On-demand RDF to Relational Query Translation in Samizdat RDF Store

Dmitry Borodaenko
Belarusian State University of Informatics and Radioelectronics
6 Brovki st., Minsk, Belarus
Email: angdraug@debian.org

***Abstract*—This paper presents an algorithm for on-demand translation of RDF queries that allows to map any relational data structure to RDF model, and to perform queries over a combination of mapped relational data and arbitrary RDF triples with a performance comparable to that of relational systems. Query capabilities implemented by the algorithm include optional and negative graph patterns, nested sub-patterns, and limited RDFS and OWL inference backed by database triggers.**

I. Introduction

A wide range of solutions that map relational data to RDF data model has accumulated to date [1]. There are several factors that make integration of RDF and relational data important for the adoption of the Semantic Web. One reason, shared with RDF stores based on a triples table, is the wide availability of mature relational database implementations which had seen decades of improvements in reliability, scalability, and performance. Second is the fact that most of structured data available online is backed by relational databases. This data is not likely to be replaced by pure RDF stores in the near future, so it has to be mapped in one way or another to become available to RDF agents. Finally, properly normalized and indexed application-specific relational database schema allows a DBMS to optimize complex queries in ways that are not possible for a tree of joins over a single triples table [12].

In the Samizdat open publishing engine, most of the data fits into the relational model, with the exception of reified RDF statements which are used in collaborative decision making process [5] and require a more generic triple store. The need for a generic RDF store with performance on par with a relational database is the primary motivation behind the design of Samizdat RDF storage module, which is different from both triples table based RDF stores and relational to RDF mapping systems. Unlike the former, Samizdat can run optimized SQL queries over application-specific tables, but unlike the latter, it is not limited by the relational database schema and can fall back, within the same query, to a triples table for RDF predicates that are not mapped to the relational model.

The following sections of this paper describe: targeted relational data, database triggers required for RDFS and OWL inference, query translation algorithm, update request execution algorithm, details of algorithm implementation in Samizdat, analysis of its performance, comparison with related work, and outline for future work.

II. Relational Data

Samizdat RDF storage module does not impose additional restrictions on the underlying relational database schema beyond the requirements of the SQL standard. Any legacy database may be adapted for RDF access while retaining backwards compatibility with existing SQL queries.

The adaptation process involves adding attributes, foreign keys, tables, and triggers to the database to enable RDF query translation and support optional features of Samizdat RDF store, such as statement reification and inference for *rdfs:subClassOf*, *rdfs:subPropertyOf*, and *owl:TransitiveProperty* rules.

Following database schema changes are required for all cases:

- create *rdfs:Resource* superclass table with autogenerated primary key;
- replace primary keys of mapped subclass tables with foreign keys referencing the *rdfs:Resource* table (existing foreign keys may need to be updated to reflect this change);
- register *rdfs:subClassOf* inference database triggers to update the Resource table and maintain foreign keys integrity on all changes in mapped subclass tables.

Following changes may be necessary to support optional RDF mapping features:

- register database triggers for other cases of *rdfs:subClassOf* entailment;
- create triples table (required to represent non-relational RDF data and RDF statement reification);
- add subproperty qualifier attributes referencing property URIref entry in the *rdfs:Resource* table for each attribute mapped to a superproperty;
- create transitive closure tables, register *owl:TransitiveProperty* inference triggers.

III. Inference and Database Triggers

Samizdat RDF storage module implements entailment rules for following RDFS predicates and OWL classes: *rdfs:subClassOf*, *rdfs:subPropertyOf*, *owl:TransitiveProperty*. Database triggers are used to minimize impact of RDFS and OWL inference on query performance:

rdfs:subClassOf inference triggers are invoked on every insert into and delete from a subclass table. When a tuple

```
1: if $o_{new} = s_\omega$ or $\langle o_{new}, \tau, s_\omega \rangle \in G^+_\tau$ then
2:     stop                                  $\triangleright$ refuse to create a cycle in $G_\tau$
3: end if
4: $G_\tau \leftarrow G'_\tau$                                  $\triangleright$ apply $\omega$
5: if $\omega \in \{update, delete\}$ then
6:     $G^+_\tau \leftarrow G^+_\tau \setminus \{\langle s, \tau, o \rangle \mid (s = s_\omega \lor \langle s, \tau, s_\omega \rangle \in G^+_\tau) \land \langle s_\omega, \tau, o \rangle \in G^+_\tau\}$    $\triangleright$ remove obsolete arcs from $G^+_\tau$
7: end if
8: if $\omega \in \{insert, update\}$ then            $\triangleright$ add new arcs to $G^+_\tau$
9:     $G^+_\tau \leftarrow G^+_\tau \cup \{\langle s_\omega, \tau, o \rangle \mid o = o_{new} \lor \langle o_{new}, \tau, o \rangle \in G^+_\tau\}$
10:    $G^+_\tau \leftarrow G^+_\tau \cup \{\langle s, \tau, o \rangle \mid \langle s, \tau, s_\omega \rangle \in G^+_\tau \land \langle s_\omega, \tau, o \rangle \in G^+_\tau\}$
11: end if
```

Fig. 1. Update transitive closure

without a primary key is inserted,[1] a template tuple is inserted into superclass table and the produced primary key is added to the new subclass tuple. Delete operation is cascaded to all subclass and superclass tables.

rdfs:subPropertyOf inference is performed during query translation, with help of a stored procedure that returns the attribute value when subproperty qualifier attribute is set, and NULL otherwise.

owl:TransitiveProperty inference uses a separate transitive closure table for each relational attribute mapped to a transitive property. Transitive closure tables are maintained by triggers invoked on each insert, update, and delete operation involving such an attribute.

The transitive closure update algorithm is presented in Fig. 1. The input to the algorithm is:

- directed labeled graph $G = \langle N, A \rangle$ where N is a set of nodes representing RDF resources and A is a set of arcs $a = \langle s, p, o \rangle$ representing RDF triples;
- transitive property τ;
- subgraph $G_\tau \subseteq G$ such that:

$$a_\tau = \langle s, p, o \rangle \in G_\tau \iff a_\tau \in G \land p = \tau\,; \qquad (1)$$

- graph G^+_τ containing transitive closure of G_τ;
- update operation $\omega \in \{insert, update, delete\}$ and its parameters $a_{old} = \langle s_\omega, \tau, o_{old} \rangle$, $a_{new} = \langle s_\omega, \tau, o_{new} \rangle$ such that:

$$G'_\tau = (G_\tau \setminus \{a_{old}\}) \cup \{a_{new}\}\,. \qquad (2)$$

The algorithm transforms G^+_τ into a transitive closure of G'_τ. The algorithm assumes that G_τ is and should remain acyclic.

IV. Query Pattern Translation

Class structure of the Samizdat RDF storage module is as follows. External API is provided by the `RDF` class. RDF storage configuration as described in section II is encapsulated in `RDFConfig` class. The concrete syntax of Squish [4], [8] and SQL is abstracted into `SquishQuery` and its subclasses. The query pattern translation algorithm is implemented by the `SqlMapper` class.

[1] Insertion into subclass table with explicit primary key is used in two-step resource insertion during execution of RDF update command (described in section V).

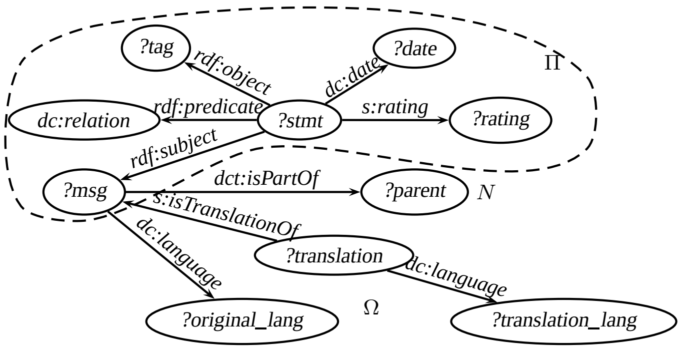

Fig. 2. Graph pattern Ψ for the example query

The input to the algorithm is as follows:

- mappings $M = \langle M_{rel}, M_{attr}, M_{sub}, M_{trans} \rangle$ where $M_{rel} : P \to R$, $M_{attr} : P \to \Phi$, $M_{sub} : P \to S$, $M_{trans} \to T$; P is a set of mapped RDF properties, R is a set of relations, Φ is a set of relation attributes, $S \subset P$ is a subset of RDF properties that have configured subproperties, $T \subset R$ is a set of transitive closures (as described in sections II and III);
- graph pattern $\Psi = \langle \Psi_{nodes}, \Psi_{arcs} \rangle = \Pi \cup N \cup \Omega$, where Π, N, and Ω are main ("must bind"), negative ("must not bind"), and optional ("may bind") graph patterns respectively, such that Π, N, and Ω share no arcs, and Π, $\Pi \cup N$ and $\Pi \cup \Omega$ are joint graphs.[2]
- global filter condition $F_g \in F$ and local filter conditions $F_c : \Psi_{arcs} \to F$ where F is a set of all literal conditions expressible in the query language syntax.

For example, consider the following Squish query and its graph pattern Ψ presented in Fig. 2.

```
SELECT ?msg
WHERE (rdf::predicate ?stmt dc::relation)
      (rdf::subject ?stmt ?msg)
      (rdf::object ?stmt ?tag)
      (dc::date ?stmt ?date)
      (s::rating ?stmt ?rating
          FILTER ?rating >= :threshold)
EXCEPT (dct::isPartOf ?msg ?parent)
OPTIONAL (dc::language ?msg ?original_lang)
      (s::isTranslationOf ?msg ?translation)
      (dc::language ?translation ?translation_lang)
LITERAL ?original_lang = :lang
     OR ?translation_lang = :lang
GROUP BY ?msg
ORDER BY max(?date) DESC
```

The output of the algorithm is a join expression F and condition W ready for composition into `FROM` and `WHERE` clauses of an SQL `SELECT` statement.

In the algorithm description below, $\mathrm{id}(r)$ is used to denote primary key of relation $r \in R$, and $\rho(n)$ is used to denote value

[2] Arcs with the same subject, object, and predicate but different bind mode are treated as distinct.

of $\mathrm{id}(Resource)$ for non-variable node $n \in \Psi_{nodes}$ where such value is known during query translation.[3]

Key steps of the query pattern translation algorithm correspond to the following private methods of `SqlMapper`:

`label_pattern_components`: Label every connected component of Π, N, and Ω with different colors K such that $K_\Pi : \Pi_{nodes} \to \mathbb{K}, K_N : N_{nodes} \to \mathbb{K}, K_\Omega : \Omega_{nodes} \to \mathbb{K}, K(n) = K_\Pi(n) \cup K_N(n) \cup K_\Omega(n)$. The Two-pass Connected Component Labeling algorithm [11] is used with a special case to exclude nodes present in Π from neighbour lists while labeling N and Ω. The special case ensures that parts of N and Ω which are only connected through a node in Π are labeled with different colors.

`map_predicates`: Map each arc $c = \langle s, p, o \rangle \in \Psi_{arcs}$ to the relational data model according to M: define mapping $M^{pos}_{attr} : \Psi_{arcs} \times \Psi_{nodes} \to \Phi$ such that $M^{pos}_{attr}(c, s) = \mathrm{id}(M_{rel}(p)), M^{pos}_{attr}(c, o) = M_{attr}(p)$; replace each unmapped arc with its reification and map the resulting arcs in the same manner;[4] for each arc labeled with a subproperty predicate, add an arc mapped to the subproperty qualifier attribute. For each node $n \in \Psi_{nodes}$, find adjacent arcs $\Psi^n_{nodes} = \{\langle s, p, o \rangle \mid n \in \{s, o\}\}$ and determine its binding mode $\beta_{node} : \Psi_{nodes} \to \{\Pi, N, \Omega\}$ such that $\beta_{node}(n) = max(\beta_{arc}(c) \forall c \in \Psi^n_{nodes})$ where $\beta_{arc}(c)$ reflects which of the graph patterns $\{\Pi, N, \Omega\}$ contains arc c, and the order of precedence used by max is $\Pi > N > \Omega$.

`define_relation_aliases`: Map each node in Ψ to one or more relation aliases $a \in \mathbb{A}$ according to the algorithm described in Fig. 3. The algorithm produces mapping $C_a : \Psi_{arcs} \to \mathbb{A}$ which links every arc in Ψ to an alias, and mappings $A = \langle A_{rel}, A_{node}, A_\beta, A_{filter} \rangle$ where $A_{rel} : \mathbb{A} \to R$, $A_{node} : \mathbb{A} \to \Psi_{nodes}$, $A_\beta : \mathbb{A} \to \{\Pi, N, \Omega\}$, $A_{filter} : \mathbb{A} \to F$) which record relation, node, bind mode, and a filter condition for each alias.

`transform`: Define bindings $B : \Psi_{nodes} \to \mathbb{B}$ where $\mathbb{B} = \{\{\langle a, f \rangle \mid a \in \mathbb{A}, f \in \Phi\}\}$ of graph pattern nodes to sets of pairs of relation aliases and attributes, such that

$$\begin{aligned} \langle a, f \rangle \in B(n) \iff & \exists c \in \Psi^n_{arcs} \\ & C_a(c) = a, M^{pos}_{attr}(c, n) = f \,. \end{aligned} \tag{3}$$

Transform graph pattern Ψ into relational query graph $Q = \langle \mathbb{A}, J \rangle$ where nodes $\mathbb{A}$ are relation aliases defined earlier and edges $J = \{\langle b_1, b_2, n \rangle \mid b_1 = \langle a_1, f_1 \rangle \in B(n), b_2 = \langle a_2, f_2 \rangle \in B(n), a_1 \neq a_2\}$ are join conditions. Ground non-variable nodes according to the algorithm defined in Fig. 4. Record list of grounded nodes $G \subseteq \Psi_{nodes}$ such that

$$\begin{aligned} n \in G \iff & n \in F_g \vee \exists \langle b_1, b_2, n \rangle \in J \\ & \vee \exists b \in B(n) \, \exists a \in \mathbb{A} \, b \in A_{filter}(a) \,. \end{aligned} \tag{4}$$

Transformation of the example query presented above will result in a relational query graph in Fig. 5.

[3]E.g. Samizdat uses *site-ns/resource-id* notation for internal resource URIrefs.

[4]M is expected to map reification properties to the triples table.

```
for all $n \in \Psi_{nodes}$ do
    for all $c = \langle s, p, o \rangle \in \Psi_{arcs} \mid s = n \wedge C_a(c) = \emptyset$ do
        if $\exists c' = \langle s', p', o' \rangle \mid n \in \{s', o'\} \wedge C_a(c') \neq \emptyset \wedge M_{rel}(p') = M_{rel}(p)$ then
            $C_a(c) \leftarrow C_a(c')$ ▷ Reuse the alias assigned to an arc adjacent to $n$ and mapped to the same relation
        else                                        ▷ Create new alias
            $a = max(\mathbb{A}) + 1$; $\mathbb{A} \leftarrow \mathbb{A} \cup \{a\}$; $C_a(c) \leftarrow a$
            $A_{node}(a) \leftarrow n$, $A_{filter}(a) \leftarrow \emptyset$
            if $M_{trans}(p) = \emptyset$ then      ▷ Use base relation
                $A_{rel}(a) \leftarrow M_{rel}(p)$
                $A_\beta(a) \leftarrow \beta_{node}(n)$
            else                                    ▷ Use transitive closure
                $A_{rel}(a) \leftarrow M_{trans}(p)$
                $A_\beta(a) \leftarrow \beta_{arc}(c)$
                        ▷ Use arc's bind mode instead of node's
            end if
        end if
    end for
end for
for all $c \in \Psi_{arcs}$ do
    $A_{filter}(C_a(c)) \leftarrow A_{filter}(C_a(c)) \cup F_c(c)$
                        ▷ Add arc filter to the linked alias filters
end for
```

Fig. 3. Define relation aliases

```
$\exists b = \langle a, f \rangle \in B(n)$              ▷ Take any binding of $n$
if $n$ is an internal resource and $\rho(n) = i$ then
    $A_{filter}(a) \leftarrow A_{filter}(a) \cup (b = i)$
else if $n$ is a query parameter or a literal then
    $A_{filter}(a) \leftarrow A_{filter}(a) \cup (b = n)$
else if $n$ is a URIref then ▷ Add a join to a URIref tuple in Resource relation
    $\mathbb{A} \leftarrow \mathbb{A} \cup \{a_r\}$; $A_{node}(a_r) = n$; $A_{rel}(a_r) = Resource$; $A_\beta(a_r) = \beta_{node}(n)$
    $B(n) \leftarrow B(n) \cup \langle a_r, \mathrm{id}(Resource) \rangle$; $J \leftarrow J \cup \{\langle b, \langle a_r, \mathrm{id}(Resource) \rangle, n \rangle\}$
    $A_{filter}(a_r) = A_{filter}(a_r) \cup (\langle a_r, literal \rangle = f \wedge \langle a_r, uriref \rangle = t \wedge \langle a_r, label \rangle = n)$
end if
```

Fig. 4. Ground non-variable nodes

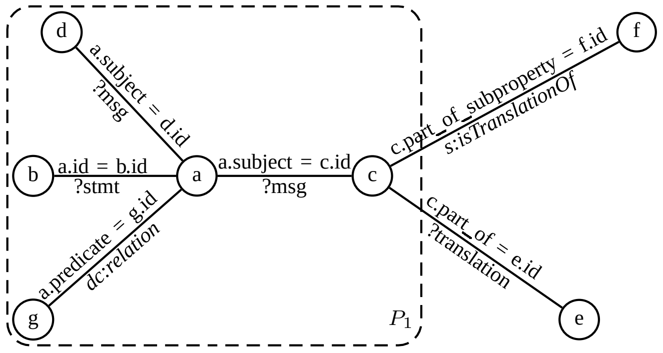

Fig. 5. Relational query graph Q for the example query

`generate_tables_and_conditions`: Produce ordered connected minimum edge-disjoint tree cover P for relational query graph Q such that $\forall P_i \in P \quad \forall j = \langle b_{j1}, b_{j2}, n_j \rangle \in P_i \quad \forall k = \langle b_{k1}, b_{k2}, n_k \rangle \in P_i$:

$$K(n_j) \cap K(n_k) \neq \emptyset \,, \tag{5}$$

$$\beta_{node}(n_j) = \beta_{node}(n_k) = \beta_{tree}(P_i) \,, \tag{6}$$

starting with P_1 such that $\beta_{tree}(P_1) = \Pi$ (it follows from definitions of Ψ and `transform` that P_1 is the only such tree and covers all join conditions $\langle b_1, b_2, n \rangle \in J$ such that $\beta_{node}(n) = \Pi$). Encode P_1 as the root inner join. Encode other trees with at least one edge as subqueries. Left join subqueries and aliases representing roots of zero-length trees into join expression F. For each P_i such that $\beta_{tree}(P_i) = N$, find a binding $b = \langle a, f \rangle \in P_i$ such that $a \in P_1 \cap P_i$ and add (b `IS NULL`) condition to W. For each non-grounded node $n \notin G$ such that $\langle a, f \rangle \in B(n) \wedge a \in P_1$, add ($b$ `IS NOT NULL`) condition to W if $\beta_{node}(n) = \Pi$, or (b `IS NULL`) condition if $\beta_{node}(n) = N$. Add F_g to W.

Translation of the example query presented earlier will result in the following SQL:

```
SELECT DISTINCT a.subject, max(b.published_date)
FROM Statement AS a
INNER JOIN Resource AS b ON (a.id = b.id)
INNER JOIN Resource AS c ON (a.subject = c.id)
INNER JOIN Message AS d ON (a.subject = d.id)
INNER JOIN Resource AS g ON (a.predicate = g.id)
  AND (g.literal = 'false' AND g.uriref = 'true'
  AND g.label = 'http://purl.org/dc/elements/1.1/relation')
LEFT JOIN (
  SELECT e.language AS _field_b, c.id AS _field_a
  FROM Message AS e
  INNER JOIN Resource AS f ON (f.literal = 'false'
         AND f.uriref = 'true' AND f.label =
'http://www.nongnu.org/samizdat/rdf/schema#isTranslationOf')
  INNER JOIN Resource AS c ON (c.part_of_subproperty = f.id)
         AND (c.part_of = e.id)
) AS _subquery_a ON (c.id = _subquery_a._field_a)
WHERE (b.published_date IS NOT NULL)
  AND (a.object IS NOT NULL) AND (a.rating IS NOT NULL)
  AND (c.part_of IS NULL) AND (a.rating >= ?)
  AND (d.language = ? OR _subquery_a._field_b = ?)
GROUP BY a.subject ORDER BY max(b.published_date) DESC
```

V. Update Command Execution

Update command uses the same graph pattern structure as a query, and additionally defines a set $\Delta \subset \Psi_{nodes}$ of variables representing new RDF resources and a mapping $U : \Psi_{nodes} \to \mathbb{L}$ of variables to literal values. Execution of an update command starts with query pattern translation using the algorithm described in section IV. The variables Ψ, A, Q, etc. produced by pattern translation are used in the subsequent stages as described below:

1) Construct node values mapping $V : \Psi_{nodes} \to \mathbb{L}$ using the algorithm defined in Fig. 6. Record resources inserted into the database during this stage in $\Delta_{new} \subset \Psi_{nodes}$ (it follows from the algorithm definition that $\Delta \subseteq \Delta_{new}$).
2) For each alias $a \in \mathbb{A}$, find a subset of graph pattern $\Psi^a_{arcs} \subseteq \Psi_{arcs}$ such that $c \in \Psi^a_{arcs} \iff C_a(c) = a$, select a key node k such that $\exists c = \langle k, p, o \rangle \in \Psi^a_{arcs}$, and collect a map $D_a : \Phi \to \mathbb{L}$ of fields to values such that $\forall c = \langle s, p, o \rangle \in \Psi^a_{arcs} \; \exists D_a(o) = V(o)$. If $k \in \Delta_{new}$ and $A_{rel}(a) \neq Resource$, transform D_a into an SQL `INSERT` into $A_{rel}(a)$ with explicit primary key assignment $\mathrm{id}_k(A_{rel}(a)) \leftarrow V(k)$. Otherwise, transform D_a into an `UPDATE` statement on the tuple in $A_{rel}(a)$ for which $\mathrm{id}_k(A_{rel}(a)) = V(k)$.
3) Execute the SQL statements produced in the previous stage inside the same transaction in the order that resolves their mutual references.

```
1:  for all $n \in \Psi_{nodes}$ do
2:      if $n$ is an internal resource and $\rho(n) = i$ then
3:          $V(n) \leftarrow i$
4:      else if $n$ is a query parameter or a literal then
5:          $V(n) \leftarrow n$
6:      else if $n$ is a variable then
7:          if $\nexists c = \langle n, p, o \rangle \in \Psi_{arcs}$ then
8:                                  ▷ If found only in object position
9:              $V(n) \leftarrow U(n)$
10:         else
11:             if $n \notin \Delta$ then
12:                 $V(n) \leftarrow \mathrm{SquishSelect}(n, \Psi^{n*})$
13:             end if
14:             if $V(n) = \emptyset$ then
15:                 Insert $n$ into $Resource$ relation
16:                 $V(n) \leftarrow \rho(n)$
17:                 $\Delta_{new} \leftarrow \Delta_{new} \cup n$
18:             end if
19:         end if
20:     else if $n$ is a URIref then
21:         Select $n$ from $Resource$ relation, insert if missing
22:         $V(n) \leftarrow \rho(n)$
23:     end if
24: end for
```

Fig. 6. Determine node values. Ψ^{n*} is a subgraph of Ψ reachable from n. $\mathrm{SquishSelect}(n, \Psi)$ finds a mapping of variable n that satisfies pattern Ψ.

VI. Implementation

The algorithms described in previous sections are implemented by the Samizdat RDF storage module, which is used as the primary means of data access in the Samizdat open publishing system. The module is written in Ruby programming language, supported by several triggers written in procedural SQL. The module and the whole Samizdat engine are available under GNU General Public License.

Samizdat exposes all RDF resources underpinning the structure and content of the site. HTTP request with a URL of any internal resource yields a page with detailed information about the resource and its relation with other resources. Furthermore, Samizdat provides a graphical interface that allows to compose arbitrary Squish queries.[5] Queries may be published so that other users may modify and reuse them, results of a query may be accessed either as plain HTML or as an RSS feed.

[5]Complexity of user queries is limited to a configurable maximum number of triples in the graph pattern to prevent abuse.

VII. Evaluation of Results

Samizdat performance was measured using Berlin SPARQL Benchmark (BSBM) [2], with following variations: a functional equivalent of BSBM test driver was implemented in Ruby and Squish (instead of Java and SPARQL); the test platform included Intel Core 2 Duo (instead of Quad) clocked at the same frequency, and 2GB of memory (instead of 8GB). In this environment, Samizdat was able to process 25287 complete query mixes per second (QMpH) on a dataset with 1M triples, and achieved 18735 QMpH with 25M triples, in both cases exceeding figures for all RDF stores reported in [2].

In production, Samizdat was able to serve without congestion peak loads of up to 5K hits per hour for a site with a dataset sized at 100K triples in a shared VPS environment. Regeneration of the site frontpage on the same dataset executes 997 Squish queries and completes in 7.7s, which is comparable to RDBMS-backed content management systems.

VIII. Comparison with Related Work

As mentioned in section I, there exists a wide range of solutions for relational to RDF mapping. Besides Samizdat, the approach based on automatic on-demand translation of RDF queries into SQL is also implemented by Federate [9], D2RQ [3], and Virtuoso [7].

While being one of the first solutions to provide on-demand relational to RDF mapping, Samizdat remains one of the most advanced in terms of query capabilities. Its single largest drawback is lack of compatibility with SPARQL; in the same time, in some regards it exceeds capabilities of other solutions.

The alternative that is closest to Samizdat in terms of query capabilities is Virtuoso RDF Views: it is the only other relational-to-RDF mapping solution that provides partial RDFS and OWL inference, aggregation, and an update language. Still, there are substantial differences between these two projects. First of all, Samizdat RDF store is a small module (1000 lines of Ruby and 200 lines of SQL) that can be used with a variety of RDBMSes, while Virtuoso RDF Views is tied to its own RDBMS. Virtuoso doesn't support implicit statement reification, although its design is compatible with this feature. Finally, Virtuso relies on SQL unions for queries with unspecified predicates and RDFS and OWL inference. While allowing for greater flexibility than the database triggers described in section III, iterative union operation has a considerable impact on query performance.

IX. Future Work

Since the SPARQL Recommendation has been published by W3C [10], SPARQL support has been at the top of the Samizdat RDF store to-do list. SPARQL syntax is considerably more expressive than Squish and will require some effort to implement in Samizdat, but, since design of the implementation separates syntactic layer from the query translation logic, the same algorithms as described in this paper can be used to translate SPARQL patterns to SQL with minimal changes. Most substantial changes are expected to be required for the explicit grouping of optional graph patterns and the associated filter scope issues [6].

Samizdat RDF store should be made more adaptable to a wider variety of problem domains. Query translation algorithm should be augmented to translate an ambiguously mapped query (including queries with unspecified predicates) to a union of alternative interpretations. Mapping of relational schema should be generalized, including support for multi-part keys and more generic stored procedures for reification and inference. Standard RDB2RDF mapping should be implemented when W3C publishes a specification to that end.

X. Conclusions

The on-demand RDF to relational query translation algorithm described in this paper utilizes existing relational databases to their full potential, including indexing, transactions, and procedural SQL, to provide efficient access to RDF data. Implementation of this algorithm in Samizdat RDF storage module has been tried in production environment and demonstrated how Semantic Web technologies can be introduced into an application serving thousands of users without imposing additional requirements on hardware resources.

References

[1] Auer, S., Dietzold, S. Lehman, J., Hellmann, S., Aumueller, D.: Triplify – Light-Weight Linked Data Publication from Relational Databases. WWW 2009, Madrid, Spain (2009) http://www.informatik.uni-leipzig.de/~auer/publication/triplify.pdf

[2] Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. International Journal On Semantic Web and Information Systems (IJSWIS), Volume 5, Issue 2 (2009) http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/

[3] Bizer, C., Seaborne, A.: D2RQ - Treating non-RDF databases as virtual RDF graphs. In: ISWC 2004 (posters) http://www.wiwiss.fu-berlin.de/bizer/D2RQ/spec/

[4] Borodaenko, Dmitry: Accessing Relational Data with RDF Queries and Assertions (April 2004) http://samizdat.nongnu.org/papers/rel-rdf.pdf

[5] Borodaenko, Dmitry: Model for Collaborative Decision Making Based on RDF Reification (April 2004) http://samizdat.nongnu.org/papers/collreif.pdf

[6] Cyganiak, R.: A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Labs (2005) http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html

[7] Erling, O., Mikhailov I.: RDF support in the Virtuoso DBMS. In: Proceedings of the 1st Conference on Social Semantic Web, volume P-113 of GI-Edition – Lecture Notes in Informatics (LNI), ISSN 1617-5468. Bonner Köllen Verlag (2007) http://virtuoso.openlinksw.com/dav/wiki/Main/VOSArticleRDF

[8] Miller, Libby, Seaborne, Andy, Reggiori, Alberto: Three Implementations of SquishQL, a Simple RDF Query Language. In: Horrocks, I., Hendler, J. (Eds) ISWC 2002. LNCS vol. 2342, pp. 423-435. Springer, Heidelberg (2002) http://ilrt.org/discovery/2001/02/squish/

[9] Prud'hommeaux, Eric: RDF Access to Relational Databases (2003) http://www.w3.org/2003/01/21-RDF-RDB-access/

[10] Prud'hommeaux, Eric, Seaborne, Andy: SPARQL Query Language for RDF. W3C Recommendation (January 2008) http://www.w3.org/TR/rdf-sparql-query/

[11] Shapiro, L., Stockman, G: Computer Vision, pp. 69-73. Prentice-Hall (2002) http://www.cse.msu.edu/~stockman/Book/2002/Chapters/ch3.pdf

[12] Schmidt, M., Hornung, T., Küchlin, N., Lausen, G., Pinkel, C.: An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In: A. Sheth et al. (Eds.) ISWC 2008. LNCS vol. 5318, pp. 82-97. Springer, Heidelberg (2008) http://www.informatik.uni-freiburg.de/~mschmidt/docs/sp2b_exp.pdf