it is worth noting here that the `tether` design transposes all of the predicates of Fig. 1 into classes or “nouns”.

4.5 Using Resources Instead of Literals

At first glance, the obvious thing to do with a database value is to put it in a literal, typed as a string, integer, date or whatever is appropriate. However, this sterilises the graph at that point: no further offshoots are possible, as a literal cannot be the source node of a triple. Figure 5, left hand side, illustrates the problems this may cause. Our site is named “Dirleton Castle”, but we subsequently find — perhaps in another RDF graph in the future — that Dirleton Castle is in East Lothian. The natural course is to make `:dirletonCastle` a resource in our data schema, that can have properties such as `:hasLocation`.

Once `:dirletonCastle` has become a resource it needs to have its type declared and we need an `rdfs:label` to preserve the original database content. The right hand side of Fig. 5 shows the result. The same argument applies to all of the literals shown in Fig. 5, each of which represents something of significance that may well have properties of its own.

So should every database value become a resource? My answer is “yes”. It would be possible to go through the database fields deciding which were likely to point to real-world objects or important concepts that should be allowed resource status, and which were purely local references such as code values, or cumbersome strings that it would seem foolish to generate URIs for. But this is schema knowledge with a vengeance! The sensible alternative seems to be to treat all data values alike, and generate URIs for them.
Having complained of data bloat earlier, I have now introduced the potential for three triples for each data item instead of one: the original inward pointing one, and outward pointing \texttt{rdf:type} and \texttt{rdfs:label} arcs. The alternative solution is to use bnodes (so \texttt{:dirletonCastle} would be replaced with a bnode) which introduces exactly the same additional triples. In fact, eschewing bnodes means that instead of immediately being multiplied by three, the number of triples increases by a much smaller percentage, as is explained below.

\subsection*{4.6 Avoiding Bnodes}

The \textit{tether} implementation uses no bnodes at all. The trouble with bnodes is that they are blank. This means that sub-graphs cannot be straightforwardly linked using them, so each one is separate even when they really need to be merged. In theory the RDB2RDF process could arrange to merge bnodes when it was certain that the data item was the same, but one can avoid a lot of error-prone work by simply generating URIs instead, so that nodes are automatically merged when appropriate.

Because bnodes are not used, the introduction of type and label arcs for each data item does not triple the size of the graph. These links are only needed once for each distinct occurrence of a particular data item. It turns out that the number of type and label arcs pointing outwards from each data item is very much smaller than the number of inward pointing “data content” arcs, so the size of the triple store is not increased three-fold, but in fact by one third, from 16.2 million to 21.6 million, or by 33\% instead of 300\%. Table 1 gives the actual numbers for the RCAHMS data. This is an interesting side-light on the amount of repetition present in this kind of data, which bodes well for the application of NLP and machine learning techniques that exploit patterns.
The table also gives, for comparison, the product of the rows and columns present in the subset of RCAHMS tables used, which is the theoretical maximum number of triples that could be generated, not counting schema triples. This is around ten times the actual value of 21 million, so clearly shows the usefulness of the various graph pruning techniques discussed.

### 4.7 Primary Keys

As has been said, *tether* operates a non-intervention policy, to permit data values to merge automatically. However, one class of data values that must never merge is the set of primary keys. In many cases the primary key is a generated running number, and it would certainly be disastrous to give site 123 the same URI as archive item 123. To ensure this, the primary keys in *tether* are always prefixed with a short label for their parent class. Thus we have site123, arc123, bib123 and so on, for all the top level classes in the hierarchy.

As a general principle, surrogate values of any kind, whether generated keys or codes pointing to lookup tables, should not be allowed to merge. Such values are arbitrary strings that are only valid in context, as opposed to grounded labels for real entities like “Edinburgh”, or “aerial view of the Forth Bridge”.

In the example shown in Fig. 4, a surrogate primary key was added to the source database table to replace the concatenated one. If bnodes were used then each part of the key would have its own triple instead, making the SPARQL joins less efficient.

### 4.8 Null Fields

There has long been debate in the relational database world about whether null values should actually be permitted in RDBs. (Ted Codd admitted nulls to his “12 rules” in 1979 but his collaborator Chris Date has always believed it was a mistake [19].) In practical implementations they are certainly here to stay, and the RCAHMS database is littered with them.

Where a field value is null no triple is generated in *tether*. This may seem obvious, but it does require schema knowledge. Though it is generally thought poor practice, a relational designer may choose to use the absence of a data value to carry meaning: the closed world assumption in effect. If the date of death field is null, the person is still alive, say. (This example illustrates some of the shortcomings: we may just not know when or if the person died.)

<table>
<thead>
<tr>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema triples — upper ontology</td>
<td>239</td>
</tr>
<tr>
<td>Distinct RDB “data content” triples (incl key-to-key links)</td>
<td>16,239,371</td>
</tr>
<tr>
<td>Distinct type and label triples (excl schema)</td>
<td>5,397,681</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>21,637,291</td>
</tr>
<tr>
<td>Rows x Columns for relevant RDB tables</td>
<td>234,242,572</td>
</tr>
</tbody>
</table>

**Table 1.** Schema statistics and triple counts for *tether* RDF graph.
4.9 De-normalising and Coded Values

Coded lookup tables are common in relational databases. The location fields mentioned above are held as short code fields on the site table. There is no point in preserving these code values in RDF as they serve merely as intermediaries, with no properties of their own. When these fields are processed, the code is expanded so that the triple points straight to the actual data item referenced.

An exception is when codes have intrinsic meaning, perhaps through long use. For example, in the RCAHMS dataset collection items are assigned “prefixes” that are known at least to specialists (such as “EE” for the Empire Exhibition collection), so are probably worth preserving. The compromise adopted is to use the short code in the URI string but expand it in the label literal.

In general, where the RDB designer seeks to normalise tables (roughly speaking, this means splitting big tables into smaller ones until dependencies between columns are eliminated), the RDF designer wants to denormalise: shortening long paths whenever possible, by cutting out intermediate nodes and classes that are not needed.

4.10 Limiting the Predicate Set

In heritage data management the key data attributes are often stated as “Who? What? Where? When?” or “People, Places, Events and Things”. With this in mind, the predicate set was designed to be very small, and yet sufficient.

Figure 2 shows the core schema classes. These are at the top of a class hierarchy that is no more than three levels deep, and contains only 60 separate classes. (Perhaps surprisingly, this does not make the RDB2RDF process irreversible, though that was not a consideration.) The set of nine relationships between the top classes form the backbone framework in tether. Just eight more predicates were added to cover the all the relationships needed to express the database contents, plus a further three to cater for text relations:

- :hasAgent
- :hasAgentRole
- :hasClassn (points to another graph, derived from domain ontologies)
- :hasDesc (for short descriptive fields)
- :hasLocation
- :hasPeriod
- :hasId (for identifying codes that are considered worth preserving)
- :hasFlag (for tags or indicators that are attached to records).
- :hasEvent
- :hasPatient
- :partOf

No suitable predicate was found in the standard vocabularies; the closest were FRBR:partOf and dc:isPartOf, but both of these are intended for bibliographic items, not the more general relationship needed here.
In addition, some standard predicates from published vocabularies for RDF, RDFS and OWL were used.

This is probably the single most significant contrast with the basic translation process outlined in Sect. 2, where the predicate set directly reflects the RDB attributes and therefore will usually be enormous.

5 Graph Querying

The RDF triples produced from the RCAHMS database are held in a triple store, and queried using SPARQL. There are many such stores available now. The two that have been used are Jena\textsuperscript{16} initially and, more recently, AllegroGraph\textsuperscript{17}. These were chosen after a survey of those available, chiefly for their design features, robustness, good documentation and usable APIs. When doing repeated loading and formatting experiments with a multi-million triple set, AllegroGraph’s remarkably good loading and indexing performance is attractive. Loading the 21 million triples into AllegroGraph takes 33 minutes and indexing takes a further 20 minutes on a standard PC. Loading and indexing the same set in Jena takes just under 15 hours. There are unresolved issues with the 32-bit version of AllegroGraph though, and a beta release of the 64-bit version is currently being tried, kindly offered by Franz Inc.

A problem faced by non-specialist users of the RCAHMS dataset is that, without good knowledge of domain terminology, it can be difficult to frame meaningful queries. One of the reasons for using a very tightly limited predicate set is to eliminate a lot of the complication, by reducing the number of categories or “facets” to search within, so that it becomes possible to summarise over each category and offer the user intermediate results with meaningful context.

Suppose the user is interested in forts. This is too broad a term to produce very manageable results, as there are over 1700 “forts” in the RCAHMS database, dating from many periods and scattered all over the country. Rather than presenting a very long list in no particular order, what may help this user is a structured summary of information about forts, from which a further search can be made. Figure 6 shows a simple example, where four forts are given with their locations, periods and the agents who are authorities on them. The right hand side of the figure shows the RDF sub-graph, whilst the left hand side is a possibly more readable version of the same thing.

Firstly we find all the site nodes linked to a “fort” classification term. For this set of nodes, an analysis by each of its other predicates is made, to calculate frequency counts for the top three or four values, to present to the user as illustrated. The query can then be refined by picking from the terms offered: say “Roman forts in Tayside”. This kind of faceted search is almost certainly impractical if the predicate set has hundreds of members, as it will do in a standard RDB2RDF conversion.

\textsuperscript{16} http://jena.sourceforge.net/
\textsuperscript{17} http://agraph.franz.com/allegrograph/
Another advantage of having a very simple schema is that it can be interpreted by software agents without enormous programming effort. A generic cultural heritage RDF schema, such as is proposed here, would enable standardised web services to explore distributed datasets and assemble results from them in a way that is quite impossible at present. Many portal sites already exist in the cultural domain, but all are specifically designed for particular databases and have generally taken years to develop. The whole procedure outlined here is adaptable to related datasets in the same domain, and could potentially deliver a distributed but connected web of cultural heritage data.

6 Conclusions

It is sometimes suggested that, for the Semantic Web to become mainstream, the priority is simply to generate more RDF data. This explains the interest in automatic tools that will take a relational database and turn it into a graph of RDF triples with minimal user intervention. However, the resulting graph may be much larger and more cumbersome than it needs to be, and the argument here is that it is worth the effort of incorporating a careful manual RDF design step to get a truly efficient dataset that will be genuinely usable. Just as the relational database was painstakingly designed, so should its RDF counterpart be. The design stage only has to be gone through once, and will then last for many years.

The pitfalls in RDB2RDF translation have been examined one by one. Most will apply to any relational database. It is hard to say precisely how great a size reduction is possible, but it has been shown that millions of triples can be eliminated by quite simple measures, and the tether graph would be around ten times as big if these steps were ignored. I have concentrated on the particular issues relevant to cultural heritage data and suggested a compact schema design, based around the data management principles established in that field and adaptable for any archive similar to the one used here.
References


