AllegroGraph RDF-Triplestore Evaluation - Joint Study

Report Draft: June 26, 2009
Last Updated: July 9, 2009

Participants:

Walter W. Chang
Adobe Advanced Technology Labs

Bill Millar
Franz Consulting, Franz, Inc.

Technical Feedback:
John Fieber & Kevin Stewart
Adobe Core Technology Group

Sponsors:
Marc LeBrun & Erica Schisler
Adobe Digital Video Engineering

This report presents a summary of the findings of a joint study conducted by DMO, ATL and Franz Inc. to evaluate the suitability of the AllegroGraph RDF-Triplestore for use in the Metasky/Triad storage system. The participants of this joint study included Walter Chang, ATL, Bill Millar, Franz Inc. Consulting, and J. Fieber, K. Stewart and Marc LeBrun who provided valuable technical feedback.

This joint study was funded by DMO; ATL developed a set of evaluation experiments, collaborated with a Franz consultant, and independently repeated and verified the results from Franz. The activities of the joint study were supported and sponsored by Marc LeBrun and Erica Schisler. While Franz AllegroGraph technology was used for our experiments, many of the methods and findings of this report will be relevant to other RDF-Triplestore technologies such as Redland/Postgres.

Three output deliverables are summarized in this report:

1. Performance and volumetric experiment results in developing a methodology for mapping Sedona2/Goldman relational databases into the AllegroGraph RDF-Triplestore.

2. The identification and characterization of high utility triplestore inferencing queries, and

3. The identification of potential performance bottlenecks, storage overhead, and design complexity issues introduced by any triplestore technology;

Milestone findings have been incrementally communicated to the key members of the Triad/Metasky team (K. Stewart, Engineering Manager, J. Fieber, Computer Scientist, and Jim Hong, Program Manager.)

DESCRIPTION OF JOINT STUDY ACTIVITIES AND PROCESS

Three major activities were conducted in support of the deliverables listed above and resulted in an ATL system technology evaluation framework that can be leveraged for any future joint study work. Findings are summarized in the three major sections that follow.

Section 1. describes the activities focused on porting and deploying the Sedona2 Adobe Story script analysis and relational metadata database into the Franz Inc. AllegroGraph 3.2 RDF-Triplestore. Included in this work are performance measurements for migrating sample movie script metadata to the triplestore, and measuring storage volumetrics.

Section 2. describes the types kinds of application triplestore queries anticipated SPARQL query patterns for these queries jointly developed by Adobe and Franz. Findings are summarized after evaluating both non-inferencing and inferencing RDF-triplestore queries.

Section 3. identifies open issues in the processing flows and query patterns developed in the first two deliverables and then identifies the areas within the RDF-Triplestore that are most likely to present performance and storage challenges to the production Triad/Metasky system.

DESCRIPTION OF JOINT STUDY TEST FRAMEWORK

Our test and evaluation framework consisted of: an instance of the 1. ATL Sedona2 media analysis platform, 2. a Franz Inc. AllegroGraph 3.2 RDF-Triplestore server at Franz, and 3. ATL, and the Metasky/TRIAD servers with both AllegroGraph and Redland/Postgres RDF-Triplestore systems running on a Metasky server. This is shown and described in detail below.

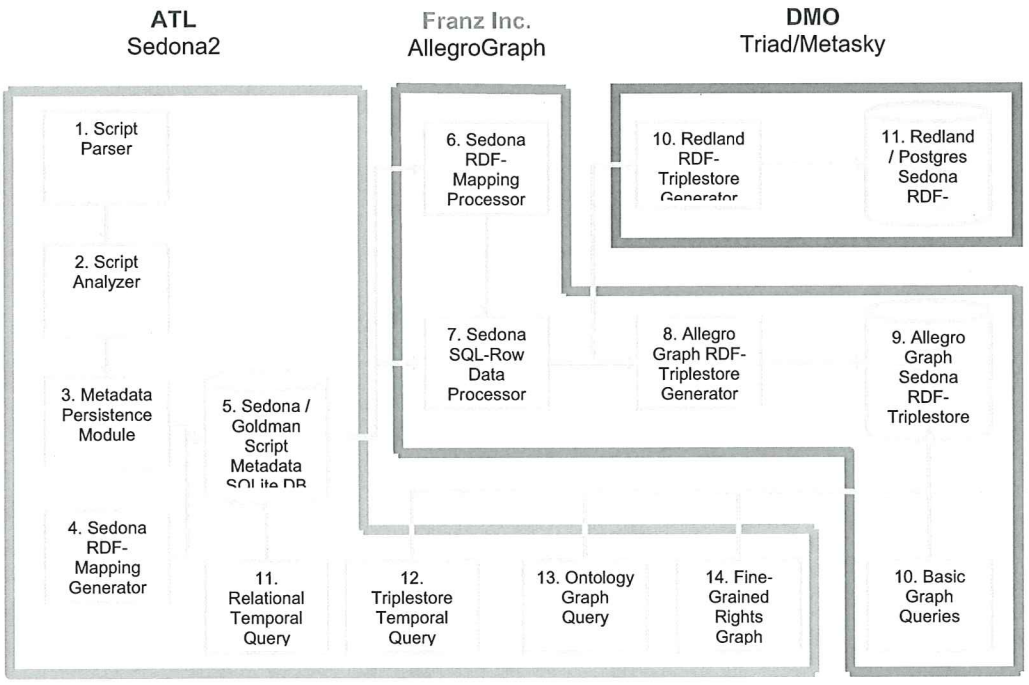


Figure 1. Joint Study System Framework

ATL Sedona2 Components

In the ATL portion of the test framework, Hollywood spec. movie scripts are input into the Sedona2/Goldman Script Parser and Analyzer components (1 & 2) and extracted script metadata was then persisted by a Sedona database module (3) into a Sedona metadata relational database (5). In order to automate the process of migration of the Sedona relational database to the target RDF-Triplestores, a mapping generator (4) is used to collect relevant relational database schema information which then writes this schema metadata into the Sedona relational database (5) for use by Franz in migrating relational rows into the AllegroGraph triplestore..

Franz Inc. AllegroGraph Components

In the Franz portion of the test framework, a Sedona relational schema and mapping interpreter was developed by Franz (6) and used to process relational databases provided by Adobe to determine how relational database rows of each Sedona table (or any table) should be mapped into RDF-Triples. The mapping information from this step is used to drive a row data processor (7) that reads the selected rows from the provided relational database. After table rows are read, a custom triple generator, persistence and indexing module (8) creates triples using PK, FK, and data columns from the source records. Triples are then stored and indexed in the AllegroGraph RDF-Triplestore (9).

DMO Triad/Metasky Components

In the Triad/Metasky portion of the system, both AllegroGraph and Redland/Postgres RDF-Triplestore systems were deployed and evaluated. Triad/Metasky engineers took the Sedona relational-to-RDF-triple mapping generator (4) and test databases (5), and the Franz mapping processing elements (6), (7), (8). In order to maintain vendor neutrality, Triad/Metasky engineers then re-implemented the Franz triple generation component (8) and created a generic triplestore generator (10) by adding a vendor – neutral triplestore persistence API supporting implementations using either AllegroGraph RDF-Triplestore, or Redland/Postgres (11).

Additional code provided by Franz for the query experiments in the joint study were a set of non-inferencing and inferencing graph query patterns (10) to evaluate query functionality and performance. Using the basic Franz query patterns, ATL then developed and characterized a set of temporal queries (11), (12), ontology queries (13), and fine-grained rights (14) queries.

Terminology:

RDF-Triplestore - A triplestore is specialized database for the storage and retrieval of Resource Description Framework (RDF) metadata. Like relational databases, information stored in a triplestore is retrieved using a query language. Unlike a relational database, a triplestore is optimized for the storage and retrieval of graph structures made up of many short statements called triples. Some triplestores can store billions of triples. The performance of a particular triplestore can be measured with the Lehigh University Benchmark (LUBM), or with real data from UniProt.

RDF-Triple – a triple of RDF values or slots consisting of a Subject, Predicate, and Object, typically written as (s, p, o). RDF-triplestores are frequently implemented as quadstores, with the addition of a named graph ID yielding (s,p,o,g), and AllegroGraph is actually a quintstore with the inclusion of a unique triple ID for each triple resulting in a 5-tuple (s,p,o,g,i).

Triad – The RDF-triplestore component of Metasky which provides a generic triplestore API. Both K., Stewart and J. Fieber have pointed out the importance of a vendor neutral implementation.

PK and FK – The Primary Key indicates the unique identity of a relational database record (also called a row) within a table. The Foreign Key indicates a reference from a relational database child record to it's parent record.

SECTION 1 – PERFORMANCE AND VOLUMETRIC ANALYSIS

In the first phase of our study, existing Sedona2 relational databases for Adobe Story (Goldman) were migrated to the AllegroGraph RDF-Triplestore to evaluate metadata persistence performance time and the resulting storage overhead. To interpret the results of our performance and volumetric experiments during this migration, the relational to triplestore mapping methodology is now described.

Our approach was to take the Sedona2 / Goldman relational movie script schema and then develop a direct mapping mechanism to convert the rows within each relational database table into explicit RDF-triples for storage into the RDF-Triplestore. The key elements of the Sedona relational database schema that were migrated into the AllegroGraph RDF-Triplestore are shown below.

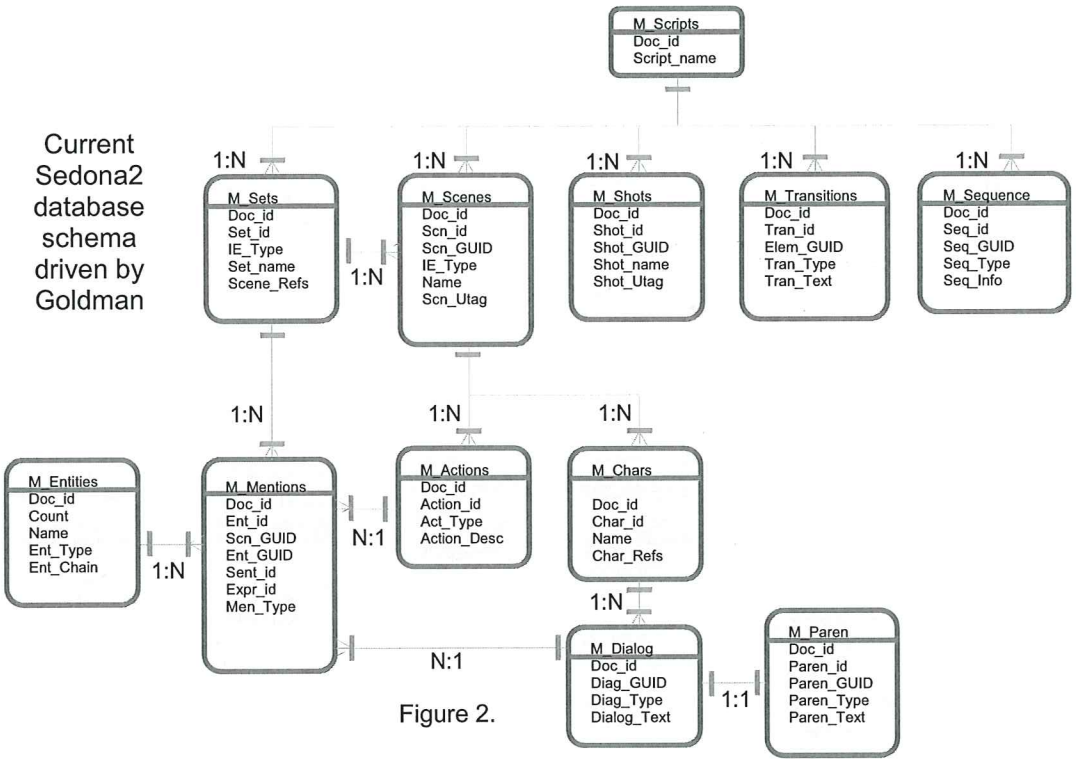


Figure 2.

After discussion with Franz and Metasky engineers, we developed a generic approach to serve all current and future ATL and CSBU relational database migrations into RDF-Triplestore(s); the mapping mechanism is general and data-driven not requiring explicit custom procedural logic for each relational application being migrated. The migration process can run in both real-time and in batch mode.

In the actual implementation (see submitted Adobe ATL Patent P921), a minimum schema graph model is constructed that includes all necessary relational per row PK, FK, and column definitions which determine the RDF-triple mappings. The schema graph is then systematically traversed and for each schema node encountered within the graph, each row of the corresponding table is expanded into M distinct (s,p,o) entity triples between the PK column(s) and each of the data columns of that row using mapping rules encoded in the schema graph node. This is shown in the diagram below.

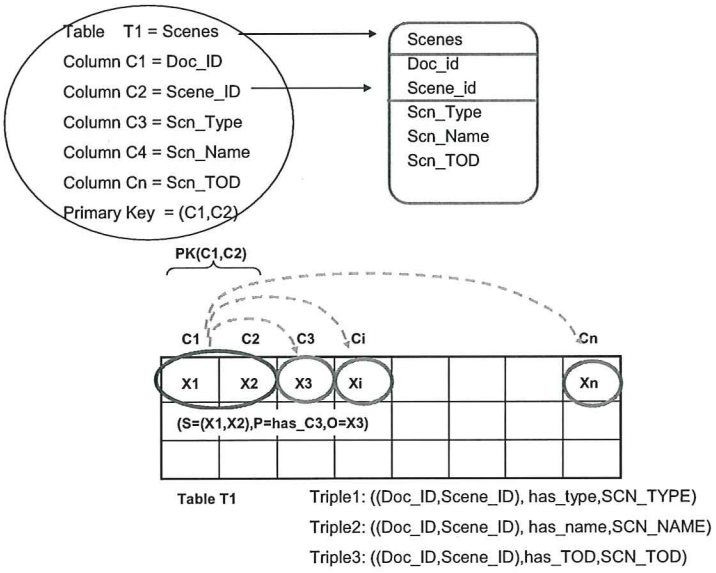


Figure 3 - Simple Relational Record to RDF-Triple mapping approach

A triple expansion/mapping list for each table is created that describes how to map and convert the N fields of any table row to M RDF-triples. Prior to the actual mapping, the ATL mapper creates one descriptor per relational table node for each of the P tables to be mapped.

In the case of PK/FK referential integrity constraints between multiple tables, a generic graph adjacency list object is created that indicates which table rows are connected to other rows through their respective PK/FK relationships. For each of pair of (PK,FK) RI constraints, the mapper creates a "link" RDF-triple between the corresponding parent-child tables.

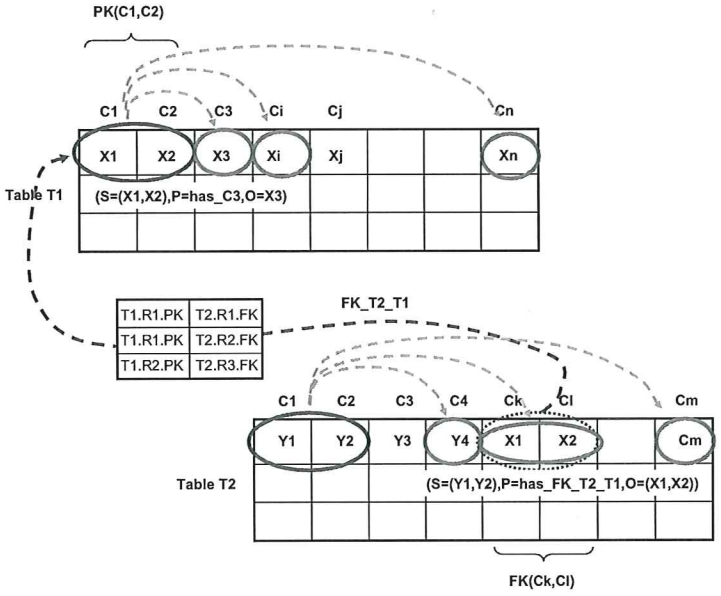


Figure 4 - Relational PK/FK constraint to RDF-Triplestore mapping

The relational to RDF-triplestore mapping system developed by ATL was submiited and accepted for patent filing and as part of Sedona2 Mesa, transferred to the DMO Triad/Metasky team. Franz Inc. provided an AllegroGraph-specific backend implementation to process the Sedona relational schema metadata in order to generate actual triples for AllegroGraph. The Metasky/Triad team then took the Franz triple generation source code and then generalized the persistence functions that used the AllegroGraph API calls so that the approach would work for Redland/Postgres.

Steps 1.4 – 1.9 in Figure 1. show the modules built as a result of the joint study to accomplish migration of Goldman/Sedona script metadata from relational format into the AllegroGraph triplestore. Steps 1.4 – 1.7 and 1.9 – 1.10 show the steps used for the migration of script metadata into Redland/Postgres.

RESULTS OF PERFORMANCE AND VOLUMETRIC EXPERIMENTS

In order to test and evaluate the relational database to RDF triplestore conversion approach, ATL provided Franz Consulting with Sedona2 SQLite databases constructed by running three Hollywood movies scripts through the Sedona2 script analysis workflow. The three movies used for this experiment were:

1. Indiana Jones – Raiders of the Lost Ark
2. Indiana Jones – The Temple of Doom
3. Indiana Jones – The Last Crusade

Each script was processed by the Sedona2 PIMA script capture and parser, then run through the TUCSON script analyzer and MESA entity extractor. All metadata was then persisted into a Sedona2 SQLite relational database.

Additionally, the triple mapping invention was used pre-populate the Sedona relational database with all of the needed record-to-RDF-triple mapping information so that script record triples and PK/FK link triples could be automatically generated. Franz Consulting then developed an AllegroGraph-specific program to process the ATL Sedona schema metadata and mapping data to create RDF-triples, persist and index these into the AllegroGraph RDF-Triplestore. In performing the direct mapping, all table rows were expanded into an average of 5 RDF-triples/row using the method outlined. (The table with the most rows expanded into 10 RDF-triples per row.) Additionally, for each anchor triple consisting of the row PK, separate FK and PK link triples were created for maintaining the referential integrity constraints of the document containership model.

STORAGE AND PERFORMANCE RESULTS

For generating both the Sedona SQLite relational metadata database and the corresponding AllegroGraph RDF-Triplestore, a dual-core 3.2 Ghz hyperthreaded Dell P670 Windows XP server with 3G of main memory was used. Using steps 3,4,5,7,8,9 in the workflow described in Figure 1., all Sedona2 script records were mapped to triples and then persisted into AllegroGraph. Results are shown below for Persist Times, mapping and triple conversion times are excluded:

Sedona2 SQLite Relational Database

Script Name	Records	Persist Time	
IJ-TLC	13500	3.1 sec	
IJ-TOD	14300	3.2 sec	
IJ-RLA	12000	2.7 sec	
TOTAL	39800	9.0 sec	= 4422 Records/sec

AllegroGraph RDF-Triplestore

	Triples	Persist Time	(S,P,O,G quads)
TOTAL	294081	84.4 sec	= 4669 Triples/sec

The measured persistence time for storing 40K Sedona2 Goldman script relational database rows was approx. 9 sec. For persisting the resulting 294K converted RDF-triples, 84 sec. was required. These initial results indicate that the AllegroGraph RDF-triplestore is approximately a factor of 9.4 X slower than using SQLite for persisting Sedona/Goldman metadata.

Storing the 40K relational database rows required a single SQLITE3 database file of 6.5M. To store the equivalent 294K RDF-Triples, a 15.3M RDF triplestore database (data portion) is required, plus 15.3M per index file for each of the (3) combinations of triple (s,p,o) slots that are indexed.

AllegroGraph 3.2 Files	Size	Description
Triples.data	15,316 K	Triple slot values
Index-spogi.1.280059.idx	15,316 K	Subject (S,P,O) index
Index-posgi.1.280059.idx	15,316 K	Predicate (P,O,S) index
Index-ospgi.1.280059.idx	15,316 K	Object (O,S,P) index
Others	6,300 K	Metadata & admin

Excluding the optional (G,*,*,*) indexes, the base size for the corresponding RDF-Triplestore (data + (S,P,O,G), (P,O,S,G), and (O,S,P,G) indexes) is 68.7M. This represents a total size increase of 10.6X from relational to triple representation. If optional graphID indexes (G,O,S,P), (G,S,P,O), and (G,P,O,S,I) are included, 15.3M will be added for each additional index.

Thus, the resulting triplestore database is approximately a factor of 10.6 X larger than the SQLite database file needed to store the equivalent script metadata and takes 9.4 X longer to persist all script metadata. This result makes sense; the three dominant Sedona2 tables M_Entities, M_Mentions, and M_Tcs_Align_Pts contain in total about 27K relational rows with each row expanding into an average of 9 triples per row or roughly 80% of all generated triples.

We note that currently, no performance or storage optimizations for AllegroGraph are being performed. Section 3 discusses various optimizations that may be possible. Franz indicates that performance may be significantly better using the 64 bit version of AllegroGraph. On the other hand, the Sedona use of SQLite is relatively well optimized; the SQLite engine itself is highly optimized due to product maturity.

Our expectation is that on the same OS and hardware platform, triple persistence performance for Triad/Metasky using Redland/Postgres should be somewhere between SQLite and untuned AllegroGraph performance values due to Redland fewer indexes. (See Section 3 for more analysis.)

While out of the original scope of the joint study, it would be useful for Triad/Metasky to capture similar measurements to compare relational-to-triple conversion and persistence performance using Redland/Postgres vs. AllegroGraph.

SECTION 2. IDENTIFICATION AND EVALUATION OF HIGH UTILITY QUERIES

A second key focus area for the joint study was the identification and characterization of anticipated triplestore queries that Metasky/Triad client applications were expected to issue. These queries fall into two categories: non-inferencing and inferencing. Both types of queries consist of zero or more filtering predicates connected by Boolean AND/OR operators. Due to the syntactic similarity of the SPARQL RDF-Triplestore query language to relational SQL, one observation is that both non-inferencing and inferencing SPARQL triple queries structurally resemble traditional SQL queries. Canonical SPARQL query templates provided by Franz. Inc. for the non-inferencing query examples provided by ATL are listed below.

Non-inferencing queries functioned as expected and exhibited comparable execution query performance when compared to similar SQLite queries. We note that AllegroGraph takes advantage of the presence of default (S,P,O,G,I) and (O,S,P,G,I) indexes for every query, while the equivalent SQLite query may not get the direct benefit on a B-tree index unless a DBA has explicitly created the relevant indexes. For completeness, the basic equivalent non-inferencing query patterns provided by Franz for specific Sedona2 relational queries are provided in Appendix 1.

2.1 TEMPORAL QUERIES

One non-inferencing Triad/Metasky query worth discussing is a temporal range query – i.e., finding all triples that contain temporal metadata such as timecode information within a specific time range. We anticipate that temporal range queries will be important and frequent, e.g. to find all metadata elements within the time boundaries of a movie scene for Premiere, or within a movie shot for OnLocation. For a relational database using a Metadata_Elements_Table to store each metadata element as a record, the following SQL query pattern is used.

2.1.1 SQL for finding metadata elements within a time interval

To locate all records with timecodes that occur between two given timecodes (00:30:00:00, 00:39:00:00), the following query is used:

```
SELECT E_BEGIN_TIMECODE, E_END_TIMECODE, ELEM_NAME, ELEM_ATTRIB1, ELEM_ATTRIB2, ...
  FROM Metadata_Elements_Table
  WHERE
        (E_BEGIN_TIMECODE > '00:30:00:00' and
         E_BEGIN_TIMECODE < '00:39:00:00')
ORDER BY  E_BEGIN_TIMECODE;
```

We list this SQL example for comparison against an equivalent time interval range query written in SPARQL.

2.1.2 SPARQL for finding triple elements within an exclusive time interval

For RDF-triplestore time interval queries, Franz Inc. Consulting provided us with the following sample SPARQL template:

```
SELECT ?s ?p ?o2
WHERE
{
      ?s <"""+BASE_URI+DB_NAME+"""/m_tcs_align_pts/timecode> ?o1 .
      ?s ?p ?o2 .
      FILTER(
         str(?o1) > str('00;30;00;00') &&
         str(?o1) < str('00;39;00;00')
      ) .
}
ORDER BY ?o1 LIMIT 500000
```

This query finds all timecode elements that lie exclusively between 0:30:00 and 0:39:00 mm:ss. When these two queries are run on the same Dell P670 server, the relational query will return all table rows where the timecode value satisfies the query conditions. When the SPARQL query is run, all triples that satisfy the query conditions are retrieved and then using the Doc_ID and element GUID fields, all related triples (that contain the equivalent row attributes) are subsequently retrieved. The following times were measured:

QUERY RESULTS:

SQL Rows:	91	Time: 31.2 msec.	Notes: C/C++
RDF Triples:	959	Time: 150.0 msec.	Notes: Python

Since each relational row is expanded into a larger number of RDF-triples, the 1:10 increase in the number of triples compared to the number of rows is expected. However, the RDF-Triplestore must do more work since it must perform approx. 10X the number of index accesses to retrieve all associated attributes. See Section 3 for a more detailed discussion of the trade-offs in using a relational vs. triplestore model for temporal metadata.

An additional comment is that we expected that inclusive range queries would work using <= and >= operators but this turned out initially not to be the case. A suspected query expansion bug in the AllegroGraph engine actually returned all timecode values when inclusion comparison operators are used. The following ATL workaround was developed and verified to work around the problem:

SPARQL:

```
SELECT ?s ?p ?o2
WHERE
{
   ?s ?p ?o2 .
    ?s <"""+BASE_URI+DB_NAME+"""/m_tcs_align_pts/timecode> ?o1 .
    FILTER
    (
       (str(?o1) > str('00;30;00;00') || str(?o1) = str('00;30;00;00')) &&
       (str(?o1) < str('00;39;00;00') || str(?o1) = str('00;39;00;00'))
    ) .
}
ORDER BY ?o1
LIMIT 500000
```

This bug was reported to Franz Inc. and a patch fixed the problem. Correct query behavior for this case should be verified in the Redland/Postgres query engine as well.

GENERALIZED TEMPORAL QUERIES

Also of particular interest are temporal metadata queries that are multimodal, i.e., queries that select different modes or channels of metadata within a specific time window. E.g., movie scripts that are time aligned with timecode information in speech-to-text documents will provide accurate time information for when scenes, characters, dialog, and movie objects appear within a video. With the addition of other video analyzers such as face-detectors, scene classifiers, and object detectors, new types of time-coded metadata are available to the system.

An anticipated use of the RDF-Triplestore will be to store all time-coded multimodal metadata and then use this information independently and in conjunction to search for specific video content, or to perform semantic analysis by considering co-occurring multimodal features to determine the major themes or “aboutness” of a specific scene or shot.

Multimodal time-coded metadata from video analyzers consist of feature vectors that are represented as RDF-triples. A feature vector has the form:

Multimodal FV = (N, T, C, BT, ET)

N = Feature name, e.g., "bat"
T = Feature type, (optional) e.g., "mammal"
C = Feature confidence, e.g., 0.30
BT = Feature begin time, e.g., "00:23:12:06"
ET = Feature end time, e.g., "00:23:47:15"

The diagram below shows examples of multimodal feature vectors that are extracted from video content and then stored into the RDF-triplestore metadata repository.

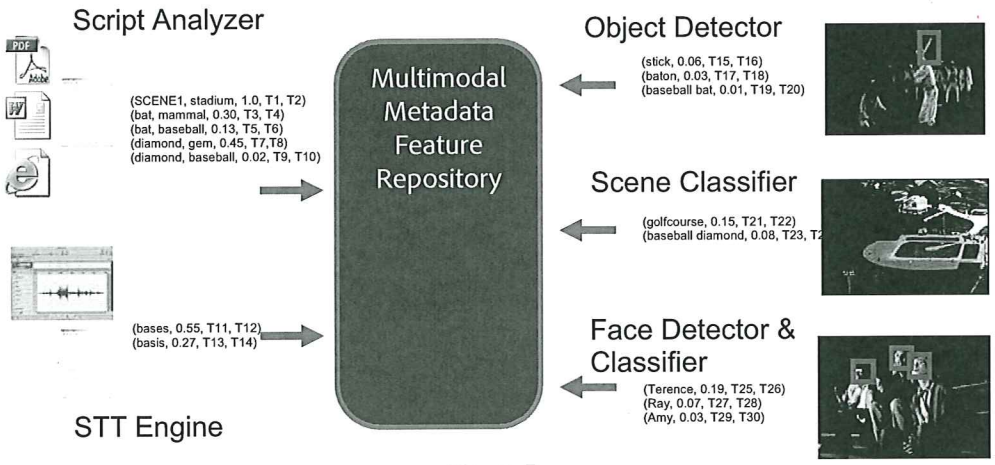


Figure 5.

Expanded forms of the query pattern from 2.1.2 are used to retrieve all triples whose timecode value indicates that the particular feature appears within a time interval of interest (typically a SCENE or SHOT)

```
SELECT ?s ?p ?o2
WHERE
{
     ?s <"""+BASE_URI+DB_NAME+"""/feature_vector/timecode> ?o1 .
     ?s ?p ?o2 .
     FILTER(
        str(?o1) >= str(BEGIN_TIME) &&
        str(?o1) <= str(END_TIME)
     ) .
}
ORDER BY ?o1 LIMIT 500000
```

In this query, the 1st WHERE expression retrieves all triples and applies the time interval comparison in the FILTER() expression, the 2nd WHERE expression finds all triples using the subject value (the ID of the Feature Vector whose timecode was qualified by the FILTER) to find all triples that contain attributes of the FV. Note: Mike Welch's ATL Summer Project will be focused on addressing the multimodal video search problem and developing similar queries to perform logical and statistical inferencing.

2.2. XMP GRAPH QUERIES

Another high frequency application operation that we anticipate will be to persist XMP metadata documents into the RDF-Triplestore and then retrieve those documents. RDF-triple persistence is accomplished using a method similar to the one developed for persisting any relational schema. In the case of XMP documents, each parent node connects to children nodes via TAG or ATTR relationships and can be expressed as RDF-triples of the form (PARENT-NODE, RELATIONSHIP, CHILD-NODE.).

The set of all triples formed this way makes up the DOM for a given document. If we are interested in retrieving all triples that form an XMP graph (or more generically, any directed acyclic document graph), Franz provides SPARQL queries that perform a complete traversal of the document graph and return all nodes in the entailment regime of the DOM structure.

2.2.1 Find entire XMP document graph from a specified root node R1

The diagram below presents a simplified XMP document graph for three documents rooted at R1, R2, and R3. All three of the documents logically share a common substructure: B1 --> B3.

```
XMP Document Query Graph G


              R1       R2   R3
            /L   \       \    \
          A        B      C     D
        /   \     /   \  /     /
      A1    A2  B2     B1----'
                      /
                    B3
```

```
SPARQL:
          PREFIX ex: <http://example.org/>
          SELECT ?node
          WHERE {
                 ex:R1 ex:L ?node
          }
```

```
QUERY RESULT:

node = http://example.org/B
node = http://example.org/A
node = http://example.org/B3
node = http://example.org/B2
node = http://example.org/B1
node = http://example.org/A2
node = http://example.org/A1

100 Root full transitivity queries took 0.640 sec. or 6.4 msec. / doc query
```

The above example performs a traversal of G to find all internal and external nodes of an XMP graph rooted at R1. While the SPARQL expression is compact, we currently see no simple way of explicitly retrieving all triples from a specified node X that forms a specified DOM subgraph without the use of PROLOG.

The SPARQL primitive "CONSTRUCT" uses triple pattern matching and can retrieve all triples for a given graph. However, this will not allow us to select the target subgraph unless the subgraph of interest uses distinguished predicate values (links) or uses specific graphIDs. Bill Milar has recently indicated that extensions to the AllegroGraph reasoner can be enabled that may allow subgraph DAGs to be selectively retrieved.

2.2.2 Find only leaf nodes of XMP DOM graph

In some cases, it may be desirable to find only the leaf nodes of the XMP DOM. This is accomplished by the addition of the following OPTIONAL/FILTER clauses to the 2.2.1 query which filters DOM nodes within internal triples.

```
SPARQL:
            PREFIX ex: <http://example.org/>
            SELECT ?leaf
            WHERE {
                   ex:R1 ex:LX ?leaf .
                   OPTIONAL { ?leaf ex:LX ?nope }
                   FILTER (!bound(?nope))
            }

QUERY RESULT:

leaf = http://example.org/B3
leaf = http://example.org/B2
leaf = http://example.org/A2
leaf = http://example.org/A1

100 Root terminal transitivity queries took 0.907 sec. of 9.07 msec. / doc query
```

Examples of where this type of query would be useful are applications that need only the data values to specific leaf nodes, e.g., retrieving all dialog text from a script in order to perform script and STT alignment.

2.2.3 Finding all XMP document root nodes from a given node value.

When Triad/Metasky is used as a CMS for XMP metadata documents, another high frequency search operation will be to locate all XMP documents that contain specific values (e.g., author, keyword, playback FPS, etc.) To satisfy these queries, the RDF-Triplestore system must perform an inverse traversal beginning at an internal or leaf node and then retrieve all root terminal nodes transitively connected to each retrieved query value.

```
Multiroot Query Graph G:

              R1       R2   R3
            /L   \       \    \
          A        B      C     D
        /   \    /   \  /     /
      A1    A2  B2     B1----'
                      /
                    B3
```

Let R1, R2, and R3 represent the root nodes of specific XMP documents persisted into the triplestore, and if a query value of "B1" is specified corresponding to the B1 node in the diagram above, the following SPARQL can be used to find only root nodes with a path to B1.

```
SPARQL:
            PREFIX ex: <http://example.org/>
            SELECT ?root
            WHERE {
                   ?root ex:L ex:B1 .
                   OPTIONAL { ?none ex:L ?root }
                   FILTER (!bound(?none))
            }

QUERY RESULT:

valid root terminal= http://example.org/R3
valid root terminal= http://example.org/R2
valid root terminal= http://example.org/R1
```

```
100 B1 terminal transitivity queries took 0.922 sec. or 9.22 msec./query
```

Franz provides the following commentary on the query pattern: "In the SPARQL query below, the first clause exploits the transitivity of relational "L" to find all L-ancestors of B1. The OPTIONAL clause tries to find an ancestor ?none for each candidate ?root node, and the FILTER clause eliminates solutions wherein ?root has an ancestor, leaving only the legitimate root nodes."

We observe that this query runs in approximately the same time as the unconstrained version. In general, for larger graphs, queries with OPTIONAL/FILTER constraints take longer than the unconstrained versions due to additional checking. This is slightly counter intuitive when compared to relational database in which query WHERE clause constraints provide the query optimizer a chance to select B-Tree indexes to accelerate the query. In network deployments, triple filtering at the server would be more desirable than sending excess triples over the network only to be filtered by the client.

2.2.4. Finding all XMP root and internal nodes from a given node value

To find all XMP document nodes, the OPTIONAL/FILTER clauses are now removed from the original query resulting in simple transitive select starting at node "B1".

```
SPARQL:
            PREFIX ex: <http://example.org/>
            SELECT ?node
            WHERE {
                   ?node ex:L ex:B1 .
            }

QUERY RESULT:

valid path node = http://example.org/C
valid path node = http://example.org/B
valid path node = http://example.org/D
valid path node = http://example.org/R3
valid path node = http://example.org/R2
valid path node = http://example.org/R1

100 B1 full transitivity queries took 0.578 sec.
```

Repeating the experiment to find all XMP root and internal nodes from a different given node value B3 produces a larger set of DMO nodes that make up the transitive closure of path nodes to all root nodes.

```
SPARQL:

            PREFIX ex: <http://example.org/>
            SELECT ?node
            WHERE {
                   ?node ex:L ex:B3 .
            }

QUERY RESULT:

valid path node = http://example.org/B1
valid path node = http://example.org/R3
valid path node = http://example.org/R2
valid path node = http://example.org/R1
valid path node = http://example.org/D
valid path node = http://example.org/B
valid path node = http://example.org/C

100 B3 full transitivity queries took 0.594 sec.
```

This set of queries shows that an RDF-Triplestore used to store XMP metadata documents can be directly searched to locate all documents that contain a specified set of attributes.

If the above example is scaled for real-life production usage, millions of XMP documents may be present within the system. For very large repositories, distributed *federated* triplestores may be required. These can be constructed using local clusters of AllegroGraph servers, or cloud-based deployments. Additionally, these queries can be further optimized by using the GraphID slot present in AllegroGraph RDF-triples to directly indicate the XMP root node by providing an additional mapping from GraphID to XMP document root UUID.

INFERENCING QUERIES

The previous set of queries addressed graph traversal operations. Inferencing in RDF-Triplestores is accomplished by computing *transitive closures* over specific relations of the RDF graph and in fact can be accomplished by the same kind of graph traversal query. Within the scope of RDF, transitive closures over graph relationships are equivalent to RDF *entailments*. Thus, RDF-Triplestores perform inferencing over triples by providing functions that compute the *entailment regime* for a set P of propositions (each represented by a triple) contained within a particular subgraph of the triplestore. In formal logic (from Wikipedia), an entailment regime is equivalent to logical implication and is a logical relation that holds between the set T of propositions and a proposition Q (the consequent inference). In symbolic form:

Eq. 1: $T \Longrightarrow Q$

and is read as "T implies Q", "T entails Q", or "Q is a (logical) consequence of T". In such an implication, T is called the antecedent, while Q is called the consequent.

In other words, Eq. 1 holds when the class of graph structures of T are a subset of the class of graph structures of Q. Without using the language of formal mathematical structures, Eq.. 1 states that the conditional formed from the conjunction of all the elements of T and Q (i.e. the corresponding conditional) is valid. That is, it is true that:

Eq. 2: $(P_1 \wedge P_2 \wedge .. P_i \wedge P_n) \Longrightarrow Q$

Where the P_i are the propositional elements of T.

E.g., Let the set P of sentences include:

a minivan is a type of car
a car is a type of motor vehicle
a motor vehicle is a type of self-propelled vehicle
a self-propelled vehicle is a type of wheeled vehicle.

If the set Q of sentences includes the following:

a minivan is a type of motor vehicle,
a minivan is a type of self-propelled vehicle
a minivan is a type of wheeled vehicle

Then P ==> Q, i.e. P entails Q, holds.

More from Wikipedia: Expressing entailment relationships between RDF graphs is the key idea which connects model-theoretic semantics to real-world applications. An assertion amounts to claiming that the world is an interpretation which assigns the value TRUE to the assertion. If P entails Q, then any interpretation that makes P true also makes Q true, so that an assertion of P already contains the same

"meaning" as an assertion of Q; one could say that the meaning of Q is contained in, or subsumed by, that of P. If P and Q entail each other, then they both "mean" the same thing, in the sense that asserting either of them makes the same claim about the world.

Within both entity-relational and relational databases, graph schemas may also be created to support triplestore structures used to form semantic networks. However, unless graph based traversal constructs are available, such as e.g., Oracle's SQL CONNECT BY / START WITH extension, traversal of the graph schema to compute transitive closures is cumbersome and requires table row navigation using procedural SQL.

2.3 ONTOLOGY QUERIES

Using ontologies to make inferences is expected to be a frequently performed classification operation for video entities and for scene elements within movie documents such as scripts. In particular, a common operation is to find the entailment regime or transitive closure relationship chains for the supertypes for a specific entity extracted from a script, STT or video scene. Building on the earlier example, an ontology can inform an application that a set of inferences using the search term "minivan" is present and can be used to determine that:

a minivan is a type of car
a car is a type of motor vehicle
a motor vehicle is a type of self-propelled vehicle
a self-propelled vehicle is a type of wheeled vehicle etc.

In many cases, the term supertype/subtype lexicon structure is not purely tree structured but instead a hierarchical directed acyclic graph. Here multiple paths may exist between parent supertype terms and children terms. In this particular case, the inference query must navigate and return all possible inferences within the entailment regime that are formed when multiple paths exist. One of the example queries created for testing the inference queries in AllegroGraph consists of all entailments possible for the term "chalet." In particular, we observe:

a chalet is a type of house
a house is a type of building
a building is a type of structure

Additionally, the WordNet lexicon ontology used for this example has a particularity in that multiple facets are present (resulting in a directed acyclic graph) since the ontology is also attempting to capture functional aspects of chalets and as well as a strict classification taxonomy. The ontology expresses this additional functional facet with the following statements:

a house is a type of dwelling
a dwelling is a type of housing
a housing is a type of structure

Support for multipath/multifacet ontologies with D.A.G. structures is especially important as these types of semantic structures frequently occur in real-world ontologies.

The basic query pattern for entailment regime retrieval provided by B. Millar was modified to query an ATL-built RDF-Triplestore sub-ontology for "structures" by beginning at the root term "entity" just above "structure", and then by finding the transitive closure of all relations between the subroot node and all directly or transitively connected children nodes. This was enabled by adding a TRANSITIVE property which enables retrieval transitivity for each predicate link of the ontology:

```
repoconn.addTriple(ts_link,RDF.TYPE,OWL.TRANSITIVEPROPERTY,None)
```

RESULTS OF THE ONTOLOGY QUERY:

A 2296 term sub-ontology was created by loading AllegroGraph with all WordNet terms rooted to the term "structure". The result is a directed acyclic ontology subgraph that covers all facets of the "chalet" term (See Appendix 2 for a listing of the actual relevant subontology.) Given this ontology, two types of ontology inference queries are of interest:

1. finding all entailment nodes from a specified root term
2. finding all possible inferences from a specific leaf or internal entity

Query 2 is of high importance because ontology-based classification uses the inferences generated by executing a query of this type. Query patterns for both types are presented and tested in the next two sections.

2.3.1 Finding all entailment nodes from a specified root (e.g., "entity")

We anticipate in some instances applications will need to retrieve portions of a given ontology for displaying portions of the ontology or for creating controlled vocabularies for export. With the TRANSITIVE property turned on, the following query performs this function.

SPARQL:

```
PREFIX ex: <http://atl.corp.adobe.com/joint_study/test_ontology/>
SELECT ?leaf
WHERE {
    ex:entity ex:LINK ?leaf
}
```

QUERY RESULT:

[2295 result queries not shown here to save space.]

Entity full entailment Ontology query time = 532 msec., 2295 ontology node elements retrieved.

Performance for this query is acceptable for interactive applications but without parallelism or clustering, may present high response latency if multiple applications were issuing similar queries simultaneously.

2.3.2 Finding all possible inferences from a specified lead node (e.g., "chalet")

In ontology-based classification, content terms are used to probe an ontology to find a chain of assertions that typically increase in generality with each inference (e.g., minivan is a car, car is a motor vehicle, etc.)

The basic query pattern for retrieving an ontology entailment regime was modified to query an ATL built RDF-Triplestore sub-ontology and to begin at the "chalet" leaf term, and then to find the transitive closure of all relations between the leaf to the root of the ontology ("entity").

SPARQL:

```
PREFIX ex: <http://atl.corp.adobe.com/joint_study/test_ontology/>
SELECT ?parent ?lnk ?child
WHERE {
    ?parent ex:LINK ex:chalet
}
```

QUERY RESULT:

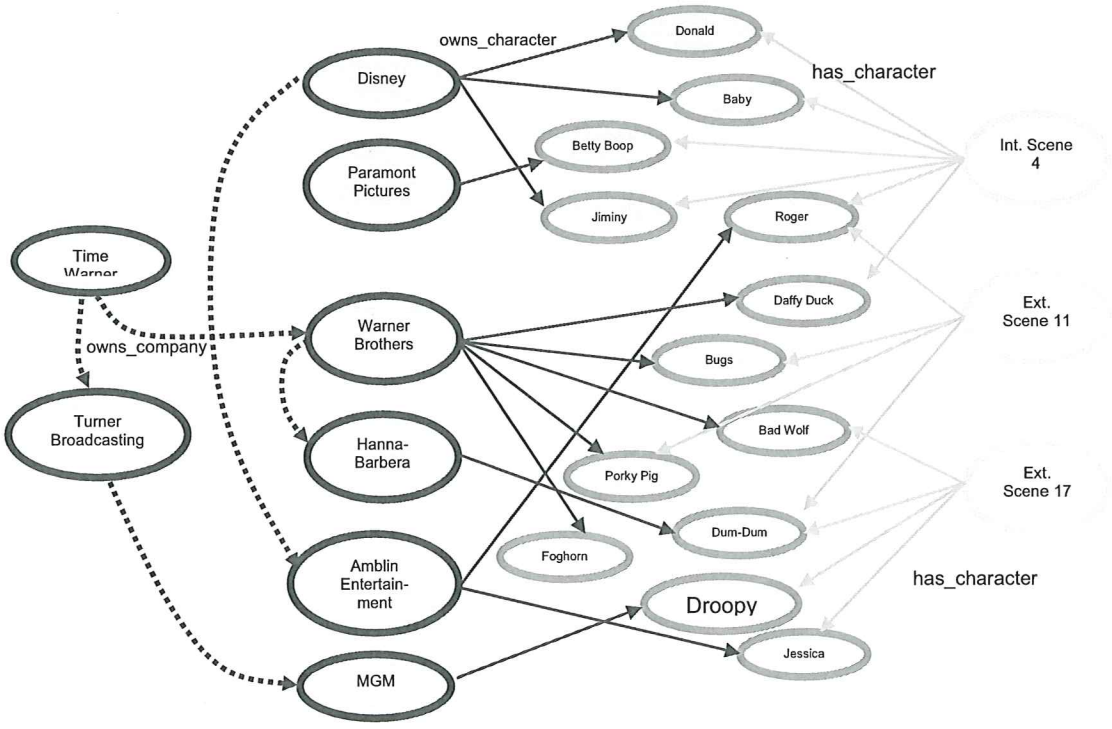
[1] http://atl.corp.adobe.com/joint_study/test_ontology/house
[2] http://atl.corp.adobe.com/joint_study/test_ontology/house (from second faceted path)
[3] http://atl.corp.adobe.com/joint_study/test_ontology/entity
[4] http://atl.corp.adobe.com/joint_study/test_ontology/structure
[5] http://atl.corp.adobe.com/joint_study/test_ontology/housing
[6] http://atl.corp.adobe.com/joint_study/test_ontology/building
[7] http://atl.corp.adobe.com/joint_study/test_ontology/dwelling

Chalet Ontology query (100 runs), time = 0.672 sec. or 6.72 msec. / full leaf-directed entailment regime.

There are three important findings from this experiment: 1) that full entailment sub-ontology retrievals are easy to express but relatively expensive to execute, 2) that OPTIONAL / FILTER constructs to remove non-terminal (internal) transitive paths are expensive and in fact, for non-networked configurations, it may be better to perform full entailments and then filter internal nodes at the application level, and 3) computing entity classification inferences (e.g., IS_A relationships) is relatively easy to express and efficient. This is important because we expect 3) to be the dominate case.

2.4. FINE-GRAINED DRM QUERIES

A closely related inferencing application is determining the direct and transitive rights held by particular parties on objects within a video or movie script. Sedona2 analysis of scripts allows for precise fine-grained identification of entities such as characters or props within specific scenes or within a certain time interval. When this information is augmented by a graph of rights holders, Triad/Metasky applications can readily determine all of the objects a rights holder currently has usage rights to. Additionally, for a given set of selected objects (e.g., a character in a specific scene), an application can determine who the direct and transitive rights holders are. A movie object rights graph can be constructed to model the relationships described above:



In the example above first suggested by M. LeBrun, for the specific movie "Who Framed Roger Rabbit", business entities such as motion picture companies will directly own the rights to specific movie cartoon characters and/or own other companies. In this diagram, assume that Time-Warner directly owns Turner Broadcasting and Warner Brothers. Further, Turner Broadcasting owns MGM which is the original owner of usage rights to the cartoon character "Droopy".

Entailment queries were constructed to determine the set of business entities that collectively own rights to the Droopy character, or conversely, given a business entity such as Time-Warner, determine all characters that Time-Warner owns rights to. When intersected with logical scene or time interval occurrence information, the Triad/Metasky system can provide precise and comprehensive fine-grained rights management information for complex video content.

RESULTS OF FINE-GRAINED RIGHTS QUERY

The above rights ontology for Who Framed Roger Rabbit was created in the RDF-Triplestore. An entailment query was issued to determine all of the business entities that have ownership rights for the MGM character Droopy. The results are summarized below:

2.4.1 Finding which root-level business entities own rights to Droopy

SPARQL:

```
PREFIX ex: <http://example.org/>
SELECT ?root
WHERE {
    ?root ?pred ex:Droopy .
    OPTIONAL { ?nope ex:owns_company ?root }
    FILTER (!bound(?nope))
}
```

[0] root = http://example.org/Time Warner
[1] root = http://example.org/Time Warner

100 terminal leaf to root transitive rights query took 0.922 sec. or 9.22 msec./queries.

When this query is run, AllegroGraph finds the correct root-level business entities that own rights to Droopy, however, we also noticed that duplicate values were being returned. This is an issue we will need to explore further to determine if the query is missing one or more constraints, or whether the AllegroGraph query engine has a bug.

2.4.2 Finding all business entities which own rights to Droopy (direct and transitive)

A related query is to determine all root-level and intermediate business entities that own rights to the Droopy character. This is accomplished by removing the OPTIONAL/FILTER clause.

SPARQL:

```
PREFIX ex: <http://example.org/>
SELECT ?root ?pred
WHERE {
    ?root ?pred ex:Droopy
}
```

[0] root = http://example.org/Metro-Goldwyn-Mayer
[1] root = http://example.org/Turner Broadcasting
[2] root = http://example.org/Time Warner
[3] root = http://example.org/Turner Broadcasting
[4] root = http://example.org/Time Warner

100 full leaf to root transitive rights query took 0.781 or 7.81 msec./query

This query succeeded in retrieving the full entailment regime and yielded the correct business entities but again duplicate values were also returned. In general, performance of these types of rights queries should be acceptable for general rights reporting and ad hoc query applications.

2.4.3 Finding all properties and characters owned by entity

When multiple asset rights holders are involved, another anticipated related rights query will be to ask what are all assets the parent holding company owns rights to. A WHERE clause specifying the root-level business entity (e.g., "Time-Warner") accomplishes this.

SPARQL:

```
PREFIX ex: <http://example.org/>
SELECT ?leaf
WHERE {
    ex:Time-Warner ex:owns ?leaf .
    OPTIONAL { ?leaf ex:owns ?nope }
    FILTER (!bound(?nope))
}
```

all terminals (characters) owned by Time-Warner

leaf = http://example.org/Dum-Dum
leaf = http://example.org/Droopy
leaf = http://example.org/Foghorn Leghorn
leaf = http://example.org/Bug Bunny
leaf = http://example.org/Bad Wolf
leaf = http://example.org/Porky Pig
leaf = http://example.org/Daffy Duck

100 Root full transitivity queries took 1.188 sec. or 0.0118 msec / query.

An initial conclusion we draw from our inferencing query experiments is that the AllegroGraph RDF-Triplestore is suitable for representing and querying many of the graph-based structures that we anticipate for CSBU applications and tools. This section has demonstrated how common non-inferencing temporal range queries can be processed in SPARQL, as well as how graphs for XMP metadata documents can be created and queried. Finally, the inferencing queries presented show how an RDF-Triplestore can be used to represent and search complex multi-faceted ontologies or fine-grained rights management structures.

SECTION 3

For certain applications, RDF-Triplestore technology offers many significant advantages over relational database; these include formalized naming and identification of all content entities and attributes, a flexible graph-based data model that naturally supports complex document and ontology structures, a high degree of indexing to allow for generalized searching of the triplestore, and the ability to perform inferencing using the various assertions represented by each triple.

Triplestores have been designed and built as database engines from ground up (e.g. Franz Inc. AllegroGraph), while others have been built on top of existing commercial or open-source relational database engines as extensions, e.g., the Oracle RDF Data Store or Redland/Postgres triplestore system. Triplestores represent a relatively new storage technology while established relational database technologies have been making significant gains in transactional, analytic, and multimedia functionality, performance and scalability over the last several decades. Pure triplestore technologies are thus are an earlier stage of their maturity curve. A triplestore system built as an extension to an existing RDBMS may be able to leverage the maturity of the relational system, but will also pose new performance and storage optimization challenges due to the differences between graph data and relational data models.

If the dominant graph and inferencing query operations for Metasky are similar to the ones evaluated in Section 2, RDF-Triplestore technology will provide leverage since graph queries are significantly more difficult to express and process using relational databases and SQL. If on the other hand metadata stored in the triplestore is derived largely from relational databases, then insert, update, retrieval and query performance may be significantly slower due to increased index accesses resulting from the triplication of the original relational data.

In either case, the selection of an underlying storage technology whether it is relational, triplestore-based, or hybrid poses several important trade-offs that the systems designers and implementers will need to make. Given the advantages of using RDF-Triplestores, this section summarizes where potential performance bottlenecks, storage overhead, and design complexity issues may be introduced by use of triplestore technology and how these issues might be addressed. These are summarized below and then described in greater detail in the following subsections.

Summary of Key Issues around RDF-Triplestores:

1. Selecting hybrid relational/triplestore vs. homogeneous triplestore storage - Data and metadata can be both relational (e.g., Sedona2/Goldman, OnLocation) or graph based (e.g., XMP, classification ontologies, rights information.) Exclusively using one RDF-Triplestore representation for all Triad/Metasky data provides certain advantages but also makes several specific assumptions about the patterns of access and queries expected.

2. Reducing redundancy and overhead in the conversion of relational rows to triples – The mapping method developed and implemented by the joint study allows for efficient and direct migration of Adobe relational information but reduces overall performance and also increases storage overhead in the RDF-Triplestore independent of the underlying triplestore technology (e.g., AllegroGraph or Redland/Postgres).

3. Index Storage Optimization – On average, triple slot values will require more indexing than equivalent data representations using relational databases. While indexing can significantly accelerate search and retrieval performance, insert and delete operations will be significantly slower due to increased index accesses. Case-by-case analysis of AllegroGraph triplestore query operations will be needed to determine whether certain triple slot indexes can be

removed. (We note that the inverse is true in relational systems where we determine what column indexes need to be added.)

4. Overhead for Triple IDs – Use of the RDF-Triplestore typically requires (s, p, o) slots to be full URIs. In relational records, PK or FK values are typically integers 4 – 8 bytes in size, or a set of relatively short character strings. In comparison, average URI/URL lengths have been measured to be in the range of 30 - 137 characters. Additionally, since multiple triples are required to convey information about some multi-attribute object, the increase in ID size and redundancy of URI/URL IDs introduces non-trivial storage overhead for triplestores. AllegroGraph ID slot compression schemes partially address this overhead problem but compression mechanisms are not uniformly used by all triplestores and do not resolve the problem of ID redundancy.

5. RDF graph query performance is good when retrieved entailment regimes are small or the length of each inference chain is small and relatively few chains are retrieved. When larger graphs are retrieved, query performance will be gated by the number of triple slot indexes that need to be accessed and the locality of any caching scheme being used. RDF-Triplestore graph queries using SPARQL patterns for basic entailment queries against expected ontologies appear to be straightforward and manageable. However, more complex inferencing queries that are beyond the scope of SPARQL will require Prolog and may make query development more challenging for Adobe application engineers.

6. Relational database design and optimization tools, formal schema descriptions, and data dictionary mechanisms are more mature than their RDF-triplestore counterparts. If Adobe is standardizing it metadata repositories on RDF-Triplestore technology, we should leverage formal RDF schema design and description methodologies. Additionally, we should adopt known best practices in the design of our ontologies.

These findings are discussed in detail below.

1. HYBRID STORAGE VS. HOMOGENEOUS STORAGE

Given the results of the initial performance and storage experiment above, an important question to answer is for which situations does an RDF-Triplestore provide significant advantage over a conventional relational database. Within the context of Metasky, there are at least three important cases:

1. Providing storage and search for XMP metadata,
2. Searching over temporal metadata (e.g., timecode information)
3. Representing and searching domain-specific ontologies.

In the case of XMP /XML serialization, use of a triplestore offers strong advantages for the system implementers when the mapping from XMP/XML to triples is performed at a logical and not physical level. Physical or literal mappings of XML to triples can lead to additional triple indirection since XMP includes many low level primitives for specifying collections consisting of details that end applications are not particularly interested in. This mapping can be later optimized as these XMP collection primitives are just an artifact of Adobe's implementation of XMP. A recommended approach is to first perform a relatively direct mapping of XMP metadata, then optimize the mapping with appropriate mapping simplifications when attribute collections are involved.

Our expectation is that XMP documents, domain vocabulary ontologies, and transitive asset rights structures will need to leverage the graph traversal and query inferencing operations provided by the underlying RDF-Triplestore technology. While any of these graph structures can be expressed using adjacency list tables within a relational database, relational graph retrieval queries typically require multiple SQL SELECT statements. We do note that some relational systems such as Oracle provide

recursive hierarchical retrieval functions such as CONNECT BY – START WITH which can fetch tree or directed acyclic graph structures directly from a table with only one query.

From experience, graph queries in SQL are significantly more difficult to express and process using relational SQL queries. Typical methods include writing a procedural query to navigate over the graph structure. In comparison, the SPARQL graph query templates used in conjunction with the AllegroGraph TRANSITIVE property shown in example in Section 2 provides an elegant method of performing graph retrieval and performing inferencing.

In contrast, certain types of metadata may consist of a collection or group of data elements for which inferencing is not meaningful or necessary. When metadata is maintained in its relational form, insert, update, and retrieval performance will be significantly faster due to decreased index accesses since triplication of the relational data is avoided.

If the application mix of relational vs. graph-structured metadata is variable or unknown, a possible strategy is a hybrid approach in which both relational storage systems and RDF-triplestores are used. In this model as graph search and inferencing applications are identified for the relational portions of the metadata, these can be migrated into the RDF-triplestore. Conversely, if performance is poor and analysis of triplestore queries indicates that graph traversal and inferencing operations do not occur for known triple subgraphs, these can be migrated to a more efficient relational representation.

2. STORAGE IMPLICATIONS DUE TO RELATONAL TO TRIPLE MAPPING

A general observation is that storage systems use physical clustering of data to improve access performance; e.g., field values for a relational record are placed on a physical database page resulting in fast retrieval of all field values for the record.

Within a triplestore, maintaining physical clustering of related triples is significantly more difficult to control. The expansion of a relational row with N fields per record results in typically (N – 1) triples allowing for highly flexible searching and inferencing capabilities. However, the cost is reduced retrieval performance for any group of related triples if physical clustering cannot be maintained within data pages of the triplestore. We can more precisely estimate the storage cost of converting relational databases into triplestores by the following equations:

If the number of relational record fields = N, and each PK and FK is not composite,

$$M = \text{\# entity triples} = \sum_{i=1}^{P} \sum_{j=2}^{N} |T_i| \times ENTITY(T_i.C_1, T_i.C_j) \tag{2.1}$$

where:

$|T_i|$ = cardinality or number of rows (records) in table Ti

$ENTITY(T_i.C_1, T_i.C_j)$ = 1 if an entity/attribute triple is to be created between the key field Ti.C1 and Ti.Cj, j >1 and j <= N

P = total number of tables

$T_i.C_1$ = primary key field

Ti.Cj = all non-key fields in table (i.e., exclude PK and FKs)

N = number of fields in Table i

The number of PK/FK link triples is then given by the equation below which provides an estimator for the total number of link triples that will need to be created to preserve referential integrity.

(2.2) $$L = \#\text{ link triples} = \sum_{i=1}^{P} \sum_{j=2}^{R} |T_i| \text{ LINK } (T.Cj)$$

where:

$|T_i|$ = cardinality or number of rows (records) in table Ti

$LINK(T_i.C_j)$ = 1 if a triple is to be created between the FK field Ti.Cj and the PK of the parent Table

Ti.Cj = FK field in table, (all other Table columns/fields are excluded)

R = number of fields in Table i

P = total number of tables

Thus, a relational database with P tables and $\sum_{i=1}^{P} |T_i|$ total rows will have:

Total RDF-Triples = (M + L)

(Note: both equations were written with the assumption that for each table T, PK or FK values contain exactly 1 field. In practice, composite PK/FKs may exist in which case the equations are similar but slightly more complex and the total number of resulting triples will be slightly smaller.)

As an example, a subset of the existing Sedona2/Goldman schema consists of two tables: SCRIPTS and SCENES containing, respectively, one Movie Script descriptor record having 3 record fields, and 100 Movie Scene records each having 6 fields. Further assume each record consists of a one field PK (ScriptID and SceneID respectively), and that the SCENE table consists of a FK(ScriptID) to enforce a referential integrity constraint between SCENE and SCRIPT records.

Using Eq. 3.2.1, the total number of entity triples needed are:

M = (1 * 2) + (100 * 5)

= 502 entity RDF-triples for 101 relational records

Using Eq. 3.2.1, the total number of line triples needed are:

L = (1 * 0) + (100 * 1)

= 100 link RDF-triples

The net effect of converting any relational database into an RDF-triplestore is the generation of additional triples based on the number of non-PK fields in the relational row. In the above example, 101 relational records expanded out into 602 RDF-triples. Higher record field to triple fan-out rates will occur when even wider tables are present. This is likely one of the largest factors in predicting how a triplestore will behave after relational data has been migrated into the triplestore.

3. OPTIMIZATION OF INDEX ALLOCATION

A high fan-out of triples from each relational row has significant performance impact on triplestore Insert, Delete and Fetch performance. In relational systems, when each of these operations is applied to a single multi-field record (and if PK and FK constraints are present), typically one B-Tree index would be used for each PK, and one B-Tree would be used for each FK present.

In a typical RDF-triplestore, each triple contains indexes on each of the (S,P,O) slots of the triple and would require at least 3 index accesses per triple. To perform the same Insert, Delete, or Fetch operation, three index accesses are required per triple, and the index access increases by a factor of the field-to-triple fan-out. E.g., a Delete of the 101 relational records in the prior example (assuming all PK and FK fields were indexed) would typically require (1 * 1) + (100 * (1 + 1)) or 201 index accesses. A delete of the equivalent 602 RDF-triples would require (602 * 3) or 1806 index accesses, an increase in B-Tree accesses by a factor of 9.

If unused indexes can be eliminated, performance can be significantly improved. More formally, in AllegroGraph, full inversion of the first three RDF (s, p, o) value slots is provided by default. This results in three indexes per triple:

Index(s, p, o, g, i)
Index(p, o, s, g, i)
Index(o, s, p, g, i)

Additionally, because a named graph ID slot is also present, a second set of g- indexes needs to be maintained:

Index(g, s, p, o, i)
Index(g, p, o, s, i)
Index(g, o, s, p, i)

Franz indicates that triplestore index storage overhead can be safely optimized by removing any indexes not used for specific queries. E.g., in applications scenarios where graphID queries are not anticipated, all indexes of the form Index(g, *, *, *, *) may be removed. Franz also indicates an important note: When AllegroGraph reads RDF-Triples from an N-Triples or RDF/XML file, it doesn't immediately index the input triples. While this doesn't preclude Metasky or Metasky applications from doing queries over the captured triples as soon as a triple data file is read, it means that all queries run using non-indexed slots will perform slower. If the client application can perform a bulk insert, then Franz recommends the application first read in all of the triple data and then subsequently start the indexing process. When the indexing processing is done, the application can then load additional triples from other file sources (or create new triples programmatically). AllegroGraph provides a function named "indexing-needed-p" to determine if the triple-store has any un-indexed triples.

4. TRIPLE SLOT IDENTIFIERS AND STORAGE OVERHEAD

To address the additional storage overhead required by triple slot URIs, all URI components of a triple slot value are prefix compressed by AllegroGraph and ultimately represented as 12 byte quantity which is used for hash lookup in a URI string table. This is an important consideration as Sedona code transferred to Metasky was refactored to follow the DMO standard of using full UUIDs as DocIDs instead of simple sequential INTs. One immediate observation and concern is that a relational SQL

database using INT DocIDs expands by over 100% in size for each ID field when the full text representation of the new DocID/UUID is stored. For binary formats, the expansion will be smaller and closer to 50% per field, but this is still significant. Measured size differences are shown:

Storage Type	DocID	DB Size
Relational DB	small INT Doc_ID	447K
RDF-Triplestore	UUID (SHA1)	943K

Metasky engineering also indicates that ID optimization may be needed to reduce storage for triple IDs. Both Franz and Metasky believe that the URI overhead is currently manageable since within the triplestore database, there is only one copy of each unique UUID string and only the 12 byte hash code identifiers might be replicated hundreds of thousands of times, and not the full URIs; this would afford sufficient compression opportunities for most URI patterns. Since AllegroGraph has a string dictionary so the full string would only be stored once, and elsewhere the much shorter references would be used. [J. Fieber, B. Millar] Metasky engineers indicate that a similar ID compression scheme could be used for the Redland/Postgres storage system currently being used in TRIAD.

An expressed concern from ATL is one consequence of standardizing on a triplestore model is a higher storage "tax" that document graph applications are forced to pay -- since every relational row results in multiple triples (on average 1:10), and that every triple must specify the URI or UUID in at least the (S) and (P) slot of each triple. This is non-trivial even if ID prefix compression and hash encoding schemes are used. Additionally, each triple statement predicate (P) also replicates the column header or field label from its relational table. Thus, for representing the same about of information in a relational database, there will be a considerable expansion in the size of the corresponding triplestore database.

Both Franz and Metasky agree that the above two concerns are true and indicate that when you have many duplicate IDs for each statement in a column-oriented store, with compression, you get potentially a large storage "tax rebate" from the URI compression mechanism. A graph with billions of triple statements may well only have a few hundred unique predicates and an indexed column structure that is compressed down to a smaller size. (For more details, see the 6/4/09 e-mail thread discussions between W. Chang and J. Fieber.)

J. Fieber indicates that within Triad/Metasky, the ID compression by the Redland/Postgres storage system is not as sophisticated as AllegroGraph. Fieber also believes there will be opportunities for performing ID space optimizations as needed. From the ATL side, one recommendation would be to perform follow-up work to evaluate ID overhead in Redland/Postgres due to the mapping of relational record PK IDs to URIs in RDF-triple slots.

5. RDF GRAPH QUERIES AND INFERENCING

Our inferencing query experiments described in Section 2 showed that the AllegroGraph RDF-Triplestore is suitable for representing and querying a wide range of document-graph and ontology-based structures that will be provided by CSBU applications and tools. If the dominant graph and inferencing query operations for Metasky are similar to the ones evaluated, RDF-Triplestore technology is desirable since graph queries are significantly more difficult to express and process using relational SQL. If on the other hand metadata stored in the triplestore is derived largely from relational databases, then insert, update, and retrieval performance may be significantly slower due increased index accesses resulting from triplication of the relational data.

Entity Identity Binding

We note that the presence of RDF and triplestores does not automatically guarantee globally consistent semantics. While the RDF-Triplestore clearly plays an essential role in representing ontologies and in particular, the controlled vocabulary portions thereof, a general issue is whether semantics between

domain ontologies or taxonomies will be consistent A significant amount of industry and academic work has been done in capturing and representing ontologies using RDF; open questions that remain are: what happens when new ontologies must be integrated into an existing ontology and, what happens when existing ontologies need to evolve their schemas? These challenges will remain and formal mechanisms will need to be in place to ensure the quality, integrity and consistency of all RDF-triples recorded into ontologies within the triplestore.

A simple example is resolving term vocabularies between different ontologies, e.g., determining that "COMPANY" in one ontology means the same thing as "BUSINESS ENTITY" in another ontology. While some progress can be made in automatic mappings using machine learning techniques, Larry Masinter et. al. will quickly point out that automatic ontology mappings are an extremely difficult problem since even a relatively simple attribute such as DC:Creator already have ambiguous and overloaded semantics. In these cases, there may be insufficient information to make meaningful inferences.

If ontology and vocabulary semantics can be reasonably addressed, a key point by J. Fieber is that use of an RDF-Triplestore is a requisite starting step that enables Metasky applications to tackle the question of how content data and metadata would link into the global web of other related similar data. If a user's data does not need to be linked to a larger body of information, then there is little value in bringing it out of a relational database silo in the first place. However, once data exists within a triplestore, challenging questions arise as to when and how semantic entities can be resolved with external RDF graphs, vocabularies, or ontologies.

January 2009, there were discussions between W. Chang, M. LeBrun, and L. Masinter on the subject of when and how triple entity identities should be resolved with other potentially equivalent entities. A particular example is the mention of an entity "Indy" which could refer to the movie character Indiana Jones in one context, or the "Indy" racecar speedway. A consensus at the time was that late or deferred entity identity binding in most cases would be preferable because of the problems around real-time identification of relevant shared vocabularies / ontologies.

In particular, there is a potentially high computation cost in creating multiple lengthy inference chains or determining there are too many possible ambiguities to resolve. A reasonable strategy would be to capture all statements and assertions while assigning distinct IDs as needed, and to defer resolution of entity identity until the needed ontologies and vocabulary context is known (e.g., explicitly provided by the application).

Rather than continuously trying to resolve entity identity as triples are captured and stored into the RDF-Triplestore, This late binding strategy avoids problems with on demand entity identity resolution by forcing specific applications to explicitly communicate the relevant context.

6. RDF-Triplestore schema design and ontology formalisms

Relational database design and optimization tools, formal schema descriptions, and data dictionary mechanisms have been in use for several decades when compared to their W3C and RDF counterparts. If Adobe is standardizing on RDF-Triplestores, we should leverage formal RDF schema description mechanisms such as RDF-Schema for describing all Triad/Metasky RDF-Triplestores. Additionally, we should employ best practice ontology design methodologies to maximize sharing of domain ontologies (e.g., movie genres, scene object classifications, video transition types). From the 2008 Semantic Technologies conference, E. Kendall and D. McGuinness provide a systematic outline for thinking about and developing ontologies for knowledge representation within the semantic web and RDF framework.

https://zerowing.corp.adobe.com/download/attachments/68227381/T03_MON_0830_Kendall_Elisa_McGuinness_Deborah_COLOR.pdf

ADDITIONAL QUESTIONS OR COMMENTS

Please contact W. Chang if you would like more information concerning this report, or if you would like full copies of the ATL and Franz. Source code used in migrating Sedona2 data to AllegroGraph, as well as copies of all of the source code used for the document graph, ontology, and rights inferencing experiments.

OTHER RELATED REFERENCES:

Wiki:
https://zerowing.corp.adobe.com/display/ATG/Metasky+and+Sedona2+Triplestore+Architecture+-++CSBU%2C+ATL%2C+and+Franz+Joint+Study

Initial Meeting Notes:
https://zerowing.corp.adobe.com/display/ATG/Metasky+and+Sedona2+-+Franz+Joint+Study+Meeting+Notes

Relational-DB-to-Triplestore Mapping Patent P921:
https://zerowing.corp.adobe.com/download/attachments/149332361/SystemForMappingRelationalDatabaseToRDF-Triplestores.doc

Appendix 1 – Examples of Non-Inferencing SPARQL Query Patterns provided by Franz Inc.

1. Basic unconstrained (S,P,O) SELECT

```
queryString = """
SELECT ?s ?p ?o
WHERE
{
        ?s ?p ?o .
}
"""
tupleQuery = repoconn.prepareTupleQuery(
                QueryLanguage.SPARQL,
                queryString)

result = tupleQuery.evaluate();
try:
        for bindingSet in result:
                s = bindingSet.getValue("s")
                p = bindingSet.getValue("p")
                o = bindingSet.getValue("o")
        print "%s %s %s" % (s, p, o)
finally:
        result.close();
```

```
SELECT ?s ?p ?o  WHERE {?s ?p ?o .}
```

Findings:

```
Prepare  SPARQL query          : took     0.000 sec.
evaluate SPARQL query          : took 132.953 sec.
iterate all 294080 triples     : took   15.282 sec.
```

Note: Sqlite3 CLI console: Select * from all tables took 16 sec.

2. Constrained SELECT (retrieve all script action elements that have DocID = 3 and ElemID = 682)

```
queryString = """
SELECT ?p ?o
WHERE
{
        <"+BASE_URI+DB_NAME+"/m_actions/3/682>  ?p  ?o
}
"""
tupleQuery = repoconn.prepareTupleQuery(
                QueryLanguage.SPARQL,
                queryString)

result = tupleQuery.evaluate();
try:
        for bindingSet in result:
                p = bindingSet.getValue("p")
                o = bindingSet.getValue("o")
                print "%s %s" % (p, o)
finally:
        result.close();
```

3. A Sedona/SPARQL text search for subjects whose "action" objects contain the word 'Indy'

```
queryString = """
SELECT ?s ?o
WHERE
{
        ?s <"+BASE_URI+"IJ-QRY5/m_actions/action> ?o . ?s fti:match 'Indy' .
}
LIMIT 50
"""

tupleQuery = repoconn.prepareTupleQuery(
                QueryLanguage.SPARQL,
                queryString)

result = tupleQuery.evaluate();

try:
        for bindingSet in result:
                s = bindingSet.getValue("s")
                o = bindingSet.getValue("o")
         print "%s %s" % (s, o)
finally:
        result.close();
```

4. Specific timecode query

```
queryString = """
SELECT ?p ?o
WHERE
{
        <"+BASE_URI+DB_NAME+"/m_tcs_align_pts/3/00:02:00:10> ?p ?o
}
"""

tupleQuery = repoconn.prepareTupleQuery(
                QueryLanguage.SPARQL,
                queryString)

result = tupleQuery.evaluate();

try:
        for bindingSet in result:
                p = bindingSet.getValue("p")
                o = bindingSet.getValue("o")
        print "%s %s" % (p, o)
finally:
        result.close();
```

Appendix 2 - "Structure" Term Ontology from WordNet for Partial RDF-Triplestore Ontology Queries

```
==> structure
    ==> housing
        ==> dwelling
            ==> house (#1) …
    ==> building (#1)
        ==> house (#1)
            ==> beach house
            ==> boarding house, boardinghouse
                ==> bed and breakfast, bed-and-breakfast
            ==> bungalow, cottage
            ==> cabin
                ==> log cabin
            ==> chalet
            ==> chapterhouse, fraternity house, frat house
            ==> country house
                ==> chateau
                ==> dacha
                ==> shooting lodge, shooting box
                ==> summer house
                ==> villa(#4)
                ==> villa(#3)
            ==> detached house, single dwelling
            ==> dollhouse, doll's house
            ==> duplex house, duplex, semidetached house
            ==> farmhouse
            ==> gatehouse
                ==> lodge
            ==> guesthouse
            ==> hacienda
            ==> lodge, hunting lodge
            ==> lodging house, rooming house
                ==> flophouse, dosshouse
            ==> maisonette, maisonnette
            ==> mansion, mansion house, manse, hall, residence
                ==> manor, manor house
                ==> palace, castle
                ==> stately home
            ==> ranch house
            ==> residence
                ==> court
                ==> deanery
                ==> manse
                ==> palace
                    ==> alcazar
                ==> parsonage, vicarage, rectory
                    ==> glebe house
                ==> religious residence, cloister
                    ==> convent
                        ==> abbey
                        ==> nunnery
                    ==> monastery
                        ==> abbey
                        ==> charterhouse
                        ==> friary
                        ==> lamasery
                    ==> priory
            ==> row house, town house
                ==> brownstone
                ==> terraced house
            ==> safe house
            ==> sod house, soddy, adobe house
            ==> solar house
            ==> stash house
            ==> tract house
            ==> villa (#1)
```

Appendix 3 – Research on expected RDF-triple ID slot URI/URL sizes:

Ducut E, Liu F, Fontelo P. An update on Uniform Resource Locator (URL) decay in MEDLINE abstracts and measures for its mitigation. BMC Medical Informatics and Decision Making 2008, 8:23. An excerpt from the abstract:

> Methods: MEDLINE records from 1994 to 2006 from the National Library of Medicine in Extensible Mark-up Language (XML) format were processed yielding 10,208 URL addresses. These were accessed once daily at random times for 30 days. Titles and abstracts were also searched for the presence of archival tools such as WebCite, Persistent URL (PURL) and Digital Object Identifier (DOI).
>
> Results: Results showed that the average URL length ranged from 13 to 425 characters with a mean length of 35 characters [Standard Deviation (SD) = 13.51; 95% confidence interval (CI) 13.25 to 13.77]. The most common top-level domains were ".org" and ".edu", each with 34%.

Huang, N.-F., Liu, R.-T., Chen, C.-H., Chen, Y.-T., and Huang, L.-W., A Fast URL Lookup Engine for Content-Aware Multigigabit Switches, in Proceedings of AINA, 2005.

> Research results conducted on the Google search engine index show that the average URL length measured in characters is 31.2 characters. WWC: When this base part is concatenated with additional namespace information and entity identification (e.g., person, place or thing name), we could expect actual triple slot URIs to easily double in size to ~ 64 bytes per ID. Using ID prefix encoding schemes, we then would see a reduction on the order of 5X down to 12 bytes per ID (as in the case of AllegroGraph) comparable to relational database ID fields. Storage redundancy of compressed IDs from a mapped relational rows into multiple triples then becomes the dominate factor.