

# SWAD–Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes

Project name:

W3C Semantic Web Advanced Development for Europe (SWAD-Europe)

Project Number:

IST-2001-34732

Workpackage name:

10. Tools for Semantic Web Scalability and Storage

Workpackage description:

<http://www.w3.org/2001/sw/Europe/plan/workpackages/live/esw-wp-10.html>

Deliverable title:

Semantic Web Scalability and Storage: Mapping Semantic Web Data with RDBMSes

URI:

[http://www.w3.org/2001/sw/Europe/reports/scalable\\_rdbms\\_mapping\\_report](http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report)

Authors:

Dave Beckett

Jan Grant

Abstract:

A public report on mapping triple stores and RDBMS concentrating on surveying the schemas used and discussing mapping approaches to and from relational schemas. It describes current best practice for using such systems for Semantic Web data including feature comparisons, recommendations for particular applications and advice on tradeoffs.

STATUS:

Completed report, published 2003-01-23. Updates will be made over the lifetime of the SWAD-Europe project as tools change, new tools emerge and other relevant materials are added.

Comments on this document are welcome and should be sent to the authors or to the [public-esw@w3.org](mailto:public-esw@w3.org) list. An archive of this list is available at

<http://lists.w3.org/Archives/Public/public-esw/>

## Contents

---

- 1 [Introduction](#)
  - 2 [Existing Work](#)
  - 3 [Triple Stores Implemented with Databases](#)
  - 4 [Database Schemas used for Triple Stores](#)
  - 5 [Mapping RDBMS schemas onto RDF](#)
  - 6 [Mapping Tools](#)
  - 7 [Summary and Conclusions](#)
  - A [References](#)
  - B [Changes](#)
- 

## 1 Introduction

This report is part of [SWAD-Europe Work package 10: Tools for Scalability and Storage](#) and addresses how semantic web data storage relates to using relational database systems, in particular those that are licensed as Free Software / Open Source (FS/OS). This work package builds on [SWAD-Europe Deliverable 10.1 Tools for Semantic Web Scalability and Storage: Survey of Free Software / Open Source RDF storage systems](#) that surveyed systems providing general triple storage and did not consider the detail of existing database approaches. This report convers relational and other database systems that might be appropriate to use; any form that is not a general triple store.

**1.1 Approach** - There are two main thrusts in this workpackage dealing with the main issues that arise when dealing with RDF triple stores and relational databases. These can be summarised as the following Frequently Asked Questions (FAQs):

1. I want to implement an RDF triple store. How can I do this with a relational database?
2. I have a legacy relational database. How can it be exposed as RDF?

These two questions can be taken as parts of the more general question of how to map RDF data and its schema vocabulary to and from a relational database.

The analysis of this question and how to approach it requires the details of what an RDF triple store requires and what features typical relational systems provide. The first steps are therefore to look at triple store requirements and

the features of RDBMSes that apply to performing this mapping between these data forms.

**1.2 Triple store requirements** - These may not all be required, depending on the application but good support for some of these triple store requirements may be crucial for efficient systems:

#### Text searching

Handling the literal string components of the triples as more than just a binary object (BLOB), performing standard word indexing, stemming, truncation and allowing searching over them using typical operators such as conjunction, phrase searching.

#### URIs

Efficient support for Universal Resource Indicators (URIs) is useful since these are used to identify all terms in the RDF graph and are contained in all parts of the triples. If they are not supported well, the triple store will not work well with such semantic web data.

#### Datatypes

RDF supports a datatypes mechanism compatible with XML Schema Datatypes [\[XSD\]](#) and it is expected that this will start to be more extensively used for very common datatypes such as decimal. These need to be efficiently supported when used.

#### RDF Containers (`rdf:Seq`, `rdf:Bag`, `rdf:Alt`, `rdf:_n`)

These constructs are built from multiple triples and can be hard to deal with without extra application support such as extra-RDF knowledge of the known size of the contents.

#### RDF Vocabulary Description Language support (aka RDF Schema)

When `rdf` Classes and Properties are declared with subclasses and subproperties, the transitive hierarchy of subsumption relationships benefits greatly from extra triple store support so that type and class queries can be answered quickly.

#### Ontological support, Inferencing

RDF is mostly conjunctive statements along with a description of the vocabulary used. When higher level applications that are using logics, there may be a need for extra functionality to support them or allow hooks to be added for working with rules and reasoners.

#### Triple provenance

When triples are added to a graph it is often useful to be able to track where they came from, how they were added, by who etc. This provenance information is outside the RDF graph (unless it uses reification) and can be expensive to track if added by hand. Some support for this aids supporting "web of trust" style applications and also allows RDF graph merging/demerging to work better.

### 1.3 Mapping considerations -

#### The database schema

The schema used by the (relational, for example) database contains a representation of the data model used by the application. This is likely to be somewhat optimised with appropriate indexes and use of extra tables for normalizing the information.

#### The particular database implementation

There are often restrictions on how the RDBMS implements the SQL standards, or has extensions that are being used. It may perform better with certain database schemas which interact with how the mapping should be used.

#### Non-relational databses such as ODBMS, XML

There are other types of non-relational databases such as ODBMS and XML databases available that have a different model of the information; not tabular, based on the relational model. These might be appropriate for triple stores.

#### Database tuning

The RDBMS representation could potentially be optimised for a particular RDF schema but that might change as new RDF vocabulary descriptions (classes, properties) are found when new triples are added. This may involve evolving the database schema or updating some of the optimized features such as subsumption support.

#### Database updates

Storing triples is a simple operation but if this is mapped to a more complex database model, a single database update may not match changing a single triple. It may require a set of triples to change in order to perform an update (SQL UPDATE).

#### Exposing the database schema

The databases has its own description of the tables, columns, types and constraints it holds. These might be useful to expose in some form, rather than just the minimum needed to map the stored database information as RDF triples.

## 2 Existing Work

There have been only a few surveys of existing work on using RDBMSes for semantic web data storage and approaches taken to mapping between these data forms. These include the online survey by Melnik in *Storing RDF in a relational database* [\[RDFRELATIONAL1\]](#) done in 2000, with submissions from the authors of the various applications and various smaller reports [\[RDFOCASE\]](#), [\[W3CACLSQL\]](#), [\[DBVIEW\]](#). There has been no major work on methods of extracting semantic web data from existing RDBMSes or on detailed issues of using RDBMSes for storing semantic web data.

**Jena** - The Jena [\[JENA\]](#) project team analysed various approaches to database schemas as part of the implementation of the Jena RDBMS storage backend. One goal was to make this work over the standard Java database abstraction system JDBC, and thus allow support for many RDBMSes. This involved various tradeoffs, so specific support was added for several particular RDBMSes, to provide optimised support (PostgreSQL, MySQL, Interbase and others). On top of this, several different database schemas were taken (and all of them with/without support for multiple models per single database). and the resulting systems tested for performance.

In *Jena relational database interface - performance notes* [\[JENARDBPERF\]](#) Reynolds summarises the analysis in section *Summary of observations* as follows:

- *Partitioning the statement table into attribute tables has no measurable performance benefit on small scale tests.*
- *The use of content hashes instead of database ID's has some performance cost on load and no performance benefit on these queries but does lead to mergable databases.*
- *The Berkeley DB storage manager (in non-transaction mode) is nearly an order of magnitude faster than most SQL database options. Though of those tested MySQL comes the closest. We assume transaction support is the primary overhead.*
- *For indexing the main statement table the two indices commonly used, namely subject+predicate and object, are indeed the best tradeoffs.*
- *So long as the database query plans are not pathological then allowing multiple models in a single database has no performance impact if you don't use it. Be careful on PostgreSQL though.*
- *Stored procedures can save around 25% in load times.*
- *Caching JDBC prepared statements can give a 2-3x improvement in performance for some databases.*

These results are useful for comparing different database backends for RDBMS storage of semantic web data and hint at the tradeoffs that can be made.

**RDF Suite** - In *The RDFSuite: Managing Voluminous RDF Description Bases* [\[MANVOLUME\]](#) the authors describe using the *RSSDB - RDF Schema Specific DataBase (RSSDB)* [\[RSSDB\]](#) part of ICS-FORTH *RDFSuite* [\[RDFSUITE\]](#) and comparing using the same RDBMS with a *generic representation* database schema using triples approach (with URI interning) versus a *RDF schema specific representation* database schema with indexes, where the tables are customised for the data. In all cases the schema-specific approach used less storage, was faster in loading and querying, and in some queries, very substantially quicker. This work was initially done over JDBC to PostgreSQL since it provided the support for subsumption that the project required. Later, support for MySQL was added but it isn't clear if the two approaches were checked over that database.

**Related Work** - This project has already reported in [SWAD-Europe Deliverable 10.1 Tools for Semantic Web Scalability and Storage: Survey of Free Software / Open Source RDF storage systems](#) on existing approaches to storing semantic web data in triple form, rather than specifically for RDBMSes. An evaluation of queries will be done later in this project in [SWAD-Europe Deliverable 7.2 Report comparing existing RDF query language functionality, documenting different scenarios and users for RDF query languages](#)

### 3 Triple Stores Implemented with Databases

This section outlines the features of the major semantic web data storage systems based on RDBMS or other backends. The main features considered are support for schemas (RDF, ontologies), inference, indexing and searching of literal strings and what are the dependencies or requirements such as implementation language or underlying relational databases. The next section describes the relational database schemas in use by the stores, where appropriate.

**JENA** - Jena [\[JENA\]](#) is a Java semantic web toolkit that provides a rich API including storage over using either Sleepycat / Berkeley DB or via JDBC to talk to a variety of RDBMSs including MySQL, PostgreSQL, Oracle, Interbase and others. Many of the support databases have specific optimisations driven from specialised configuration files, along with multiple forms off database schema that allow the user to pick an appropriate one for the application, such as allowing multiple models in a single RDBMS database, or choosing to use stored procedures. The RDBMS api can be easily customised for other databases, and a generic schema is available that will likely work with standard SQL. The performance of the approaches and databases has been analysed and optimised, and advice on the tradeoffs of these are documented [\[JENARDBPERF\]](#). The Jena RDBMS backend is restricted to work only via JDBC directly, but a more specific RDBMS database storage system could be written relatively easily, which could enable the optimising of, for example, query operations into more easily optimizable forms.

License: Apache/BSD-style license without advertising.

**KAON** - The *KAON project's* [\[KAON\]](#) RDF Server as described in *Karlsruhe Ontology and Semantic Web Infrastructure Developer's Guide* [\[KAONDEV\]](#) provides an RDF repository as an Enterprise Java Beans (EJB) that can be used on J2EE application servers, with the data persisted in a relational database via JDBC. The default persistence is provided by the EJB container mechanism but an enhanced "Engineering" relational schema is provided as part of the Engineering Server for use when building ontologies, with a complex indexing custom scheme over the raw content. The *KAON Server* [\[KAON-SERVER\]](#) (produced as part of IST Project 2001-33052 WonderWeb) provides a high-level onotological interface including modules accessing triple store or RQL-based

repositories and also allows pluggable inferencers to be added for higher level logics. The 2002-10-02 version works with any SQL2-compatible RDB, tested with MS SQL Server, and run on PostgreSQL, IBM DB2 and Oracle 9i.

License: LGPL

**Parka Database** - *The Parka Database* [PARKASW], [PARKADB] - part of the Parka-KB is a knowledge representation system based on semantic networks, layered on frames using a relational database beneath (presently an internal one, although it has been used over Oracle). It uses fixed table sizes for predicates for speed of operation and has similar optimised structures for handling inheritance of classes and properties, which are always stored in memory. The property tables are moved between disk and memory on demand. The system has been used with over 2M frames (not quite assertions) and although applied to the KR world, is now being updated for semantic web data (with URIs) such as RDF.

License: MIT License with advertising not required.

**RDFSuite** - ICS-FORTH's Java RDFSuite [RDFSUITE] as described in *The RDFSuite: Managing Voluminous RDF Description Bases* [MANVOLUME] describes how it was designed and implemented and the approach taken to create an efficient persistent store based on an ORDBMS model. Two approaches were taken - a generic store or a specific RDBMS schemas for the application. It creates the latter from schema knowledge to automatically generate an Object-Relational (SQL3) representation of RDF metadata. Internally it has tables for Class, Property, SubClass, SubProperty and the particular classes (instances) and properties (source, target) in the RDF schema being used. It also handles XML Schema data types for literal values, grouping and filtering primitives and sorting.

License: Under the [RDFSuite License](#) (C-Web license on the web page) which allows free (price) use of the software as long as credit is kept.

**Sesame and SAIL** - Sesame [SESAMEPROJ] as described in *Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema* [SESAMETR], [SESAME] and [BenchRDFS] was designed to use existing storage systems such as the various DBMSes available that which have different strengths and weaknesses. These can then be used via sesame in various ways to store RDF data. This variety was captured by the concentration of the DBMS-specific code into a *Storage And Inference Layer* (SAIL) which interfaces between the RDF-specific methods and the database API. There is also, an in-memory SAIL implementation which uses files for persistent storage.

The SAIL provides interfaces to sesame modules that implement the RQL query language, administration (including loading of RDF data and schema, deleting) and exporting RDF as RDF/XML. SAIL is a high-level and lightweight Java API that includes support for RDF schema semantics and data-streaming operation. At the time it was designed the existing Java APIs didn't support all of these features. SAIL layers stack so that actions are passed between layers until they are handled, for example a schema-caching SAIL was created without disturbing other layers.

SAIL can work on top of any RDBMS, ODBMS, existing RDF stores, RDF files or network services. In the current version (Sesame 0.7.1), Sesame supports using any DBMS with a JDBC-driver and currently explicitly supports PostgreSQL (7.0.2 or later), MySQL (3.23.47 or later) and Oracle9i. The PostgreSQL store uses the object-relational RDBMS features to support (transitive) subtable relations between tables, which are appropriate for providing RDF Schema class and property subsumption. The MySQL store also supports subsumption by another mechanism.

The SAIL uses a dynamic database schema inspired by that described in *Querying Community Web Portals* [OCOMWEBPOR] in which new tables are added for each new class and subclass relationships cause the table to be a subtable of the superclass tables. The same method is also used for properties.

Sesame also has an RDF model theory inferencer that can work directly on the SQL schema used, which although it use knowledge of the tables that are in use, it is independent of the actual RDBMS. A variant implementation of this inferencer also tracks which statements were used to infer other statements, called the *Truth Maintenance System (TMS)*. This is available for the PostgreSQL and MySQL backing stores only.

### Sesame SAIL Query Module

The query module turns the RQL query into a model, optimises it to another model. The evaluation is then done in the RQL query engine over calls to the SAIL. The RQL query itself could have been passed to the repository, which could then be used directly if it supported it, but this would restrict it to only repositories that supported RQL. RDFSuite took the opposite approach and requires all repositories to support RQL for querying.

### Sesame SAIL Admin Module

This provides adding/removing data, emptying a repository, adding a stream of data (such as from an RDF parser) and includes RDF schema information such as checking statements versus their schema and inferring implied information.

### Sesame SAIL Export Module

Allows serializing a repository to the RDF exchange format, RDF/XML.

### Sesame Issues

The PostgreSQL store is rather slow and in particular, schema changes since table creation is very expensive. Adding subclasses between two existing classes is very slow since it cannot be done on existing tables.

License: GPL/LGPL

**TAP** - The TAP Project [\[TAPPROJ\]](#) is a system written in C as an Apache module (*TAPache*) that implements an RDF storage, query and network/web service API and client libraries in Perl and Java. It supports RDF schema subclass and sub property queries and indexing in an efficient manner based on either a BerkeleyDB store or MySQL. The MySQL schema is optimised for the *TAP Knowledge Base* [\[TAPKB\]](#) properties hard-coded into the source, but could be customised to index additional schemas. It has a query language *GetData* [\[GETDATA\]](#) implemented as a SOAP XML interface that allows cross-server queries with RDFS subsumption across TAP servers.

License: Apache/BSD-style license with advertising.

## 4 Database Schemas used for Triple Stores

**JENA** - Several database schemas are available for Jena over multiple relational database backends. The detail of the implementation is described in *Jena relational database interface - introduction* [\[JENARDBINTRO\]](#) and shows that the flavours that can be chosen are related to using MD5 hashes for resource and literal IDs (**Hash**), multiple models in a single database (**MM**), particular optimised database schemas for an implementation (PostgreSQL, MySQL, Interbase, Oracle).

Schema 1: Jena Relational Database (RDB) Backend Tables

Table RDF_STATEMENTS		
Column name	Type	Comments
subject	id-ref	
predicate	id-ref	
object	id-ref	
object_isliteral	smallint	flags whether "object" is in literal or resource table
model	id-ref	only used in multiple-model variants
isreified	smallint	not used at present

Table RDF_LITERALS		
Column name	Type	Comments
id	id-ref	
language	varchar	xml:lang value if available
literal_idx	varchar	the literal itself or the largest subset of that which is indexable by the database
literal	blob	the full literal value if the literal won't fit in literal_idx
int_ok	smallint	flag to indicate that an parse of the literal into an integer is available
int_literal	int	the integer value of the literal, only valid if int_OK=1
well_formed	smallint	preserve jena flag that the literal is well-formed xml

Table RDF_RESOURCES		
Column name	Type	Comments
id	id-ref	
namespace	id-ref	pointer to namespace table
localname	varchar	

Table RDF_NAMESPACES		
Column name	Type	Comments
id	id-ref	
uri	varchar	

Table RDF_MODELS		
Column name	Type	Comments
id	id-ref	
name	varchar	Used when reopening a persistent model in a database that supports more than one model.

Table RDF_LAYOUT_INFO - name/value pairs which define the layout properties		
Column name	Type	Comments
name	varchar	
val	varchar	

The layouts currently defined are:

Layout	Supports multiple-models?	Uses hash ids?	Comments
Generic	no	no	See above for details
MMGeneric	yes	no	
GenericProc	no	no	Variant on generic that uses stored procedures for updates
MMGenericProc	yes	no	Variant on generic that uses stored procedures for updates
Hash	no	yes	
MMHash	yes	yes	

Taken from *Jena relational database interface - introduction* [JENARDBINTRO]

Jena uses variants of the schemas given in the layouts described in *Schema 1* the tradeoffs of which were analysed in *Jena relational database interface - performance notes* [JENARDBPERF] for typical operations.

**KAON** - KAON uses the standard Java Entity Bean storage for standard applications - either the container managed persistence or bean managed. For more powerful applications such as working on a complex ontology, there is a specialised "engineering server" relational database schema.

One of the storage systems is described as suitable for Ontology Engineering - *Engineering Server* which is a rich database schema which is highly indexed and unlike Sesame, does not have a table per class so does not require rebuilding as they are added and related.

It requires a SQL2-compatible DBMS and has been tested with MS SQL Server 2000, PostgreSQL, IBM DB2 and Oracle 9i.x

Schema 2: KAON Engineering Server Schema

From EJB classes in Java package `edu.unika.aifb.rdf.rdfserver.ejb`

Table Model ( <code>edu.unika.aifb.rdf.rdfserver.ejb.ModelBean</code> )		
Column name	Type	Comments
alias	java.lang.String	model name
LogicalURI	java.lang.String	URI of model

Table Counter ( <code>edu.unika.aifb.rdf.rdfserver.ejb.CounterBean</code> )		
Column name	Type	Comments
type	varchar(255)	type of bean; used to give a counter for each bean type
counter	int	

Table Statement ( <code>edu.unika.aifb.rdf.rdfserver.ejb.StatementBean</code> )		
Column name	Type	Comments
id	java.lang.Integer	
modelKey	java.lang.Integer	pointer to Model table (alias)
subKey	java.lang.Integer	pointer to Resource table (id)
predKey	java.lang.Integer	pointer to Resource table (id)
objKey	java.lang.Integer	pointer to Resource table (id) or Literal table (id)
literal	java.lang.Integer	true if objKey is a literal (in Literal table, else in Resource)

Table Resource ( <code>edu.unika.aifb.rdf.rdfserver.ejb.ResourceBean</code> )		
Column name	Type	Comments
id	java.lang.Integer	
label	java.lang.String	resource URI

Table Literal ( <code>edu.unika.aifb.rdf.rdfserver.ejb.LiteralBean</code> )		
Column name	Type	Comments
id	java.lang.Integer	
label	varchar(255)	literal string

The KAON engineering server schema additionally provides (from 2002-02-10 version) an enhanced schema with models, supermodels as well as indexing of advanced ontological relationships and constraints.

Table PKCounter		
Column name	Type	Comments
type	varchar(255)	
counter	int	

Table OIModel		
Column name	Type	Comments
modelID	int	
logicalURI	varchar(255)	

Table IncludedOIModel		
Column name	Type	Comments
includingModelID	int	
includedModelID	int	

Table AllIncludedOIModels		
Column name	Type	Comments
includedModelID	int	
includingModelID	int	

Table OIModelEntity		
Column name	Type	Comments
entityID	int	
modelID	int	
entityURI	varchar(255)	
conceptVersion	int	
propertyVersion	int	
isAttribute	smallint	
isSymmetric	smallint	
isTransitive	smallint	
inversePropertyID	int	
inversePropertyModelID	int	
instanceVersion	int	

Table ConceptHierarchy		
Column name	Type	Comments
modelID	int	
superConceptID	int	
subConceptID	int	

Table ConceptInstance		
Column name	Type	Comments
modelID	int	
conceptID	int	
instanceID	int	

Table PropertyHierarchy		
Column name	Type	Comments
modelID	int	
superPropertyID	int	
subPropertyID	int	

Table PropertyDomain		
Column name	Type	Comments
modelID	int	
propertyID	int	
conceptID	int	
minimumCardinality	int	
maximumCardinality	int	

Table PropertyRange		
Column name	Type	Comments
modelID	int	
propertyID	int	
conceptID	int	

Table RelationInstance		
Column name	Type	Comments
modelID	int	
propertyID	int	
sourceInstanceID	int	
targetInstanceID	int	

Table AttributeInstance		
Column name	Type	Comments
modelID	int	
propertyID	int	
sourceInstanceID	int	
textValue	varchar (255)	

**Parka Database** - Parka is a frame-based system and does not use an external relational store using SQL schema. It uses a lightweight internal mini-relational store designed in the frames/slots KR approach predating RDF-style triples. The frames (RDF triple subjects) are identified by integers and contain frame properties that are the non-hierarchical relationships. The relational tables store the domain and range of the properties. The persistent store of properties is complemented with an in-memory cache of the *structural links* or hierarchical properties; the relations ISA, SUBCAT, INSTANCE-OF, INSTANCE so that class and property queries can be carried out more efficiently.

**RDFSuite and SAIL** - RDFSuite has a persistent RDF store based on an object-relational DBMS (ODBMS) with two types of schema - *The RDF Schema Specific Database (RSSDB)* also called SpecRepr and a generic, simple list of triples called GenRepr

The core model of the SpecRepr schema is represented by 4 tables with separation of data and schema information. Class tables store URIs of the resources (instances) of that class. Property tables store the uris of the source and target nodes of the property. It also has tables Class, Property, SubClass, SubProperty for recording the class and property details and their relationships. Sub tables are used for relating tables that are subclass/properties.

Schema 3: RDFSuite SpecRepr Schema

Table Class		
Column name	Type	Comments
id	int	
nsid	int	pointer to namespace URI of the class
lpart	text	local name of the class

Table Property		
Column name	Type	Comments
id	int	
nsid	int	pointer to namespace URI of the property
lpart	text	local name of the property
domainid	int	pointer to the domain class

rangeid	int	pointer to the range class
---------	-----	----------------------------

Table SubClass		
Column name	Type	Comments
subid	int	pointer to the sub-class
superid	int	pointer to the super-class

Table SubProperty		
Column name	Type	Comments
subid	int	pointer to the sub-property
superid	int	pointer to the super-property

Table NameSpace		
Column name	Type	Comments
id	int	
URI	text	URI of namespace (of class/property)

Table Types		
Column name	Type	Comments
type	text	names of RDF/S built-in-types such as rdf:Property, rdf:Bag, ... and literal types (string, integer, date)

Instance Table (for a particular property)		
Column name	Type	Comments
source	text	URI of property source
target	text	URI of property target

Class Table (for a particular class)		
Column name	Type	Comments
URI	text	URI of class instance

Indices are constructed on the attributes URI, source and target of the tables in [schema 3](#) in order to speed up joins and the selection of specific tuples of the tables. Indices are also constructed on the attributes lpart, nsid and id of the tables Class and Property and on the attribute subid of the tables SubClass and SubProperty. Instance tables also connected through the subtable relationship of ORDBMSs.

Schema 4: RDFSuite GenRepr Schema

Table Triples		
Column name	Type	Comments
predid	int	
subid	int	
objid	int	Triple object resource pointer
objvalue	text	Triple literal value string.

Table Resources		
Column name	Type	Comments
id	int	
uri	text	Resource URI

**Sesame and SAIL** - The SAIL uses a dynamic database schema inspired by [\[OCOMWEBPQR\]](#) in which new tables are added for each new class and subclass relationships cause the table to be a subtable of the superclass tables. Similarly for properties.

The schema in [schema 5](#) is from Sesame 0.7.1 (with MySQL, but it applies to PostgreSQL or other RDBMSes with the appropriate SQL changes). All resources and literal values are mapped to a unique ID (resources and literals tables). These are then used to form the triples (triples table) and the relationships between the classes and properties. Indexes are made on several of the tables to enhance domain, range, superclass etc. lookups.

Schema 5: Sesame Schema

Explicitly added statements during a transaction (can have duplicates)

Table addedTriples		
--------------------	--	--

Column name	Type	Comments
subject	unsigned integer	
predicate	unsigned integer	
object	unsigned integer	
explicit	bool	

All statements that were inferred by one inference rule

Table allInferred		
Column name	Type	Comments
subject	unsigned integer	
predicate	unsigned integer	
object	unsigned integer	
explicit	bool	

All statements that were added during a transaction

Table allNewTriples		
Column name	Type	Comments
subject	unsigned integer	
predicate	unsigned integer	
object	unsigned integer	
explicit	bool	

Table class		
Column name	Type	Comments
id	unsigned integer	pointer to class instance (resource) identifier

All direct rdfs:subClassOf relations

Table direct_subclassof		
Column name	Type	Comments
sub	unsigned integer	
super	unsigned integer	

All direct rdfs:subPropertyOf relations

Table direct_subpropertyof		
Column name	Type	Comments
sub	unsigned integer	
super	unsigned integer	

Table domain		
Column name	Type	Comments
property	unsigned integer	
class	unsigned integer	

All statements that were inferred by one inference rule.

Table inferred		
Column name	Type	Comments
subject	unsigned integer	
predicate	unsigned integer	
object	unsigned integer	
explicit	bool	

All instance relations

Table instanceof		
Column name	Type	Comments

inst	unsigned integer	
class	unsigned integer	

Table literals		
Column name	Type	Comments
id	unsigned integer	
language	varchar(4) binary	
value	(text)	

Table namespaces		
Column name	Type	Comments
id	unsigned integer	
prefix	varchar(16) binary	
name	(text)	Namespace URI
export	bool	

All added statements that are actually new (i.e. not yet in the triples table)

Table newTriples		
Column name	Type	Comments
subject	unsigned integer	
predicate	unsigned integer	
object	unsigned integer	
explicit	bool	

All direct instance relations.

Table proper_instanceof		
Column name	Type	Comments
inst	unsigned integer	
class	unsigned integer	

Table property		
Column name	Type	Comments
id	unsigned integer	

Table range		
Column name	Type	Comments
property	unsigned integer	
class	unsigned integer	

Repository metadata

Table rep_metadata		
Column name	Type	Comments
mkey	character varying(255)	
mvalue	character varying(255)	

Table resources		
Column name	Type	Comments
id	unsigned integer	
namespace	unsigned integer	
localname	character varying(255)	

Table subclassof		
Column name	Type	Comments
sub	unsigned integer	
super	unsigned integer	

Table subpropertyof		
Column name	Type	Comments
sub	unsigned integer	
super	unsigned integer	

Table triples		
Column name	Type	Comments
subject	unsigned integer	
predicate	unsigned integer	
object	unsigned integer	
explicit	bool	

**TAP** - TAP uses a mostly hard-coded database schema for MySQL only, but can create tables for particular predicates. The standard form is to index only the *type* (rdf:type) predicate; the remaining triples live in the 'tp' table. It also has built-in knowledge of namespaces which are stripped out of the URIs before they are added to the tables; so although namespaces are used, they are not in the database schema.

Schema 6: TAP MySQL Schema

Table tp		
Column name	Type	Comments
source	text	namespace-shortened URI
arc	text	namespace-shortened URI
target	text	namespace-shortened URI

The table 'io' (which TAP internally calls type) is always present and stores statements with the `rdf:type` property in the schema of the 'tp' table above.

Table lexicon (title, label, plural, adjective arcs)		
Column name	Type	Comments
source	text	namespace-shortened resource URI
str	text	each word of literal
fulltitle	string	full literal

This tables is used to perform the literal indexing for certain property arcs. It is generally queried like:  

```
select source from 'lexicon' where str="word"
```

to identify candidate relevant resources.

## 5 Mapping RDBMS schemas onto RDF

There is a large quantity of existing data that is stored using relational database technology. We now compare relational schemas and idioms to features in RDF, and consider how best to expose relationally-stored data to RDF processors and other Semantic Web tools.

At the moment we only consider the problems of a data-dump into some serialised RDF format. Issues arising from wrapping existing relational database schemas with RDF triple-matching and querying APIs will be dealt with elsewhere in [SWAD-Europe Deliverable 7.2 Report comparing existing RDF query language functionality, documenting different scenarios and users for RDF query languages](#).

**5.1 The relational model** - Clear descriptions of the relational algebra [\[CODD\]](#) and its relationship to modern RDBMS are readily available elsewhere. Here we present a two-line "executive summary":

- Relational data is organised in *relations*, or *tables* as the RDBMS world terms it, with each record or row of a relation consisting of the same set of attributes (or fields).
- RDBMS utilise primary and foreign keys to create joins between tables - that is, to express relationships between the entities described by the contents of two table rows.

**5.2 A naive approach** - A simple approach to making data available in a format suitable for import or further manipulation with native RDF tools is obvious: simply generate a node for every row of every relation. To that node, attach property arcs (one per column in the relation / field in the table) with the field content as the property value.

Such an approach is reasonably straightforward to extend to provide a typical triple-based API. Such APIs need to associate private (model-specific) identifying information with blank nodes in order to preserve their identity. That mechanism can be used to associate with the generated blank nodes a row identifier for the underlying table (either using a ROWID feature, if present, or by encapsulating the values of sufficient fields to uniquely identify the row).

### Problems with the simple approach

There are some conceptual problems with this simple approach, however. These may be effectively summed up as follows: the approach exposes an RDF description of the relational database, not the conceptual entities which the relational description is attempting to capture.

In particular, the expression of various problem-domain concepts within the idiom of relational algebra (particularly as implemented in a RDBMS) may end up using the same small set of RDBMS constructions to model several different problem-domain constraints. That is, there is not necessarily a perfect mapping from the problem-domain into the relational model (nor is it reasonable to expect there to be).

In addition, a developer of a RDBMS application may make a number of assumptions and optimisations that are possible due to (tacit) knowledge of the problem domain and/or experience with efficient database application implementation.

Therefore, a fully automatic solution to convert data from a relational schema into an RDF schema together with instance data can never prove to be fully satisfactory (although it may suffice in some circumstances). Instead, there are additional steps that may be taken (guided by knowledge of the initial problem domain) that can produce a more "natural" expression of the data using an idiom more familiar to RDF users.

**5.3 Improvements on the naive approach** - We look at the case of a single relation first. Consider the relation:

$(p_1, \dots, p_n, a_1, \dots, a_m)$

where the  $p_i$  are the attributes that form the primary key, and the  $a_j$  are the other attributes in the relation.

Where a primary key exists, we may consider a table row as describing the properties of an entity associated with that row in the table.

**5.4 Naming and identifying entities** - The first question that arises is how to name those entities within RDF. There are basically two ways that resources are represented within RDF.

- We can create URIs and use URI-labelled resource nodes in an RDF graph.
- Alternatively, we can create blank nodes and link properties off them that represent the properties expressed by the relation.

The first is a simple solution but it poses problems: do we construct the URIs such that they encode primary key information? Do we need a separate table to map URIs to rows?

The second approach seems a more natural way to expose the information in a single relation, particularly where no obvious URI exists to name a particular entity. This approach is the one adopted here; but should a relation describe a resource that has a "natural" or obvious URI, it makes sense to use the first option.

We therefore transform the row into the following RDF (expressed here in an N-Triples-like format [\[NTRIPLES\]](#)):

$_:x(p_1, \dots, p_n) <eg:p1> p_1 .$

...

$_:x(p_1, \dots, p_n) <eg:pn> p_n .$

$_:x(p_1, \dots, p_n) <eg:a1> a_1 .$

...

$_:x(p_1, \dots, p_n) <eg:am> a_m .$

where  $_:x(p_1, \dots, p_n)$  is a blank node. For the purposes of serialising a graph, the blank node is allocated a local name using a function based on the primary key from this relation, thereby uniquely identifying the entity or resource being described. (For the purposes of wrapping a RDBMS within an RDF API, a "blank node" object would similarly have to carry such identifying information "beneath the hood".)

In fact, where tables are not completely normalised, several entities that exist in a one-to-one relationship may be described by the same relation. Let us assume, for example, that attributes  $a_j \dots a_k$  actually describe a conceptually separate entity. Then the following RDF might be a more natural expression:

$_:x(p_1, \dots, p_n) <eg:r1> _:y(p_1, \dots, p_n) .$

$_:y(p_1, \dots, p_n) <eg:aj> a_j .$

...

$_:y(p_1, \dots, p_n) <eg:ak> a_k .$

Here,  $_:y(p_1, \dots, p_n)$  is a second blank node generated similarly, with sufficient identifying information to be able to locate the source row.

Such denormalisation is common in production databases. In general, this sort of consideration is not generally reflected using the usual machinery of a RDBMS; recourse to an underlying ER diagram or other specific knowledge of the problem-domain is required to make the determination that multiple entities may be represented in a single relation.

**5.5 Choice of property names** - In the above example, we used arbitrary  $<eg:p1> \dots <eg:am>$  arc labels to connect a resource to its attributes. The selection of more appropriate attribute labels (possibly from existing schemas) is an integral part of the translation process.

It is conceivable that the same property label may arise naturally in several places, from several source relations. There are RDFS considerations arising from this, since range and domain declarations in RDFS are not class-contextualised.

Although a full treatment of adapting a legacy RDBMS to a "triple-matching" API is beyond the scope of this document (we consider primarily the export of data here), it is worthwhile noting that having the same property arc arise from several translation rules will likely give rise to (potentially expensive) union queries when trying to answer questions of the form:

Find all  $X$  and  $Y$  such that  
 $X$  *<property-name>*  $Y$ .

### 5.6 Foreign keys and multiple relations - There is a natural expression in RDF of a two-table join:

Table 1:  $(p_1, \dots, p_n, a_1, \dots, a_r)$

Table 2:  $(f_1, \dots, f_n, b_1, \dots, b_s)$

where the  $(f_i)$  form a foreign key. The simplest approach is to create multiple arcs as follows (with the  $_:x(p_1, \dots, p_n)$  generated as above, naming the entity in table 1).

For each row in table 2 with  $(f_i) = (p_i)$ , generate

$_:x(p_1, \dots, p_n) <eg:b1> b_1$  .

...

$_:x(p_1, \dots, p_n) <eg:bs> b_s$  .

Such a simple approach may well suffice in many cases. However, where a one-to-many relationship exists between table 1 and table 2, this approach provides no way of distinguishing between corresponding groups of triples produced by multiple rows from table 2. Therefore, an accurate expression of these relations in RDF requires the creation of a (blank) node for each row in table 2:

For each row in table 2 with  $(f_i) = (p_i)$ , generate

$_:x(p_1, \dots, p_n) <eg:t1t2> _:y(p_1, \dots, p_n)$  .

$_:y(p_1, \dots, p_n) <eg:b1> b_1$  .

...

$_:y(p_1, \dots, p_n) <eg:bs> b_s$  .

Here,  $<eg:t1t2>$  is an arc linking the entity described in table 1 to a set of values from a row in table 2.

We also note in passing that RDF and RDFS can say little or nothing about any cardinality constraints here - more expressive languages (for example, OWL) are required to express such meta-information.

### Foreign keys as enumerated types

Occasionally, and where the underlying RDBMS does not support a better expression of an enumerated type, a database may utilise a foreign key to capture that notion. In such a situation it may be better to map the foreign key directly to either: a datatyped literal, or: a URIref-named resource denoted the value of the enumerated type. Determining when such an alternative mapping may be appropriate is not generally automatable from a data dictionary.

### 5.7 Many-many relationships - Where a many-to-many relationship exists between entities, a RDBMS generally has to model this using a joining table containing foreign keys from each of the tables representing the entities in the many-many relationship.

In the absence of additional fields in the joining table (that is, where the joining table is simply an artifact of the expression of the many-many relationship between the other two tables), such a relationship may be expressed directly in RDF via the use of a simple property arc. In such a situation the only choice remaining is of the name (and direction) of that property arc.

Should additional information exist on the joining table (ie, there are fields that are not part of a foreign key), or should the joining table relate more than two other tables, an additional node will have to be introduced into the resulting RDF to model the relationship directly.

That is, we have two alternatives: either -

Table 1:  $(p_1, \dots, p_k, a_1, \dots, a_l)$

Table 2:  $(q_1, \dots, q_n, b_1, \dots, b_m)$

Joining table:  $(f_1, \dots, f_k, g_1, \dots, g_n)$

Generate the many-many triple as follows -

For each row in the joining table with  $(f_i) = (p_i)$  and  $(g_j) = (q_j)$ , generate

$_:x(p_1, \dots, p_k) <eg:join> _:y(q_1, \dots, q_n)$  .

or the more explicit alternative:

Generate the many-many triples as follows -

For each row in the joining table with  $(f_i) = (p_i)$  and  $(g_j) = (q_j)$ , generate

$_:x(p_1, \dots, p_k) <eg:join> _:join(p_1, \dots, p_k, q_1, \dots, q_n)$  .

$_:y(q_1, \dots, q_n) <eg:join> _:join(p_1, \dots, p_k, q_1, \dots, q_n)$  .

**5.8 Dealing with ordering and sequences** - The relational model does not impose an ordering on the rows comprising a relation: the rows of a relation form a flat, unordered bag (or set). Instead, a RDBMS permits queries to specify ordering criteria using the contents of fields in a query. To improve query efficiency, indexes on fields often used for ordering results may be created.

RDF has two constructs which may be utilised for ordering purposes: containers (the `rdf:_1, ...` construct) and collections.

Mapping the flat relational data onto either of these RDF constructs is likely to be counter-productive. If the original data is orderable, then it must contain fields which permit its ordering. To permit the most simple reflection of such data into RDF, it simply suffices to ensure that all the field contents that are used to create the ordering are reflected into RDF. By the manipulation of the resulting RDF, the data may then be extracted in the original order (assuming the RDF querying technology used permits the ordering of results).

**5.9 Use of RDF datatyping** - With the recent revision of the RDF specifications (at time of writing, approaching last call) [\[RDFCORE\]](#), a mechanism for specifying a datatype to be attached to a literal has been introduced.

The adoption of such a datatyping mechanism has practically zero cost from the point of view of mapping relational data into RDF; and it has the additional benefit that typing information from the original database can be preserved.

For example, where a field exists which contains an integer value, the resulting RDF may utilise the XSD integer datatype rather than the "untyped" RDF literal. That is:

```
_:x(p1, ..., pn) <eg:intValuedProperty> "10"^^<xsd:integer> .
```

rather than:

```
_:x(p1, ..., pn) <eg:intValuedProperty> "10" .
```

For the basic types most commonly found within a relational database, existing datatypes (for example, from the XML Schema Datatypes [\[XSD\]](#)) will suffice.

**5.10 Expressing the resulting schema in RDFS or higher-level languages** - Thus far, attention has been on the mapping of instance data, not on meta-data (the relational schema). Clearly, some mapping is possible. For example, it is trivial to create a class for the entities described in a particular relation, and to generate *rdf:type* arcs as required.

It is natural, in performing a relational-RDF mapping, to consider what support RDF gives for expressing schema-level constraints. In this regard, RDFS is expressively weak in many aspects.

There is no support in RDFS for expressing cardinality constraints. While a number of tools are capable of identifying two blank nodes as denoting the same entity (on the basis of equality conditions expressed over one or more properties of those nodes), the rules for such an inference are not expressible in RDFS.

The semantics for *rdfs:range* and *rdfs:domain* are also universal and conjunctive, not class-contextualised. Therefore, care needs to be taken while making range and domain assertions.

RDF does not have a mechanism for *declaring* new datatypes, although it can represent literals with arbitrary datatypes.

However, RDF and RDFS form a foundation on which a number of more expressive languages are expected to be built. It is hoped that these languages will fill in some of the missing features outlined here.

**5.11 RDF features that have not been utilised** - In the description above of the process of mapping from a relational database to RDF, some of the features present in RDF have not been used:

- We have not considered reification at all above. There is generally no notion of reflection in a relational schema; the mapping outlined above is purely concerned with exposing instance data.
- The container constructions have been somewhat deprecated in the description above. The ordering of data in a RDBMS is done by field content, which can be exposed directly.
- The collection construction has been similarly deprecated. In addition to the considerations affecting collections, the first-rest construct introduces a large number of "artificial" nodes into the RDF graph. The relational model has a much more "flat" feel to it.

**5.12 Relational / RDBMS features that may present complications** - We finish with a rundown of some features in modern RDBMS that may present additional complications in the relational-RDF mapping.

### 5.12.1 Complex field types

Many modern RDBMS permit a wide variety of field types. Some of these may be mapped directly to datatyped literals for preexisting datatypes (eg. *xsd:integer*). For some types, however, it is possible that no corresponding XSD datatype exists, and a new datatype may be required to complete the mapping.

Some modern systems include compound types for fields - for example, "array of integers". Mapping such a construct into a single, large datatyped literal is somewhat unsatisfactory. Instead, RDF containers (or even collections) may be used.

### 5.12.2 Object-relational databases

Some systems include features which might be described as "object-relational". These include the "inheritance" of one relation's properties by another. While the detail of such features varies between products, the notion of class inheritance may be adequately modelled using RDFS.

### 5.12.3 The use of NULL values

The NULL value does not appear directly in relational algebra. With sufficient normalisation, it is possible to "factor out" the need for NULL values. However, pragmatic concerns such as a desire for efficient operation often lead to NULL values becoming necessary.

Unfortunately, the NULL value may be tacitly overloaded with a number of meanings: "not appropriate", "not available", "don't know", even such things as "this person is an adult" (when present in a "date-of-birth" field), and so on; the semantics for a particular NULL value are often only captured by application code.

Therefore, no automatic rule is possible to deal with the many uses and misuses of NULL. Instead, the semantics of each occurrence must be individually considered, and an appropriate expression in RDF found. In many cases, it suffices to not emit triples whose object would be a NULL value.

## 6 Mapping Tools

**KAON REVERSE** - *KAON REVERSE* [KAONREVERSE] from the *KAON project* [KAON] is an early prototype "for mapping relational database content to ontologies enabling both storage of instance data in such databases and querying the database through the conceptualisation of the database". The mapper is intended to be merged with the *Harmonise Mapping Framework* [HMAFRA] tool and the user interface into the KAON OIModeler. The work is ongoing at the current date 2003-02-18.

**D2R MAP** - *D2R MAP* [D2RMAP] is a mapping language for turning a relational database into RDF along with a LGPL-licensed processor that implements the mapping, emitting RDF/XML, N3 or N-Triples.

## 7 Summary and Conclusions

We surveyed existing work mapping on triple stores and databases, especially relational ones and gave an overview of the major implementations with their schemas.

We have looked at the mapping of relational data into RDF. This discussion has been principally from the point of view of *exposing* relational data in a format suitable for import and processing by RDF tools. However, similar considerations apply when wrapping a legacy RDBMS for use with an RDF triple-matching API.

- At the meta level, RDFS is somewhat limited in its ability to capture constraints present in a relational schema.
- Both the relational model and RDF(S) permit idiomatic approximations of a conceptual problem-domain model. These are close enough that the export of data automatically is possible; however, some knowledge of the problem domain is usually required for a more "natural" expression of the data in RDF.
- Further, more formal, work in this area is needed

## A References

[XSD]

[XML Schema Part 2: Datatypes](#), Paul V. Biron, Ashok Malhotra (Eds.), W3C Recommendation, 2 May 2001

[RDFRELATIONAL1]

[Storing RDF in a relational database](#), Sergey Melnik, Stanford University, 2000-2001

[RDFQCASE]

[RDF, SQL and the Semantic Web - a case study](#), D. Brickley and L. Miller, University of Bristol, UK

[W3CACLSQL]

[RDF SQL Mapping for W3C ACLs](#), Eric Prud'hommeaux, W3C

[DBVIEW]

[dbview -- view an SQL DB thru RDF glasses](#), Dan Connolly, W3C

[JENA]

[Jena Semantic Web Toolkit](#), [HP Labs Semantic Web Activity](#)

[JENARDBPERF]

[Jena relational database interface - performance notes](#), Dave Reynolds, part of the documentation for the [Jena Semantic Web Toolkit](#), [HP Labs Semantic Web Activity](#)

[MANVOLUME]

*The RDFSuite: Managing Voluminous RDF Description Bases* ([HTML](#), [PDF](#)), S. Alexaki and V. Christophides and G. Karvounarakis and D. Plexousakis and K. Tolle, Technical report, ICS-FORTH, Heraklion, Greece, 2000.

[RSSDB]

[RSSDB - RDF Schema Specific DataBase \(RSSDB\)](#), ICS-Forth, 2002

[RDFSUITE]

[RDFSutie Project](#)

[KAON]

[The Karlsruhe Ontology and Semantic Web Tool Suite \(KAON\)](#)

[KAONDEV]

[The Karlsruhe ONtology and Semantic Web Infrastructure Developer's Guide](#), [KAON project](#), FZI

and AIFB, University of Karlsruhe, August 2002



[KAON-SERVER]

[KAON SERVER prototype](#), Daniel Oberle, Raphael Volz, Boris Motik, Steffen Staab, Deliverable 6, EU IST Project 2001-33052 WonderWeb, 13 January 2003.

[PARKASW]

[ParkaSW - Parka inferencing database](#), open source release, [MINDswap group](#), University of Maryland, College Park, Maryland, USA.

[PARKADB]

[PARKA-DB: A Scalable Knowledge Representation System](#)

[SESAMEPROJ]

[Sesame Open Source RDF Schema-based Repository and Querying facility](#)

[SESAMETR]

[Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema](#), OTK-del-10,

[On-To-Knowledge](#) deliverable 10.

[SESAME]

*Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema* [PDF](#) ([Springer subscriber-only link](#)), Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen in proceedings of [First International Semantic Web Conference \(ISWC'02\)](#), Sardinia, Italy, June 9-12, 2002.

[BenchRDFS]

*Benchmarking RDF Schemas for the Semantic Web* ([PDF](#)), A. Maganaraki, S. Alexaki, V. Christophides, and Dimitris Plexousakis, ICS-Forth, Heraklion, Greece in proceedings of [First International Semantic Web Conference \(ISWC'02\)](#), Sardinia, Italy, June 9-12, 2002.

[QCOMWEBPOR]

*Querying Community Web Portals*, G. Karvounarakis, V. Christophides, D. Plexousakis, and S. Alexaki, Technical Report, ICS-FORTH, Heraklion, Greece.

[TAPPROJ]

[TAP Project](#), Stanford University, 2002

[TAPKB]

[TAP Knowledge Base](#), Stanford University, 2002

[GETDATA]

[GetData Data Query Interface](#), TAP Project, Stanford University, 2002

[JENARDBINTRO]

[Jena relational database interface - introduction](#), Dave Reynolds, part of the documentation for the [Jena Semantic Web Toolkit](#), [HP Labs Semantic Web Activity](#)

[CODD]

[A Relational Model of Data for Large Shared Data Banks](#), E.F. Codd, Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[NTRIPLES]

[N-Triples](#), part of [RDF Test Cases](#), W3C Working Draft, Jan Grant, Dave Beckett (editors), Work in progress, 12 November 2002

[RDFCORE]

[RDF Core Working Group](#), World Wide Web Consortium

[KAONREVERSE]

[KAON REVERSE](#), prototype GUI for mapping relational database content to ontologies, [KAON project](#), FZI and AIFB, University of Karlsruhe.

[HMAFRA]

[Harmonise Mapping Framework](#) tools supporting semantic mapping definition and data reconciliation between ontologies.

[D2RMAP]

[D2R MAP - Database to RDF Mapping Language and Processor](#), Chris Bizer, Freie Universität Berlin