W3C Recommendation

R2RML: RDB to RDF Mapping Language

W3C Recommendation 27 September 2012

This version:
http://www.w3.org/TR/2012/REC-r2rml-20120927/

Latest version:
http://www.w3.org/TR/r2rml/

Previous version:
http://www.w3.org/TR/2012/PR-r2rml-20120814/

Editors:
Souripriya Das, Oracle
Seema Sundara, Oracle
Richard Cyganiak, DERI, National University of Ireland, Galway

Please refer to the **errata** for this document, which may include some normative corrections.

See also **translations**.

Copyright © 2012 W3C® (MIT, ERCIM, Keio), All Rights Reserved. W3C liability, trademark and document use rules apply.

Abstract

This document describes R2RML, a language for expressing customized mappings from relational databases to RDF datasets. Such mappings provide the ability to view existing relational data in the RDF data model, expressed in a structure and target vocabulary of the mapping author's choice. R2RML mappings are themselves RDF graphs and written down in Turtle syntax. R2RML enables different types of mapping implementations. Processors could, for example, offer a virtual SPARQL endpoint over the mapped relational data, or generate RDF dumps, or offer a Linked Data interface.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document was published by the RDB2RDF Working Group. Comments on this document should be sent to public-rdb2rdf-comments@w3.org, a mailing list with a public archive. The following related documents have been made available:

- A wiki page with additional information for users and implementers of R2RML,
- a color-coded diff of all changes since the previous draft,
- the implementation report used by the director to transition to W3C Recommendation.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

- **1 Introduction**
  - 1.1 Document Conventions
- **2 R2RML Overview and Example (Informative)**
  - 2.1 Example Input Database
  - 2.2 Desired RDF Output
  - 2.3 Example: Mapping a Simple Table
  - 2.4 Example: Computing a Property with an R2RML View
  - 2.5 Example: Linking Two Tables
  - 2.6 Example: Many-to-Many Tables
  - 2.7 Example: Translating database type codes to IRIs
- **3 Conformance**
- **4 R2RML Processors and Mapping Documents**
  - 4.1 Mapping Graphs and the R2RML Vocabulary
  - 4.2 RDF-based Turtle Syntax; Media Type
  - 4.3 Data Errors
  - 4.4 Default Mappings
- **5 Defining Logical Tables**
  - 5.1 Base Tables and SQL Views (`rr:tableName`)
  - 5.2 R2RML Views (`rr:sqlQuery`, `rr:sqlVersion`)
- **6 Mapping Logical Tables to RDF with Triples Maps**
  - 6.1 Creating Resources with Subject Maps
  - 6.2 Typing Resources (`rr:class`)
  - 6.3 Creating Properties and Values with Predicate-Object Maps
- **7 Creating RDF Terms with Term Maps**
  - 7.1 Constant RDF Terms (`rr:constant`)
  - 7.2 From a Column (`rr:column`)
  - 7.3 From a Template (`rr:template`)
  - 7.4 IRIs, Literal, Blank Nodes (`rr:termType`)
  - 7.5 Language Tags (`rr:language`)
  - 7.6 Typed Literals (`rr:datatype`)
  - 7.7 Inverse Expressions (`rr:inverseExpression`)
- **8 Foreign Key Relationships among Logical Tables (`rr:parentTriplesMap`, `rr:joinCondition`, `rr:child` and `rr:parent`)**
- **9 Assigning Triples to Named Graphs**
  - 9.1 Scope of Blank Nodes
- **10 Datatype Conversions**
  - 10.1 Introduction (Informative)
  - 10.2 Natural Mapping of SQL Values
  - 10.3 Datatype-override Mapping of SQL Values
  - 10.4 Non-String Columns in String Contexts
  - 10.5 Summary of XSD Lexical Forms (Informative)
- **11 The Output Dataset**
  - 11.1 The Generated RDF Triples of a Triples Map
  - 11.2 The Generated RDF Term of a Term Map
- **A. RDF Terminology (Informative)**
- **B. Index of R2RML Vocabulary Terms (Informative)**
  - B.1 Classes

B.2 Properties
B.3 Other Terms

C. References
C.1 Normative References
C.2 Other References

D. Acknowledgements (Informative)

---

1 Introduction

This specification describes R2RML, a language for expressing customized mappings from relational databases to RDF datasets. Such mappings provide the ability to view existing relational data in the RDF data model, expressed in a structure and target vocabulary of the mapping author's choice.

This specification has a companion that defines a direct mapping from relational databases to RDF [DM]. In the direct mapping of a database, the structure of the resulting RDF graph directly reflects the structure of the database, the target RDF vocabulary directly reflects the names of database schema elements, and neither structure nor target vocabulary can be changed. With R2RML on the other hand, a mapping author can define highly customized views over the relational data.

Every R2RML mapping is tailored to a specific database schema and target vocabulary. The input to an R2RML mapping is a relational database that conforms to that schema. The output is an RDF dataset [SPARQL], as defined in SPARQL, that uses predicates and types from the target vocabulary. The mapping is conceptual; R2RML processors are free to materialize the output data, or to offer virtual access through an interface that queries the underlying database, or to offer any other means of providing access to the output RDF dataset.

R2RML mappings are themselves expressed as RDF graphs and written down in Turtle syntax [TURTLE].

The intended audience of this specification is implementors of software that generates or processes R2RML mapping documents, as well as mapping authors looking for a reference to the R2RML language constructs. The document uses concepts from *RDF Concepts and Abstract Syntax* [RDF] and from the SQL language specifications [SQL1][SQL2]. A reader's familiarity with the contents of these documents, as well as with the Turtle syntax, is assumed.

The R2RML language is designed to meet the use cases and requirements identified in *Use Cases and Requirements for Mapping Relational Databases to RDF* [UCNR].

1.1 Document Conventions

In this document, examples assume the following namespace prefix bindings unless otherwise stated:

Prefix	IRI
`rr:`	`http://www.w3.org/ns/r2rml#`
`rdf:`	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
`rdfs:`	`http://www.w3.org/2000/01/rdf-schema#`
`xsd:`	`http://www.w3.org/2001/XMLSchema#`
`ex:`	`http://example.com/ns#`

Throughout the document, boxes containing Turtle markup and SQL data will appear. These boxes are color-coded. Gray boxes contain RDFS definitions of R2RML vocabulary terms:

R2RML vocabulary definition

```
# This box contains RDFS definitions of R2RML vocabulary terms
```

Yellow boxes contain example fragments of R2RML mappings in Turtle syntax:

Example R2RML mapping

```
# This box contains example R2RML mappings
```

Blue tables contain example input into an R2RML mapping:

EXAMPLE

ID INTEGER PRIMARY KEY	**DESC** VARCHAR(100)
1	This is an example input table.
2	The table name is EXAMPLE.
3	It has six rows.
4	It has two columns, ID and DESC.
5	ID is the table's primary key and of type INTEGER.
6	DESC is of type VARCHAR(100)

Green boxes contain example output:

Example output data

```
# This box contains example output RDF triples or fragments
```

2 R2RML Overview and Example (Informative)

This section gives a brief overview of the R2RML mapping language, followed by a simple example relational database with an R2RML mapping document and its output RDF. Further R2RML examples can be found in the *R2RML and Direct Mapping Test Cases* [TC].

An R2RML mapping refers to logical tables to retrieve data from the input database. A logical table can be one of the following:

1. A base table,
2. a view, or
3. a valid SQL query (called an “R2RML view” because it emulates a SQL view without modifying the database).

Each logical table is mapped to RDF using a triples map. The triples map is a rule that maps each row in the logical table to a number of RDF triples. The rule has two main parts:

1. A subject map that generates the subject of all RDF triples that will be generated from a logical table row. The subjects often are IRIs that are generated from the primary key column(s) of the table.
2. Multiple predicate-object maps that in turn consist of predicate maps and object maps (or referencing object maps).

Triples are produced by combining the subject map with a predicate map and object map, and applying these three to each logical table row. For example, the complete rule for generating a set of triples might be:

- **Subjects:** A template `http://data.example.com/employee/{empno}` is used to generate subject IRIs from the `empno` column.
- **Predicates:** The constant vocabulary IRI `ex:name` is used.
- **Objects:** The value of the `ename` column is used to produce an RDF literal.

By default, all RDF triples are in the default graph of the output dataset. A triples map can contain graph maps that place some or all of the triples into named graphs instead.

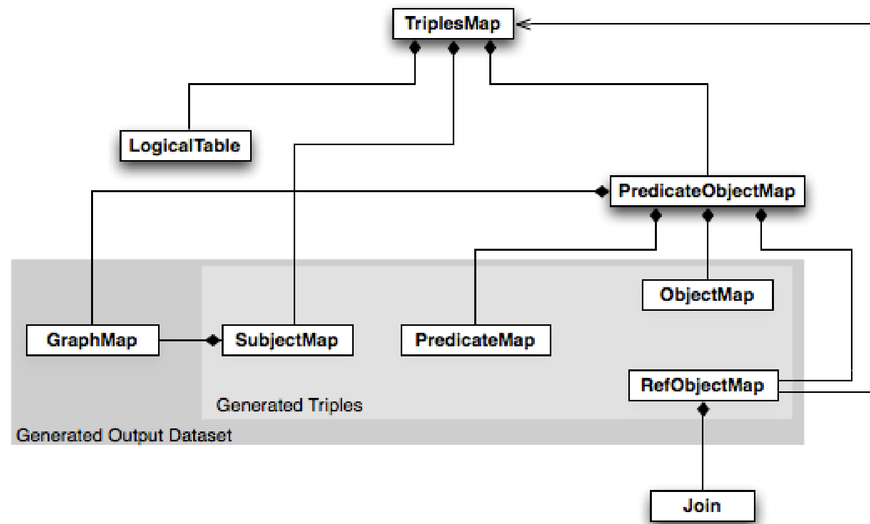

Figure 1: An overview of R2RML

2.1 Example Input Database

The following example database consists of two tables, `EMP` and `DEPT`, with one row each:

EMP

EMPNO INTEGER PRIMARY KEY	**ENAME** VARCHAR(100)	**JOB** VARCHAR(20)	**DEPTNO** INTEGER REFERENCES DEPT (DEPTNO)
7369	SMITH	CLERK	10

DEPT

DEPTNO INTEGER PRIMARY KEY	**DNAME** VARCHAR(30)	**LOC** VARCHAR(100)
10	APPSERVER	NEW YORK

2.2 Desired RDF Output

The desired RDF triples to be produced from this database are as follows:

Example output data

```
<http://data.example.com/employee/7369> rdf:type ex:Employee.
<http://data.example.com/employee/7369> ex:name "SMITH".
<http://data.example.com/employee/7369> ex:department <http://data.example.com/department/10>.

<http://data.example.com/department/10> rdf:type ex:Department.
<http://data.example.com/department/10> ex:name "APPSERVER".
<http://data.example.com/department/10> ex:location "NEW YORK".
<http://data.example.com/department/10> ex:staff 1.
```

Note in particular:

- Construction of custom IRI identifiers for departments and employees;
- use of a custom target vocabulary (`ex:Employee`, `ex:location` etc.);
- the `ex:staff` property has the total number of staff of a department; this value is not stored directly in the database but has to be computed.
- the `ex:department` property relates an employee to their department, using the identifiers of both entities;

2.3 Example: Mapping a Simple Table

The following partial R2RML mapping document will produce the desired triples from the `EMP` table (except the `ex:department` triple, which will be added later):

Example R2RML mapping

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix ex: <http://example.com/ns#>.

<#TriplesMap1>
    rr:logicalTable [ rr:tableName "EMP" ];
    rr:subjectMap [
        rr:template "http://data.example.com/employee/{EMPNO}";
        rr:class ex:Employee;
    ];
    rr:predicateObjectMap [
        rr:predicate ex:name;
        rr:objectMap [ rr:column "ENAME" ];
    ].
```

Example output data

```
<http://data.example.com/employee/7369> rdf:type ex:Employee.
<http://data.example.com/employee/7369> ex:name "SMITH".
```

2.4 Example: Computing a Property with an R2RML View

Next, the `DEPT` table needs to be mapped. Instead of using the table directly as the basis for that mapping, an "R2RML view" will be defined based on a SQL query. This allows computation of the staff number. (Alternatively, one could define this view directly in the database.)

Example SQL query

```
<#DeptTableView> rr:sqlQuery """
SELECT DEPTNO,
       DNAME,
       LOC,
       (SELECT COUNT(*) FROM EMP WHERE EMP.DEPTNO=DEPT.DEPTNO) AS STAFF
FROM DEPT;
""".
```

The definition of a triples map that generates the desired `DEPT` triples based on this R2RML view follows.

Example R2RML mapping

```
<#TriplesMap2>
    rr:logicalTable <#DeptTableView>;
    rr:subjectMap [
        rr:template "http://data.example.com/department/{DEPTNO}";
        rr:class ex:Department;
    ];
    rr:predicateObjectMap [
        rr:predicate ex:name;
        rr:objectMap [ rr:column "DNAME" ];
    ];
    rr:predicateObjectMap [
        rr:predicate ex:location;
        rr:objectMap [ rr:column "LOC" ];
    ];
    rr:predicateObjectMap [
        rr:predicate ex:staff;
        rr:objectMap [ rr:column "STAFF" ];
    ].
```

Example output data

```
<http://data.example.com/department/10> rdf:type ex:Department.
<http://data.example.com/department/10> ex:name "APPSERVER".
<http://data.example.com/department/10> ex:location "NEW YORK".
<http://data.example.com/department/10> ex:staff 1.
```

2.5 Example: Linking Two Tables

To complete the mapping document, the `ex:department` triples need to be generated. Their subjects come from the first triples map (`<#TriplesMap1>`), the objects come from the second triples map (`<#TriplesMap2>`).

This can be achieved by adding another `rr:predicateObjectMap` to `<#TriplesMap1>`. This one uses the other triples map, `<#TriplesMap2>`, as a parent triples map:

Example R2RML mapping

```
<#TriplesMap1>
    rr:predicateObjectMap [
        rr:predicate ex:department;
        rr:objectMap [
            rr:parentTriplesMap <#TriplesMap2>;
            rr:joinCondition [
                rr:child "DEPTNO";
                rr:parent "DEPTNO";
            ];
        ];
    ].
```

This performs a join between the `EMP` table and the R2RML view, on the `DEPTNO` columns. The objects will be generated from the subject map of the parent triples map, yielding the desired triple:

Example output data

```
<http://data.example.com/employee/7369> ex:department <http://data.example.com/department/10>.
```

This completes the R2RML mapping document. An R2RML processor will generate the triples listed above from this mapping document.

2.6 Example: Many-to-Many Tables

The following example will assume that a many-to-many relationship exists between the extended versions of `EMP` table and the `DEPT` table shown below. This many-to-many relationship is captured by the content of the `EMP2DEPT` table. The database consisting of the `EMP`, `DEPT`, and `EMP2DEPT` tables are shown below:

EMP

EMPNO INTEGER PRIMARY KEY	**ENAME** VARCHAR(100)	**JOB** VARCHAR(20)
7369	SMITH	CLERK
7369	SMITH	NIGHTGUARD
7400	JONES	ENGINEER

DEPT

DEPTNO INTEGER PRIMARY KEY	**DNAME** VARCHAR(30)	**LOC** VARCHAR(100)
10	APPSERVER	NEW YORK
20	RESEARCH	BOSTON

EMP2DEPT
PRIMARY KEY (EMPNO, DEPTNO)

EMPNO INTEGER REFERENCES EMP (EMPNO)	**DEPTNO** INTEGER REFERENCES DEPT (DEPTNO)
7369	10
7369	20
7400	10

Example output data

```
<http://data.example.com/employee=7369/department=10>
    ex:employee   <http://data.example.com/employee/7369> ;
    ex:department <http://data.example.com/department/10> .

<http://data.example.com/employee=7369/department=20>
    ex:employee <http://data.example.com/employee/7369> ;
    ex:department <http://data.example.com/department/20> .

<http://data.example.com/employee=7400/department=10>
    ex:employee <http://data.example.com/employee/7400> ;
    ex:department <http://data.example.com/department/10> .
```

The following R2RML mapping will produce the desired triples listed above:

Example R2RML mapping

```
<#TriplesMap3>
    rr:logicalTable [ rr:tableName "EMP2DEPT" ];
    rr:subjectMap [ rr:template "http://data.example.com/employee={EMPNO}/department={DEPTNO}" ];
    rr:predicateObjectMap [
        rr:predicate ex:employee;
        rr:objectMap [ rr:template "http://data.example.com/employee/{EMPNO}" ];
    ];
    rr:predicateObjectMap [
        rr:predicate ex:department;
        rr:objectMap [ rr:template "http://data.example.com/department/{DEPTNO}" ];
    ].
```

However, if one does *not* require that the subjects in the desired output uniquely identify the rows in the `EMP2DEPT` table, the desired output may look as follows:

Example output data

```
<http://data.example.com/employee/7369>
    ex:department <http://data.example.com/department/10> ;
    ex:department <http://data.example.com/department/20> .

<http://data.example.com/employee/7400>
    ex:department <http://data.example.com/department/10>.
```

The following R2RML mapping will produce the desired triples:

Example R2RML mapping

```
<#TriplesMap3>
    rr:logicalTable [ rr:tableName "EMP2DEPT" ];
    rr:subjectMap [
        rr:template "http://data.example.com/employee/{EMPNO}";
    ];
    rr:predicateObjectMap [
      rr:predicate ex:department;
      rr:objectMap [ rr:template "http://data.example.com/department/{DEPTNO}" ];
    ].
```

2.7 Example: Translating database type codes to IRIs

Sometimes, database columns contain codes that need to be translated into IRIs, but a direct syntactic translation using string templates is not possible. For example, consider a `JOB` column in the `EMP` table with the following possible values, and IRIs corresponding to those database values in the RDF output:

Value	Corresponding RDF IRI
`CLERK`	`http://data.example.com/roles/general-office`
`NIGHTGUARD`	`http://data.example.com/roles/security`
`ENGINEER`	`http://data.example.com/roles/engineering`

The IRIs are not found in the original database and therefore the mapping from database codes to IRIs has to be specified in the R2RML mapping. Such translations can be achieved using an “R2RML view”. The view is defined based on a SQL query that computes the IRI based on the database value. SQL's `CASE` statement is convenient for this purpose. (Alternatively, one could define this view directly in the database.)

Example R2RML mapping

```
<#TriplesMap1>
    rr:logicalTable [ rr:sqlQuery """

        SELECT EMP.*, (CASE JOB
            WHEN 'CLERK' THEN 'general-office'
```

```
            WHEN 'NIGHTGUARD' THEN 'security'
            WHEN 'ENGINEER' THEN 'engineering'
        END) ROLE FROM EMP

        """ ];
    rr:subjectMap [
        rr:template "http://data.example.com/employee/{EMPNO}";
    ];
    rr:predicateObjectMap [
        rr:predicate ex:role;
        rr:objectMap [ rr:template "http://data.example.com/roles/{ROLE}" ];
    ].
```

With the example input database, this mapping would yield the following triple:

Example output data

```
<http://data.example.com/employee/7369> ex:role <http://data.example.com/roles/general-office>.
```

3 Conformance

As well as sections marked as non-normative in the section heading, all diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words MUST, MUST NOT, REQUIRED, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this specification are to be interpreted as described in RFC 2119 [RFC2119].

This specification describes conformance criteria for:

- R2RML mapping documents
- R2RML mapping graphs
- R2RML processors
- R2RML data validators
- R2RML default mapping generators

A collection of test cases for R2RML processors and R2RML data validators is available in the *R2RML and Direct Mapping Test Cases* [TC].

This specification defines R2RML for databases that conform to *Core SQL 2008*, as defined in *ISO/IEC 9075-1:2008* [SQL1] and *ISO/IEC 9075-2:2008* [SQL2]. Processors and mappings may have to deviate from the R2RML specification in order to support databases that do not conform to this version of SQL.

Where SQL queries are embedded into R2RML mappings, SQL version identifiers can be used to indicate the specific version of SQL that is being used.

4 R2RML Processors and Mapping Documents

An ***R2RML mapping*** defines a mapping from a relational database to RDF. It is a structure that consists of one or more triples maps.

The input to an R2RML mapping is called the ***input database***.

An ***R2RML processor*** is a system that, given an R2RML mapping and an input database, provides access to the output dataset.

There are no constraints on the method of access to the output dataset provided by a conforming R2RML processor. An R2RML processor MAY materialize the output dataset into a file, or offer virtual access through an interface that queries the input database, or offer any other means of providing access to the output dataset.

An R2RML processor also has access to an execution environment consisting of:

- A ***SQL connection*** to the input database,
- a ***base IRI*** used in resolving relative IRIs produced by the R2RML mapping.

The SQL connection is used by the R2RML processor to evaluate SQL queries against the input database. It MUST be established with sufficient privileges for read access to all base tables and views that are referenced in the R2RML mapping. It MUST be configured with a ***default catalog*** and ***default schema*** that will be used when tables and views are accessed without an explicit catalog or schema reference.

How the SQL connection is established, or how users are authenticated against the database, is outside of the scope of this document.

The base IRI MUST be a valid IRI. It SHOULD NOT contain question mark ("`?`") or hash ("`#`") characters and SHOULD end in a slash ("`/`") character.

Note

To obtain an absolute IRI from a relative IRI, the term generation rules of R2RML use simple string concatenation, rather than the more complex algorithm for resolution of relative URIs defined in Section 5.2 of [RFC3986]. This ensures that the original database value can be reconstructed from the generated absolute IRI. Both algorithms are equivalent if all of the following are true:

1. The base IRI does not contain question marks or hashes,
2. the base IRI ends in a slash,
3. the relative IRI does not start with a slash, and
4. the relative IRI does not contain any "`.`" or "`..`" path segments.

An ***R2RML data validator*** is a system that takes as its input an R2RML mapping, a base IRI, and a SQL connection to an input database, and checks for the presence of data errors. When checking the input database, a data validator MUST report any data errors that are raised in the process of generating the output dataset.

An R2RML processor MAY include an R2RML data validator, but this is not required.

4.1 Mapping Graphs and the R2RML Vocabulary

An R2RML mapping is represented as an RDF graph. In other words, RDF is used not just as the target data model of the mapping, but also as a formalism for representing the R2RML mapping itself.

An RDF graph that represents an R2RML mapping is called an ***R2RML mapping graph***.

The ***R2RML vocabulary*** is the set of IRIs defined in this specification that start with the `rr:` namespace IRI:

`http://www.w3.org/ns/r2rml#`

An R2RML mapping graph:

- SHOULD NOT include any IRIs that start with the `rr:` namespace IRI, but are not defined in the R2RML vocabulary.
- SHOULD NOT include IRIs from the R2RML vocabulary where such use is not explicitly allowed or required by a clause in this specification.
- SHOULD contain only mapping components that are referenced by some triples map (in other words, all mapping components should actually be "used" in the mapping).
- MAY contain arbitrary additional triples whose terms are not from the R2RML vocabulary. In particular, a valid mapping graph MAY contain documentation in the form of `rdfs:label`, `rdfs:comment` and similar properties.
- MAY assign IRIs or blank node identifiers to any mapping component in order to enable reuse of mapping components within the mapping graph. For example, an IRI that represents a subject map may be used as the subject map of multiple triples maps; and may even be used as an object map of another triples map if it has the right properties.

The R2RML vocabulary also includes the following ***R2RML classes***:

- `rr:TriplesMap` is the class of triples maps.
- `rr:LogicalTable` is the class of logical tables. It has two subclasses:
  - `rr:R2RMLView` is the class of R2RML views.
  - `rr:BaseTableOrView` is the class of SQL base tables or views.
- `rr:TermMap` is the class of term maps. It has four subclasses:
  - `rr:SubjectMap` is the class of subject maps.
  - `rr:PredicateMap` is the class of predicate maps.

  - `rr:ObjectMap` is the class of object maps.
  - `rr:GraphMap` is the class of graph maps.
- `rr:PredicateObjectMap` is the class of predicate-object maps.
- `rr:RefObjectMap` is the class of referencing object maps.
- `rr:Join` is the class of join conditions.

The members of these classes are collectively called ***mapping components***.

> **Note**
>
> Many of these classes differ only in capitalization from properties in the R2RML vocabulary.

Explicit typing of the resources in a mapping graph with R2RML classes is OPTIONAL and has no effect on the behaviour of an R2RML processor. The mapping component represented by any given resource in a mapping graph is defined by the presence or absence of certain properties, as defined throughout this specification. A resource SHOULD NOT be typed as an R2RML class if it does not meet the definition of that class.

4.2 RDF-based Turtle Syntax; Media Type

An ***R2RML mapping document*** is any document written in the Turtle [TURTLE] RDF syntax that encodes an R2RML mapping graph.

The media type for R2RML mapping documents is the same as for Turtle documents in general: `text/turtle`. The content encoding of Turtle content is always UTF-8 and the `charset` parameter on the media type SHOULD always be used: `text/turtle;charset=utf-8`. The file extension `.ttl` SHOULD be used.

A conforming R2RML processor SHOULD accept R2RML mapping documents in Turtle syntax. It MAY accept R2RML mapping graphs encoded in other RDF syntaxes.

4.3 Data Errors

A ***data error*** is a condition of the data in the input database that would lead to the generation of an invalid RDF term. The following conditions give rise to data errors:

1. A term map with term type `rr:IRI` results in the generation of an invalid IRI.
2. A term map whose natural RDF datatype is overridden with a specified datatype produces an ill-typed literal (see datatype-override RDF literal).

When providing access to the output dataset, an R2RML processor MUST abort any operation that requires inspecting or returning an RDF term whose generation would give rise to a data error, and report an error to the agent invoking the operation. A conforming R2RML processor MAY, however, allow other operations that do not require inspecting or returning these RDF terms, and thus MAY provide partial access to an output dataset that contains data errors. Nevertheless, an R2RML processor SHOULD report data errors as early as possible.

The presence of data errors does not make an R2RML mapping non-conforming.

> **Note**
>
> Data errors cannot generally be detected by analyzing the table schema of the database, but only by scanning the data in the tables. For large and rapidly changing databases, this can be impractical. Therefore, R2RML processors are allowed to answer queries that do not “touch” a data error, and the behavior of such operations is well-defined. For the same reason, the conformance of R2RML mappings is defined without regard for the presence of data errors.
>
> R2RML data validators can be used to explicitly scan a database for data errors.

4.4 Default Mappings

An R2RML processor MAY include an ***R2RML default mapping generator***. This is a facility that introspects the schema of the input database and generates an R2RML mapping, possibly in the form of an R2RML mapping document, intended for further customization by a mapping author. Such a mapping is known as a ***default***

mapping.

The default mapping SHOULD be such that its output is the Direct Graph [DM] corresponding to the input database.

Duplicate row preservation: For tables without a primary key, the Direct Graph requires that a fresh blank node is created for each row. This ensures that duplicate rows in such tables are preserved. This requirement is relaxed for R2RML default mappings: They MAY reuse the same blank node for multiple duplicate rows. This behaviour does not preserve duplicate rows. R2RML default mapping generators that provide default mappings based on the Direct Graph MUST document whether the generated default mapping *preserves duplicate rows* or not.

5 Defining Logical Tables

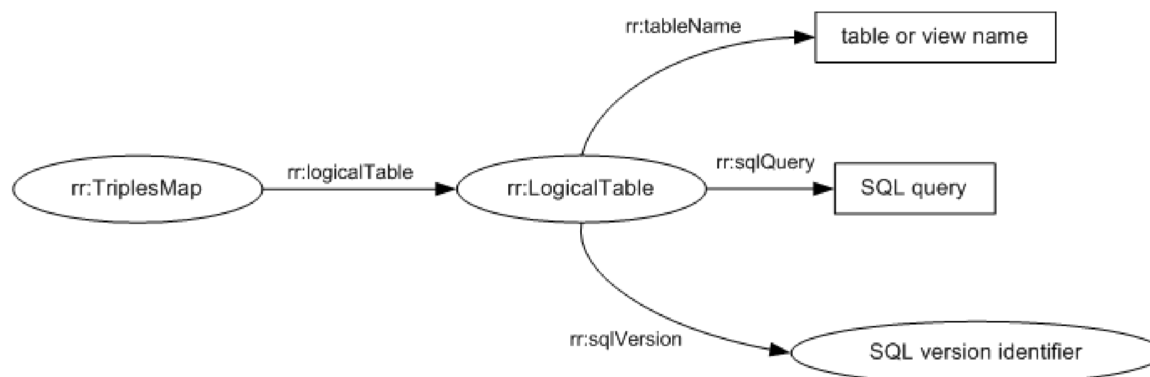

Figure 2: The properties of logical tables

A ***logical table*** is a tabular SQL query result that is to be mapped to RDF triples. A logical table is either

- a SQL base table or view, or
- an R2RML view.

Every logical table has an ***effective SQL query*** that, if executed over the SQL connection, produces as its result the contents of the logical table.

A ***logical table row*** is a row in a logical table.

A ***column name*** is the name of a column of a logical table. A column name MUST be a valid SQL identifier. Column names do not include any qualifying table, view or schema names.

A ***SQL identifier*** is the name of a SQL object, such as a column, table, view, schema, or catalog. A SQL identifier MUST match the `<identifier>` production in [SQL2]. When comparing identifiers for equality, the comparison rules of [SQL2] MUST be used.

Note

An informative summary of SQL identifier syntax rules:

1. SQL identifiers can be delimited identifiers (with double quotes), or regular identifiers.
2. Regular identifiers must start with a Unicode character from any of the following character classes: upper-case letter, lower-case letter, title-case letter, modifier letter, other letter, or letter number. Subsequent characters may be any of these, or a nonspacing mark, spacing combining mark, decimal number, connector punctuation, and formatting code.
3. Regular identifiers are case-insensitive.
4. Delimited identifiers can contain any character.
5. A double-quote character inside a delimited identifier is escaped by appending a second double-quote character.
6. Delimited identifiers are case-sensitive.
7. `deptno` and `"deptno"` are not equivalent (delimited identifiers that are not all-upper-case are not equivalent to any undelimited identifiers).
8. `DEPTNO` and `"DEPTNO"` are equivalent (all-upper-case delimited and undelimited identifiers are equivalent).
9. Five examples of valid column names: `deptno`, `dept_no`, `"dept_no"`, `"Department Number"`,

```
"Identifier ""with quotes""".
```

Note that Turtle string syntax requires escaping of double quotes with a backslash, so the identifiers from the list above might be written like this if occurring inside an R2RML mapping document:

```
[] rr:column "deptno".
[] rr:column "dept_no".
[] rr:column "\"dept_no\"".
[] rr:column "\"Department Number\"".
[] rr:column "\"Identifier \"\"with quotes\"\"\"".
```

These rules are for *Core SQL 2008*. See Section 3, *Conformance* regarding databases that do not conform to this version of SQL.

5.1 Base Tables and SQL Views (`rr:tableName`)

A ***SQL base table or view*** is a logical table containing SQL data from a base table or view in the input database. A SQL base table or view is represented by a resource that has exactly one `rr:tableName` property.

The value of `rr:tableName` specifies the ***table or view name*** of the base table or view. Its value MUST be a valid schema-qualified name that names an existing base table or view in the input database.

A ***schema-qualified name*** is a sequence of one, two or three valid SQL identifiers, separated by the dot character (“.”). The three identifiers name, respectively, a catalog, a schema, and a table or view. If no catalog or schema is specified, then the default catalog and default schema of the SQL connection are assumed.

The effective SQL query of a SQL base table or view is:

```
SELECT * FROM {table}
```

with *{table}* replaced with the table or view name.

The following example shows a logical table specified using a schema-qualified table name.

Example R2RML mapping

```
[] rr:tableName "SCOTT.DEPT".
```

The following example shows a logical table specified using an unqualified table name. The SQL connection's default schema will be used.

Example R2RML mapping

```
[] rr:tableName "DEPT".
```

5.2 R2RML Views (`rr:sqlQuery`, `rr:sqlVersion`)

An ***R2RML view*** is a logical table whose contents are the result of executing a SQL query against the input database. It is represented by a resource that has exactly one `rr:sqlQuery` property, whose value is a literal with a lexical form that is a valid SQL query.

Note

R2RML mappings sometimes require data transformation, computation, or filtering before generating triples from the database. This can be achieved by defining a SQL view in the input database and referring to it with `rr:tableName`. However, this approach may sometimes not be practical for lack of database privileges or other reasons. R2RML views achieve the same effect without requiring changes to the input database.

Note

Note that unlike “real” SQL views, an R2RML view can not be used as an input table in further SQL queries.

A ***SQL query*** is a `SELECT` query in the SQL language that can be executed over the input database. The string MUST conform to the production `<direct select statement: multiple rows>` in [SQL2] with an OPTIONAL trailing

semicolon character and OPTIONAL surrounding white space (excluding comments) as defined in [TURTLE]. It MUST be valid to execute over the SQL connection. The result of the query execution MUST NOT have duplicate column names. Any columns in the `SELECT` list derived by projecting an expression SHOULD be named, because otherwise they cannot be reliably referenced in the rest of the mapping.

Database objects referenced in the SQL query MAY be qualified with a catalog or schema name. For any database objects referenced without an explicit catalog name or schema name, the default catalog and default schema of the SQL connection are assumed.

For example, the following `SELECT` query is **not a valid** R2RML SQL query because the result contains a duplicate column name `DEPTNO`:

Example SQL query

```
SELECT EMP.DEPTNO, 1 AS DEPTNO FROM EMP;
```

As a further example, the following `SELECT` query SHOULD NOT be used, because it contains an unnamed column derived from a `COUNT` expression:

Example SQL query

```
SELECT DEPTNO, COUNT(EMPNO) FROM EMP GROUP BY DEPTNO;
```

An R2RML view MAY have one or more ***SQL version identifiers***. They MUST be valid IRIs and are represented as values of the `rr:sqlVersion` property. The following SQL version identifier indicates that the SQL query conforms to Core SQL 2008:

```
http://www.w3.org/ns/r2rml#SQL2008
```

The absence of a SQL version identifier indicates that no claim to Core SQL 2008 conformance is made.

Note

No further identifiers besides `rr:SQL2008` are defined in this specification. The RDB2RDF Working Group intends to maintain a non-normative list of identifiers for other SQL versions [SQLIRIS].

The effective SQL query of an R2RML view is the value of its `rr:sqlQuery` property.

The following example shows a logical table specified as an R2RML view conforming to Core SQL 2008.

Example R2RML mapping

```
[] rr:sqlQuery """
        Select ('Department' || DEPTNO) AS DEPTID
             , DEPTNO
             , DNAME
             , LOC
          from SCOTT.DEPT
    """;
    rr:sqlVersion rr:SQL2008.
```

6 Mapping Logical Tables to RDF with Triples Maps

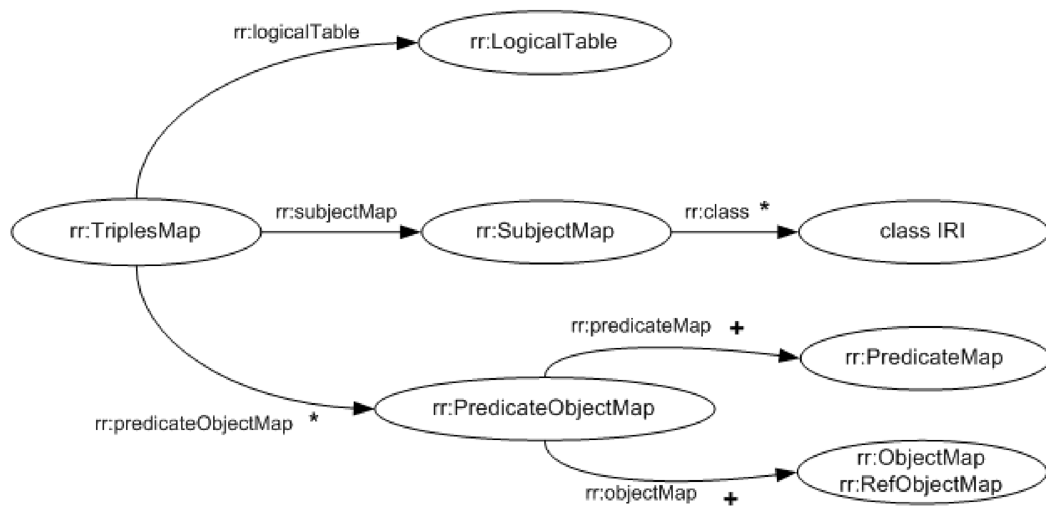

Figure 3: The properties of triples maps

A ***triples map*** specifies a rule for translating each row of a logical table to zero or more RDF triples.

The RDF triples generated from one row in the logical table all share the same subject.

A triples map is represented by a resource that references the following other resources:

- It MUST have exactly one `rr:logicalTable` property. Its value is a logical table that specifies a SQL query result to be mapped to triples.
- It MUST have exactly one subject map that specifies how to generate a subject for each row of the logical table. It may be specified in two ways:
  1. using the `rr:subjectMap` property, whose value MUST be the subject map, or
  2. using the constant shortcut property `rr:subject`.
- It MAY have zero or more `rr:predicateObjectMap` properties, whose values MUST be predicate-object maps. They specify pairs of predicate maps and object maps that, together with the subjects generated by the subject map, may form one or more RDF triples for each row.

The referenced columns of all term maps of a triples map (subject map, predicate maps, object maps, graph maps) MUST be column names that exist in the term map's logical table.

The following example shows a triples map including its logical table, subject map, and two predicate-object maps.

Example R2RML mapping

```
[]
    rr:logicalTable [ rr:tableName "DEPT" ];
    rr:subjectMap [ rr:template "http://data.example.com/department/{DEPTNO}" ];
    rr:predicateObjectMap [
        rr:predicate ex:name;
        rr:objectMap [ rr:column "DNAME" ];
    ];
    rr:predicateObjectMap [
        rr:predicate ex:location;
        rr:objectMap [ rr:column "LOC" ];
    ].
```

6.1 Creating Resources with Subject Maps

A ***subject map*** is a term map. It specifies a rule for generating the subjects of the RDF triples generated by a triples map.

6.2 Typing Resources (`rr:class`)

A subject map MAY have one or more ***class IRIs***. They are represented by the `rr:class` property. The values of the `rr:class` property MUST be IRIs. For each RDF term generated by the subject map, RDF triples with predicate `rdf:type` and the class IRI as object will be generated.

Note

Mappings where the class IRI is not constant, but needs to be computed based on the contents of the database, can be achieved by defining a predicate-object map with predicate `rdf:type` and a non-constant object map.

In the following example, the generated subject will be asserted as an instance of the `ex:Employee` class.

Example R2RML mapping

```
[] rr:template "http://data.example.com/employee/{EMPNO}";
   rr:class ex:Employee.
```

Using the example `EMP` table, the following RDF triple will be generated:

Example output data

```
<http://data.example.com/emp/7369> rdf:type ex:Employee.
```

6.3 Creating Properties and Values with Predicate-Object Maps

A ***predicate-object map*** is a function that creates one or more predicate-object pairs for each logical table row of a logical table. It is used in conjunction with a subject map to generate RDF triples in a triples map.

A predicate-object map is represented by a resource that references the following other resources:

- One or more predicate maps. Each of them may be specified in one of two ways:
  1. using the `rr:predicateMap` property, whose value MUST be a predicate map, or
  2. using the constant shortcut property `rr:predicate`.
- One or more object maps or referencing object maps. Each of them may be specified in one of two ways:
  1. using the `rr:objectMap` property, whose value MUST be either an object map, or a referencing object map.
  2. using the constant shortcut property `rr:object`.

A ***predicate map*** is a term map.

An ***object map*** is a term map.

7 Creating RDF Terms with Term Maps

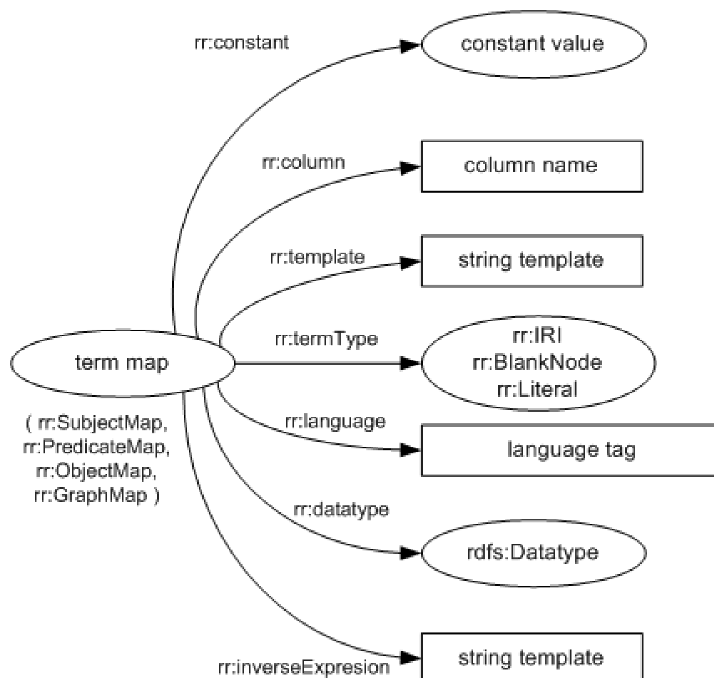

Figure 4: The properties of term maps

An ***RDF term*** is either an IRI, or a blank node, or a literal.

A ***term map*** is a function that generates an RDF term from a logical table row. The result of that function is known as the term map's generated RDF term.

Term maps are used to generate the subjects, predicates and objects of the RDF triples that are generated by a triples map. Consequently, there are several kinds of term maps, depending on where in the mapping they occur: subject maps, predicate maps, object maps and graph maps.

A term map MUST be exactly one of the following:

- a constant-valued term map,
- a column-valued term map,
- a template-valued term map.

The ***referenced columns*** of a term map are the set of column names referenced in the term map and depend on the type of term map.

7.1 Constant RDF Terms (`rr:constant`)

A ***constant-valued term map*** is a term map that ignores the logical table row and always generates the same RDF term. A constant-valued term map is represented by a resource that has exactly one `rr:constant` property.

The ***constant value*** of a constant-valued term map is the RDF term that is the value of its `rr:constant` property.

If the constant-valued term map is a subject map, predicate map or graph map, then its constant value MUST be an IRI.

If the constant-valued term map is an object map, then its constant value MUST be an IRI or literal.

The referenced columns of a constant-valued term map is the empty set.

Constant-valued term maps can be expressed more concisely using the ***constant shortcut properties*** `rr:subject`, `rr:predicate`, `rr:object` and `rr:graph`. Occurrences of these properties MUST be treated exactly as if the following triples were present in the mapping graph instead:

Triple involving constant shortcut property	Replacement triples
`?x rr:subject ?y.`	`?x rr:subjectMap [ rr:constant ?y ].`
`?x rr:predicate ?y.`	`?x rr:predicateMap [ rr:constant ?y ].`

?x rr:object ?y.	?x rr:objectMap [rr:constant ?y].
?x rr:graph ?y.	?x rr:graphMap [rr:constant ?y].

The following example shows a predicate-object map that uses a constant-valued term map both for its predicate and for its object.

Example R2RML mapping

```
[] rr:predicateMap [ rr:constant rdf:type ];
   rr:objectMap [ rr:constant ex:Employee ].
```

If added to a triples map, this predicate-object map would add the following triple to all resources `?x` generated by the triples map:

Example output data

```
?x rdf:type ex:Employee.
```

The following example uses constant shortcut properties and is equivalent to the example above:

Example R2RML mapping

```
[] rr:predicate rdf:type;
   rr:object ex:Employee.
```

7.2 From a Column (`rr:column`)

A ***column-valued term map*** is a term map that is represented by a resource that has exactly one `rr:column` property.

The value of the `rr:column` property MUST be a valid column name. The ***column value*** of the term map is the data value of that column in a given logical table row.

The referenced columns of a column-valued term map is the singleton set containing the value of the term map's `rr:column` property.

The following example defines an object map that generates literals from the `DNAME` column of some logical table.

Example R2RML mapping

```
[] rr:objectMap [ rr:column "DNAME" ].
```

Using the sample row from the `DEPT` table as a logical table row, the column value of the object map would be "`APPSERVER`".

7.3 From a Template (`rr:template`)

A ***template-valued term map*** is a term map that is represented by a resource that has exactly one `rr:template` property. The value of the `rr:template` property MUST be a valid string template.

A ***string template*** is a format string that can be used to build strings from multiple components. It can reference column names by enclosing them in curly braces ("`{`" and "`}`"). The following syntax rules apply to valid string templates:

- Pairs of unescaped curly braces MUST enclose valid column names.
- Curly braces that do not enclose column names MUST be escaped by a backslash character ("`\`"). This also applies to curly braces within column names.
- Backslash characters ("`\`") MUST be escaped by preceding them with another backslash character, yielding "`\\`". This also applies to backslashes within column names.
- There SHOULD be at least one pair of unescaped curly braces.
- If a template contains multiple pairs of unescaped curly braces, then any pair SHOULD be separated from the next one by a ***safe separator***. This is any character or string that does not occur anywhere in any of the data values of either referenced column; or in the IRI-safe versions of the data values, if the term type is `rr:IRI` (see note below).

The ***template value*** of the term map for a given logical table row is determined as follows:

1. Let `result` be the template string
2. For each pair of unescaped curly braces in `result`:
   1. Let `value` be the data value of the column whose name is enclosed in the curly braces
   2. If `value` is `NULL`, then return `NULL`
   3. Let `value` be the natural RDF lexical form corresponding to `value`
   4. If the term type is `rr:IRI`, then replace the pair of curly braces with an IRI-safe version of `value`; otherwise, replace the pair of curly braces with `value`
3. Return `result`

The ***IRI-safe version*** of a string is obtained by applying the following transformation to any character that is not in the `iunreserved` production in [RFC3987]:

1. Convert the character to a sequence of one or more octets using UTF-8 [RFC3629]
2. Percent-encode each octet [RFC3986]

The following table shows examples of strings and their IRI-safe versions:

String	IRI-safe version
42	42
Hello World!	Hello%20World%21
2011-08-23T22:17:00Z	2011-08-23T22%3A17%3A00Z
~A_17.1-2	~A_17.1-2
葉篤正	葉篤正

Note

R2RML always performs percent-encoding when IRIs are generated from string templates. If IRIs need to be generated without percent-encoding, then `rr:column` should be used instead of `rr:template`, with an R2RML view that performs the string concatenation.

Note

In the case of string templates that generate IRIs, any single character that is legal in an IRI, but percent-encoded in the IRI-safe version of a data value, is a safe separator. This includes in particular the eleven `sub-delim` characters defined in [RFC3987]: `!$&'()*+,;=`

The referenced columns of a template-valued term map is the set of column names enclosed in unescaped curly braces in the template string.

The following example defines a subject map that generates IRIs from the `DEPTNO` column of a logical table.

Example R2RML mapping

```
[] rr:subjectMap [ rr:template "http://data.example.com/department/{DEPTNO}" ].
```

Using the sample row from the `DEPT` table as a logical table row, the template value of the subject map would be:

Example output data

```
http://data.example.com/department/10
```

The following example shows how an IRI-safe template value is created:

Example R2RML mapping

```
[] rr:subjectMap [ rr:template "http://data.example.com/site/{LOC}" ].
```

Using the sample row from the `DEPT` table as a logical table row, the template value of the subject map would be:

Example output data

```
http://data.example.com/site/NEW%20YORK
```

The space character is not in the `iunreserved` set, and therefore percent-encoding is applied to the character, yielding “`%20`”.

The following example shows the use of backslash escapes in string templates. The template will generate a fancy title such as

```
{{{ \o/ Hello World! \o/ }}}
```

from a string “`Hello World!`” in the `TITLE` column. By default, `rr:template` generates IRIs. Since the intention here is to create a literal instead, the term type has to be set.

Example R2RML mapping

```
[] rr:objectMap [
    rr:template "\\{\\{\\{ \\\\o/ {TITLE} \\\\o/ \\}\\}\\}";
    rr:termType rr:Literal;
].
```

Note that because backslashes need to be escaped by a second backslash in the Turtle syntax [TURTLE], a double backslash is needed to escape each curly brace, and to get one literal backslash in the output one needs to write four backslashes in the template.

7.4 IRIs, Literal, Blank Nodes (`rr:termType`)

The ***term type*** of a column-valued term map or template-valued term map determines the kind of generated RDF term (IRIs, blank nodes or literals).

If the term map has an optional `rr:termType` property, then its term type is the value of that property. The value MUST be an IRI and MUST be one of the following options:

- If the term map is a subject map: `rr:IRI` or `rr:BlankNode`
- If the term map is a predicate map: `rr:IRI`
- If the term map is an object map: `rr:IRI`, `rr:BlankNode`, or `rr:Literal`
- If the term map is a graph map: `rr:IRI`

If the term map does not have a `rr:termType` property, then its term type is:

- `rr:Literal`, if it is an object map and at least one of the following conditions is true:
  - It is a column-based term map.
  - It has a `rr:language` property (and thus a specified language tag).
  - It has a `rr:datatype` property (and thus a specified datatype).
- `rr:IRI`, otherwise.

Note

Term maps with term type `rr:IRI` cause data errors if the value is not a valid IRI (see generated RDF term for details). Data values from the input database may require percent-encoding before they can be used in IRIs. Template-valued term maps are a convenient way of percent-encoding data values.

Note

Constant-valued term maps are not considered as having a term type, and specifying `rr:termType` on these term maps has no effect. The type of the generated RDF term is determined directly by the value of `rr:constant`: If it is an IRI, then an IRI will be generated; if it is a literal, a literal will be generated.

7.5 Language Tags (`rr:language`)

A term map with a term type of `rr:Literal` MAY have a ***specified language tag***. It is represented by the `rr:language` property on a term map. If present, its value MUST be a valid language tag.

A specified language tag causes generated literals to be language-tagged plain literals. In the following example, plain literals with language tag “`en-us`” (U.S. English) will be generated for the data values in the `DNAME` column.

Example R2RML mapping

```
[] rr:objectMap [ rr:column "DNAME"; rr:language "en-us" ].
```

7.6 Typed Literals (`rr:datatype`)

A ***datatypeable term map*** is a term map with a term type of `rr:Literal` that does not have a specified language tag.

Datatypeable term maps may generate typed literals. The datatype of these literals can be automatically determined based on the SQL datatype of the underlying logical table column (producing a natural RDF literal), or it can be explicitly overridden using `rr:datatype` (producing a datatype-override RDF literal).

A datatypeable term map MAY have a `rr:datatype` property. Its value MUST be an IRI. This IRI is the ***specified datatype*** of the term map.

A term map MUST NOT have more than one `rr:datatype` value.

A term map that is not a datatypeable term map MUST NOT have an `rr:datatype` property.

The ***implicit SQL datatype*** of a datatypeable term map is `CHARACTER VARYING` if the term map is a template-valued term map; otherwise, it is the SQL datatype of the respective column in the logical table row.

See generated RDF term for further details on generating literals from term maps.

Note

One cannot explicitly state that a plain literal without language tag should be generated. They are the default for string columns. To generate one from a non-string column, a template-valued term map with a template such as `"{MY_COLUMN}"` and a term type of `rr:Literal` can be used.

The following example shows an object map that overrides the default datatype of the logical table with an explicitly specified `xsd:positiveInteger` type. A datatype-override RDF literal of that datatype will be generated from whatever is in the `EMPNO` column.

Example R2RML mapping

```
[] rr:objectMap [ rr:column "EMPNO"; rr:datatype xsd:positiveInteger ].
```

7.7 Inverse Expressions (`rr:inverseExpression`)

An ***inverse expression*** is a string template associated with a column-valued term map or template-value term map. It is represented by the value of the `rr:inverseExpression` property. This property is OPTIONAL and there MUST NOT be more than one for a term map.

Inverse expressions are useful for optimizing term maps that reference derived columns in R2RML views. An inverse expression specifies an expression that allows “reversing” of a generated RDF term and the construction of a SQL query that efficiently retrieves the logical table row from which the term was generated. In particular, it allows the use of indexes on the underlying relational tables.

Every pair of unescaped curly braces in the inverse expression is a ***column reference in an inverse expression***. The string between the braces MUST be a valid column name.

An inverse expression MUST satisfy the following condition:

- Let *t* be the logical table associated with this term map
- Every column reference in the inverse expression MUST be an existing column in *t*
- Given a logical table row *r* in *t*, let *instantiation(r)* be the result of replacing each column reference *c* in the inverse expression with:
  - the quoted and escaped data value of column *c* in *r*, if *c* is a referenced column in the term map
  - the column name of column *c*, otherwise
- Given a logical table row *r* in *t*, let *same-term(r)* be the set of logical table rows in *t* that are the result of executing the following SQL query over the SQL connection:

  ```
  SELECT * FROM ({query}) AS tmp WHERE {expr}
  ```

  where *`{query}`* is the effective SQL query of *t*, and *`{expr}`* is *instantiation(r)*

- For every logical table row *r* in *t* whose generated RDF term *g* is not `NULL`, *same-term(r)* MUST be exactly the set of logical table rows in *t* whose generated RDF term is also *g*.

For example, for the `DEPTID` column in the logical table used for mapping the `DEPT` table in this example mapping, an inverse expression could be defined as follows:

Example R2RML mapping

```
[] rr:column "DEPTID";
   rr:inverseExpression "{DEPTNO} = SUBSTRING({DEPTID}, CHARACTER_LENGTH('Department')+1)";
```

This facilitates the use of an existing index on the `DEPTNO` column of the DEPT table.

A ***quoted and escaped data value*** is any SQL string that matches the `<literal>` or `<null specification>` productions of [SQL2]. This string can be used in a SQL query to specify a SQL data value. Examples:

- `27`
- `'foo'`
- `'foo''bar'`
- `TRUE`
- `DATE '2011-11-11'`
- `NULL`

8 Foreign Key Relationships among Logical Tables (`rr:parentTriplesMap`, `rr:joinCondition`, `rr:child` and `rr:parent`)

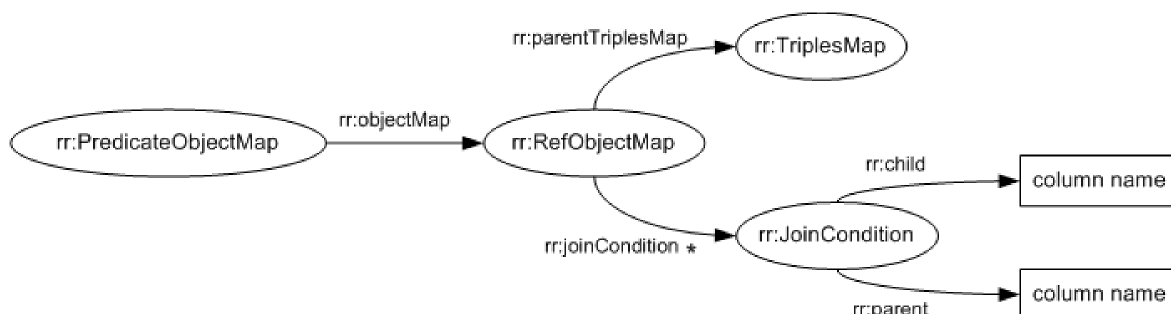

Figure 5: The properties of referencing object maps

A ***referencing object map*** allows using the subjects of another triples map as the objects generated by a predicate-object map. Since both triples maps may be based on different logical tables, this may require a join between the logical tables. This is not restricted to 1:1 joins.

A referencing object map is represented by a resource that:

- has exactly one `rr:parentTriplesMap` property, whose value MUST be a triples map, known as the referencing object map's ***parent triples map***.
- MAY have one or more `rr:joinCondition` properties, whose values MUST be join conditions.

A ***join condition*** is represented by a resource that has exactly one value for each of the following two properties:

- `rr:child`, whose value is known as the join condition's ***child column*** and MUST be a column name that exists in the logical table of the triples map that contains the referencing object map
- `rr:parent`, whose value is known as the join condition's ***parent column*** and MUST be a column name that exists in the logical table of the referencing object map's parent triples map.

The ***child query*** of a referencing object map is the effective SQL query of the logical table of the term map containing the referencing object map.

The ***parent query*** of a referencing object map is the effective SQL query of the logical table of its parent triples map.

If the child query and parent query of a referencing object map are not identical, then the referencing object map MUST have at least one join condition.

The ***joint SQL query*** of a referencing object map is:

- If the referencing object map has no join condition:

```
SELECT * FROM ({child-query}) AS tmp
```

- If the referencing object map has at least one join condition:

```
SELECT * FROM ({child-query}) AS child,
              ({parent-query}) AS parent
        WHERE child.{child-column1}=parent.{parent-column1}
          AND child.{child-column2}=parent.{parent-column2}
          AND ...
```

  where `{child-query}` is the referencing object map's child query, `{parent-query}` is its parent query, `{child-column1}` and `{parent-column1}` are the child column and parent column of its first join condition, and so on. The order of the join conditions is chosen arbitrarily.

The joint SQL query is used when generating RDF triples from referencing object maps.

The following example shows a referencing object map as part of a predicate-object map:

Example R2RML mapping

```
[] rr:predicateObjectMap [
    rr:predicate ex:department;
    rr:objectMap [
        rr:parentTriplesMap <#TriplesMap2>;
        rr:joinCondition [
            rr:child "DEPTNO";
            rr:parent "DEPTNO";
        ];
    ];
].
```

If the logical table of the surrounding triples map is `EMP`, and the logical table of `<#TriplesMap2>` is `DEPT`, this would result in a join between these two tables with the condition

```
EMP.DEPTNO = DEPT.DEPTNO
```

and the objects of the triples would be generated using the subject map of `<#TriplesMap2>`.

Given the two example tables, and subject maps as defined in the example mapping, this would result in a triple:

Example output data

```
<http://data.example.com/employee/7369> ex:department <http://data.example.com/department/10>.
```

The following example shows a referencing object map that does not have a join condition. It creates two kinds of resources from the `DEPT` table: departments and sites.

Example R2RML mapping

```
<#DeptTriplesMap>
    rr:logicalTable [ rr:tableName "DEPT" ];
    rr:subjectMap [
        rr:template "department/{DEPTNO}";
        rr:class ex:Department;
    ];
    rr:predicateObjectMap [
        rr:predicate ex:location;
        rr:objectMap [ rr:parentTriplesMap <#SiteTriplesMap> ];
    ].

<#SiteTriplesMap>
    rr:logicalTable [ rr:tableName "DEPT" ];
    rr:subjectMap [
        rr:template "site/{LOC}";
        rr:class ex:Site;
    ];
    rr:predicateObjectMap [
        rr:predicate ex:siteName;
        rr:objectMap [ ex:column "LOC" ];
    ].
```

An `ex:Site` resource is created for each distinct value in the `LOC` column, using the `<#SiteTriplesMap>`. Departments and sites are linked by `ex:location` triples, and the objects of these triples are specified using a referencing object map that references the sites triples map. No join condition is needed as both triples maps use the same logical table (the base table `DEPT`). Given the example table, this mapping would result in four triples (assuming an appropriate base IRI):

Example output data

```
<http://data.example.com/department/10> rdf:type ex:Department.
<http://data.example.com/department/10> ex:location <http://data.example.com/site/NEW%20YORK>.
<http://data.example.com/site/NEW%20YORK> rdf:type ex:Site.
<http://data.example.com/site/NEW%20YORK> ex:siteName "NEW YORK".
```

9 Assigning Triples to Named Graphs

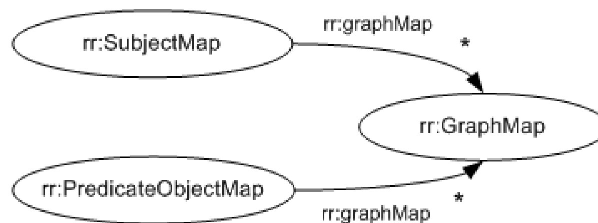

Figure 6: The properties of graph maps

Each triple generated from an R2RML mapping is placed into one or more graphs of the output dataset. Possible target graphs are the unnamed default graph, and the IRI-named named graphs.

Any subject map or predicate-object map MAY have one or more associated ***graph maps***. They are specified in one of two ways:

1. using the `rr:graphMap` property, whose value MUST be a graph map,
2. using the constant shortcut property `rr:graph`.

Graph maps are themselves term maps. When RDF triples are generated, the set of target graphs is determined by taking into account any graph maps associated with the subject map or predicate-object map.

If a graph map generates the special IRI `rr:defaultGraph`, then the target graph is the default graph of the output dataset.

In the following subject map example, all generated RDF triples will be stored in the named graph `ex:DepartmentGraph`.

Example R2RML mapping

```
[] rr:subjectMap [
    rr:template "http://data.example.com/department/{DEPTNO}";
    rr:graphMap [ rr:constant ex:DepartmentGraph ];
].
```

This is equivalent to the following example, which uses a constant shortcut property:

Example R2RML mapping

```
[] rr:subjectMap [
    rr:template "http://data.example.com/department/{DEPTNO}";
    rr:graph ex:DepartmentGraph;
].
```

In the following example, RDF triples are placed into named graphs according to the job title of employees:

Example R2RML mapping

```
[] rr:subjectMap [
    rr:template "http://data.example.com/employee/{EMPNO}";
    rr:graphMap [ rr:template "http://data.example.com/jobgraph/{JOB}" ];
].
```

The triples generated from the `EMP` table would be placed in the named graph with the following IRI:

Example output data

```
<http://data.example.com/jobgraph/CLERK>
```

9.1 Scope of Blank Nodes

Blank nodes in the output dataset are scoped to a single RDF graph. If the same blank node identifier occurs in multiple RDF triples that are in the same graph, then the triples will share the same blank node. If, however, the same blank node identifier occurs in multiple graphs, then a distinct blank node be created for each graph. An R2RML-generated blank node can never be shared by two triples in two different graphs.

This implies that triples generated from a single logical table row will have different subjects if the subjects are blank nodes and the triples are placed into different graphs.

10 Datatype Conversions

This section defines mappings from SQL data values to RDF literals.

10.1 Introduction (Informative)

This section defines the following mappings from SQL data values:

1. The **natural RDF literal** is a mapping to literals. It is used in R2RML and in the *Direct Mapping of Relational Data to RDF* [DM] as the default mapping when literals are created. It maps SQL datatypes to corresponding XML Schema datatypes [XMLSCHEMA2] and loosely follows *ISO/IEC 9075-14:2008* [SQL14].
2. The **natural RDF lexical form** is similar, but produces only the lexical form of the typed literal and *recommends* that implementations perform XSD canonicalization. It is used in R2RML when non-string columns are used in a string context, for example when a `TIMESTAMP` is used in an IRI template.
3. The **canonical RDF lexical form** is again similar, but *requires* XSD canonicalization. It is used in the Direct Mapping when IRIs are generated.
4. The **datatype-override RDF literal** is a mapping that constructs typed literals by using the natural RDF lexical form and applying a specified datatype IRI. The mapping author is responsible for ensuring that the generated lexical form is valid for the datatype. It is used in R2RML when the target datatype of a literal-generating term map is overridden using `rr:datatype`.

The mappings cover all predefined Core SQL 2008 datatypes except `INTERVAL`. The natural mappings may be extended with custom handling for other types, such as vendor-specific SQL datatypes. In the absence of such extensions, the natural mappings fall back on a simple cast to string for all unsupported SQL datatypes.

The mappings are referenced in the R2RML term generation rules.

An informative summary of XSD lexical forms is provided to aid implementers.

10.2 Natural Mapping of SQL Values

The ***natural RDF literal*** corresponding to a SQL data value is the result of applying the following steps:

1. Let *dt* be the SQL datatype of the SQL data value.
2. If *dt* is a character string type (in Core SQL 2008: `CHARACTER`, `CHARACTER VARYING`, `CHARACTER LARGE OBJECT`, `NATIONAL CHARACTER`, `NATIONAL CHARACTER VARYING`, `NATIONAL CHARACTER LARGE OBJECT`), then the result is a plain literal without language tag whose lexical form is the SQL data value.
3. Otherwise, if *dt* is listed in the table below: The result is a typed literal whose datatype IRI is the IRI indicated in the *RDF datatype* column in the same row as *dt*. The lexical form may be any lexical form that represents the same value as the SQL data value, according to the definition of the RDF datatype. If there are multiple lexical forms available that represent the same value (e.g., `1`, `+1`, `1.0` and `1.0E0`), then the choice is implementation-dependent. However, the choice MUST be made so that given a target RDF datatype and value, the same lexical form is chosen consistently (e.g., `INTEGER 5` and `BIGINT 5` must be mapped to the same lexical form, as both are mapped to the RDF datatype `xsd:integer` and are equal

values; mapping one to `5` and the other to `+5` would be an error). The canonical lexical representation [XMLSCHEMA2] MAY be chosen. (See also: Summary of XSD Lexical Forms)
4. Otherwise, the result is a plain literal without language tag whose lexical form is the SQL data value cast to string.

SQL datatype	RDF datatype	Lexical transformation (informative)
`BINARY`, `BINARY VARYING`, `BINARY LARGE OBJECT`	`xsd:hexBinary`	`xsd:hexBinary` lexical mapping
`NUMERIC`, `DECIMAL`	`xsd:decimal`	none required
`SMALLINT`, `INTEGER`, `BIGINT`	`xsd:integer`	none required
`FLOAT`, `REAL`, `DOUBLE PRECISION`	`xsd:double`	none required
`BOOLEAN`	`xsd:boolean`	ensure lowercase (`true`, `false`)
`DATE`	`xsd:date`	none required
`TIME`	`xsd:time`	none required
`TIMESTAMP`	`xsd:dateTime`	replace space character with "`T`"
`INTERVAL`	undefined	undefined

Note

R2RML extensions that handle vendor-specific or user-defined datatypes beyond those of Core SQL 2008 are expected to behave as if the table above contained additional rows that associate the SQL datatypes with appropriate RDF-compatible datatypes (e.g., the XML Schema built-in types [XMLSCHEMA2]), and appropriate lexical transformations where required. Note however that future versions of R2RML may also normatively add additional rows to this table.

Note

The translation of `INTERVAL` is left undefined due to the complexity of the translation. [SQL14] describes a translation of `INTERVAL` to `xdt:yearMonthDuration` and `xdt:dayTimeDuration`.

Note

In [SQL2], the precision of many SQL datatypes is not fixed, but left implementation-defined. Therefore, the mapping to XML Schema datatypes must rely on arbitrary-precision types such as `xsd:decimal`, `xsd:integer` and `xsd:dateTime`. Implementers of the mapping may wish to set upper limits for the supported precision of these XSD types. The XML Schema specification allows such *partial implementations of infinite datatypes* [XMLSCHEMA2], and defines specific minimum requirements.

The ***natural RDF datatype*** corresponding to a SQL datatype is the value of the *RDF datatype* column in the row corresponding to the SQL datatype in the table above.

The ***natural RDF lexical form*** corresponding to a SQL data value is the lexical form of its corresponding natural RDF literal, with the additional constraint that the canonical lexical representation [XMLSCHEMA2] SHOULD be chosen.

The ***canonical RDF lexical form*** corresponding to a SQL data value is the lexical form of its corresponding natural RDF literal, with the additional constraint that the canonical lexical representation [XMLSCHEMA2] MUST be chosen.

Cast to string is an implementation-dependent function that maps SQL data values to equivalent Unicode strings. It is undefined for the following kinds of SQL datatypes: collection types, row types, user-defined types without a user-defined string `CAST`, reference types whose referenced type does not have a user-defined string `CAST`, binary types.

Note

Cast to string is a fallback that handles vendor-specific and user-defined datatypes not supported by the R2RML processor. It can be implemented in a number of ways, including explicit SQL casts ("`CAST(`*`value`*` AS VARCHAR(`*`n`*`))`", where *n* is an arbitrary large integer), implicit SQL casts (concatenation with the empty string), or by employing a database access API that presents return values as strings.

10.3 Datatype-override Mapping of SQL Values

The ***datatype-override RDF literal*** corresponding to a SQL data value *v* and a datatype IRI *dt*, is a typed literal whose lexical form is the natural RDF lexical form corresponding to *v*, and whose datatype IRI is *dt*. If the typed literal is ill-typed, then a data error is raised.

A typed literal is ***ill-typed*** in R2RML if its datatype IRI denotes a validatable RDF datatype and its lexical form is not in the lexical space of the RDF datatype identified by its datatype IRI. (See also: Summary of XSD Lexical Forms)

The set of ***validatable RDF datatypes*** includes all datatypes in the *RDF datatype* column of the table of natural datatype mappings, as defined in [XMLSCHEMA2]. This set MAY include implementation-defined additional RDF datatypes.

For example, `"X"^^xsd:boolean` is ill-typed because `xsd:boolean` is a validatable RDF datatype in R2RML, and "`X`" is not in the lexical space of `xsd:boolean` [XMLSCHEMA2].

10.4 Non-String Columns in String Contexts

The same non-character-string SQL data value can typically be represented in multiple different string forms. For example, the `DOUBLE` value 1 can be represented as `1`, `+1`, `1.0` and `1.0E0`. This can cause interoperability issues when such values are used in string contexts, for example when using them to generate IRIs. Two IRIs that are character-for-character equivalent, except one contains `1` where the other contains `1.0`, will not "link up" in an RDF graph – they are two different nodes.

To reduce portability issues arising from such conversions, this specification recommends that implementations convert non-string data values to a canonical form (see natural RDF lexical form). However, this is not a strict requirement. Therefore, when portability between R2RML implementations is a concern, mapping authors SHOULD NOT use non-character-string columns in contexts where strings are produced:

- with `rr:column` when IRIs or blank nodes are produced,
- with `rr:column` when `rr:language` or an `rr:datatype` other than the natural RDF datatype is used, and
- with `rr:template`.

In these contexts, if portability is to be maximized, then mapping authors SHOULD use an R2RML view instead and explicitly convert the non-string column to a string column using an SQL expression.

Note that this is not a problem when natural RDF literals are generated from such columns, because the resulting literal has a corresponding non-string XSD datatype, and equivalences between different lexical forms within these datatype are well-defined.

10.5 Summary of XSD Lexical Forms (Informative)

The natural mappings make reference to various XSD datatypes and require that SQL data values be converted to strings that are appropriate as lexical forms for these datatypes. This subsection gives examples of these lexical forms in order to aid implementers of the mappings. This subsection is non-normative; the normative definitions of the lexical spaces as well as the canonical lexical mappings are found in *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes* [XMLSCHEMA2].

A general approach that may be used for implementing the natural mappings is as follows:

1. Identify the SQL datatype of the input SQL data value.
2. Look up its corresponding natural RDF datatype.
3. Apply cast to string to the SQL data value.
4. Ensure that the resulting string is in the lexical space of the target RDF datatype; that is, it must be in a form such as those listed in either column of the table below. This may require some transformations of the string, in particular for `xsd:hexBinary`, `xsd:dateTime` and `xsd:boolean`.
5. If the goal is to obtain a canonical lexical representation, then further string transformations may be required to obtain a form such as those listed in the *Canonical lexical forms* column of the table below.

RDF datatype	Non-canonical lexical forms	Canonical lexical forms	Comments
			Convert from SQL by applying

xsd:hexBinary	5232524d4c	5232524D4C	xsd:hexBinary lexical mapping.
xsd:decimal	.224	0.224	
	+001	1	
	42.0	42	
	-5.9000	-5.9	
xsd:integer	-05	-5	
	+333	333	
	00	0	
xsd:double	-5.90	-5.9E0	Also supports INF, -INF, NaN and -0.0E0, but these do not appear in standard SQL.
	+0.00014770215000	1.4770215E-4	
	+01E+3	1.0E3	
	100.0	1.0E2	
	0	0.0E0	
xsd:boolean	1	true	Must be lowercase.
	0	false	
xsd:date		2011-08-23	Dates in SQL don't have timezone offsets. They are optional in XSD.
xsd:time	22:17:34.885+00:00	22:17:34.885Z	May or may not have timezone offset.
	22:17:34.000	22:17:34	
	22:17:34.1+01:00	22:17:34.1+01:00	
xsd:dateTime	2011-08-23T22:17:00.000+00:00	2011-08-23T22:17:00Z	May or may not have timezone offset. Convert from SQL by replacing space wiht “T”.

11 The Output Dataset

The ***output dataset*** of an R2RML mapping is an RDF dataset that contains the generated RDF triples for each of the triples maps of the R2RML mapping. The output dataset MUST NOT contain any other RDF triples or named graphs besides these. However, R2RML processors MAY provide access to datasets that contain additional triples or graphs beyond those in the output dataset, such as inferred triples or provenance information.

If a table or column is not explicitly referenced in a triples map, then no RDF triples will be generated for that table or column.

Conforming R2RML processors MAY rename blank nodes when providing access to the output dataset. This means that client applications may see actual blank node identifiers that differ from those produced by the R2RML mapping. Client applications SHOULD NOT rely on the specific text of the blank node identifier for any purpose.

Note

RDF syntaxes and RDF APIs generally represent blank nodes with blank node identifiers. But the characters allowed in blank node identifiers differ between syntaxes, and not all characters occurring in the values produced by a term map may be allowed, so a bijective mapping function from values to valid blank node identifiers may be required. The details of this mapping function are implementation-dependent, and R2RML processors may have to use different functions for different output syntaxes or access interfaces. Strings matching the regular expression `[a-zA-Z_][a-zA-Z_0-9-]*` are valid blank node identifiers in all W3C-recommended RDF syntaxes (as of this document's publication).

Note

RDF datasets may contain empty named graphs. R2RML cannot generate such output datasets.

11.1 The Generated RDF Triples of a Triples Map

This subsection describes the process of ***generating RDF triples*** from a triples map. This process adds RDF triples to the output dataset. Each generated triple is placed into one or more graphs of the output dataset.

The generated RDF triples are determined by the following algorithm. R2RML processors MAY use other means than implementing this algorithm to compute the generated RDF triples, as long as the result is the same.

1. Let *sm* be the subject map of the triples map
2. Let *rows* be the result of evaluating the effective SQL query of the triples map's logical table using the SQL connection
3. Let *classes* be the class IRIs of *sm*
4. Let *sgm* be the set of graph maps of *sm*
5. For each logical table row *row* in *rows*, apply the following steps:
    1. Let *subject* be the generated RDF term that results from applying *sm* to *row*
    2. Let *subject_graphs* be the set of the generated RDF terms that result from applying each term map in *sgm* to *row*
    3. For each *class* in *classes*, add triples to the output dataset as follows:

        Subject: *subject*
        Predicate: `rdf:type`
        Object: *class*
        Target graphs: If *sgm* is empty: `rr:defaultgraph`; otherwise: *subject_graphs*

    4. For each predicate-object map of the triples map, apply the following steps:
        1. Let *predicates* be the set of generated RDF terms that result from applying each of the predicate-object map's predicate maps to *row*
        2. Let *objects* be the set of generated RDF terms that result from applying each of the predicate-object map's object maps (but not referencing object maps) to *row*
        3. Let *pogm* be the set of graph maps of the predicate-object map
        4. Let *predicate-object_graphs* be the set of generated RDF terms that result from applying each graph map in *pogm* to *row*
        5. For each possible combination <*predicate*, *object*> where *predicate* is a member of *predicates* and *object* is a member of *objects*, add triples to the output dataset as follows:

            Subject: *subject*
            Predicate: *predicate*
            Object: *object*
            Target graphs: If *sgm* and *pogm* are empty: `rr:defaultGraph`; otherwise: union of *subject_graphs* and *predicate-object_graphs*

6. For each referencing object map of a predicate-object map of the triples map, apply the following steps:
    1. Let *psm* be the subject map of the parent triples map of the referencing object map
    2. Let *pogm* be the set of graph maps of the predicate-object map
    3. Let *n* be the number of columns in the logical table of the triples map
    4. Let *rows* be the result of evaluating the joint SQL query of the referencing object map
    5. For each *row* in *rows*, apply the following steps:
        1. Let *child_row* be the logical table row derived by taking the first *n* columns of *row*
        2. Let *parent_row* be the logical table row derived by taking all but the first *n* columns of *row*
        3. Let *subject* be the generated RDF term that results from applying *sm* to *child_row*
        4. Let *predicates* be the set of generated RDF terms that result from applying each of the predicate-object map's predicate maps to *child_row*
        5. Let *object* be the generated RDF term that results from applying *psm* to *parent_row*
        6. Let *subject_graphs* be the set of generated RDF terms that result from applying each graph map of *sgm* to *child_row*
        7. Let *predicate-object_graphs* be the set of generated RDF terms that result from applying each graph map in *pogm* to *child_row*
        8. For each *predicate* in *predicates*, add triples to the output dataset as follows:

            Subject: *subject*
            Predicate: *predicate*
            Object: *object*
            Target graphs: If neither *sgm* nor *pogm* has any graph maps: `rr:defaultGraph`; otherwise: union of *subject_graphs* and *predicate-object_graphs*

"***Add triples to the output dataset***" is a process that takes the following inputs:

- **Subject**, an IRI or blank node or *empty*
- **Predicate**, an IRI or *empty*
- **Object**, an RDF term or *empty*
- **Target graphs**, a set of zero or more IRIs

Execute the following steps:

1. If *Subject*, *Predicate* or *Object* is *empty*, then abort these steps.
2. Otherwise, generate an RDF triple <*Subject*, *Predicate*, *Object*>
3. If the set of target graphs includes `rr:defaultGraph`, add the triple to the default graph of the output dataset.
4. For each IRI in the set of target graphs that is not equal to `rr:defaultGraph`, add the triple to a named graph of that name in the output dataset. If the output dataset does not contain a named graph with that IRI, create it first.

RDF graphs cannot contain duplicate RDF triples. Placing multiple equal triples into the same graph has the same effect as placing it into the graph only once. Also note the scope of blank nodes.

11.2 The Generated RDF Term of a Term Map

A term map is a function that generates an RDF term from a logical table row. The result of that function can be:

- *Empty* – if any of the referenced columns of the term map has a `NULL` value,
- An RDF term – the common case,
- A data error.

The ***generated RDF term*** of a term map for a given logical table row is determined as follows:

- If the term map is a constant-valued term map, then the generated RDF term is the term map's constant value.
- If the term map is a column-valued term map, then the generated RDF term is determined by applying the term generation rules to its column value.
- If the term map is a template-valued term map, then the generated RDF term is determined by applying the term generation rules to its template value.

The ***term generation rules***, applied to a *value*, are as follows:

1. If *value* is `NULL`, then no RDF term is generated.
2. Otherwise, if the term map's term type is `rr:IRI`:
   1. Let *value* be the natural RDF lexical form corresponding to *value*.
   2. If *value* is a valid absolute IRI [RFC3987], then return an IRI generated from *value*.
   3. Otherwise, prepend *value* with the base IRI. If the result is a valid absolute IRI [RFC3987], then return an IRI generated from the result.
   4. Otherwise, raise a data error.
3. Otherwise, if the term type is `rr:BlankNode`:
   1. Return a blank node that is unique to the natural RDF lexical form corresponding to *value*. (Note: On Blank Node Identifiers, Scope of Blank Nodes)
4. Otherwise, if the term type is `rr:Literal`:
   1. If the term map has a specified language tag, then return a plain literal with that language tag and with the natural RDF lexical form corresponding to *value*.
   2. Otherwise, if the term map has a non-empty specified datatype that is different from the natural RDF datatype corresponding to the term map's implicit SQL datatype, then return the datatype-override RDF literal corresponding to *value* and the specified datatype.
   3. Otherwise, return the natural RDF literal corresponding to *value*.

A. RDF Terminology (Informative)

This appendix lists some terms normatively defined in other specifications.

The following terms are defined in *RDF Concepts and Abstract Syntax* [RDF] and used in R2RML:

- ***RDF graph***
- ***RDF triple***

- ***IRI*** (corresponds to the Concepts and Abstract Syntax term *RDF URI reference*)
- ***literal***
- ***plain literal***
- ***typed literal***
- ***language tag***
- ***lexical form***
- ***datatype IRI*** (corresponds to the Concepts and Abstract Syntax term *datatype URI*)
- ***lexical space***
- ***blank node***
- ***blank node identifier***

The following terms are defined in *SPARQL Query Language for RDF* [SPARQL] and used in R2RML:

- ***RDF dataset***
- ***default graph***
- ***named graph***

B. Index of R2RML Vocabulary Terms (Informative)

This appendix lists all the classes, properties and other terms defined by this specification within the R2RML vocabulary.

An RDFS representation of the vocabulary is available from the namespace IRI.

B.1 Classes

The following table lists all R2RML classes.

The third column contains minimum conditions that a resource has to fulfil in order to be considered member of the class. Where multiple conditions are listed, all must be fulfilled.

Class	Represents	Minimum conditions
`rr:BaseTableOrView`	SQL base table or view	Having an `rr:tableName` property
`rr:GraphMap`	graph map	Being an `rr:TermMap` Being value of an `rr:graphMap` property
`rr:Join`	join condition	Having an `rr:parent` property Having an `rr:child` property
`rr:LogicalTable`	logical table	Being one of its subclasses, `rr:BaseTableOrView` or `rr:R2RMLView`
`rr:ObjectMap`	object map	Being an `rr:TermMap` Being value of an `rr:objectMap` property
`rr:PredicateMap`	predicate map	Being an `rr:TermMap` Being value of an `rr:predicateMap` property
`rr:PredicateObjectMap`	predicate-object map	Having at least one of `rr:predicate` and `rr:predicateMap` Having at least one of `rr:object` and `rr:objectMap`
`rr:R2RMLView`	R2RML view	Having an `rr:sqlQuery` property
`rr:RefObjectMap`	referencing object map	Having an `rr:parentTriplesMap` property
`rr:SubjectMap`	subject map	Being an `rr:TermMap` Being value of an `rr:subjectMap` property
`rr:TermMap`	term map	Having exactly one of `rr:constant`, `rr:column`, `rr:template`
`rr:TriplesMap`	triples map	Having an `rr:logicalTable` property Having exactly one of `rr:subject` and `rr:subjectMap`

As noted earlier, a single node in an R2RML mapping graph may represent multiple mapping components and thus be typed as several of these classes. However, the following classes are disjoint:

- `rr:TermMap` and `rr:RefObjectMap`

- `rr:BaseTableOrView` and `rr:SQLQuery`
- constant-valued, column-valued and template valued term maps (indicated by the presence of the properties `rr:constant`, `rr:column` and `rr:template`, respectively)

B.2 Properties

The following table lists all properties in the R2RML vocabulary.

The cardinality column indicates how often this property occurs within its context. Note that additional constraints not stated in this table might apply. The actual cardinality of some properties may depend on the presence or absence of other properties, and their values. Properties where this applies are indicated by an exclamation mark.

Property	Represents	Context	Cardinality	
`rr:child`	child column	join condition	1	
`rr:class`	class IRI	subject map	0…∞	
`rr:column`	column name	column-valued term map	1	
`rr:datatype`	specified datatype	term map	0…1	!
`rr:constant`	constant value	constant-valued term map	1	
`rr:graph`	constant shortcut property	subject map, predicate-object map	0…∞	
`rr:graphMap`	graph map			
`rr:inverseExpression`	inverse expression	term map	0…1	!
`rr:joinCondition`	join condition	referencing object map	0…∞	
`rr:language`	specified language tag	term map	0…1	!
`rr:logicalTable`	logical table	triples map	1	
`rr:object`	constant shortcut property	predicate-object map	1…∞	
`rr:objectMap`	object map, referencing object map			
`rr:parent`	parent column	join condition	1	
`rr:parentTriplesMap`	parent triples map	referencing object map	1	
`rr:predicate`	constant shortcut property	predicate-object map	1…∞	
`rr:predicateMap`	predicate map			
`rr:predicateObjectMap`	predicate-object map	triples map	0…∞	
`rr:sqlQuery`	SQL query	R2RML view	1	
`rr:sqlVersion`	SQL version identifier	R2RML view	0…∞	
`rr:subject`	constant shortcut property	triples map	0…1	
`rr:subjectMap`	subject map			
`rr:tableName`	table or view name	SQL base table or view	1	
`rr:template`	string template	template-valued term map	1	
`rr:termType`	term type	term map	0…1	!

B.3 Other Terms

Term	Denotes	Used with property
`rr:defaultGraph`	default graph	`rr:graph`
`rr:SQL2008`	Core SQL 2008	`rr:sqlVersion`
`rr:IRI`	IRI	`rr:termType`
`rr:BlankNode`	blank node	`rr:termType`
`rr:Literal`	literal	`rr:termType`

C. References

C.1 Normative References

[DM]
A Direct Mapping of Relational Data to RDF, Alexandre Bertails, Marcelo Arenas, Eric Prud'hommeaux, Juan Sequeda, Editors. World Wide Web Consortium, 27 September 2012. This version is http://www.w3.org/TR/2012/REC-rdb-direct-mapping-20120927/. The latest version is http://www.w3.org/TR/rdb-direct-mapping/.

[RDF]
Resource Description Framework (RDF): Concepts and Abstract Syntax, Graham Klyne, Jermey J. Carroll, Editors. World Wide Web Consortium, 10 February 2004. This version is http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. The latest version is http://www.w3.org/TR/rdf-concepts/.

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, March 1997. Internet RFC 2119, http://tools.ietf.org/html/rfc2119.

[RFC3629]
UTF-8, a transformation format of ISO 10646, F. Yergeau. November 2003. Internet RFC 3629, http://tools.ietf.org/html/rfc3629.

[RFC3986]
Uniform Resource Identifier (URI): Generic Syntax, T. Berners-Lee, R. Fielding, L. Masinter. January 2005. Internet RFC 3986, http://tools.ietf.org/html/rfc3986.

[RFC3987]
Internationalized Resource Identifiers (IRIs), M. Duerst, M. Suignard. January 2005. Internet RFC 3987, http://tools.ietf.org/html/rfc3987.

[SPARQL]
SPARQL Query Language for RDF, Eric Prud'hommeaux, Andy Seaborne, Editors. World Wide Web Consortium, 15 January 2008. This version is http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. The latest version is http://www.w3.org/TR/rdf-sparql-query/.

[SQL1]
ISO/IEC 9075-1:2008 SQL - Part 1: Framework (SQL/Framework). International Organization for Standardization, 27 January 2009.

[SQL2]
ISO/IEC 9075-2:2008 SQL - Part 2: Foundation (SQL/Foundation). International Organization for Standardization, 27 January 2009.

[TURTLE]
Turtle - Terse RDF Triple Language, Eric Prud'hommeaux, Gavin Carothers. World Wide Web Consortium, 10 July 2012. This version is http://www.w3.org/TR/2012/WD-turtle-20120710/. The latest version is http://www.w3.org/TR/turtle/. This document is work in progress.

[XMLSCHEMA2]
W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, David Peterson, Shudi Gao, Ashok Malhotra, C. M. Sperberg-McQueen, Henry S. Thompson. World Wide Web Consortium, 5 April 2012. This version is http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/. The latest version is http://www.w3.org/TR/xmlschema11-2/.

C.2 Other References

[SQL14]
ISO/IEC 9075-14:2008 SQL - Part 14: XML-Related Specifications (SQL/XML). International Organization for Standardization, 27 January 2009.

[SQLIRIS]
SQL Version IRIs, Editors of the W3C Semantic Web Standards wiki. The latest version is http://www.w3.org/2001/sw/wiki/RDB2RDF/SQL_Version_IRIs. This is a public wiki page.

[TC]
R2RML and Direct Mapping Test Cases, Boris Villazón-Terrazas, Michael Hausenblas, Editors. World Wide Web Consortium, 14 August 2012. This version is http://www.w3.org/TR/2012/NOTE-rdb2rdf-test-cases-20120814/. The latest version is http://www.w3.org/TR/rdb2rdf-test-cases/.

[UCNR]

Use Cases and Requirements for Mapping Relational Databases to RDF, Eric Prud'hommeaux, Michael Hausenblas, Editors. World Wide Web Consortium, 8 June 2010. This version is http://www.w3.org/TR/2010/WD-rdb2rdf-ucr-20100608/. The latest version is http://www.w3.org/TR/rdb2rdf-ucr/. This document is work in progress.

D. Acknowledgements (Informative)

The Editors would like to give special thanks to the following contributors: David McNeil greatly improved the quality of the specification with detailed reviews and comments. Nuno Lopes and Eric Prud'hommeaux contributed to the design of the mapping from SQL data values to RDF literals. Eric also worked on the mechanism for SQL compatibility. Boris Villazón-Terrazas drew the diagrams throughout the text, and kept them up-to-date throughout many iterations.

In addition, the Editors gratefully acknowledge contributions from: Marcelo Arenas, Sören Auer, Samir Batla, Alexander de Leon, Orri Erling, Lee Feigenbaum, Enrico Franconi, Howard Greenblatt, Wolfgang Halb, Harry Halpin, Michael Hausenblas, Patrick Hayes, Ivan Herman, Nophadol Jekjantuk, Li Ma, Nan Ma, Ashok Malhotra, Ivan Mikhailov, Percy Enrique Rivera Salas, Juan Sequeda, Ben Szekely, Ted Thibodeau, and Edward Thomas.