IDA

INSTITUTE FOR DEFENSE ANALYSES

Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure (DFS) Portal Implementation: Part 2, Technical Feasibility, Affordability, and Architecture Integration Options

Francisco L. Loaiza-Lemos, *Project Leader*

Dale Visser

1 June 2017

IDA Document
D-8516

Log: H 2017-000309

Copy

INSTITUTE FOR DEFENSE
ANALYSES
4850 Mark Center Drive
Alexandria, Virginia 22311-1882

About This Publication

This work was conducted by the Institute for Defense Analyses (IDA) under contract HQ0034-14-D-0001, Task BC-5-4277, "Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure Portal Implementation," for Army CIO/G-6 (SAIS-AOD). The views, opinions, and findings should not be construed as representing the official position of either the Department of Defense or the sponsoring organization.

Acknowledgments

Steven P. Wartik

For more information:

Francisco L. Loaiza-Lemos, Project Leader
floaiza@ida.org, 703-845-687

Margaret E. Myers, Director, Information Technology and Systems Division
mmyers@ida.org, 703-578-2782

INSTITUTE FOR DEFENSE ANALYSES

IDA Document D-8516

Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure (DFS) Portal Implementation:

Part 2, Technical Feasibility, Affordability, and Architecture Integration Options

Francisco L. Loaiza-Lemos, *Project Leader*

Dale Visser

Executive Summary

This document was prepared by the Institute for Defense Analyses (IDA) in support of the FY 16 Army Study "Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure (DFS) Portal Implementation."

This document constitutes the second deliverable under the project description and addresses the study's objective of assessing the maturity and applicability of graph database technology as a practicable materiel solution that reflects legacy system realities, while effectively and efficiently delivering the needed *at-rest* and *in-motion* force structure products for the planned DFS portal.

Specifically, the objective of the second deliverable is to determine the technical feasibility, affordability, and necessary architecture integration needed to include graph databases in the mix of technologies that will support the planned Army DFS Portal. Rapid prototyping techniques have been applied, as part of the continuing evaluation of alternatives, to stress the implementations of the graph databases chosen for the study. Data collected during those activities will be used to continue maturing the decision process needed to determine the best-of-breed options. The assessments leverage the metrics elaborated in the initial phase of the study, which were documented in the first deliverable.[1]

Background

This phase of the study is aligned with the goals and objectives of the Department of Defense (DoD), as expressed in its Global Force Management Data Initiative (GFM DI), whereby DoD is seeking the standardization of all authorized force structure data so that it can be understandable to, and usable by, both warfighting and business systems across the DoD Enterprise.[2] As noted in the first deliverable under this project, the challenge in all of the related activities is the harmonization of data that currently resides in a large number of relational legacy systems, so that it can be readily used in the generation of *at-rest* and *in-motion* force structure products.

[1] IDA Document D-8345, *Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure (DFS) Portal Implementation: Part 1, Preliminary Characterization of Data Sources, Representation Options, Test Scenarios and Objective Metrics*, F. Loaiza, D. Visser, DRAFT FINAL, March 2017.

[2] http://www.prim.osd.mil/init/init_osdmanpower.html

Because of the potential for cost reduction and the procedural simplicity associated with an approach that directly recasts the legacy source data in the form of Resource Description Framework (RDF) triples,[3] collects them in a graph data store, and then uses the triples directly to generate the force structure products, this phase of the study explores more carefully the technical aspects involved in those steps and the impact on performance of the various technical options. The analytical results were then used to inform the selection of possible solution architecture alternatives to support the integration of graph databases into the mix of technologies needed to implement the planned Army DFS Portal.

Document Structure

This document is organized as follows:

1. Section 1 analyzes a fairly comprehensive catalog of data structures likely to be in use within the relational legacy data stores that constitute the source of the initial data population for the Army DFS Portal, and how best to re-express data captured in those types of structures in the form RDF triples.
2. Section 2 documents the degree of performance degradation that may be associated with some of the representation choices presented in Section 1.
3. Section 3 documents solution architecture options that support the integration of graph databases into the mix of technologies needed to implement the planned Army DFS Portal.
4. Section 4 provides the current set of conclusions and recommendations for this phase of the study.
5. Appendix A contains a follow-on survey of proprietary and open source graph database implementations that complements what was covered in the first deliverable under the project description.
6. Appendix B contains a number of Python scripts used for the generation of test data used in the "six degrees of separation" (SDOS) scenarios. The code is licensed for free reuse, and it is intended to help other groups in their evaluations.

Scope

As was the case for the first deliverable, the results described in this document do not address any of the complexities inherent in the policies and procedures embedded in the "as-is" systems that currently support the population of the Army Organization Server under the GFM DI initiative, and which would come into play for scenarios in which the source data to be converted into RDF triples is in the form of XML instance documents

[3] https://www.w3.org/RDF/

that conform to the GFM DI specifications. It is, therefore, assumed that those XML instance documents can be generated and would be accessible as inputs for subsequent manipulations required by the graph database approach.

We continue to note that in spite of repeated attempts by the sponsor to obtain data dictionaries and sanitized sample data, it has not been possible for the IDA team to collect actual legacy source data. Therefore, as was the case in the previous phase of the study, the simulations are based on realistic but nevertheless synthetic data. The IDA team continues to feel that the use of notional data is appropriate because the transformation of data from a relational database representation into RDF triples that can be stored in a graph database does not depend on the nature of the tables and columns implemented in the physical schema of relational data stores. In fact, generally speaking, the conversions needed to recast source legacy data into RDF triples are all syntactic and involve basic string manipulations that apply in all cases.

Since essentially all the Army legacy data pertinent to the generation of force structure products resides in relational data stores, the IDA team did not perform a comparison among all possible NoSQL alternatives to the relational paradigm but chose to concentrate on RDF triple stores as one of the most salient and mature representatives of the graph database technologies. Depending on time and funding, the IDA team may be able to conduct preliminary testing of a technology that represents data in the form of key-value pairs in one of the subsequent phases of the study.

Finally, the study does not cover the performance implications of using special-purpose hardware and software when testing medium-size RDF triples stores, i.e., on the order of one to ten billion triples, because that would have required acquiring it and setting up an appropriate environment in which to use it, which not only would be costly but arguably would not add anything special to the analysis, other than to show that, generally speaking, a special-purpose server machine most likely will load and retrieve RDF triples faster than a regular laptop.

Analytical Approach

The work performed for this phase of the study concentrated on answering the following questions:

- What are the data structures likely to be found in the Army relational legacy data stores that contain the source data needed to populate the planned Army DFS Portal?
- What are the choices for implementing the conversion process into RDF triples for the data resident in the data structures that the Army relational legacy data stores use?

- Do any of the choices – as identified in this phase of the study – for representing the legacy data in the form of RDF triples negatively impact data retrieval performance?
- What are the most likely types of solution architecture options that support the integration of graph databases in the mix of technologies needed to implement the planned Army DFS Portal?
- What are the lessons learned and how can they help inform the decision process needed to determine best-of-breed options?
- What are the implications for the Army of adopting a graph database approach?

Conclusions and Recommendations

Based on the analytical work performed during this phase, the IDA team concluded the following:

- All the data structures that are likely to be found in the pertinent Army relational legacy data stores – namely, those containing the source data needed to populate the planned Army DFS Portal – can be re-expressed in a straightforward manner using RDF triples. The difference in the degree of complexity of the transformation chosen for the relational data structures obeys strategic considerations, such as reuse and expansion of the data to satisfy novel and emerging uses.
- Use of a "semantic layer" in the form of an appropriately sized ontology is quite useful for organizing the resources in an RDF triple store in the same way that data is bundled in relational data stores under the concept of a "table." The semantic layer could also be used to retain traceability back to the data sources.
- Certain types of data structures common in relational data stores can lead to very poor data retrieval performance – such as in the canonical example of multiple layers of node dependencies found in networks which has been popularized under the rubric of "six degrees of separation." Pre-filtering and the use of materialized views essentially eliminate the performance issue in the relational stores, although they reduce flexibility and add complexity to the physical schema. Similar approaches can also be used to improve the performance of RDF triple stores, but the downside implications may be handled more elegantly through judicious use of federated triple stores and special-purpose hardware and software.
- The solution architecture options that can support the integration of graph databases in the mix of technologies needed to implement the planned Army DFS Portal are generally satisfactory, and a final determination will require the inclusion and analysis of the concept of operations for the planned DFS portal and the timelines associated with the key Army information systems.

- Similarly, the selection of best-of-breed options may be more sensitive to the concept of operations for the planned DFS portal than to factors of size, scalability, and data retrieval performance.

For this stage of the study, the preliminary recommendations are as follows:

- Continue during the next phase the evaluation of available graph database implementations, both proprietary and open source, and expand the scope to include other promising NoSQL choices.
- Continue to use rapid prototyping techniques to collect performance statistics that can inform both the selection process of the optimal graph database implementation and its integration into the mix of technologies needed to implement the planned Army DFS Portal.
- Explore applicable mitigation strategies for potential risks that would arise from the adoption of graph database technology as a materiel solution for the planned Army DFS portal.

Contents

Figures and Tables

1. A Catalog of Typical Relational Data Structures and Their Representations Via RDF Triples

A. Notation for Depicting Relational Data Structures

As briefly noted in the Executive Summary, the analysis presented in this document assumes that the vast majority of the Army legacy data stores pertinent to the generation of force structure products are implemented using standard relational database technologies. Due to fiscal and time constraints, this document cannot provide an in-depth introduction to the relational database paradigm. Therefore, the discussion of relational database technologies covered in this and subsequent sections assumes on the part of the reader an intermediate level of familiarity with the main concepts covered in database modeling and their depiction in entity-relationship diagrams (ERDs), such as one-to-many relationships between parent and child entities, the concept of relationship cardinality, and the concept of subtype hierarchies.[4] In addition, the materials are presented here under the assumption that the reader has a basic understanding of the notion of normal forms in relational data stores.

A number of graphical modeling languages can be used to represent both the logical and the physical schemata of relational databases. In this document, we use the graphical notation from the data modeling language called IDEF1X.[5]

Excellent introductions to relational database modeling using IDEF1X exist, and the reader is encouraged to use those resources when encountering unfamiliar concepts.[6] A short summary of the IDEF1X graphical notation is shown in Figure 1-2. The basic construct shown at the top of the figure is a line that joins two rectangles. The rectangles represent the entities in the model, e.g., Person, Organization. The line joining the rectangles represents the relationship between the entities. It can be either solid or dashed. The line

[4] *Designing Quality Databases with IDEF1X Information Models*, by Thom Bruce, Dorset House Publishing Company, Inc. (1992).

[5] IDEF stands for ICAM Definition, where ICAM is the acronym for Integrated Computer Aided Manufacturing. IDEF 1 is the method in that series intended to support the development of "information models" capturing the structure and semantics of the information within the environment or system. After the original specification was released, *extensions* were made that led to the addition of the X in the name, hence, IDEF1X. The specification was later made into a NIST standard (FIPS PUB 184) and was widely used in the 80s and 90s within DoD under various data reengineering efforts.

[6] See Footnote 3 above.

starts at an entity normally referred to as the *parent* entity and ends at the entity called the *child* entity.

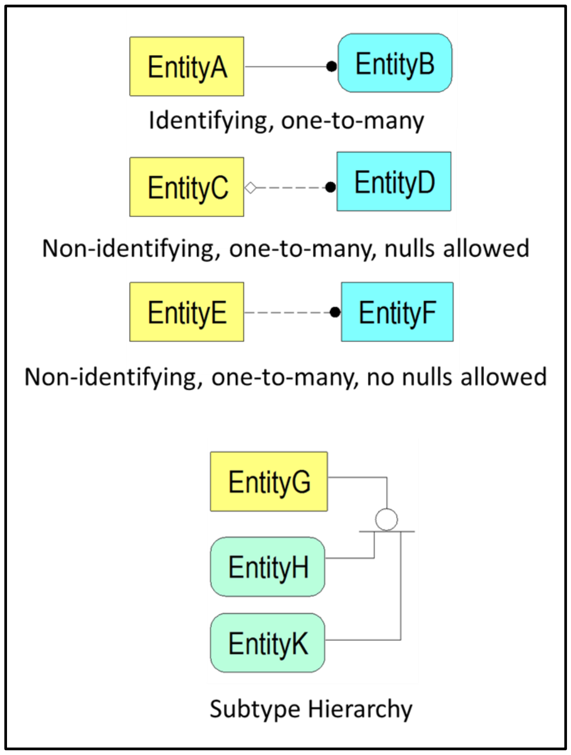


Figure 1-1. Summary of the IDEF1X Graphical Notation

A solid line corresponds to an *identifying* relationship, meaning that the attribute used to uniquely point to an instance in the parent entity, is also needed to identify the instance associated to it in the child entity. A dashed line indicates a *non-identifying* relationship, meaning in this case that the attribute that is passed from the parent entity to the child entity is not intended to be used for the purpose of identifying instances in the child entity.

In addition, the dashed line can start with a diamond to indicate that nulls are allowed, or it can start without a diamond to signal that no nulls are allowed. In other words, for non-identifying relations, the column corresponding to the attribute that migrates from the parent to the child can (nulls allowed) or cannot (no nulls allowed) be a null respectively, and the presence or absence of the diamond signals that constraint.

In all cases the line is drawn from the parent to the child, and the child side corresponds to the *many* side of the relationship, which in the IDEF1X notation is indicated by a large dot. So, for example, if the parent is the entity Building and the child is the entity Floor, then a relationship between the entity Building and the entity Floor, which would express the linkage between a specific building and all its floors, would be graphically depicted using a solid line starting at the entity Building and ending at the entity Floor. Since, typically, an instance of Building has one or more instances of Floor, the entity Floor is the *many* side of the relation and the line would have a large dot on the end that touches the entity Floor.

IDEF1X supports a number of additional embellishments that allow the modeler to more precisely specify cardinality constraints intended to control the relationship. So, for example, adding a numeric value at the many side of the relationship enables the modeler to state that the many side must have exactly that number of instances. In the case of the relationship between Building and Floor that we have been using as an example, adding the number 10 on the many side would restrict the instances of Building in that model to those that have exactly ten floors.

A special notation is also supported in IDEF1X to depict subtype hierarchies (shown at the bottom of Figure 1-1). This notation is of particular interest when modeling taxonomies at the logical level, and, in IDEF1X, it implies that the elements of a given taxonomy are *disjoint*, i.e., an instance of the supertype specializes to an instance of at most one of the subtypes that comprise the taxonomy.[7]

In other words, the specialization of the instances of the supertype via the subtypes cannot span multiple subtypes. The canonical example is that of land vehicles specializing into either trucks or sedans. In such a model, an instance of LandVehicle cannot be both an instance of Truck and an instance of Sedan.[8]

If one does not know what specialization corresponds to an instance of LandVehicle, then there is no need to enter anything in the tables corresponding to the subtypes, and that is why the cardinality is zero or one. At the physical level, a subtype relationship is therefore equivalent to a Z-relationship, i.e., a relationship that says that every instance of the parent entity is associated to zero or one instances of the child entity. Most implementations, either proprietary or open source, of the relational database paradigm do not enforce the disjunction constraint implicit in the IDEF1X notation. In other words, custom code must be added either in the application accessing the database, or in stored procedures and triggers within the relational database to ensure that, for example, an instance of LandVehicle is not typed as being both a BMW luxury car and a refueling truck.

Finally, it is to be noted that the IDEF1X modeling language supports the generation of ERDs at various stages of definition, ranging from the high-level conceptual models all the way down to fully attributed ERDs, ready to generate the physical schema of a relational database.

---

[7] Besides the support for the notion of subtypes, IDEF1X also can indicate whether or not the taxonomy being modeled is to be considered complete. The icon with a single line (see Figure 1-1) is used for incomplete taxonomies. The icon with two lines is for complete taxonomies.

[8] Note that this does not mean that one cannot have constructs that have semantic overlaps. It simply means that if modelers choose the IDEF1X subtype notation they are committing to a disjoint taxonomy.

B. Common Relational Data Structures and Their Representation using RDF Triples

1. The Basic Conversion of a Relational Table Structure into RDF Triples

The implementation of a graph database approach using RDF triples to power the planned Army DFS portal requires the conversion of the legacy source data into a format that can be loaded into the target RDF triple store. As noted in the preceding discussion, the relational paradigm encompasses both entities and relations among the entities. First we discuss the conversion of entities/tables to RDF triples.

2. Options for Converting Relational Table Content to RDF Triples

In a relational data store each entity contains one or more attributes. For example, the entity Person may include attributes such as a first name (shortened, for example, to fname in the physical table), a last name (shortened, for example, to lname in the physical table), and a date of birth (shortened, for example, to dob in the physical table), as shown in Figure 1-2.

Modern implementations of relational data stores have also adopted the use of attributes – mostly numeric – whose sole purpose is to serve as a tag to enable the retrieval of records in the table when using a query language such as Structured Query Language (SQL). These attributes are called the keys – or more specifically, the primary key attribute(s) – of the table. In our example the primary key is the attribute named perID in Figure 1-2.

```
Person
+-------+---------------+------+-----+---------+-------+
| Field | Type          | Null | Key | Default | Extra |
+-------+---------------+------+-----+---------+-------+
| perID | decimal(10,0) | NO   | PRI | NULL    |       |
| fname | varchar(50)   | YES  | MUL | NULL    |       |
| lname | varchar(100)  | YES  | MUL | NULL    |       |
| dob   | varchar(10)   | YES  |     | NULL    |       |
+-------+---------------+------+-----+---------+-------+
```

Figure 1-2. Component of the Physical Schema corresponding to the Entity Person

A snippet of the contents from the table Person is shown in Figure 1-3. To avoid inadvertently listing actual personal identification information (PII) the instances of the last name in Figure 1-3 are randomly generated alphanumeric strings, with four alpha characters followed by nine digits. The date of birth is also randomly generated and is set in the next century.

```
Person
+------------+-----------+--------------+------------+
| perID      | fname     | lname        | dob        |
+------------+-----------+--------------+------------+
| 1000000005 | OLIVE     | JYZZ593823293 | 2188-9-28  |
| 1000000010 | ALLYN     | XZDT440292416 | 2153-8-1   |
| 1000000015 | AURORA    | TFEG256160276 | 2159-4-29  |
| 1000000020 | DONG      | IMNM613397034 | 2129-8-15  |
| 1000000025 | CALLIE    | AEGN604310404 | 2127-3-12  |
| 1000000030 | DALTON    | IJFY233276724 | 2123-1-31  |
| 1000000035 | MALVINA   | BODH578098673 | 2166-9-19  |
| 1000000040 | GWENDOLYN | JJSJ309934750 | 2168-7-21  |
| 1000000045 | GLENN     | KACM616170748 | 2183-4-23  |
| 1000000050 | ALI       | JCKY848499644 | 2141-11-26 |
+------------+-----------+--------------+------------+
```

Figure 1-3. Sample Data Content of the Notional Person Table

The content of any record in a table such as the one shown in Figure 1-3 maps to a number of triples equal to the number of attributes, i.e., columns, in the table. Using, for example, the Terse RDF Triple Language serialization style (commonly referred to as *Turtle*),[9] one can convert each row, i.e., record, into the corresponding four triples. For example the first two records in Figure 1-3 can be converted to the following triples:

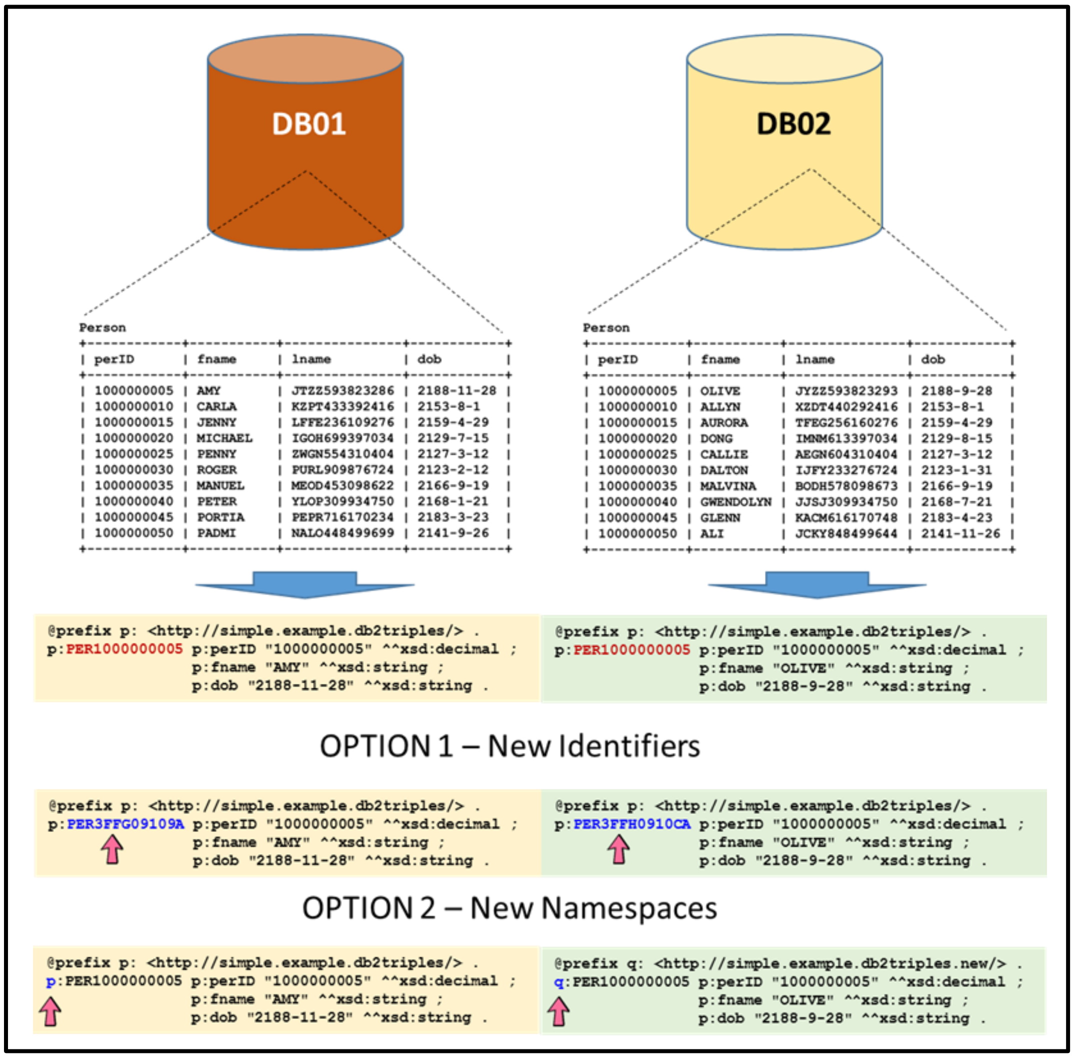
```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix p: <http://simple.example.db2triples/> .

p:PER1000000005 p:perID "1000000005"^^xsd:decimal ;
                p:fname "OLIVE"^^xsd:string ;
                p:lname "JYZZ593823293"^^xsd:string ;
                p:dob "2188-9-28"^^xsd:string .

p:PER1000000010 p:perID "1000000010"^^xsd:decimal ;
                p:fname "ALLYN"^^xsd:string ;
                p:lname "XZDT440292416"^^xsd:string ;
                p:dob "2153-8-1"^^xsd:string .
```

Note that in the listing above the subject of the triples is generated by pre-pending the string “PER” to the numeric value of the perID in the record, e.g., `PER1000000005` for the record with perID equal to `1000000005`. Also note that each attribute has a **prefix** that is normally in the form of a universal resource identifier (URI).[10] After the prefixes have been defined, they are available via the qualified notation, i.e., **<prefix>:<attribute>**, which

[9] See http://www.w3.org/TR/turtle/

[10] For the definitive guide on the RDF syntactical variants the reader can consult the official website maintained by the World Wide Web Consortium (W3C): https://www.w3.org/RDF/

is the one used in the listing. In Turtle for each one of the successive assertions one can also leave out the subject of a triple if it is the same as the one above. In that case each new triple is separated by a semicolon. The end of the record is indicated by a final period. Triples can have either another resource or a literal as their object. In the latter case, the literal may include a datatype as defined under the XML Schema Definition (XSD) specification.[11]



```
Person
+------------+------------+---------------+------------+
| perID      | fname      | lname         | dob        |
+------------+------------+---------------+------------+
| 1000000005 | AMY        | JTZZ593823286 | 2188-11-28 |
| 1000000010 | CARLA      | KZPT433392416 | 2153-8-1   |
| 1000000015 | JENNY      | LFFE236109276 | 2159-4-29  |
| 1000000020 | MICHAEL    | IGOH699397034 | 2129-7-15  |
| 1000000025 | PENNY      | ZWGN554310404 | 2127-3-12  |
| 1000000030 | ROGER      | PURL909876724 | 2123-2-12  |
| 1000000035 | MANUEL     | MEOD453098622 | 2166-9-19  |
| 1000000040 | PETER      | YLOP309934750 | 2168-1-21  |
| 1000000045 | PORTIA     | PEPR716170234 | 2183-3-23  |
| 1000000050 | PADMI      | NALO448499699 | 2141-9-26  |
+------------+------------+---------------+------------+
```

```
Person
+------------+------------+---------------+------------+
| perID      | fname      | lname         | dob        |
+------------+------------+---------------+------------+
| 1000000005 | OLIVE      | JYZZ593823293 | 2188-9-28  |
| 1000000010 | ALLYN      | XZDT440292416 | 2153-8-1   |
| 1000000015 | AURORA     | TFEG256160276 | 2159-4-29  |
| 1000000020 | DONG       | IMNM613397034 | 2129-8-15  |
| 1000000025 | CALLIE     | AEGN604310404 | 2127-3-12  |
| 1000000030 | DALTON     | IJFY233276724 | 2123-1-31  |
| 1000000035 | MALVINA    | BODH578098673 | 2166-9-19  |
| 1000000040 | GWENDOLYN  | JJSJ309934750 | 2168-7-21  |
| 1000000045 | GLENN      | KACM616170748 | 2183-4-23  |
| 1000000050 | ALI        | JCKY848499644 | 2141-11-26 |
+------------+------------+---------------+------------+
```

```
@prefix p: <http://simple.example.db2triples/> .
p:PER1000000005 p:perID "1000000005" ^^xsd:decimal ;
                p:fname "AMY" ^^xsd:string ;
                p:dob "2188-11-28" ^^xsd:string .
```

```
@prefix p: <http://simple.example.db2triples/> .
p:PER1000000005 p:perID "1000000005" ^^xsd:decimal ;
                p:fname "OLIVE" ^^xsd:string ;
                p:dob "2188-9-28" ^^xsd:string .
```

```
@prefix p: <http://simple.example.db2triples/> .
p:PER3FFG09109A p:perID "1000000005" ^^xsd:decimal ;
                p:fname "AMY" ^^xsd:string ;
                p:dob "2188-11-28" ^^xsd:string .
```

```
@prefix p: <http://simple.example.db2triples/> .
p:PER3FFH0910CA p:perID "1000000005" ^^xsd:decimal ;
                p:fname "OLIVE" ^^xsd:string ;
                p:dob "2188-9-28" ^^xsd:string .
```

```
@prefix p: <http://simple.example.db2triples/> .
p:PER1000000005 p:perID "1000000005" ^^xsd:decimal ;
                p:fname "AMY" ^^xsd:string ;
                p:dob "2188-11-28" ^^xsd:string .
```

```
@prefix q: <http://simple.example.db2triples.new/> .
q:PER1000000005 p:perID "1000000005" ^^xsd:decimal ;
                p:fname "OLIVE" ^^xsd:string ;
                p:dob "2188-9-28" ^^xsd:string .
```

Figure 1-4. Options for Deconflicting Triples Subject Names during RDF Triple Generation

From the above discussion, it then follows that the conversion of legacy data content residing in a relational data store in most cases can be a fairly mechanical process. Specifically, for tables that have a very simple key structure, e.g., a single numeric attribute, such as perID in the above-mentioned table Person, one can create the subject of

[11] https://www.w3.org/TR/xmlschema-2/

the triples using a method like the one presented above, i.e., pre-pending some meaningful prefix to the numeric value of the key attribute, and then using the labels from each of the columns as the respective predicates for the triples. The objects are the literals that reside in each of the cells of the record.

One could also adopt a more generic approach in which instead of using legacy identifiers, one generates new numeric identifiers for the names of the resources that may be used across multiple legacy data stores. This would make sense in situations where one is dealing with legacy data stores that contain similarly named tables and similar datatypes and ranges for their respective keys. In such cases, the simple approach suggested above for the subject name of the triples will create duplicate resource names, which would then need to be deconflicted before the triples are loaded into the triple store data lake.[12] Another approach may be to use the prefixes as a kind of namespace that ensures uniqueness. Figure 1-4 shows these two options. There is no reason why both techniques may not be used together to minimize the potential for unintentional duplicate resource names for the subjects of the RDF triples. However, a proliferation of namespaces may increase the overhead when querying the entire collection within the data lake.

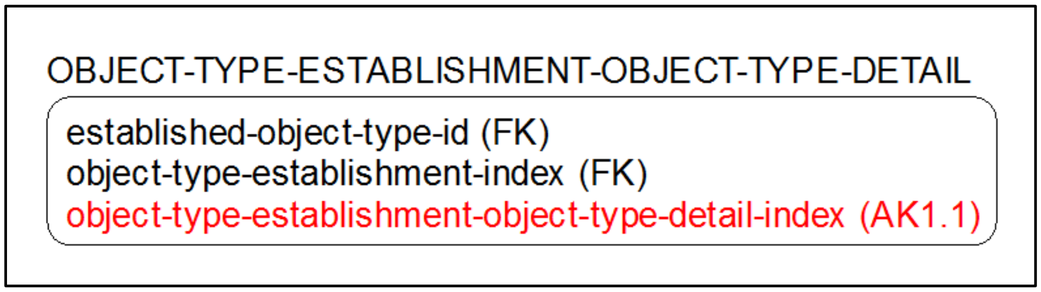

OBJECT-TYPE-ESTABLISHMENT-OBJECT-TYPE-DETAIL
established-object-type-id (FK)
object-type-establishment-index (FK)
object-type-establishment-object-type-detail-index (AK1.1)

Figure 1-5. Example of an Entity with a Complex Primary Key Structure

Where the key structure in the legacy data store tables is more complex, i.e., involves two or more attributes, such as in the example shown in Figure 1-5, reusing the values of the key attributes, by, for example, concatenating them, may lead to unwieldy names for the subjects of the triples. In those cases, it is highly recommended that a method be adopted that uses new alpha-numeric strings. One such approach could be to use a hexadecimal representation of a large number that acts as the new key and that is incremented for each new record (see Figure 1-4 above).

[12] "A data lake is a storage repository that holds a vast amount of raw data in its native format until it is needed." http://searchaws.techtarget.com/definition/data-lake

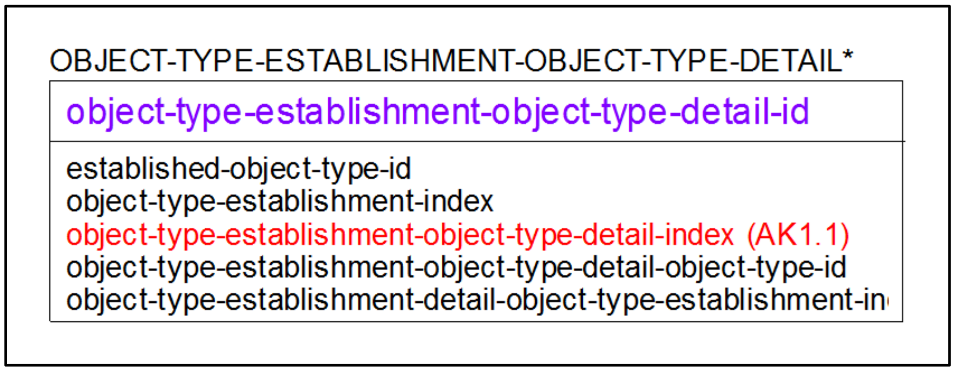

OBJECT-TYPE-ESTABLISHMENT-OBJECT-TYPE-DETAIL*
object-type-establishment-object-type-detail-id
established-object-type-id
object-type-establishment-index
object-type-establishment-object-type-detail-index (AK1.1)
object-type-establishment-object-type-detail-object-type-id
object-type-establishment-detail-object-type-establishment-in

Figure 1-6. Example of a Key Structure Simplification for Tables with multiple attributes making up the Key

This replacement of composite keys by a single new value is essentially the same as restructuring the original entity so that all the attributes that originally constituted the primary key are moved below the line – and thus no longer functioning as the key of the table – while inserting a new key composed of a single attribute (see Figure 1-6).[13]

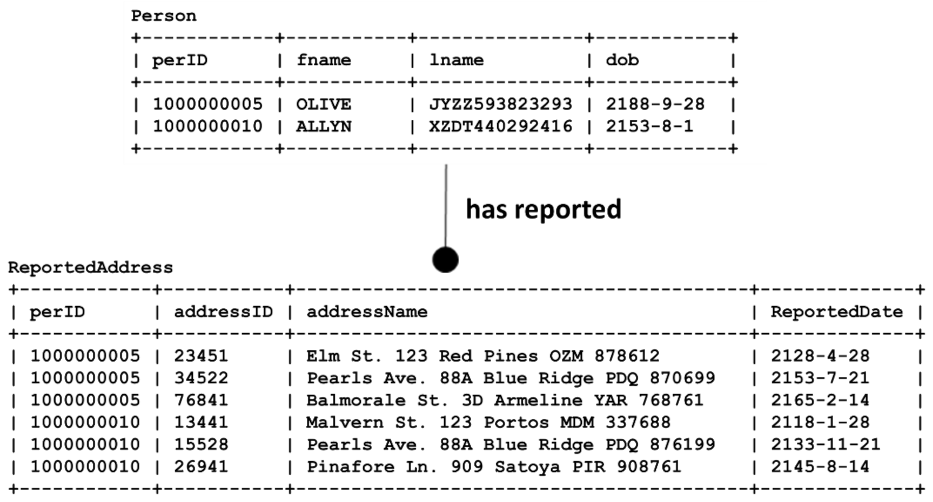
When adopting this "restructuring" at the time the triples are generated, the table content conversion becomes essentially the same as that of converting a table with a single key attribute (see Figure 1-3 above).

3. Options for Representing Relations among Entities in an RDF Triple Store

As was mentioned above, the relational paradigm encompasses both entities and relations among the entities. We now discuss the relations.

In a relational data store, the way a record in a given table is linked to a record in another table is via additional attributes placed in the child entities. These additional attributes contain the value of the respective keys of the parent entities. These are the so-called *migrated keys* in the child table (see Figure 1-7). They allow the correlation between the parent records and their respective child records.

[13] The reader should note that we are not recommending re-architecting the legacy database, but simply showing how to map between legacy databases and generated triples using IDEF1X notation to highlight what the end state would be equivalent to.



```
Person
+------------+-----------+---------------+------------+
| perID      | fname     | lname         | dob        |
+------------+-----------+---------------+------------+
| 1000000005 | OLIVE     | JYZZ593823293 | 2188-9-28  |
| 1000000010 | ALLYN     | XZDT440292416 | 2153-8-1   |
+------------+-----------+---------------+------------+
```

has reported

```
ReportedAddress
+------------+-----------+----------------------------------------+--------------+
| perID      | addressID | addressName                            | ReportedDate |
+------------+-----------+----------------------------------------+--------------+
| 1000000005 | 23451     | Elm St. 123 Red Pines OZM 878612       | 2128-4-28    |
| 1000000005 | 34522     | Pearls Ave. 88A Blue Ridge PDQ 870699  | 2153-7-21    |
| 1000000005 | 76841     | Balmorale St. 3D Armeline YAR 768761   | 2165-2-14    |
| 1000000010 | 13441     | Malvern St. 123 Portos MDM 337688      | 2118-1-28    |
| 1000000010 | 15528     | Pearls Ave. 88A Blue Ridge PDQ 876199  | 2133-11-21   |
| 1000000010 | 26941     | Pinafore Ln. 909 Satoya PIR 908761     | 2145-8-14    |
+------------+-----------+----------------------------------------+--------------+
```

Figure 1-7. A Parent Child Relationship Depicting Migrated Keys

Two relatively straightforward strategies can be used to convert content from tables that are linked via a parent-child relationship, as shown in Figure 1-7, where instances of Person may be related to zero or more instances of ReportedAddress. The first one is to treat the child in the same manner as the parent. In other words, the fact that a table stands in a parent-child relationship to another table represents a logical connection, but at the data level the child table is just like any other one in the physical schema, in the sense that it has a number of attributes, some of which may be used as the record key when running a SELECT query. Beyond that, nothing is different with regard to the ReportedAddress table when compared with the Person table. Therefore, one could have triples such as the ones shown below for the first two records in the ReportedAddress table of Figure 1-7.

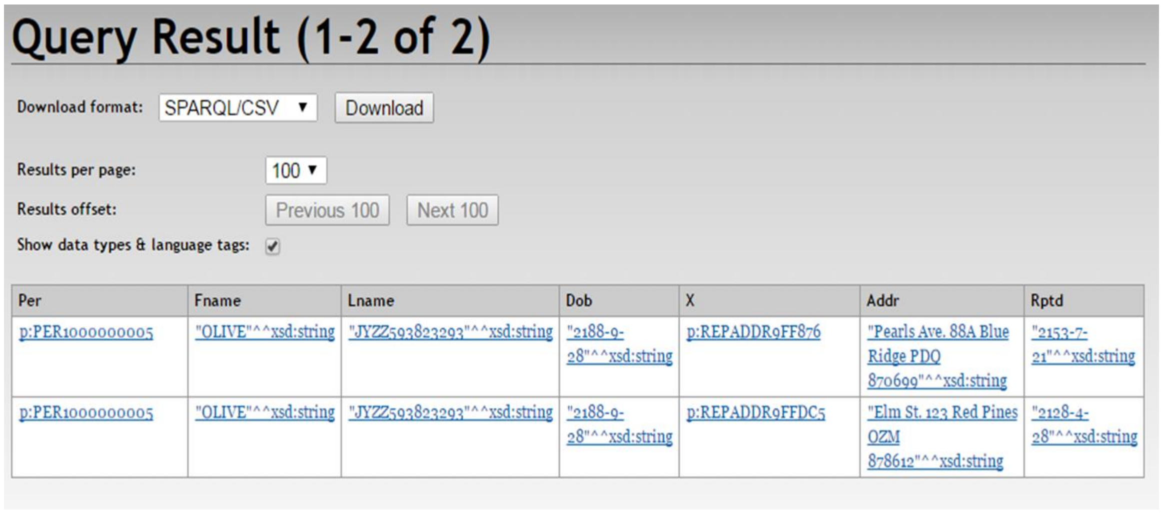
```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix p: <http://simple.example.db2triples/> .

p:REPADDR9FFDC5 p:perID "1000000005"^^xsd:decimal ;
                p:addressID "23451"^^xsd:decimal ;
                p:addressName   "Elm   St.   123   Red   Pines
                                 OZM 878612" ^^xsd:string;
                p:ReportedDate "2128-4-28" ^^xsd:string .

p:REPADDR9FF876 p:perID "1000000005"^^xsd:decimal ;
                p:addressID "34522"^^xsd:decimal ;
                p:addressName  "Pearls  Ave.  88A  Blue  Ridge
                                PDQ 870699" ^^xsd:string;
                p:ReportedDate "2153-7-21"^^xsd:string .
```

Having converted both the parent and the child tables to RDF triples, and thus eliminating the “table” construct that exists in relational data stores, the question becomes how to retrieve all the child records associated with a parent record. It turns out that the retrieval of the addresses in the ReportedAddress table for every instance of Person simply requires stating in the SPARQL query that the value of the perID attribute associated with the records of Person be the same as the value of the perID attribute associated with the records of ReportedAddress. Using the variable ?per for the triples corresponding to the records of the Person table and the variable ?X for the subject of the triples corresponding to the content of the ReportedAddress table, one can write a query that looks like this:

```
PREFIX p: <http://simple.example.db2triples/>
select ?per ?fname ?lname ?dob ?X ?addr ?rptd
where{?per p:perID ?z .
      ?per p:fname ?fname .
      ?per p:lname ?lname .
      ?per p:dob ?dob .
      ?X p:perID ?z .
      ?X p:addressName ?addr .
      ?X p:ReportedDate ?rptd }
```

The SPARQL query is simply asking the RDF triple store to find all the triples that satisfy the restriction that the value of the p:perID predicate for both the resources ?per and ?X be the same, namely **?z**, thus essentially linking the parent records to the child records at execution time. Figure 1-8 shows the dataset obtained when executing the SPARQL query above. As can be seen, the query correctly retrieves the two addresses associated with the instance of Person that has a perID with the value 1000000005. The query also retrieves the values of the subjects of the triples in the Person and ReportedAddress tables, as well as all their corresponding attributes.[14]

[14] One obvious disadvantage from using this approach is that it results in a large number of disconnected graphs. As such, it doesn't facilitate graph traversal unless one understands the structure of the DB from which data was translated.


Query Result (1-2 of 2)

Download format: SPARQL/CSV

Results per page: 100

Show data types & language tags: ✓

Per	Fname	Lname	Dob	X	Addr	Rptd
p:PER1000000005	"OLIVE"^^xsd:string	"JYZZ593823293"^^xsd:string	"2188-9-28"^^xsd:string	p:REPADDR9FF876	"Pearls Ave. 88A Blue Ridge PDQ 870699"^^xsd:string	"2153-7-21"^^xsd:string
p:PER1000000005	"OLIVE"^^xsd:string	"JYZZ593823293"^^xsd:string	"2188-9-28"^^xsd:string	p:REPADDR9FFDC5	"Elm St. 123 Red Pines OZM 878612"^^xsd:string	"2128-4-28"^^xsd:string

Figure 1-8. Parent Child Tables Dynamically Joined via SPARQL Query

Although the preceding approach shows that one can create the links between parent and child entities at query time – assuming that the data captured in the triples supports the relationship that is implied in the SPRQL query – one could also make the relationships among parent and child entities more explicit by including triples that assert that specific instances of the parent entity are related to specific instances of the child entity. In the example under discussion this would mean adding the following triples:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix p: <http://simple.example.db2triples/> .

p:PER1000000005 p:hasRptdAddress p:REPADDR9FFDC5;
                p:hasRptdAddress p:REPADDR9FF876 .
```

These triples state that the resource `p:PER1000000005` is directly connected to the resources `p:REPADDR9FFDC5` and `p:REPADDR9FF876` via the predicate `p:hasRptdAddress`. With this addition, the SPARQL query can now be rewritten as follows:

```
PREFIX p: <http://simple.example.db2triples/>
select ?per ?fname ?lname ?dob ?x ?addr ?rptd
where{?per p:fname ?fname .
      ?per p:lname ?lname .
      ?per p:dob ?dob .
      ?per p:hasRptdAddress ?X .
      ?X p:addressName ?addr .
      ?X p:ReportedDate ?rptd }
```

The line highlighted in green leverages the assertion that triples corresponding to some instances of the Person table are linked to triples corresponding to some instances of the ReportedAddress table. The dataset obtained when executing the new SPARQL query is identical to the one shown in Figure 1-8.

Table 1-1 summarizes the options available when converting legacy data content from relational data stores to RDF triples.

Table 1-1. Summary of Conversion Options from Relational Data Stores to RDF Triples

Conversion of Relational Legacy Source Data to RDF Triples	
Relational Construct	**RDF Representation**
• Instances of Entities/Tables (i.e., records)	• Subjects of the RDF triples. Specifically, each record is logically associated with the subject of a triple • The RDF resource must be uniquely named • This can be accomplished by naming the resource as a concatenation of an alpha prefix and the numeric value of the attribute used as the primary key for the record • When the record has a composite key, i.e., the key is made of more than one attribute, one can replace the composite key with a new key value and then concatenate it with a meaningful prefix, making the conversion identical to the case of records with a simple key structure • Alternatively, one can use different namespaces to ensure uniqueness, especially, if the legacy data stores may contain similarly named tables and key values with the same datatype and range
• Entity attributes / Table column names	• Predicates of the RDF triples. Thus if one has a table such as Person, with attributes fname, lname, and dob, then each of those attribute labels will become predicates in the respective triples
• Values of each record cell	• Objects of the RDF triples. The content of a record in a relational data store is generally some kind of literal with its associated datatype, and, therefore, the objects of the triples will be the literals that reside in the cells of each record
Conversion of Relationships	
Relational Construct	**RDF Representation**
• Migrated key in the child entity	• **Alternative 1:** Treat the key as any other attribute in the child entity, i.e., create a predicate to represent the migrated key and capture the value of the migrated key as the literal object for the corresponding triple • **Alternative 2:** State explicitly the parent-child relationship via additional triples

In the preceding paragraphs, the majority of the triples discussed have been of the type **<subject** [resource]>**<predicate** [resource]><literal>. This approach may be adequate in general. However, one should keep in mind that once the object of a triple is cast as a literal there is no way to further characterize the content beyond what can be said about a literal, e.g., its language, its XSD datatype.

On the other hand, if the object of a triple is itself a resource, then additional triples can be written to capture other metadata. For example, in the Person table one could have triples for the content of the last name written as:

```
p:PER1000000005 p:lname p:PER1000000005lname .
p:PER1000000005lname p:content "JYZZ593823293"^^xsd:string .
```

The insertion of the resource `p:PER1000000005lname` would make it possible to state things like the national origin of the last name, whether it is the maiden last name or the married last name of the individual, the date when it was changed, the reason it was changed. Going directly to a literal as the object for the content of the records precludes the possibility of adding directly further context to it. It is, therefore, recommended that careful consideration be given to any potential future uses of the triples to see whether the additional triples are justified.

4. Additional Relational Data Structure Patterns – The Associative Entity Pattern

Besides the basic parent-child relationship, there are some relational data structures that occur quite frequently and merit some discussion. The first one to consider is the associative entity pattern shown in Figure 1-9. It can be considered as a more flexible or extended version of the parent-child relationship in that now it is possible to relate instances of one parent entity to the same instances of another parent entity under multiple perspectives.

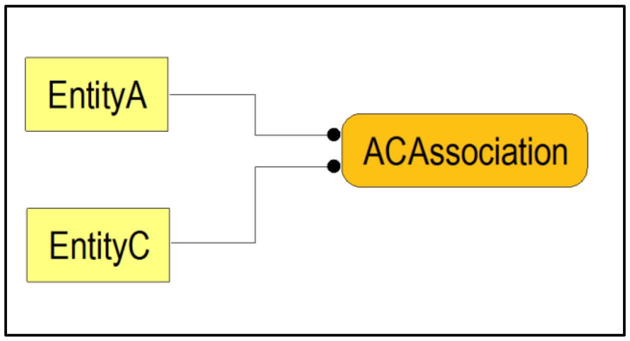


Figure 1-9. The Associative Entity Pattern

For example, the pattern can be used to relate instances of Person to instances of Address (see Figure 1-10), but with this pattern, not only can an address be associated with multiple individuals, or one individual with multiple addresses, as was possible with the parent-child pattern discussed in the preceding section, but now the same instance of Address and the same instance of Person can be related to each other under multiple use types, e.g., to express that the individual uses the location as a business address, as a postal address for sending or receiving merchandise, and as a domicile address. Also, the same address and the same individual can be associated to each other at different time intervals through the use of attributes such as associationStartDate and associationEndDate. This means that in addition to the attributes one expects for the two parent entities, i.e., the migrated foreign keys, the associative entity can also have attributes of its own.

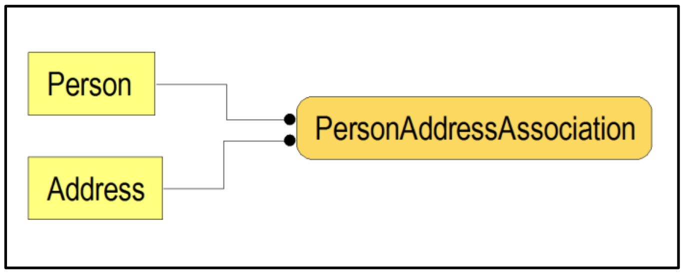


Figure 1-10. Example of the Associative Entity Pattern for Person and Address

The conversion of the content for these tables can be done just as in the basic mode discussed in the preceding section, and all that is required at query time is that the values of the key attributes in the parent entities and in the associative entity be constrained to be the same.

Alternatively, at conversion time one could create additional triples to explicitly state the relation between the instances of the two parent entities to the associative entity. These triples could be stated from the point of view of the associative entity. They could also be stated from the point of view of each of the parent entities. In some cases it may be advisable to have the linkages in both directions to improve the efficiency of the graph traversals, although this would not be a strict necessity for a correct mapping. Retaining the association entity at mapping time can be viewed as a form of reification of the relationship, and it is similar to the modeling of associations done in UML.

5. Additional Relational Data Structure Patterns – The N-Ary Associative Entity Pattern

The logic of the simple associative entity can be extended to N-Ary associations as shown in Figure 1-11. Adding triples that explicitly link an instance of the N-Ary association to all the respective instances of the parent entities is highly recommended in these cases to eliminate any ambiguity with regard to the linkages of the parent resources to each other. Similarly, simplifying the composite key structure as discussed previously (see Figure 1-6 above) can streamline the naming of the subjects for the triples and make data retrieval faster by reducing the number of constraints to be tested. Beyond these two points the conversion of the content of the tables that participate in this pattern does not offer any other challenge.

It should be noted that the N-Ary association pattern can be easily decomposed into a series of binary associations (see Figure 1-12). In this case, the entity that participates in more than one of the binary associations would constitute the hub of the construct, and the SPARQL queries would include the constraints needed to ensure the correct response.

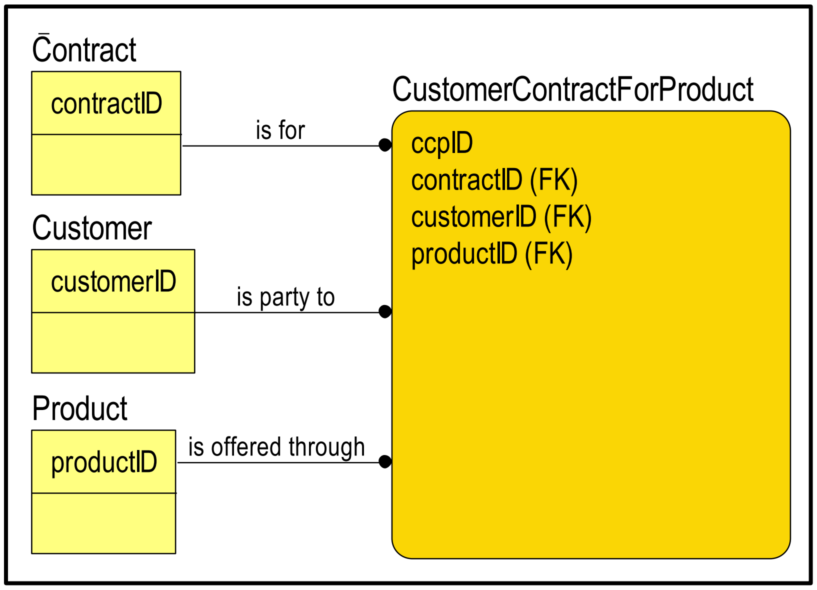


Figure 1-11. Example of an N-Ary Associative Entity Pattern for Products sold to Customers under a Contract

In the example shown in Figure 1-12, the query would ensure that the same instance of Contract is the one pointing to the pertinent instance(s) of Customer and the pertinent instance(s) of Product.

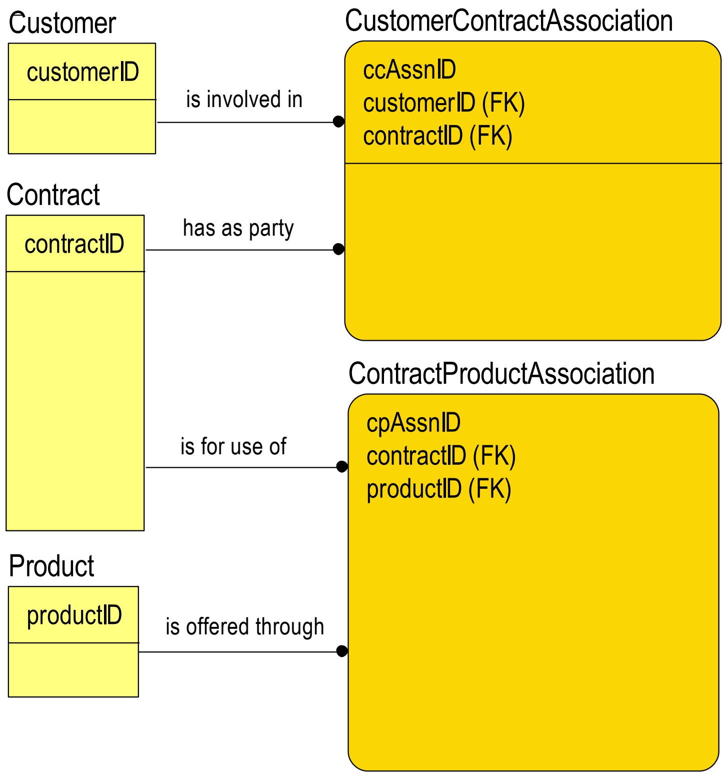


Figure 1-12. Decomposition of N-Ary Association into a Series of Binary Associations

6. Additional Relational Data Structure Patterns – The Subtype Hierarchy Pattern

As was noted briefly in Section 1A above, subtype relationships are parent child relations with cardinality zero or one, which makes the content of entities involved in a relational subtype hierarchy pattern rather simple to convert to RDF triples because each instance of a supertype can be linked to at most one instance of any of the subtypes in the hierarchy. Having triples that explicitly document the supertype–subtype relation for the instances involved not only simplifies the SPARQL queries but also, unlike a typical one-to-many relationship, does so with very little overhead.[15]

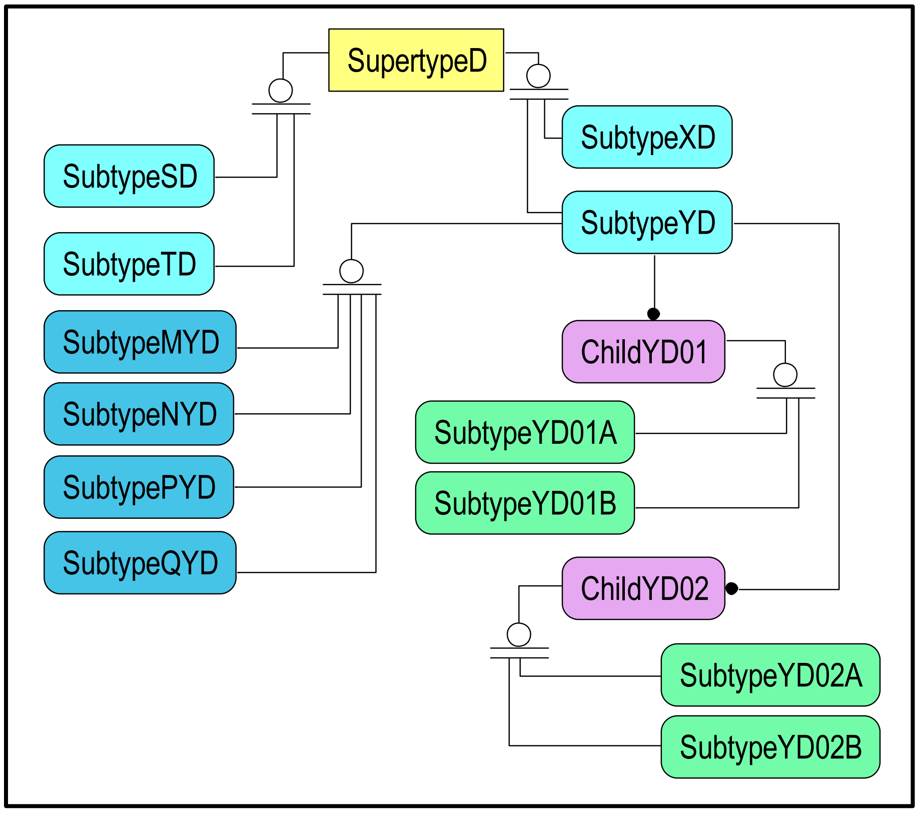


Figure 1-13. Example of the Subtyping Pattern

This can be readily seen in the notional case depicted in Figure 1-13, where one of the subtype branches can go three levels down. The additional triples would make the traversal of the graph simpler and faster by reducing the number of equality tests needed

[15] The predicate in this cases would be simply specializesTo when reading the graph from supertype to subtype, and generalizesTo if reading the graph in the opposite direction.

to retrieve the data.[16] A final point to consider is the situation where the subtypes have identically named attributes. In such cases one has to account for possible naming conflicts. One straightforward solution is to generate the names of the predicates that go into the triples using a concatenation of the table and attribute name.

7. Additional Relational Data Structure Patterns – The Self-Association Pattern

The final common pattern to consider is the self-association pattern, which generally takes the form shown in Figure 1-14. This pattern is extensively used to create functional decompositions, such as those that appear in force structure products, where administrative or command and control relations among military units are documented.

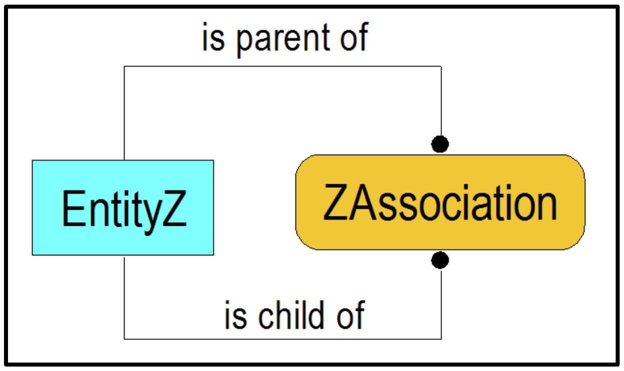


Figure 1-14. The Relational Self Association Pattern

For example in the case of a self-association for the Organization entity, instances of the Organization entity act as parents of other instances of the same Organization entity. Figure 1-15 shows explicitly how the OrganizationAssociation table relates the parent instances of the Organization entity to the child instances of the Organization entity (shown as an entity with the name Organization starred to highlight that it is a copy of the original entity). This is in fact how relational data stores implement this pattern at query time, namely, by creating a virtual copy of the parent table to serve as the child and then traversing the records in the OrganizationAssociation table to retrieve the respective children for each of the parent records.

[16] Another way to handle a type hierarchy, would be to generate a set of triples with a single subject, since by definition, the key of a given instance of a supertype entity is the same as the key of any of its subtype entities. This would make SPARQL queries even simpler.

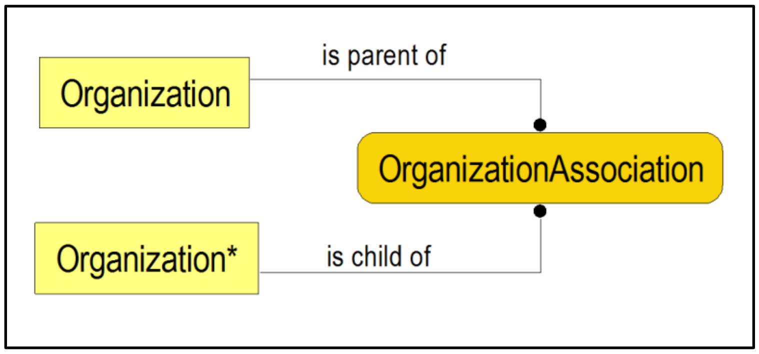


Figure 1-15. Example of the Self-Association Pattern for the Organization Entity

As will be discussed in the following chapter, use of this pattern in relational data stores can lead to very expensive queries when the tables are large because finding all the children requires repeated traversals of the entire self-associative table, e.g., the OrganizationAssociation table in the previous example. On non-optimized machines, the response times for such queries may become unacceptably long. The conversion of data stored in relational stores using this data structure can follow the pattern discussed in Section 2B.5 above, but in cases where the association does not have any attributes of its own, a much simpler approach is to add a triple that states the child resource to which the parent resource is linked to.

The preceding sections have described straightforward, workable approaches to translating into RDF triples data stored in a relational database. The emphasis there has been on the simplicity of the methods, so that the transformation can be applied almost mechanically to any legacy relational data store of interest. One should, however, note that a conversion that focuses on what the purpose and goal of the data is, rather than slavishly following the artifacts imposed by the relational paradigm, may have long term benefits, although it may be more time consuming and costlier up front.

C. The Semantic Layer

In the preceding sections, the focus has been on the straightforward conversion to RDF triples of the data content from legacy relational data stores, and we have just briefly noted in passing that when that is done, the resulting triples no longer state what they mean – in other words, unlike the relational paradigm, an RDF triple store contains no *tables*, everything is a triple. This means that once a record from the table Person is mapped to a resource `PER1000000005` with all its corresponding predicates and values, there is nothing explicitly indicating the origin or the meaning of that triple. All we know is that said resource has a number of predicates such as `p:fname`, `p:lname`, and `p:dob` and what values they have. If we had used a more cryptic naming convention, it would be very hard or potentially impossible to figure out with complete certainty what the triples are about. Fortunately, it is possible to add the required semantics to the data by adding an additional set of triples that allow us to state what the resources mean.

Technically speaking, a collection of RDF triples that one can generate out of the data content from the legacy relational data stores constitutes the *assertion component* or ABox of the corresponding knowledge base, in other words, a set of facts such as stating that the resource `p:PER1000000005` has an attribute `p:fname` with value `"OLIVE"^^xsd:string`. The conceptualization associated with the set of facts residing in the ABox is where the semantics of the data can be captured. It is commonly referred to as the *terminological component* or TBox of the knowledge base, and, typically, when using RDF triples, the TBox is written using the Web Ontology Language (OWL).[17]

The importance of having a TBox component is that one can now state what each of the resources is. For example, one can state that the resource `p:Person` is an OWL class and that the resource `p:PER1000000005` is of type `p:Person`, as shown in the listing below.

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
PREFIX p: <http://simple.example.db2triples/>

p:Person rdf:type owl:Class .
p:PER1000000005 a p:Person .
```

This could be used to eliminate possible ambiguities arising from the use of the prefix PER for triples from a different table, such as one called PlantEliminationReport. All that would be required would be the inclusion of this fact in the pertinent SPARQL query so that only those triples dealing with resources of type p:Person would be examined, and resources from the PlantEliminationReport table would be ignored.

Table 1-2 summarizes the preceding discussion regarding the use of a TBox in a knowledgebase implemented as an RDF triple store.

[17] See https://www.w3.org/TR/owl2-overview/#.

Table 1-2. Utilization of the Semantic Layer in RDF Triple Stores

Adding Semantics to RDF Triples	
Relational Construct	**RDF Representation**
• Entities/Tables	• OWL classes. Each resource in the RDF triple store can be related to the pertinent class, essentially establishing an analogy to the concept of table that exists in a relational store. The TBox can be either a fully-fledged ontology or a minimal set of assertions sufficient to capture only those aspects that support the operational scenarios. Generally speaking, having an ontology enables the use of automated reasoning and classification of resources. It also offers the possibility of managing at the semantic level additional relations among the classes, such as their equivalence, or making subclass restrictions for required attributes, or constraining an individual to never have more than one literal attribute value. If the TBox is held separately and one uses the capability to federate RDF triple stores, as is offered by many open source and proprietary implementations of the RDF triple store paradigm, one could "turn on" or "turn off" these semantic perspectives depending on the need. For example, one could assert the equivalence of the classes Business and Organization for one type of query but treat them as distinct classes for a different query.
• Database	• OWL classes, object properties and datatype properties. The hierarchy implicit in the fact that tables are defined in database physical schemata can also be captured using the TBox. This would allow the resources to keep traceability back to the original sources, with as much metadata as operationally required.

2. Performance Degradation Associated with Specific Representations of Data Via RDF Triples

A. The "Six Degrees of Separation" Use Case

Although relational databases use highly optimized indexing and search algorithms which, generally speaking, guarantee their ability to retrieve in milliseconds data from a single table containing a very large dataset, their performance can be substantially affected when the search requires traversing multiple table joins or making millions of passes through the same very large association table.

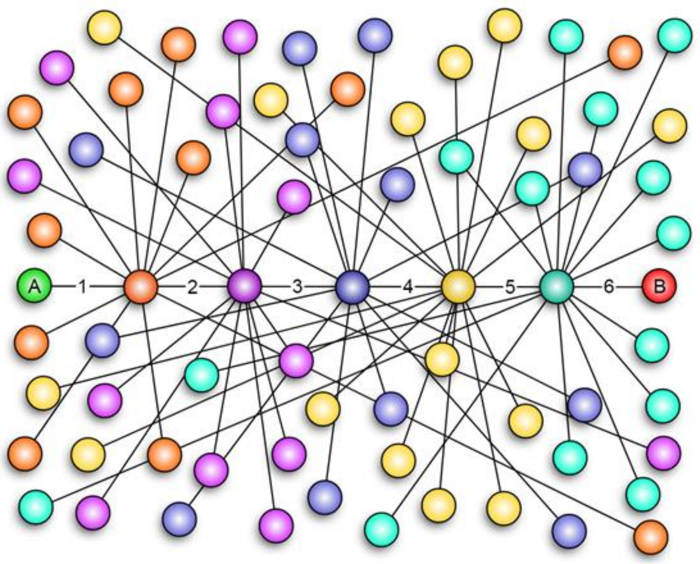


Figure 2-1. Notional Depiction of a Person-Knows-Person Graph[18]

In this section we discuss the latter scenario to use it as the basis for comparison with the performance of an RDF triple store. The use case to be analyzed is depicted in Figure 2-1. The nodes in that figure are meant to represent instances of Person, and the edges stand for the relation "*knows,*" so that starting from any node, one can have a series of threads that read *personX1*-knows-*personX2*-knows-…-knows-*personXN*. The data in this graph can be

[18] Six degrees of separation. (2017, April 23). In *Wikipedia, The Free Encyclopedia*. Retrieved 17:55, May 11, 2017, from https://en.wikipedia.org/w/index.php?title=Six_degrees_of_separation&oldid=776826562.

captured using a relational model such as the one depicted in Figure 2-2, where the nodes would be the records in the Person table and the edges would be encoded in the double-associative table PersonAssociation. The key structure of the PersonAssociation table shows the primary key perId from the Person table appropriately role-named to prevent key unification.

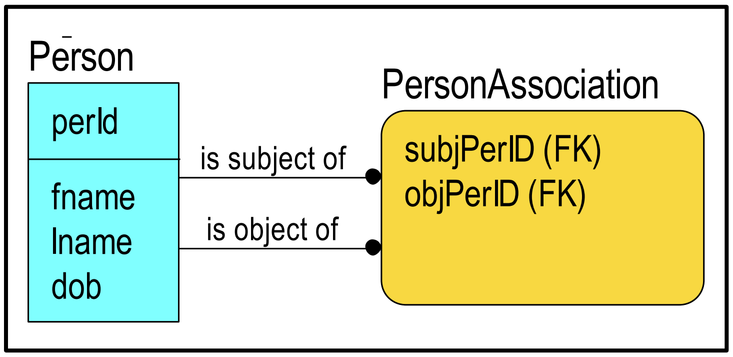
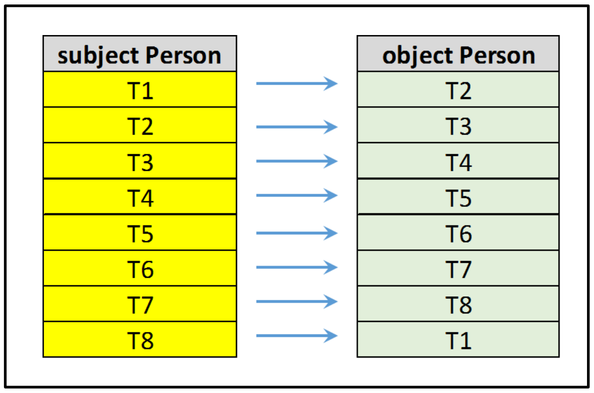


Figure 2-2. Modeling of Person-Knows-Person in a Relational Data Store

The tests described in this section were run on a DELL PC Optiplex 9020, with 16 GIG of RAM and an Intel Pentium 7 with eight cores. The relational data store engine was MySQL 5.7.18-0 running in a Linux box that used the Ubuntu 16.04 LTS distribution. To stress the relational data store, three databases were created. In the first one, the Person table was populated with 128 million records created algorithmically, as discussed in Section 1B.2 above (see Figure 1-3). Similarly, the PersonAssociation table was populated with 128 million associations constructed by partitioning the instances in the Person table into 8 groups of 16 million records each (T1 through T8) and then associating them as shown in Figure 2-3.



subject Person		object Person
T1	→	T2
T2	→	T3
T3	→	T4
T4	→	T5
T5	→	T6
T6	→	T7
T7	→	T8
T8	→	T1

Figure 2-3. Structure of the PersonAssociation Table Content

As can be seen in the figure, the associations constitute a circle, which means that starting from an instance of Person in any subset Tx one can hop any number of times N to reach an instance of Person in another subset Ty. In this scenario, referring back to Figure 2-1 above, the *degrees of separation* indicate the number of edges one needs to traverse from a *start* Person node until one gets to the desired *target* Person node. For example, the node labeled A in Figure 2-1 requires the traversal of six edges before reaching the node labeled B. For simplicity, in this phase of the study, we consider only a maximum of six degrees of separation (SDOS). We also impose an additional restriction on the value of fname (the person's first name) to pare down the number of associations. Specifically, we ask that the *start* Person node have fname = "TAYLOR" and that the *target* Person node have fname = "DORIAN". The SQL queries correspond to the questions shown in Table 2-1.

Table 2-1. Representation in Natural Language of the Six Degrees of Separation Scenario

SDOS Queries	
Label	**Query**
• SDOS-1	• How many instances of Person named "TAYLOR" know an instance of Person named "DORIAN"?
• SDOS-2	• How many instances of Person named "TAYLOR" know an instance of Person who knows an instance of Person named "DORIAN"?
• SDOS-3	• How many instances of Person named "TAYLOR" know an instance of Person who knows an instance of Person who knows an instance of Person named "DORIAN"?
• SDOS-4	• How many instances of Person named "TAYLOR" know an instance of Person who knows an instance of Person who knows an instance of Person who knows an instance of Person named "DORIAN"?
• SDOS-5	• How many instances of Person named "TAYLOR" know an instance of Person who knows an instance of Person who knows an instance of Person who knows an instance of Person who knows an instance of Person named "DORIAN"?
• SDOS-6	• How many instances of Person named "TAYLOR" know an instance of Person who knows an instance of Person who knows an instance of Person who knows an instance of Person who knows an instance of Person who knows an instance of Person named "DORIAN"?

The relational model for SDOS-1 is the one already shown in Figure 2-3. The relational model for SDOS-2 though SDOS-6 is depicted in Figure 2-4.

As mentioned above, two additional databases were created to gain a feeling for the performance of the relational data stores with data loads comparable to what may be expected for an operational implementation of the Army DFS portal. The second data store had also a Person table and a PersonAssociation table. The Person table was populated with 256 million records, and so did the PersonAssociation table, but the T1 through T8 subsets used to create the associations in the PersonAssociation table were now configured to be 32 million in size. For the third database, 512 million records were loaded into the Person table as well as into the PersonAssociation table, with a corresponding increase in the size of the T1 through T8 subsets to 64 million in size. In all cases, the records for the respective Person tables were automatically generated using a random selection of the values for the fname attribute from a list of some 5,000 commonly used first names and a random generation of the strings for the lname and the dob, as discussed earlier.

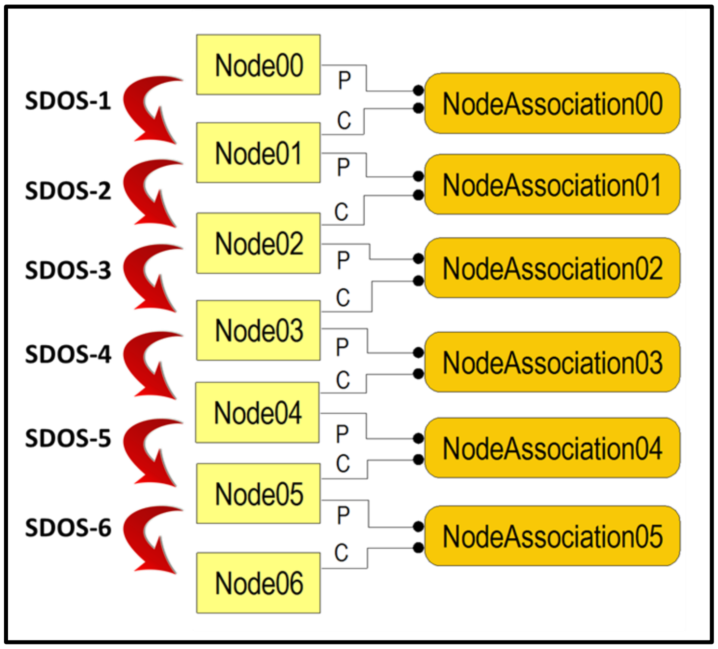
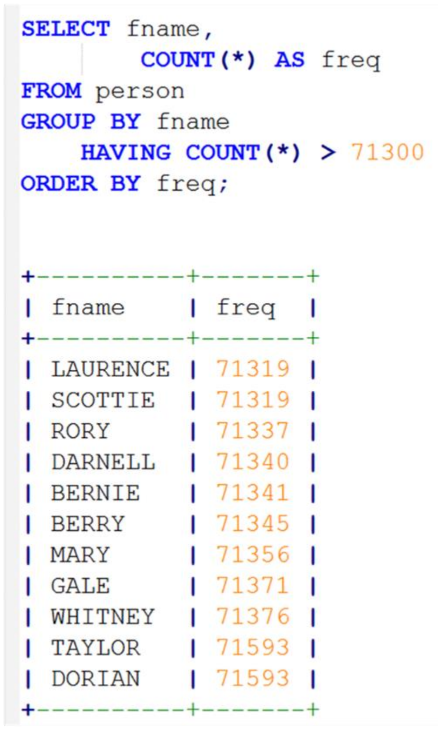


Figure 2-4. Relational Model for the SDOS-6 Query

B. Statistics for the Relational Implementation

1. SDOS-1 Case

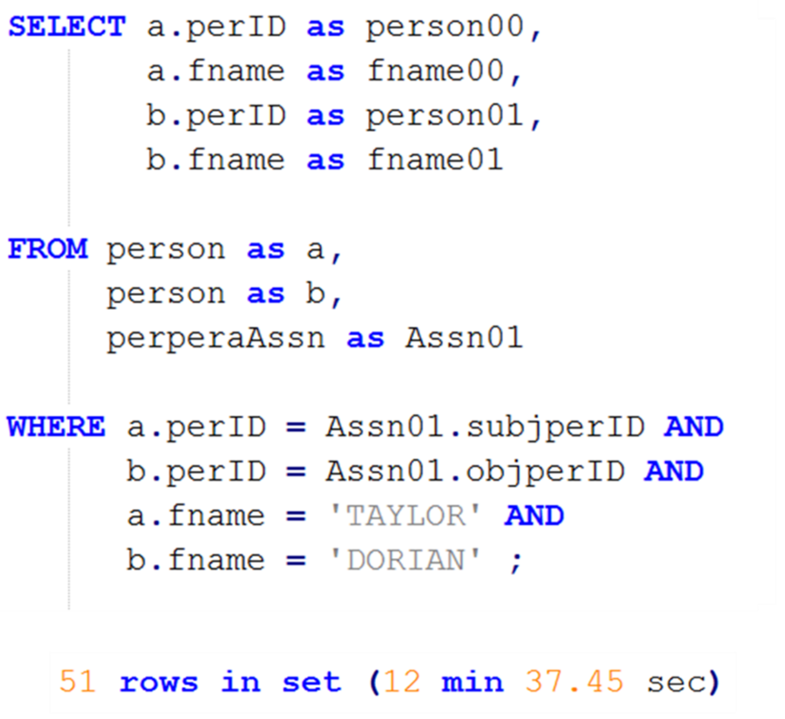


```
SELECT fname,
        COUNT(*) AS freq
FROM person
GROUP BY fname
    HAVING COUNT(*) > 71300
ORDER BY freq;


+----------+-------+
| fname    | freq  |
+----------+-------+
| LAURENCE | 71319 |
| SCOTTIE  | 71319 |
| RORY     | 71337 |
| DARNELL  | 71340 |
| BERNIE   | 71341 |
| BERRY    | 71345 |
| MARY     | 71356 |
| GALE     | 71371 |
| WHITNEY  | 71376 |
| TAYLOR   | 71593 |
| DORIAN   | 71593 |
+----------+-------+
```

Figure 2-5. Values of fname with Highest Frequencies in the chosen Sample Dataset

The values of fname with the highest frequencies for the 128 million record set loaded in the Person table of the first database are shown Figure 2-5. Since the values "TAYLOR" and "DORIAN" are the most frequent names in that set and occur the same number of times, they were chosen for the testing. An index for each of the attributes fname, lname and dob was added to the Person table prior to running the tests, but for the PersonAssociation table no additional indexing was done, leaving that table with just the primary key index automatically given to the composite key made up of the subjperID and the objperID attributes.

The SELECT query used for the SDOS-1 case, as well as the execution time and the number of records retrieved is shown in Figure 2-6.



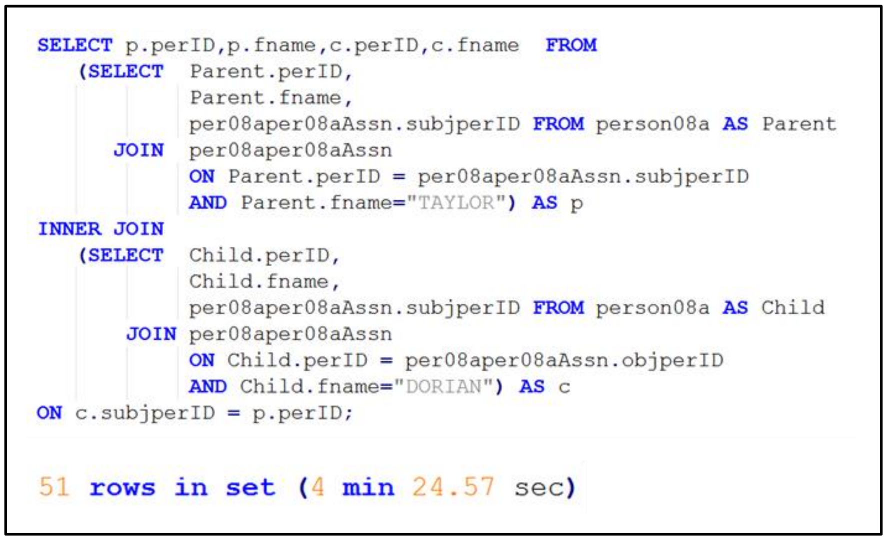
```
SELECT a.perID as person00,
       a.fname as fname00,
       b.perID as person01,
       b.fname as fname01

FROM person as a,
     person as b,
     perperaAssn as Assn01

WHERE a.perID = Assn01.subjperID AND
      b.perID = Assn01.objperID AND
      a.fname = 'TAYLOR' AND
      b.fname = 'DORIAN' ;
```

```
51 rows in set (12 min 37.45 sec)
```

Figure 2-6. SELECT Query and Results for the SDOS-1 Case

These results make it quite apparent that a traversal of a small data set of 128 million records in the Person table to find those instances of Person that have fname = “TAYLOR”, and for each one of those records the subsequent traversal of the 128 million records in the PersonAssociation table to find the instances of Person that they are linked to, followed by the testing of its fname attribute to satisfy the restriction that it be equal to “DORIAN”, is a very time-consuming process. The immediate implication is that to bring the response time for this query into the millisecond regime, as required for example for interactive usage in a web portal, one needs to deploy at a minimum a machine with much more computational capacity, as well as optimized disk architecture.

It should be noted, however, that at times, the execution plan chosen by a database engine to carry out a SELECT query may be not be as optimal as one would expect. In that regard, it is worth mentioning that generic queries where all the equality tests are placed in the WHERE clause, such as the one shown Figure 2-6, may be less efficient than queries that use an inner join. This appears to be the case here. When running the equivalent query using an inner join for the SDOS-1 case, one notices a marked improvement (see Figure 2-7). However, even though the SELECT query now runs about three times faster than before, the overall time is still too slow for an interactive session, and the need for hardware optimization is still valid.



```
SELECT p.perID,p.fname,c.perID,c.fname  FROM
   (SELECT  Parent.perID,
            Parent.fname,
            per08aper08aAssn.subjperID FROM person08a AS Parent
      JOIN  per08aper08aAssn
            ON Parent.perID = per08aper08aAssn.subjperID
            AND Parent.fname="TAYLOR") AS p
INNER JOIN
   (SELECT  Child.perID,
            Child.fname,
            per08aper08aAssn.subjperID FROM person08a AS Child
       JOIN per08aper08aAssn
            ON Child.perID = per08aper08aAssn.objperID
            AND Child.fname="DORIAN") AS c
ON c.subjperID = p.perID;

51 rows in set (4 min 24.57 sec)
```

Figure 2-7. SELECT Query Using INNER JOIN and Results for the SDOS-1 Case

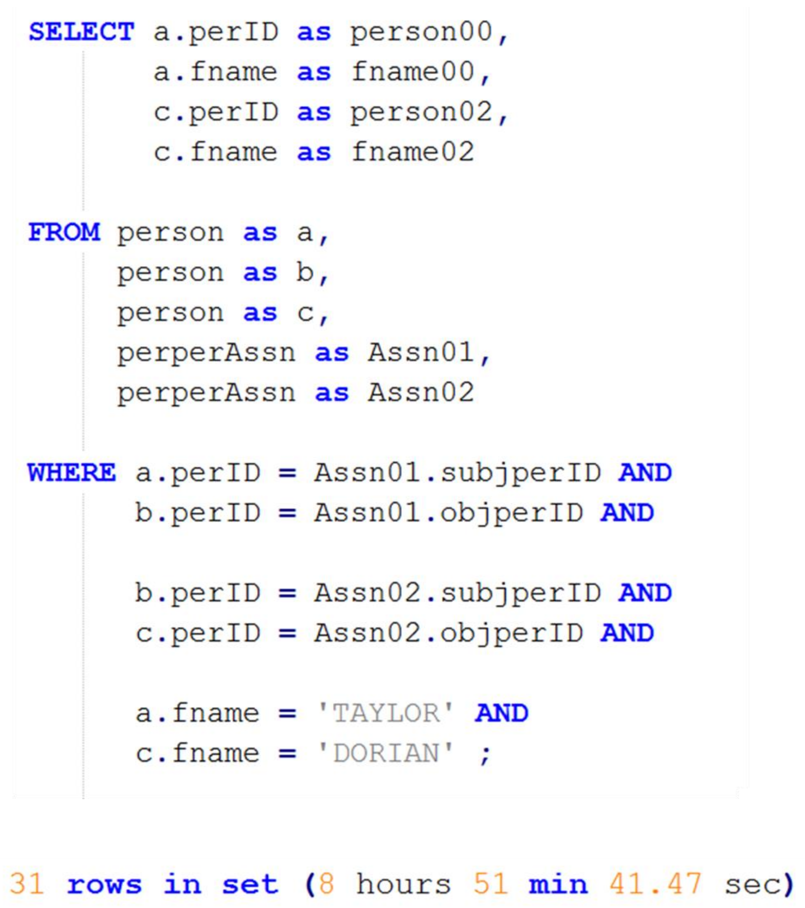
2. SDOS-2 Case

The SELECT query used for the SDOS-2 case, as well as the execution time and the number of records retrieved is shown in Figure 2-8. The execution time is now approximately 42 times longer than that required to complete the un-optimized SDOS-1 query (see Figure 2-6 above), and clearly shows that with standard hardware and software, this type of query cannot be used in interactive environments, where users expect response times under one second.

It should be noted that the PersonAssociation table does not have inverse relationships. That is, it does not have a row stating that Taylor knows Dorian, as well as a row stating that Dorian knows Taylor. The query still works, but only because in this example we are checking connections between people with different names. The query would have to be modified if one wanted to query whether a person named Taylor knows someone, who knows someone named Taylor. In that case one would need to add to the WHERE clause:

AND a.perID != c.perID

Otherwise, the query would return a row for every person association whose subject is Taylor. Also note that because the PersonAssociation table does not have inverse relationships, then presumably this query won't return the same results if one swaps the names.



```
SELECT a.perID as person00,
       a.fname as fname00,
       c.perID as person02,
       c.fname as fname02

FROM person as a,
     person as b,
     person as c,
     perperAssn as Assn01,
     perperAssn as Assn02

WHERE a.perID = Assn01.subjperID AND
      b.perID = Assn01.objperID AND

      b.perID = Assn02.subjperID AND
      c.perID = Assn02.objperID AND

      a.fname = 'TAYLOR' AND
      c.fname = 'DORIAN' ;


31 rows in set (8 hours 51 min 41.47 sec)
```

Figure 2-8. SELECT Query and Results for the SDOS-1 Case

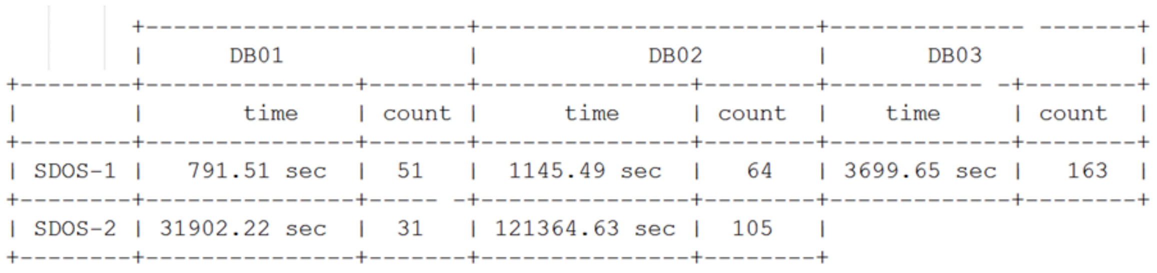
3. Summary of Results for the Larger Data Sets

Figure 2-9 gives the statistics regarding data set size and highest frequencies for the first names for the databases used in the additional experiments, together with the results mentioned for the first database (see Figure 2-5 above).

```
+--------+---------------+-------------+-----------+
| Store  |     Size      | First Name  | Frequency |
|        | (in Millions) |             |           |
+--------+---------------+-------------+-----------+
| DB01   |      128      | DORIAN      | 71593     |
|        |               | TAYLOR      | 71593     |
+--------+---------------+-------------+-----------+
| DB02   |      256      | CLEO        | 142720    |
|        |               | OLLIE       | 142944    |
+--------+---------------+-------------+-----------+
| DB03   |      512      | GAIL        | 284902    |
|        |               | FRANKIE     | 285198    |
+--------+---------------+-------------+-----------+
```

Figure 2-9. Summary of Size, and First Name Frequencies

During this phase of the study we found that the performance of the relational engine continued to degrade as the size of the data sets increased. Figure 2-10 summarizes the results obtained for the SDOS-1 and SDOS-2 cases. As can be seen in the figure, the time taken to retrieve data from the database with 256 million records for the SDOS-2 case was already so long that the IDA team decided there was no point in conducting further testing with the third database containing 512 million records. Instead, the IDA team decided to explore whether an approach using essentially materialized views of the queries could bring the performance within the millisecond regime. The results of that approach are presented in the next section.



```
         +-----------------------+-----------------------+-------------- -------+
         |       DB01            |            DB02       |       DB03           |
+--------+---------------+-------+---------------+--------+----------- -+--------+
|        |        time   | count |       time    | count  |    time      | count  |
+--------+---------------+-------+---------------+--------+-------------+--------+
| SDOS-1 |   791.51 sec  |  51   |  1145.49 sec  |   64   | 3699.65 sec |   163  |
+--------+---------------+----- -+---------------+--------+-------------+--------+
| SDOS-2 | 31902.22 sec  |  31   | 121364.63 sec |  105   |
+--------+---------------+-------+---------------+--------+
```

Figure 2-10. Performance Statistics for Larger Data Sets

4. Use of Materialized Views as an Approach to Speed Up Performance

As briefly alluded above, queries involving joins create an additional overhead during the execution of a SELECT query. Database administrators, therefore, routinely create so-called *materialized views*, that is, data objects that contain the results of complex and costly SELECT queries so that the results are effectively cached, and, therefore, one does not need to re-run the original, costly SELECT queries, but can subsequently access the precomputed values from a flat table with minimal overhead.

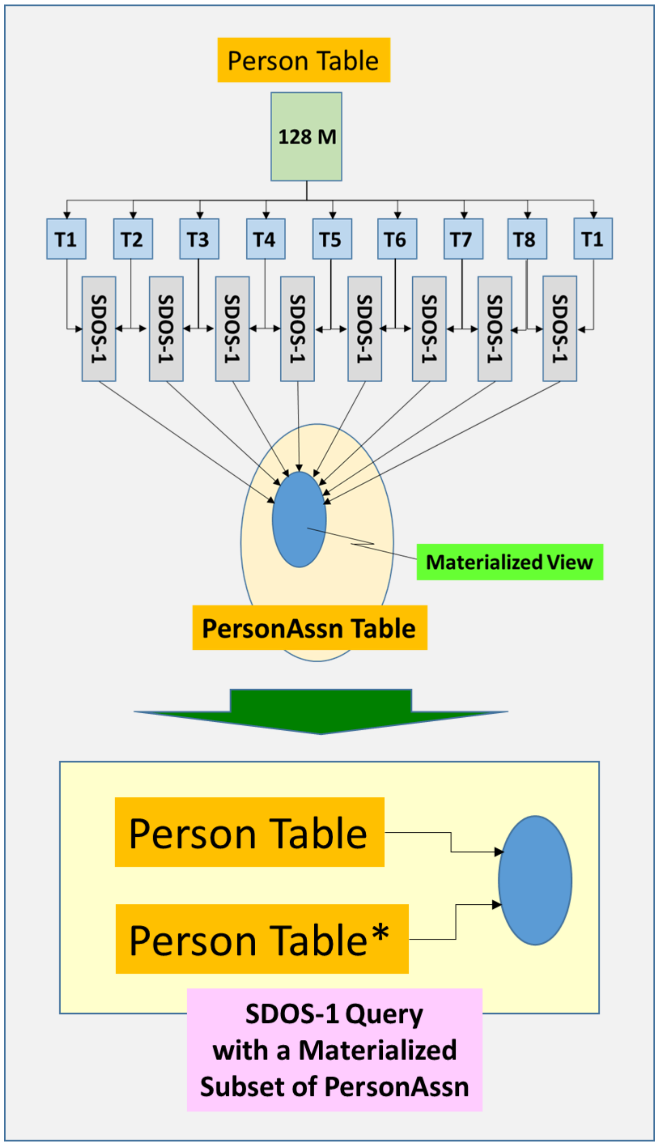
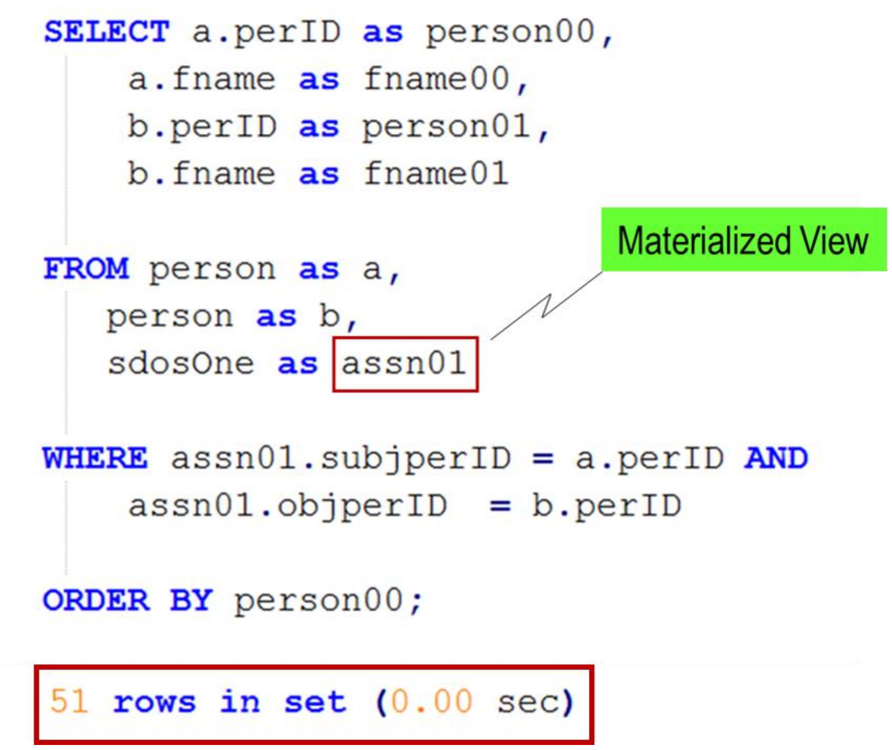


Figure 2-11. Schematic Depiction of the Process to Generate a Materialized View Containing a Subset of PersonAssociation for the SDOS-1 Test Case

Since by design the records in the PersonAssociation table were chunked into eight distinct subsets corresponding to the associations among the T1 through T8 sets in the Person table (see Figure 2-3 above), one can find for each of the T-pairs the records that satisfy the SDOS-1 case, that is, the associations of instances of Person in each source T-subset with fname = "TAYLOR" that are linked to instances of Person in the target T-subset with fname = "DORIAN" and then combine those results into a new, substantially smaller subset of the PersonAssociation table that can serve as the materialized view of the costly SDOS-1 query. The entire process can be executed as a script that creates temporary tables to store the T subsets, runs the SDOS-1 query for each of the pairs, and finally collects the individual subsets into a substantially pared-down version of the PersonAssociation table that can now be used to run the SDOS-1 query (see Figure 2-6 above). The process is schematically shown in Figure 2-11. Using the resulting materialized view for the association table but still using the full Person table the query now runs in millisecond time (see Figure 2-12).



```
SELECT a.perID as person00,
    a.fname as fname00,
    b.perID as person01,
    b.fname as fname01

FROM person as a,
   person as b,
   sdosOne as assn01

WHERE assn01.subjperID = a.perID AND
    assn01.objperID  = b.perID

ORDER BY person00;
```

```
51 rows in set (0.00 sec)
```

Figure 2-12. SELECT Query and Results for the SDOS-1 Case with a Materialized View of the PersonAssociation Table

5. Preliminary Lessons Learned

The tests described in the preceding section were conducted as part of an effort to baseline the performance of typical relational data stores, so that one can compare them with the performance of graph databases. They do not represent anything novel, except the fact that most users seldom have the need to deal with really large data sets, and, therefore, they seldom experience any serious performance degradation. The tests confirm the following points:

- Substantial performance degradation in relational data stores can occur when attempting to manipulate record sets with a size as small as 128 million records. It should be noted, however, that the above statement applies for arbitrary queries. When dealing with frequently executed queries, a database will likely have a design that specifically addresses those needs, which, coupled with judicious indexing, can noticeably improve database performance.

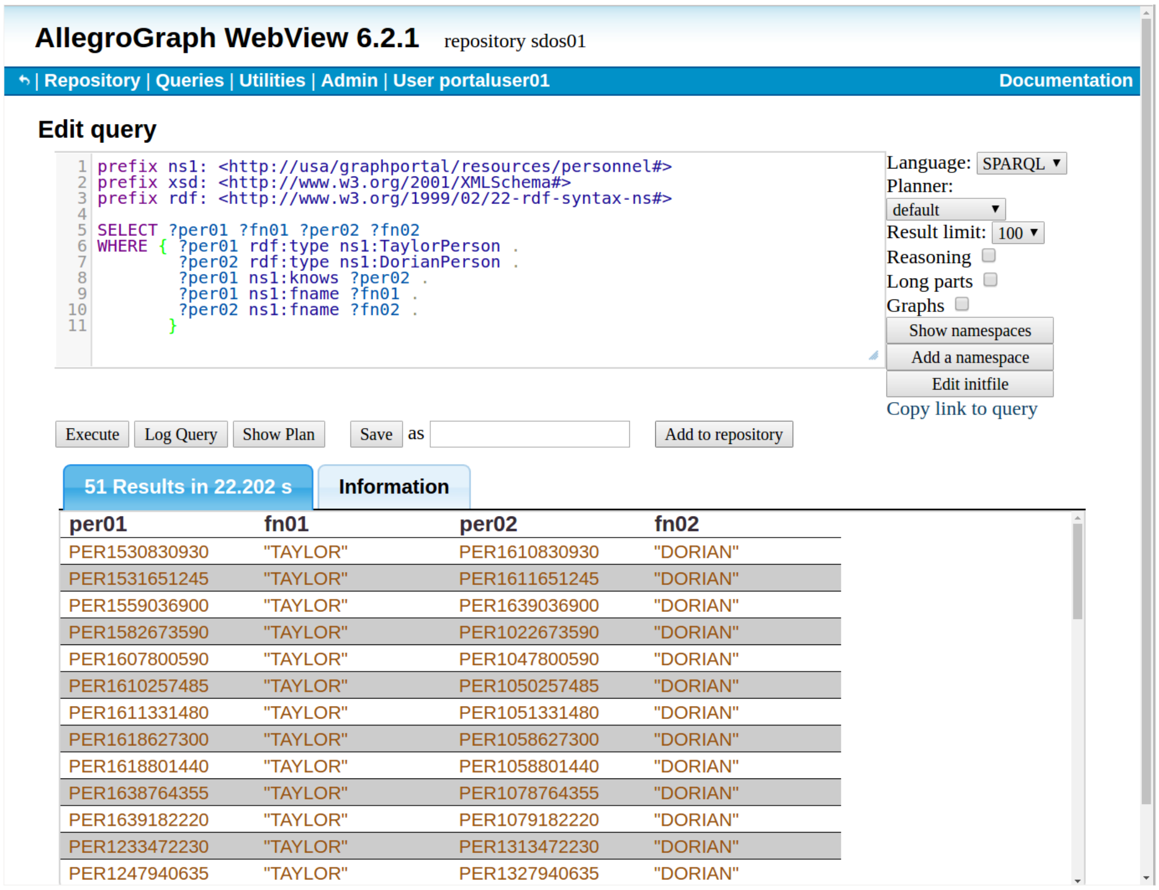
- Performance degradation in relational data stores for certain types of queries appears to be directly proportional to the size of the record sets, but improvement in performance can be obtained by judicious choice of query style (generic SELECT queries perform less well than SELECT queries that use inner joins).
- The execution times obtained with the software and hardware available during this phase of the study should not be considered normative or definitive when comparing NoSQL alternatives to relational data stores. They are meant to highlight general trends when operating with data sets larger than what is encountered in most day-to-day operations.
- Machines with substantially more computational power and optimized disk architectures should be considered when working with large datasets.
- The suggestion for using materialized views as described here does not scale well, in the sense that to be able to run the SDOS-1 case with any possible combination of fname values would require creating on the order of 5 million such tables, which is obviously impractical. On the other hand, the number of attributes that may be relevant for force structure cases may be substantially smaller and thus more tractable. When that is the case, the performance of relational data stores continues to be excellent.
- The impact of large data sets extends beyond the immediate use of the relational data stores. Thus, for example, libraries used to generate RDF triples and serialize them substantially slow down if one attempts to generate the complete set of triples out of a mid-size table (e.g., 128 million records) since the libraries do not appear to perform intermediate data flushes but rather wait until the entire set has been processed before writing to disk. Rewriting the code to process the data in small chunks substantially improves the overall performance of the scripts.

C. Statistics for a Graph Database Implementation

To gain some insight into the performance of RDF triple stores that could be employed in the future Army DFS Portal, the project sponsor approved the acquisition of a commercial license for the Allegro Graph triple store. The license was configured to have no limitations with respect to the number of RDF triples that can be loaded into any given repository. Allegro Graph licenses are tied to the number of cores that the application can use. The fee for the activation of each core is on the order of $1,300. Two cores were activated for the trial described in this deliverable.

To be able to compare the relational data store engine with the Allegro Graph triple store, the 128 million records from the Person table of the first database were converted to RDF triples following the approach described in Section 1B.3 above. As suggested in that section, the associations in the PersonAssociation table were converted into explicit triples using the predicate p:knows. In addition, two semantic statements were made for each resource. The first one stated that the resources used as the subject of the triples generated from the Person table were of type p:Person. This was done to exemplify the mapping of the concept of a relational entity/table to a semantic concept in the TBox of the knowledgebase as discussed in Chapter 1. The second statement was a triple added to those resources from the Person table whose fname values were either “TAYLOR” or “DORIAN” and which essentially mirrored in part the effect of having a materialized view for the SDOS-1 case in the RDF triple store. This represented an additional 143,186 triples loaded into the knowledgebase used for the testing. These triples were of the form:

```
ns1:PER1000000005 a ns1:TaylorPerson .
ns1:PER1000490075 a ns1:DorianPerson .
```



AllegroGraph WebView 6.2.1 repository sdos01

Edit query

```
prefix ns1: <http://usa/graphportal/resources/personnel#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?per01 ?fn01 ?per02 ?fn02
WHERE { ?per01 rdf:type ns1:TaylorPerson .
        ?per02 rdf:type ns1:DorianPerson .
        ?per01 ns1:knows ?per02 .
        ?per01 ns1:fname ?fn01 .
        ?per02 ns1:fname ?fn02 .
      }
```

Language: SPARQL
Planner: default
Result limit: 100

51 Results in 22.202 s

per01	fn01	per02	fn02
PER1530830930	"TAYLOR"	PER1610830930	"DORIAN"
PER1531651245	"TAYLOR"	PER1611651245	"DORIAN"
PER1559036900	"TAYLOR"	PER1639036900	"DORIAN"
PER1582673590	"TAYLOR"	PER1022673590	"DORIAN"
PER1607800590	"TAYLOR"	PER1047800590	"DORIAN"
PER1610257485	"TAYLOR"	PER1050257485	"DORIAN"
PER1611331480	"TAYLOR"	PER1051331480	"DORIAN"
PER1618627300	"TAYLOR"	PER1058627300	"DORIAN"
PER1618801440	"TAYLOR"	PER1058801440	"DORIAN"
PER1638764355	"TAYLOR"	PER1078764355	"DORIAN"
PER1639182220	"TAYLOR"	PER1079182220	"DORIAN"
PER1233472230	"TAYLOR"	PER1313472230	"DORIAN"
PER1247940635	"TAYLOR"	PER1327940635	"DORIAN"

Figure 2-13. Results and Performance for the SDOS-1 Case using the Proprietary RDF Triple Store Allegro Graph

One of the evident benefits of using that approach is that one can eliminate right away the need for string value comparisons via the FILTER feature supported in SPARQL, which, depending on the maturity of the implementation, can noticeably increase the execution time of the SPARQL query. Instead, one can explicitly identify the resources pertinent for the SDOS-1 case, namely those that have a predicate fname = “TAYLOR” or a predicate fname = “DORIAN” in the SPARQL query, as shown below.

```
PREFIX ns1: <http://usa/graphportal/resources/personnel#>
PREFIX xsd: <http://www.wc3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.wc3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?per01 ?fn01 ?per02 ?fn02
WHERE { ?per01 rdf:type ns1:TaylorPerson .
        ?per02 rdf:type ns1:DorianPerson .
        ?per01 ns1:knows ?per02 .
        ?per01 ns1:fname ?fn01 .
        ?per02 ns1:fname ?fn02 .
      }
```

Figure 2-13 shows a screen capture of the results. The query execution time is given as 22.202 seconds, which, although substantially better that what was obtained for the SDOS-1 case using a relational data store (see Figure 2-6 and Figure 2-7) is still too slow for an interactive scenario. We also note that on the Linux machine used for the testing, the application actually took substantially longer to refresh the screen. This may not be the fault of the Allegro Graph tool but rather an artifact of the current Linux kernel, which has been reported to freeze randomly. The IDA team is investigating the matter and plans to update the Allegro Graph results in the next deliverable.[19]

[19] In discussions with personnel from the Allegrograph technical support team, it became clear that the statistics displayed in the WebView interface at the end of query executions do not correspond to the total time taken by the application to conduct the queries. The support team suggested one should not rely on those values for comparison purposes, but to use the HTTP interface with curl.

3. Solution Architecture Options that Enable Integration of Graph Databases in a DFS Portal Implementation

A. Integration Options

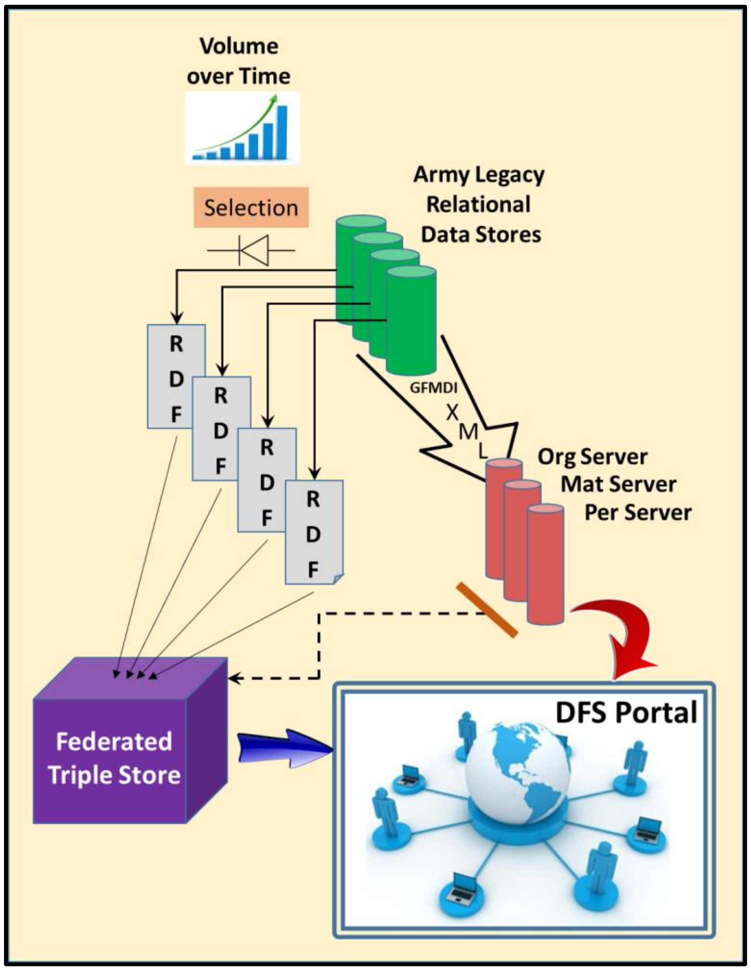


Figure 3-1. High-Level View of Key Components and Architecture Integration Options

Some of the key components to be considered when discussing options for the solution architecture intended to support the planned Army DFS Portal are depicted in Figure 3-1. One has, to begin with, all the legacy relational data stores that currently contain the information needed to generate force structure products. It is not clear at this point whether some of these systems will continue to operate after the DFS Portal has become operational. In that case, one would need to understand whether their data will be exported for reuse using only an approved exchange format, such as the GFM DI XML, or whether tailored extraction, transformation, and loading (ETL) processes will be maintained to move the data.

Closely related to the above question is the fact that the Army has been a strong supporter of the GFM DI activities. Under that umbrella, it has worked on the development of an Army Organization Server. An effort to stand up additional servers was originally envisioned and, in that case, there will be, at some later date, an Army Materiel Server and an Army Personnel Server that will constitute the authoritative data sources needed by the Army for a substantial portion of the force structure products.

From the above, it follows that the timelines associated with the various activities is of paramount importance when looking for optimal integration options for the graph database technology to support the Army DFS Portal. In this phase of the study, it is arguably sufficient to look at the choices that derive from certain assumptions one could make regarding said timelines. Table 3-1 summarizes the choices considered.

Table 3-1. Architecture Integration Options for Graph Databases in Support of the Army DFS Portal

Architecture Integration Options Overview	
Assumption	**Role for Graph Databases**
• All legacy relational data stores that collect and maintain data pertinent to the generation of force structure products will be replaced by the Army DFS Portal, once it is fully operational	**Main Role:** • As shown in Chapter 1 above, graph databases can be used effectively and efficiently as a replacement for the costly, time-consuming and labor intensive ETL processes that otherwise would be needed to move all the legacy data to the final repository that will support the DFS Portal operations. The two options for how to use graph databases in this scenario are: o **Option 1:** After the legacy data has been collected in an RDF data lake, it could serve as the backend data store for directly powering the functionality of the Army DFS Portal. o **Option 2:** The RDF data lake could serve as an intermediate data staging place from where to conduct in a cost effective manner the cleanup and harmonization of the legacy data before moving it into the final repository intended to support the Army DFS Portal. (This option does not make any assumptions regarding the technology used in the final repository. In other words, one could continue to use a relational data store as the backend of the DFS Portal, or use a NoSQL implementation.) **Issues to consider:** • In this scenario the migration of all the legacy data does not include the GFM DI servers. Therefore, the role of those servers needs to be assessed and a final determination must be made on how they will support the Army DFS Portal.

Architecture Integration Options Overview	
Assumption	**Role for Graph Databases**
• Some of the legacy relational data stores that collect and maintain data pertinent to the generation of force structure products will continue to operate after the Army DFS Portal is up and running.	**Main Role:** • With respect to the legacy data stores that will be terminated after the DFS Portal becomes operational, the use of graph database technology and the role of the RDF data lake would be the same as the one discussed above. **Issues to consider:** • If the vision is to continue extracting data from the legacy systems that remain in place after the DFS Portal is operational, then, given the advantages offered by graph databases in terms of data integration, the Army should consider using the RDF data lake as the place to collect the extracts from the legacy systems. The same two options discussed above would apply here as well. • One aspect that also must be addressed is whether these legacy systems are queried each time one needs data, or whether some application in the RDF data lake periodically refreshes the triples. This must be assessed in light of operational needs, namely, whether one needs access to live data, or whether one can live with data that may be slightly out of date – perhaps by 24 hours, if one refreshes the triples daily. • Also, what was said with respect to the GFM DI servers would apply under this scenario.

It is probably too early to decide whether in the case of using an RDF data lake as the final repository to power the DFS Portal there would be any substantive advantages to having a single data lake or a number of those data lakes that can be federated as needed.

Another aspect of the solution architecture that needs to be considered is whether the integration of graph database technology is intended to happen quickly or whether due to other factors, such as contractual obligations and funding levels, one could proceed slowly at first and then ramp up the utilization of this technology. Figure 3-1 alludes to these two aspects by noting that the conversion of legacy data into RDF triples could be applied only to selected components of the legacy physical schemata, and that the volume of utilization could increase over time as appropriate.

Finally, although the role of the GFM DI servers in the context of the DFS Portal has to be determined and finalized, Figure 3-1 also suggests that one possible scenario may be to have the GFM DI servers continue their existence for the purpose of supporting specialized functions outside what the DFS Portal is intended to cover, but then to feed their data into the RDF data lake for subsequent utilization by the DFS Portal.

4. Conclusions and Recommendations

A. Conclusions

Based on the analytical work performed during this phase, the IDA team concluded the following:

- All the data structures that are likely to be found in the pertinent Army relational legacy data stores – namely, those containing the source data needed to populate the planned Army DFS Portal – can be re-expressed in a straightforward manner using RDF triples. The degree of complexity of the transformation chosen depends on strategic considerations, such as reuse and expansion of the data to satisfy novel and emerging uses.
- Use of a "semantic layer" in the form of an appropriately sized ontology is quite useful for organizing the resources in an RDF triple store in the same way that data is bundled in relational data stores under the concept of a "table." The semantic layer could also be used to retain traceability back to the data sources.
- Certain types of data structures common in relational data stores can lead to very poor data retrieval performance – such as in the canonical example of multiple layers of node dependencies found in networks, which has been popularized under the rubric of "six degrees of separation." Pre-filtering and the use of materialized views essentially eliminates the performance issue in the relational stores, although it reduces flexibility and adds complexity to the physical schema. Similar approaches can also be used to improve the performance of RDF triple stores, but the downside implications may be handled more elegantly through judicious use of federated triple stores and special-purpose hardware and software.
- The solution architecture options that can support the integration of graph databases in the mix of technologies needed to implement the planned Army DFS Portal are generally satisfactory, and a final determination will require the inclusion and analysis of the concept of operations for the planned DFS portal and the timelines associated with the key Army information systems.
- Similarly, the selection of best-of-breed options may be more sensitive to the concept of operations for the planned DFS portal than to factors of size, scalability, and data retrieval performance.

B. Recommendations

For this stage of the study, the preliminary recommendations are as follows:

- Continue during the next phase the evaluation of available graph database implementations, both proprietary and open source, and expand the scope to include other promising No-SQL choices.
- Continue to use rapid prototyping techniques to collect performance statistics that can inform both the selection process of the optimal graph database implementation and its integration into the mix of technologies needed to implement the planned Army DFS Portal.
- Explore applicable mitigation strategies for potential risks that would arise from the adoption of graph database technology as a materiel solution for the planned Army DFS portal.

Appendix A
Survey of Graph Database Implementations

1. Introduction

This appendix constitutes a follow-on to the survey initiated in the previous phase of the study. As was the case before, the main purpose of the appendix is to highlight the types of features that one should be focusing on when considering a graph database as a possible material solution for a project. The IDA team considers that in the context of the planned Army DFS Portal among the most important characteristics to focus on should be the programming language used for the implementation of the graph database, the availability of one or more application program interfaces (APIs), whether the vendor/provider offers good documentation for their implementation, and the availability of any other type of technical support such as discussion groups, training courses, etc. Finally, it is also important to consider the types of licenses for the products being offered and their availability via download.

2. Basic Definitions

A graph database, also called a graph-oriented database, is a type of NoSQL database that uses graph theory to store, map, and query relationships. A graph database is essentially a collection of nodes and edges. Each node represents an entity (such as a person or a business) and each edge represents a connection or relationship between two nodes. In a graphical visualization, the edges are the lines that connect any two nodes. Every node in a graph database is defined by a unique identifier, a set of outgoing edges and/or incoming edges, and a set of properties expressed as key/value pairs. Each edge is also defined by a unique identifier, a starting-place (node), and an ending-place (node), together with a set of properties. This essentially means that if one can draw a diagram of the data on a blackboard, then one can also store that data in a graph database.

It should be noted that the terms "node" and "edge" are interchangeable with "vertex" and "arc" respectively, with one or the other terminology being more common depending on the specific context, such as data modelling versus mathematics, or general graphs versus directed graphs (digraphs), etc.

RDF databases, also known as RDF triple stores, are a kind of graph database and thus belong in the family of NoSQL solutions. The basic unit of data storage in an RDF triple store is the triple, which is analogous to a declarative statement in natural language, composed of a subject (node), a predicate (edge), and an object (node), but where all the pieces of the statement conform to the RDF syntax. For example, the declarative statement *John works at IBM*, maps to the triple:

<http://example.com/people#John><http://example.com/roles#worksAt><http://example.com/organization#IBM>

where the strings enclosed in brackets constitute the node-edge-node basic unit of data storage.[20] RDF was designed in a slightly earlier era more concerned with knowledge representation, standards, and portability. These are still its strengths. Some triple stores described here hew quite closely to the standards. Others are designed with some range of tactical use cases in mind. These have added features to make them faster, or more scalable, etc., in order to be more competitive with other types of graph databases.

Unless otherwise stated, all RDF triple stores have ways of supporting the W3C standard SPARQL query language, which is a declarative pattern-matching language using SQL-inspired syntax. It is capable of the full suite of Create, Read, Update, and Delete (CRUD) operations.[21]

The following sections provide short descriptions – initiated in Phase 1 of the study, and documented in the first deliverable – for some of the most commonly used graph databases available today.

3. Graph Engine

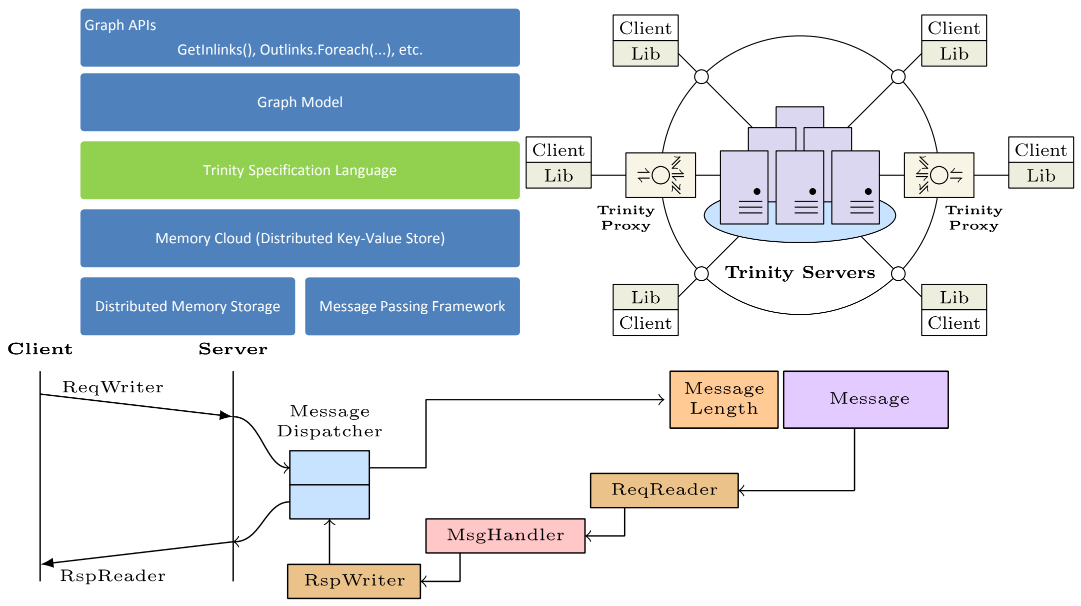
Graph Engine is a proof-of-principle demonstration from Microsoft, code-named "Trinity" prior to its open source release. It was announced in 2013, as a new database engine capable of efficiently serving Satori, a knowledge base supporting the Bing search engine.[22] The mention in its documentation of "triples" and "SPARQL" queries strongly implies that Satori is RDF-based. Graph Engine itself, though, is a .NET technology for building scalable database applications. It comprises a domain-specific schema language,[23] efficient network protocols, and in-memory compact data representations, as well as Visual Studio and GUI tools (see Figure A-1). Its main goal is to support fast queries, traversals, computations, etc., by keeping the entire data set in RAM, perhaps across a cluster of cooperating servers. The specific needs of an application, though, are what drive the schema designs.

[20] The syntax of RDF is extensively covered in https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/. A node in RDF is also referred to as a *resource*, and in RDF all resources must be uniquely identified using an International Resource Identifier (IRI), a generalization of the Uniform Resource Identifier (URI). The most common type of an IRI, is the Universal Resource Locator (URL), which is a resolvable web address used on the Internet to reach web pages and web content in general.

[21] https://www.w3.org/TR/sparql11-query/ and http://www.w3.org/TR/sparql11-update/

[22] https://arstechnica.com/information-technology/2012/06/inside-the-architecture-of-googles-knowledge-graph-and-microsofts-satori/

[23] Trinity Specification Language (TSL)



Graph APIs
GetInlinks(), Outlinks.Foreach(...), etc.
Graph Model
Trinity Specification Language
Memory Cloud (Distributed Key-Value Store)
Distributed Memory Storage
Message Passing Framework
Client
Lib
Trinity Proxy
Trinity Servers
Client
Server
ReqWriter
Message Dispatcher
Message Length
Message
ReqReader
MsgHandler
RspWriter
RspReader

Figure A-1. Overview of the Graph Engine Components

The communication protocols may be synchronous, asynchronous, or HTTP-based. HTTP protocols are a special type of synchronous protocol for building REST APIs.

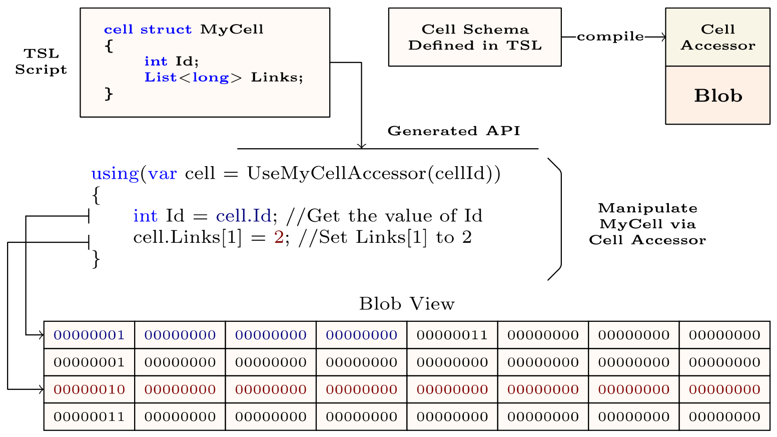
a. API Support

The site, https://graphengine.io/, has a demo applications section, largely using C# as its programming language and making liberal use of Microsoft's Language Integrated Query (LINQ) for processing the data. In fact, using LINQ is recommended, since it allows for many behind-the-scenes optimizations to occur, boosting performance. LINQ is supported in C#, Visual Basic, and F# via query expressions.

b. Language Support

As stated in the previous section, applications can be coded using any language targeting the .NET platform, although the LINQ features of C#, F#, and Visual Basic make those languages preferable.

The Trinity Specification Language (TSL) is the C-like data specification language at the core of Graph Engine. It is used to specify data schemata, communication protocols, and servers (which serve defined protocols). Once the TSL has been authored, MSBuild.exe compiles it into a .NET assembly that can be referenced by both server-side and client-side application code for accessing, querying, performing computations on, etc., with the data (see Figure A-2).


TSL Script
Cell Schema Defined in TSL
compile
Cell Accessor
Blob
Generated API
Manipulate MyCell via Cell Accessor
Blob View

```
cell struct MyCell
{
    int Id;
    List<long> Links;
}
```

```
using(var cell = UseMyCellAccessor(cellId))
{
    int Id = cell.Id; //Get the value of Id
    cell.Links[1] = 2; //Set Links[1] to 2
}
```

00000001	00000000	00000000	00000000	00000011	00000000	00000000	00000000
00000001	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000010	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000011	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figure A-2. Data Access via Trinity Specification Language (TSL)

c. Query Language Support

Aside from the .NET data accessors, and their cooperation with LINQ, no built-in query languages are supported by Graph Engine.

d. Third-Party Integrations

None known.

e. Download and Licensing Options

Graph Engine requires Windows, i.e., nowhere in the documentation is it stated that it would run using something like Mono on Linux. It must be run on a version of the Windows OS that is modern enough to support PowerShell 3.0 or later. To develop software using Graph Engine, Visual Studio (VS) 2012 or later is required and the Graph Engine VS Extension must be downloaded. For the embedded version, the OS version may be Windows 7 or later. The IDA team tested and found out that “GE embedded” works on Visual Studio 2017 Community Edition running on Windows 7 Professional.

For server and proxy roles in a cluster configuration, the OS must be Windows Server 2008 R2 or later. An available GUI tool (GE Configuration Editor) allows one to design the topology of a cluster of cooperating graph data servers and proxy servers. The configuration is saved in a human-readable (and editable) file called `trinity.xml`.

The Graph Engine code is also made available on GitHub,[24] released under an MIT open source license.

[24] https://github.com/Microsoft/GraphEngine

f. Documentation and Customer Support

All documentation is available at https://graphengine.io/ and a support page provides contact information, blog links, and social media accounts for informal support. [25]

4. Gun

Gun DB is a pure JavaScript solution for storing JSON-like objects/documents in a data store, which can be kept synchronized with a server and client peers. By design, local copies of the data store may be used when connectivity is lost, and they are re-synced with remote copies when connectivity resumes. Local Objects may reference each other, building up a graph of relations.

The following is a simple social graph traversal example:

```
>> dog = {name: 'Fido'};
>> cat = {name: 'Fluffy'};
>> dog.friend = cat;
>> cat.friend = dog;
>> test = gun.get('foo').put(dog);
>> log_it = function(v){console.log(v);};
>> test.path('name').val(log_it);
Fido
>> test.path('friend.name').val(log_it);
Fluffy
>> test.path('friend.friend.name').val(log_it);
Fido
```

a. API Support

The Gun JavaScript library can be deployed to clients, i.e., web browsers, or used on a server. The latter case is supported by using **node.js** to develop server-side code. All create, read, update, and delete (CRUD) operations are supported through a fast JavaScript API,[26] as well as the ability to subscribe to changes in views/queries of the database contents, often useful for updating a web page view.

b. Query Languages

Other than the fast interface described in the previous section, Gun does not support any query languages.

[25] https://www.graphengine.io/support.html

[26] https://en.wikipedia.org/wiki/Fluent_interface#JavaScript

c. Third-Party Integrations

It is possible to use Amazon S3 storage[27] with Gun.

d. Download and Licensing Options

The source can be downloaded directly at the GitHub project page.[28] Alternatively it is possible to install the "now" script with Node Package Manager (npm),[29] then install gun from the command-line.[28] An official Docker image is also available. Gun is released under the Apache 2.0 open source license.

5. HANA

SAP's HANA is a proprietary solution designed to be a highly parallel and performant in-memory database. It is a "columnar" database, meaning data is stored contiguously by column, to optimize for performance of aggregation operations, e.g., totals and averages. It offers many enterprise features, such as multiple authentication options, high availability and disaster recovery. Out of the box, it supports graph data schema via property graphs. Graph data capability is integrated with functionality for geospatial, text analytics, and predictive analysis. Its remote data sync feature allows data to be synchronized between it and external SAP SQLAnywhere databases. HANA's model is, therefore, a hybrid of relational and graph models.

a. API Support

HANA supports analytics with the "R" programming language. It also provides an HTML5 and JavaScript framework for developing responsive[30] web applications. SAP makes a Web IDE available, and HANA also supports SAP's own ABAP application programming language.[31]

b. Query Languages

HANA supports its own version of SQL, which has support for "fuzzy" text search in columns as well as stored documents such as PDFs and Microsoft Office files. Geospatial queries are supported in HANA's SQL. It has its own language called SQLScript, used for building stored procedures, i.e., procedures that run directly on the server for speed.

[27] https://github.com/amark/gun/wiki/Using-Amazon-S3-for-Storage

[28] https://github.com/amark/gun

[29] https://www.npmjs.com/

[30] "Responsive" in this context means HTML/CSS/JavaScript that adapts to different client screens from smartphones up to full PC web browsers.

[31] https://en.wikipedia.org/wiki/ABAP

SQLScript can also be used for stream processing of real-time data. When using property graph data, HANA supports Cypher, an open standard query language for graph data introduced by the NEO4J graph database implementation.

c. Third-Party Integrations

HANA can be used to analyze Apache Hadoop data and to access the Hadoop distributed file system. HANA also makes integrations available for Apache Spark and Apache Hive.

d. Download and Licensing Options

HANA is sold via tailored hardware appliances for on-premises deployments.[32] SAP also makes HANA available through their own cloud or third-party offerings such as AWS, Azure, and IBM SoftLayer. Developers and IT professionals are able to preview HANA's capabilities using a free express edition.[33]

e. Documentation and Customer Support

SAP provides Enterprise support with its HANA appliances and cloud offerings,[34] including help with deployment and operations. Call center and online communications are included. At the highest level of customer incident, response times are as low as 1 hour.

Training courses are periodically offered at SAP training centers and virtually via the web. As an example, a 1-week HANA installation and operations course aimed at IT professionals has a tuition ranging from $3,500 to $4,500.

SAP also hosts a HANA online community. See https://www.sap.com/product/technology-platform/hana/community.html for details.

6. Horton

Horton is a project of Microsoft Research (MSR) from the 2010–2012 timeframe. Its stated goal is "to enable querying large distributed graphs." It was designed with data center use in mind, and it stores graph partitions in memory distributed across clusters of machines. Horton's data schema is a directed or undirected graph, where nodes are typed entities with associated sets of key-value pairs. Edges relating the entities may also have types and associated data.

[32] http://global.sap.com/community/ebook/2014-09-02-hana-hardware/enEN/appliances.html

[33] http://go.sap.com/developer/topics/sap-hana-express.html

[34] https://www.sap.com/product/technology-platform/hana/support.html

Horton leveraged another MSR project, “Orleans,” which was an actor-based framework for building distributed interactive applications without the need for learning complex concurrency paradigms. “Orleans” is described as a distributed run-time application for cloud implementations. It lives on in various high-scale Azure services, including cloud services for the popular Halo game. Horton, however, appears to exist only as a memory recorded in research and conference papers.

a. API Support

Horton was written in C# for the .NET framework, using the .NET Task Parallel Library and “Orleans.”

b. Query Languages

Horton provides its own query language that enables formulation of graph reachability queries, as well as its own query execution engine and query optimizer. Its syntax is a simpler version of openCypher’s[35] MATCH clauses and is reminiscent of regular expression pattern matching applied to graph connection patterns.

c. Third-Party Integrations

None known.

d. Download and Licensing Options

None known.

e. Documentation and Customer Support

Not applicable.

7. HyperGraphDB

HyperGraphDB is an open source Java-focused database that uses directed hypergraphs. Hypergraphs are a generalization of the more familiar graph data structure. Edges in a hypergraph are not limited to two node connections, but may in fact connect an arbitrary number of nodes together. In the figure below,[36] nodes (vertices) are represented by dots, and edges are represented by colored regions (see Figure A-3).

[35] http://www.opencypher.org/

[36] Taken from https://en.wikipedia.org/wiki/File:Hypergraph-wikipedia.svg

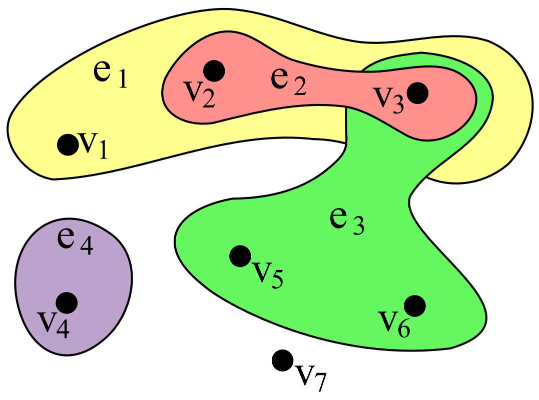


Figure A-3. Visualization of Hypergraphs with "Edges" Rendered as Regions Encompassing Multiple Nodes

HyperGraphDB is intended for knowledge management and semantic web applications (e.g., RDF), but it can also be used as an embedded object-oriented database for Java software projects or as a graph database.

a. API Support

HyperGraphDB exposes a Java API. The Javadoc documentation may be found at http://www.hypergraphdb.org/docs/javadoc/index.html.

b. Query Languages

HyperGraphDB does not include any query languages. Instead, it provides a special-purpose Java query API, where query conditions are composed using functional composition, submitted as a query, and the results are iterated over. Similarly, graph traversals are performed using another included Java API.

c. Third-Party Integrations

A number of projects leverage HyperGraphDB for various purposes. Notable ones include an RDF4J SAIL plugin using HyperGraphDB as the storage engine, and integration with the TuProlog interpreter for reasoning over hypergraphs.

d. Download and Licensing Options

HyperGraphDB requires a system capable of running Java 6 or later. Unix/Linux, Windows and Mac are recommended. A downloads page[37] describes how to directly download ZIP files or tarballs that contain all core components, plus API documentation

[37] http://www.hypergraphdb.org/?project=hypergraphdb&page=Downloads

and source code. Software projects utilizing Maven for dependency management may also pull in HyperGraphDB components from the HyperGraphDB Maven repository.

e. Documentation and Customer Support

Extensive online documentation is available.[38] A community page[39] also provides a link to a community discussion group hosted on Google Groups, as well as instructions on contributing to the HyperGraphDB code on GitHub.

8. IBM System G Native Graph Store

IBM System G is a set of graph computing tools and solutions. Native Graph Store (NGS) is one of the Graph Database components of System G. It handles various types of graphs, including property graphs and “RDF-like” graphs. It is advertised to scale up on a single machine and scale out on clusters. At the lowest level, NGS provides several in-memory and on-disk property graph storage engines.

a. API Support

NGS includes a set of C++ graph programming APIs and a command-line shell (gShell). It also has a socket client upon which a REST API is provided, as well as a socket GUI. It includes a visualization toolkit. A set of Python wrappers for all gShell commands is also available.

b. Query Languages

A Java Native Interface translation layer provides TinkerPop Blueprints interfaces, upon which Gremlin is supported for graph traversals, as well as a Jena-based SPARQL query engine.

c. Third-Party Integrations

See the previous section.

d. Download and Licensing Options

System G, being an IBM research project, is freely downloadable[40] and usable in any project, commercial or non-commercial. Packages for several Linux distributions, Mac OS

[38] http://www.hypergraphdb.org/?project=hypergraphdb&page=LearnHyperGraphDB

[39] http://www.hypergraphdb.org/?project=hypergraphdb&page=Community

[40] http://systemg.research.ibm.com/download.html

X, and Power 8 are also available. The use, however, is at your own risk. IBM System G can also be used via IBM's cloud.[41]

e. Documentation and Customer Support

A feedback e-mail address is provided,[42] but no official support. Documentation is included in the docs/ folder of the downloadable package, and links to the documentation are also made available on the download page.[40] There is an announcements message board at http://systemg.research.ibm.com/message-board.html.

9. InfiniteGraph

InfiniteGraph[43] is a partially deprecated graph database from Objectivity, Inc. It has been superseded by Objectivity's ThingSpan. It is only being recommended to Java developers who wish to use graph analytics outside of an Apache Spark environment.

a. API Support

InfiniteGraph exposes a Java API.

b. Query Languages

It supports graph traversals via Gremlin.

c. Third-Party Integrations

None known.

d. Download and Licensing Options

Objectivity, Inc., must be directly contacted to obtain a copy.

e. Documentation and Customer Support

Objectivity, Inc., must be directly contacted to obtain documentation. The level of support provided to InfiniteGraph users is unknown.

[41] http://systemg.research.ibm.com/cloud.html

[42] systemg@us.ibm.com

[43] http://www.objectivity.com/products/infinitegraph/

10. InfoGrid Graph Database

InfoGrid Graph Database (IG) is an open source “web graph database” developed in Java. The “web” part refers to the fact that all data objects in IG are automatically assigned RESTful URLs. It is typically meant to be used by Java-based web applications.

IG is composed of several sub-projects (see Figure A-4):

- *IG Grid* - adds distributed capabilities to IG.
- *IG Stores* – provides various SQL and NoSQL underlying storage alternatives.
- *UG User Interface* – maps IG database contents to RESTful interfaces, in customizable ways.
- *IG Light-Weight Identity Project* – authentication module, supports LID, OpenID, et al.
- *IG Model Library* – “Defines a library of reusable object models that can be used as schemas for InfoGrid applications.”
- *IG Probe* – allows to access arbitrary data sources on the Internet as if they consist of objects (MeshObject = nodes with properties; Relationship = edges with type(s) and directions).
- *IG Utilities* – Common object frameworks and utility code used throughout IG.

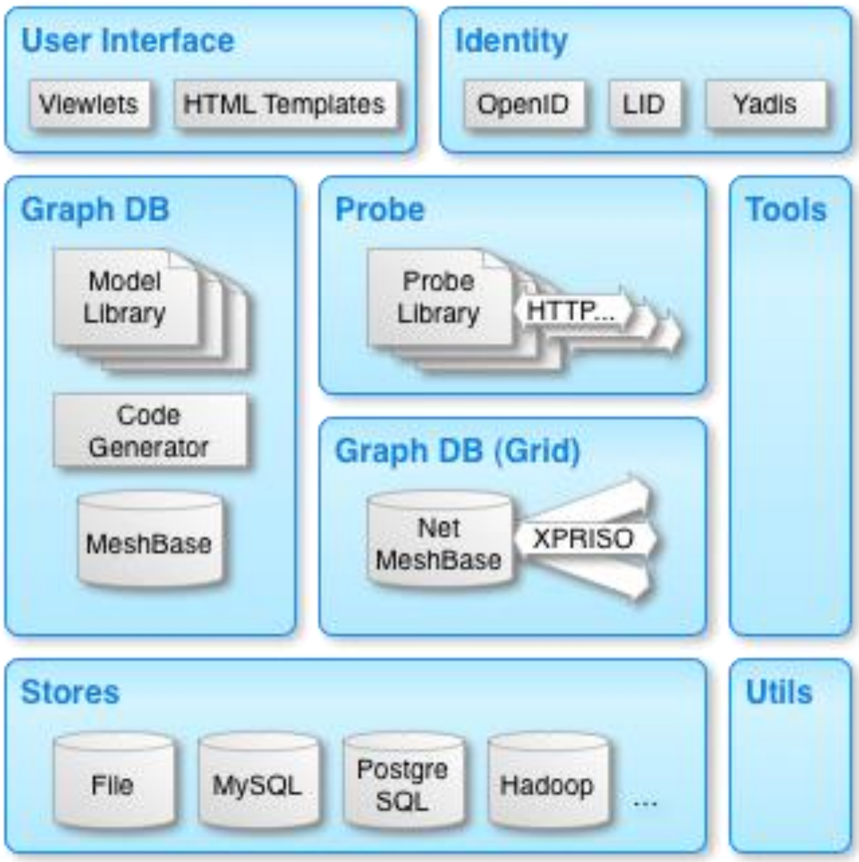


Figure A-4. Summary of InfoGrid Subprojects

a. API Support

IG exposes a Java API, and an IG database application usually will expose REST APIs.

b. Query Languages

InfoGrid databases are primarily queried/traversed via Java and the REST APIs.

c. Third-Party Integrations

IG is designed to be used on Java Enterprise Editions servers, such as Tomcat. While it has its own file and in-memory store implementations, it may also be used with MySQL, PostgreSQL, Hadoop, S3, and other storage solutions.

d. Download and Licensing Options

IG is released as an open source project. No commercial support is available. Releases may be downloaded at http://infogrid.org/trac/wiki/Docs/Downloads. In addition, the same page has links to instructions for source code access through Subversion.

e. Documentation and Customer Support

Documentation is available in Wiki form at http://infogrid.org/trac/wiki/Docs/Overview. For developers, the documentation for the Java API is available on the download page.

11. MarkLogic

MarkLogic is advertised as a database product that can easily load data from diverse siloes and give visibility to all of an organization's data. They advertise many high-profile projects/success stories in their portfolio such as Deutsche Bank, KPMG, HealthCare.gov, and NBC SNL viewer app. The server itself is a NoSQL database that is document-centric, multi-model with support for commonly used concrete syntaxes (JSON, XML, RDF, et al.), with Geospatial, and semantic and enterprise features (back-up, replication, etc.).

a. API Support

Multiple programming language APIs are available such as Java, Node.js, and XCC Java/.NET.

b. Query Languages

MarkLogic Server has SQL[44] and SPARQL[45] query engines.

c. Third-Party Integrations

MarkLogic provides Hadoop and ODBC connectors.

d. Download and Licensing Options

MarkLogic makes a free, limited use edition[46] available for download[47] by developers. In order to use it, one must first join the developer community[48] and request a developer license from within the software interface. The developer license may also be used on a cloud deployment such as AWS, subject to the same restrictions as a local deployment.

Paid licenses with support are called Essential Enterprise, and are available on a subscription basis for $18k+/year per eight processor cores.[49] Perpetual licenses are also

[44] http://docs.marklogic.com/guide/sql/intro

[45] http://docs.marklogic.com/guide/semantics/semantic-searches

[46] 1 TB of data, development use only.

[47] http://developer.marklogic.com/free-developer see also http://developer.marklogic.com/products

[48] http://developer.marklogic.com/people/signup

[49] http://www.marklogic.com/what-is-marklogic/pricing/

available. The sales department must be consulted in either case. Semantics (RDF), Tiered Storage, and Geospatial capabilities are each additional license options available at extra cost.

e. Documentation and Customer Support

Full documentation is available at http://docs.marklogic.com/. Free online training is available for developers and administrators.[50] Teacher-led courses, tutorials, and professional certifications are also available. A variety of paid support options are outlined at http://www.marklogic.com/services/support/, and the main support portal is hosted at https://help.marklogic.com/.

12. OpenCog AtomSpace

OpenCog is an open source software initiative project for creating "artificial general intelligence (AGI)." AGI appears to be a repatriation of the old term, "artificial intelligence (AI)," which used to mean machine simulation/implementation of human-like cognition. The term AI very recently has been diluted to more of a marketing term. It usually refers to current machine learning technologies, especially as applied in business, big data, and video games.

OpenCog's AtomSpace[51] project defines data storage according to a hypergraph model (see the HyperGraphDB discussion, above). In their own words, AtomSpace is a "knowledge representation (KR) database together with the associated query/reasoning engine needed to fetch and manipulate that data, and to perform reasoning on it."

a. API Support

AtomSpace is developed in C++. There are Python, Haskell, and Scheme bindings[52] as well.

b. Query Languages

AtomSpace has a language for constructing knowledge graphs called "Atomese" that, while human-readable, is not really meant for human-machine interaction. Rather it is used for machine learning and process-to-process communication. In practice, querying is performed using the bindings mentioned in the API section above.

[50] http://www.marklogic.com/training/

[51] https://github.com/opencog/atomspace

[52] https://github.com/opencog/atomspace/tree/master/examples

c. Third-Party Integrations

The core AtomSpace project uses PostgreSQL as its storage backend, but there is at least one alternative implementation using NEO4J.

d. Download and Licensing Options

OpenCog is licensed under the AGPL 3.0, and no commercial licenses are available.

e. Documentation and Customer Support

In addition to the README files included with the source code, one may refer to the OpenCog wiki (http://www.opencog.org/) and the documentation site (http://docs.opencog.org/). Community discussions occur in the GitHub issue trackers, a Google Groups e-mail discussion list,[53] and a FreeNode IRC channel, #opencog.[54]

13. OrientDB

OrientDB bills itself as a scalable, high-performance,[55] polyglot[56] database. It supports graph, document, key/value, and object data models.[57] OrientDB claims that its core engine truly supports all these models natively, making it possible to leverage the advantages of more than one model at a time. It also claims easy scaling with “zero-configuration multi-master architecture.”

a. API Support

OrientDB has drivers for an exhaustive list of programming languages[58] that encompass Java (including JDBC and Spring), PHP, .NET, Python, Go, JavaScript (including node.js), Ruby, Groovy, Scala, R, Elixir, Clojure, and Perl. There is also an Android port of OrientDB.

[53] https://groups.google.com/forum/#!forum/opencog

[54] http://webchat.freenode.net/?channels=opencog

[55] They claim the ability to write 120k records/second to local storage on a Core i7 CPU with 16 GB of RAM and SSD. However, they caveat this number by saying “no indexes” and using local on-device storage, i.e., not over a network.

[56] https://en.wikipedia.org/wiki/Polyglot_persistence

[57] http://orientdb.com/docs/last/Tutorial-Document-and-graph-model.html

[58] http://orientdb.com/docs/last/Programming-Language-Bindings.html

b. Query Languages

OrientDB supports graph traversals using the TinkerPop Gremlin language.[59] It also supports traditional SQL queries.

c. Download and Licensing Options

A freely downloadable Community Edition is available under the Apache 2 license.[60] To facilitate test-running the product, a runnable Docker image is also made available. OrientDB Enterprise is the product offering under a commercial license when used in production, although it is free of charge for development usage under a 45-day free trial. Production usage is offered at three service levels: Bronze, Silver, and Gold. Prices start at $4k/server/year. See http://orientdb.com/support/ for details.

OrientDB Enterprise offers additional features not available in the Community version:

- Non-Stop Incremental Backup and Restore,
- Metrics Recording,
- High Availability,
- Configurable Alerts,
- Teleporter, i.e., Easy RDBMS data importing,
- Query Profiler,
- Auditing Tools,
- Live Monitor.

d. Documentation and Customer Support

The support page given in the previous section shows that two different SLA levels are offered for production, 24/7 (round the clock) and 10/5 (business hours). 24/7 is only offered at the silver and gold support levels.

A manual is available at http://orientdb.com/docs/.

14. Profium Sense

Profium Sense is the sole product of Profium, a Finnish company that has offices in the United States. Technical information on their web site is sparse, but the descriptions make clear that it is closed source and Java 8-based. It is billed as having two major

[59] https://tinkerpop.apache.org/gremlin.html

[60] http://orientdb.com/download

components: an in-memory graph database and a real-time rule engine. The database has native support for RDF and OWL2 RL, and it also supports JSON-LD.[61] It supports RDF, including RDF Schema and Web Ontology Language (OWL). It is packaged in four configurations for different application foci:

- Semantic Search,
- Situation Awareness,
- Digital Asset Management,
- Context-Aware Services.

Profium is deployed as a Java EE application. High availability and scalability are accomplished by leveraging Java EE configuration.

a. API Support

Customization of the rules engine can be accomplished with RDFS, OWL, and JavaScript.

b. Query Languages

Profium Sense supports SPARQL, including geo-semantic and full-text search extensions. New indices may be added to running instances in order to tune search performance.

c. Third-Party Integrations

Profium Sense makes use of many open standards:

- Geography Markup Language,
- Web Feature Service 2.0,
- Monitoring of RSS/Atom feeds,
- Simple Knowledge Organization System (SKOS),
- Data ingestion – File-based, FTP, HTTP, SOAP with attachments,
- Adobe Extensible Metadata Platform (XMP),
- International Press Telecommunications Council – IIM and IPTC core,
- Online Information Exchange (ONIX),
- NewsML.

[61] LD = Linked Data

d. Download and Licensing Options

Profium Sense is available under a commercial license. Pricing information is not published on their website. Inquiries may be made at https://www.profium.com/en/contact-us.

15. Oracle NoSQL Database

Oracle offers NoSQL Database as a scalable non-relational data store that may collect documents, key-value pairs, or RDBMS-style tables. It uses sharding to distribute records across clusters based on primary key hash values, with records automatically replicated to support automatic failover. Java 8 is required to run Oracle's NoSQL Database.

a. API Support

Drivers are available to use NoSQL from Java, JavaScript, C (separate versions for Key-Value storage versus Table storage), and Python. Also, it is possible to natively expose REST interfaces to the data.

b. Query Languages

Oracle's NoSQL Database supports a version of SQL for queries of table-like data and JSON datatypes, referred to as JSON query. Below is a sample from a whitepaper containing the code to create a table definition:[62]

```
CREATE TABLE patient (
    pid INTEGER,
    patientInfo JSON,
    PRIMARY KEY (pid)
)

put table -name patient-json '{
    "pid":1,
    "patientInfo": {
    "firstName": "Bob",
    "lastName": "Smith",
    "birthdate": "1940-01-04",
    "prescriptions":  [
          {
          "medicine": "Tylenol", "frequency": "Hourly"
          }  ,
          {
          "medicine":"Advil", "frequency":"Daily"
          }
```

[62] http://www.oracle.com/technetwork/database/database-technologies/nosqldb/learnmore/4-3-features-3513236.pdf

```
        ]
        }
    }'

    # Serialization to JSON
    #
    SELECT p.pid, p.patientinfo.firstName, p.patientinfo.lastName
    FROM patient p
    WHERE p.patientinfo.prescriptions.medicine =any "Advil" ;

    Result:
    {"pid":1,"firstName":"Bob","lastName":"Smith"}
```

c. Third-Party Integrations

Oracle's NoSQL Database can be integrated with Hadoop to support Map/Reduce jobs.

d. Download and Licensing Options

Oracle's NoSQL Database is available in both the open source Apache-licensed Community Edition (CE) and a closed source, commercially licensed Enterprise Edition (EE). A fully supported version of EE[63] called Basic Edition (BE) is now bundled at no extra charge with Oracle Database Enterprise Edition, versions 11 and 12. Oracle's NoSQL Database CE lacks the following features that the others have:

- Integration with Oracle Database for pushing data from NoSQL to the RDBMS,
- Integration with Oracle Enterprise Manager (GUI monitoring interface),
- Integration with Oracle Event Processing Engine,
- Integration with Oracle Coherence,
- Integration with Oracle Big Data Spatial and Graph (see next section),
- Enterprise 24x7 support,
- Kerberos integration.

16. Oracle Spatial and Graph

Oracle Spatial and Graph (OS&G) is a general-purpose property graph database, with graph analytic features. It supports RDF linked data applications. As hinted by the *Spatial* in the name, it supports geospatial data, with standard features such as querying by

[63] Licenses are perpetual on either a per-named-user or per-machine basis.

boundary. It also supports general-purpose property graphs. OS&G can provide access control either on a graph basis or as fine-grained as individual vertices and edges.

a. API Support

OS&G supports both Sesame/RDF4J and Jena flavored Java APIs. Both the development and the management operations are supported by a Groovy[64] console. APIs for Python and R are also supported.

b. Query Languages

SQL and SPARQL are both supported. In fact, SQL queries may be augmented with SPARQL pattern matching. The GeoSPARQL extension to SPARQL is supported.

c. Third-Party Integrations

Apache Lucene and Apache SolrCloud may be leveraged for fast text searching.

d. Download and Licensing Options

OS&G is listed as a "key feature" of Oracle Database 12c. Therefore, it is assumed that one must have an Oracle Database license in order to use the OS&G feature set.

17. Sparksee

Sparksee from Sparsity Technologies, based in Spain, is billed as a "scalable high-performance graph database." It works on a wide variety of operating systems, and Sparsity claims it is the first graph database available on Android and iOS (Sparksee Mobile). The fact that Sparksee has a mobile version highlights that it is intended as an embedded database. They claim that they use "bitmap" data structures, which optimize performance and memory/storage utilization, e.g., fitting the bitmaps within disk pages for fast retrieval. The data structure it works with is termed a "labeled[65] and directed attributed[66] multigraph."[67]

a. API Support

Sparksee is implemented in C++, but supports a variety of APIs such as .Net, C++, Python, Objective-C and Java.

[64] Groovy (http://groovy-lang.org/) is a popular, powerful language targeted at the JVM, which easily integrates with Java, and is suitable for scripting.

[65] "Labeled" means all nodes and edges have types (labels).

[66] "Attributed" means both nodes and edges may have one or more properties.

[67] "Multigraph" means nodes may be connected with multiple edges.

b. Query Languages

All queries and graph traversals are performed via function calls in the various APIs.

c. Third-Party Integrations

None known.

d. Download and Licensing Options

Full and Mobile versions of Sparksee are freely downloadable. They are downloaded as API-specific packages (e.g., Python). Its official Java API libraries are also available via Maven for easier integration into Java software projects. The downloads come with a limited Evaluation License (see below), and usage of the Mobile version explicitly requires a license key.

A freely usable version licensed for evaluation purposes is downloadable. It is limited to 1 million stored objects and a single session, with high availability (HA) functionality disabled. A free Research License is available for small (10M objects), medium (100M), and large (1B) instances, with the HA feature enabled. Similar capability is available prior to deployment under a commercial license, where a company may request free Development Licenses. Standard commercial SME Licenses are available with a cost ranging from €2,200/year (Small, No HA) to €11,640/year (Very Large, >1B objects, HA enabled). A Corporate License is also available (for medium, large, and very large) that includes a higher level of support, and bundles four development licenses as part of the deal. Pricing for the Corporate License is available by contacting Sparsity Technologies.

The various support channels are described at http://sparsity-technologies.com/StartingGuide/Support.html.

18. Sqrrl Enterprise

Sqrrl Enterprise is not really marketed as a graph database per se, although it does have graph database capability. It is sold as a cybersecurity solution, that is, as a product capable of ingesting large quantities of cybersecurity-relevant information, such as server logs, and creating what they call a "Behavior Graph." Exploration and search of this Behavior Graph is enabled by web, command-line, Java, and Python APIs. At the data storage level, Sqrrl leverages Apache Accumulo[68] running on top of Apache Hadoop, enabling data set sizes up to multi-petabytes.

[68] Apache Accumulo is an open source distributed key-value store designed for scalability and based on Google's BigTable.

a. API Support

Both Java and Python APIs are supported.

b. Query Languages

Unknown.

c. Third-Party Integrations

Unknown.

d. Download and Licensing Options

A "Test Drive" virtual machine is downloadable for evaluation purposes. The company must be contacted for pricing details, although the website does indicate that licenses are offered on an annual subscription basis.

19. Teradata Aster

The Teradata Aster architecture includes a row store (similar to typical RDBMS), a column store (column data stored contiguously, more efficient for some purposes), and a file store. It uses unified computation and management layers in order to support a variety of data operations at large scale (hundreds of TBs to petabytes). It uses massively parallel processing for all operations: loading, querying, exporting, backups, recovery, installation, and upgrades.

a. API Support

Aster supports APIs for R, Python, Java, C, and C++, utilizing database access standards like OLE DB, ADO.NET, ODBC, JDBC, and Psycopg (Python).

b. Query Languages

Aster uses its own variant of SQL, as well as SQL engines optimized for MapReduce (SQL-MR) and graph analysis (SQL-GR). ANSI-compliant SQL-92, SQL-99, and SQL-03 extensions are available.

c. Third-Party Integrations

Aster-Teradata and Aster-Hadoop adapters are available, which allow fast, parallel data transfers between Aster and Hadoop. Aster is compatible with Quest Authentication Services.

d. Download and Licensing Options

Aster is deployed either on Teradata hardware solutions or on Teradata-certified commodity server hardware.

20. ThingSpan

ThingSpan is a product of Objectivity, Inc., and is advertised to be a "highly scalable real-time graph analytics platform" built on top of Apache Spark and the Hadoop Distributed File System (HDFS).[69] The "highly scalable" claim points to its ability to both "scale up" and "scale out." "Scale up" refers to scaling to fully utilize hardware of differing capabilities. "Scale out" refers to the ability to add many compute and storage devices, up to petabytes of data. ThingSpan takes advantage of Hadoop's YARN and Apache Mesos for management of ThingSpan clusters.

The ThingSpan […] graph analytics platform […] optimizes the execution of JOINs when using Spark SQL by taking advantage of the underlying graph semantics.[70]

Vertices and edges may be untyped, or they may inherit types according to user-defined schemata. Edges may be 1-1, 1-many, many-1, many-many, and can be unidirectional or bi-directional.

a. API Support

ThingSpan provides its own Java, C++, Python, C#, and REST APIs. In addition, the Spark APIs may be used, which have bindings for Java, Scala, and Python:

- *Spark Streaming*, which is a language-integrated stream processing API,
- *MLib*, a collection of popular machine learning algorithms,
- *GraphX*, fast collection of graph algorithms, incl., e.g., PageRank,
- *Spark SQL DataFrame* API.

b. Query Languages

ThingSpan provides its own query language called DO. DO has "adopted many best-of-breed techniques from existing declarative languages such as SQL, XPATH, OCL, SPARQL, Cypher, and LINQ."[70] A command-line read, evaluate, print loop (REPL) is also provided for DO.

ThingSpan's use of Apache Spark means it provides two additional query languages:

---

[69] HDFS is a "distributed file system designed to be run on commodity hardware." https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction

[70] ThingSpan Technical Whitepaper, downloadable upon request.

- HiveQL, a SQL-like language for reading, writing, and managing large datasets residing on distributed storage;
- Spark SQL, for querying structured data.

c. Third-Party Integrations

Spark SQL provides JDBC and ODBC connectors, which enable existing RDBMS-based business tools to interface with ThingSpan.

d. Download and Licensing Options

A Linux installer is downloadable from http://support.objectivity.com/downloads. Presumably this is a developer preview, and to scale up, one needs to purchase support. No information is given on the website about licensing, and, therefore, one must contact Objectivity, Inc., for this information.

21. TinkerGraph-Gremlin

TinkerGraph is the reference implementation of graph storage for the TinkerPop3 framework. It is a single-machine, in-memory (optional persistence) non-transactional graph engine. It provides both Online Transaction Processing (OLTP) and Online Analytic Processing (OLAP).[71]

a. API Support

TinkerGraph's primary reason for existing is to demonstrate the TinkerPop3 APIs. TinkerPop3 is natively used with Java and Groovy, but interfaces to other popular languages, such as Gremlin-Python, are available.

b. Query Languages

The TinkerPop3 graph traversal APIs are the primary mode of interacting with TinkerGraph.

c. Third-Party Integrations

None known.

[71] OLTP is concerned with dealing with high-volume data transactions. OLAP is more concerned with analysis and reporting.

d. Download and Licensing Options

TinkerGraph is open source software, freely downloadable as Java .jar files, or using development package managers such as Maven.

22. Titan Distributed Graph Database

Titan appears is an open source project intended to support very large graphs. “Very large” in this case means graphs so large that graph processing needs are beyond what a single machine can perform. As described below, it is very customizable to the capabilities needed for a particular application. It supports:

- Scalability, both in dataset size and users,
- Replication, fault tolerance, high availability,
- Hot backups,
- ACID and eventual consistency.

a. API Support

Titan exposes a Java API for server management purposes. Users are strongly encouraged to use only the Java API for this purpose, and to use Gremlin for querying data.

b. Query Languages

The Gremlin query language is available through integration with the TinkerPop graph stack. By adding various data indexers like Solr or ElasticSearch, the performance can be optimized for particular use cases.

c. Third-Party Integrations

In addition to an in-memory store for embedded use cases, Titan supports various storage backends such as Apache Cassandra, Apache HBase, and Oracle BerkeleyDB. The choice depends on application needs (see Figure A-5).[72]

[72] Figure is from the documentation: http://s3.thinkaurelius.com/docs/titan/1.0.0/benefits.html

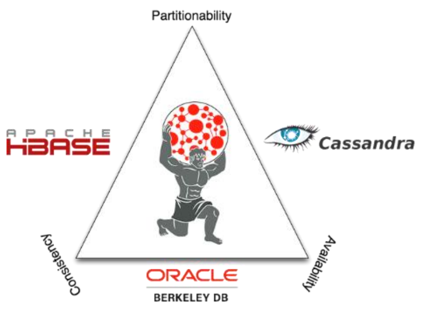


Figure A-5. Data Stores Supported by Titan

Graph data analytics is supported via integrations with Apache Spark, Apache Giraph, and Apache Hadoop. Additional search capabilities are available via integrations with ElasticSearch, Solr, and Lucene. Graph traversals are supported via integration with the TinkerPop graph stack.

d. Download and Licensing Options

Titan is freely downloadable and usable under an Apache open source license. No commercial licenses or support are offered. Downloads can be in the form of either a .zip archive, using the Maven package dependency manager, or via cloning the source from GitHub.

23. VelocityGraph

VelocityGraph, from VelocityDB, Inc., is a .NET-focused database that is compatible with open source .NET alternatives mono and Xamarin. It appears to be an extension of the core product, VelocityDB, which is primarily an object-oriented database, where suitably defined serializable C# objects and object graphs may be efficiently persisted and retrieved. VelocityGraph appears to add a property graph object model and API for easily storing and retrieving property graphs. An administration tool called *DatabaseManager*[73] is made available. VelocityDB supports distributed storage via sharding using defined storage locations. ACID-compliant transactions are supported.

a. API Support

VelocityGraph natively supports a C# interface, although due to the nature of .NET, it is almost certain that alternative .NET languages such as F# and VisualBasic.NET are also usable. For a fee, VelocityDB, Inc., can provide interfaces for your data schema using other languages, such as Java or Python, or even a REST interface. It supports an in-

[73] http://www.velocitydb.com/UserGuide.aspx#_Toc433626343

memory storage option that, in combination with the utilization of multiple servers, provides high availability.

b. Query Languages

No query language support. Usage of the C# data retrieval API is strongly encouraged. Language Integrated Query (LINQ) usage is recommended, though.

c. Third-Party Integrations

VelocityDB has a driver for the LINQPad[74] development tool. It can integrate with Microsoft's ASP.NET Identity Core[75] in addition to using Windows Authentication when accessing servers. It also can support geocoding via GeoHash.[76]

d. Download and Licensing Options

The compiled .NET assemblies are freely downloadable at the web site. Licenses may be obtained on the web site, with a range of prices. Paid licenses include support:

- Free:
  - Trial usage, 10 days only, no more than 3 data types persisted;
  - Nonprofits such as schools or open source projects, issued on case-by-case basis;
- Standalone Client Applications (no VelocityDB Server):
  - $100 per 6 months single-machine developer license;
  - $200/user "perpetual" developer license never expires, but upgrades available only for 1 year.
- Server/Client:
  - $300/year annual developer license;
  - $400/user "perpetual" developer license;
  - $6000/year source code license, includes Git access to VelocityDB code repository.

74 http://www.linqpad.net/

75 https://www.nuget.org/packages/Microsoft.AspNet.Identity.Core/

76 https://en.wikipedia.org/wiki/Geohash

24. VertexDB

VertexDB's home page tersely describes the product as a

> *Graph database server with AJAX API and support for queries and queues. Built on top of TokyoCabinet and libevent.*

VertexDB is open source and written in C. While libevent (for asynchronous socket and HTTP support) appears to be a maintained open source project, both TokyoCabinet (a B-tree key-value disk store) and VertexDB appear to have been unmaintained for at least the past 4 years, with the exception of some Docker instructions added in 2016.

a. API Support

VertexDB primarily exposes an REST API. The graph nature of VertexDB lies in resources (returned as JSON documents) being able to reference one another, much like URLs.

b. Query Languages

VertexDB is primarily used via its REST API.

c. Third-Party Integrations

None.

d. Download and Licensing Options

VertexDB is released under a BSD open source license, and the source is available for download at https://github.com/stevedekorte/vertexdb/.

25. Virtuoso Universal Server

Virtuoso is a multi-paradigm server (i.e., supporting column-store SQL RDBMS, SPARQL RDF-based quad store, content management, web, RDF-based linked data server, SOAP, REST) that is capable of deployment on Windows and popular Linux and Unix distributions (see Figure A-6).[77]

[77] Diagram taken from https://virtuoso.openlinksw.com/

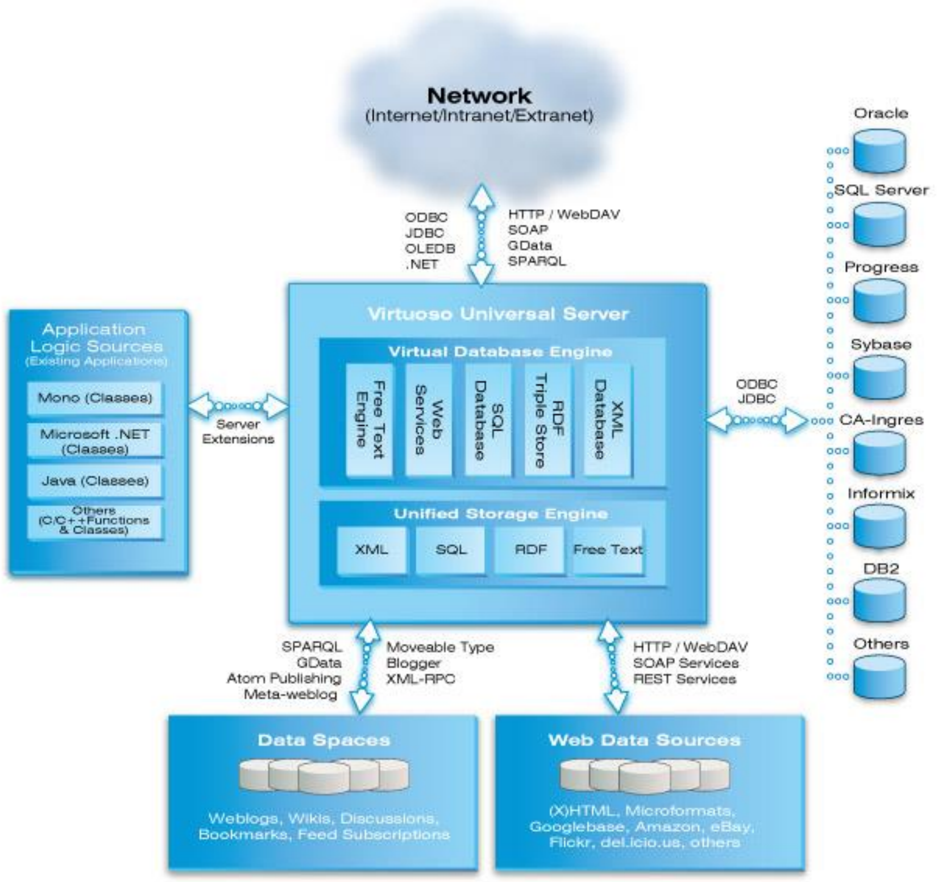


Figure A-6. High-Level Depiction of the Virtuoso Universal Server Architecture

As can be seen from the architecture diagram in **Error! Reference source not found.**, key capability of Virtuoso, aside from its own “universal storage engine,” is the ability to expose data from existing data sources in desired formats, such as XML or RDF.

a. API Support

Runtime hosting of the .NET CLR, Mono CLI and the JVM is supported, which allows developers to extend the functionality of the server, such as, e.g., defining new SQL functions. Similarly, a Virtuoso Server Extension Interface allows one to extend the server using C.

b. Query Languages

Virtuoso supports SQL, SPARQL, and XQuery.

c. Third-Party Integrations

Providers are available for the use of Virtuoso as the data store for RDF4J or Jena. Virtuoso supports JDBC, ODBC, ADO.NET, and OLEDB database connectivity, and it can act as a connective layer to existing data resources, such as Oracle DB, SQL Server, etc.

d. Download and Licensing Options

A GPL-licensed version of Virtuoso is available, and is called the Virtuoso Open-Source Edition (VOS).[78] Virtuoso is also offered in a variety of commercial license levels:

- Perpetual on-site:
  - Single seat:
    - Personal, $750, 4 cores, 5 concurrent DB sessions,
    - Developer, $1050, 8 cores, 5 concurrent DB sessions;
  - Project, $1875, 8 cores, 10 concurrent DB sessions;
  - Workgroup, $3750, same as Project + server-class OS deployment and cluster features enabled;
  - Department, $7,500, same as Workgroup, but 16 cores.
- AWS Cloud:
  - M3 Large, $1914-$3828/year, 2 cores, 7 GB RAM, 32 GB storage;
  - R3 Large, $2272-$4545/year, 2 cores, 15 GB RAM, 80 GB storage;
  - R3 X-Large, $4506-$9012/year, 4 cores, 30.5 GB RAM, 32 GB storage.

26. Weaver

Weaver is an open source, distributed, graph store for usage with Python. It is advertised to be “scalable, fast, [and] consistent.” It is targeted at very dynamic graphs such as social networks and knowledge graphs. It is built on top of the HyperDex[79] cluster-enabled key-value store, which means it is primarily targeted at the 64-bit Linux architecture. It is possible to bulk load data into a cluster while starting the shards. Weaver optimizes cluster performance in a number of ways, including caching graph computation results on nodes and automatically migrating data between shards to improve graph locality and reduce the need for inter-node communication.

[78] Downloadable from https://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/

[79] http://hyperdex.org/ HyperDex is interesting in itself, with API bindings for C, C++, Python, Ruby, Java, Go, Node.js, and Rust.

In their FAQ (http://weaver.systems/#faq), the Weaver team claims better performance than Titan, due to a different transaction locking mechanism. However, it must be stated that they consider Weaver to be pre-release, and not yet suitable for production usage.

e. API Support

Weaver is primarily intended to be used in combination with CPython. An easy introduction is posted at http://weaver.systems/#docs. The API appears to be very "Pythonic," i.e., it makes a lot of use of built-in Python types like lists and tuples. There is also a native C++ binding.

a. Query Languages

Weaver doesn't support any query languages, but it does offer a fast Python graph traversal API.

b. Third-Party Integrations

None known.

c. Download and Licensing Options

Weaver is released under the BSD-3 clause "New" or "Revised" license. Instructions for downloading and installing it are available at http://weaver.systems/#install, and the source code is accessible at https://github.com/dubey/weaver.

27. Ontotext GraphDB

Ontotext is a company operating in the European Union (EU) that for some time has been heavily involved in European Commission projects related to RDF and the Semantic Web, including contributing to RDF4J as part of the On-To-Knowledge project. Their GraphDB product builds on RDF4J libraries, with its own triple store implementing the SAIL API.

GraphDB is billed as a "semantic repository," with emphasis on utilizing formal ontologies to make sense of triple store data. It is available in Free, Standard, and Enterprise editions. The Free and Standard editions lack search connector integrations, as well as features like clustering that enable greater scaling. The Free edition is further limited in the number of concurrent queries allowed, and it lacks a service-level agreement.

GraphDB provides its own web interface, called *Workbench*. As of the latest version, GraphDB 8.1, a visual graph view is made available in *Workbench* (see Figure A-7). It allows visualization of the semantic class hierarchies, domain-range view on particular semantic classes, an overall view of class relationships, as well as graphs of specific resources and their relationships.[80]

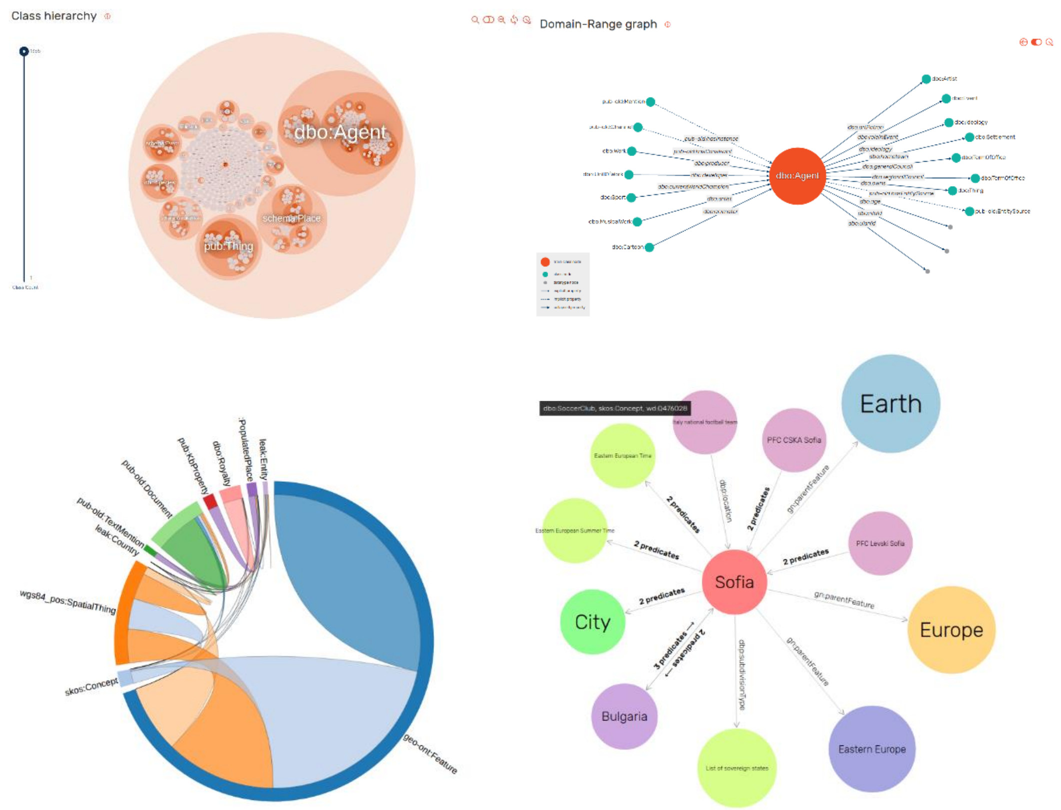


Figure A-7. GraphDB Visualization Capabilities

Online documentation is available at http://graphdb.ontotext.com/documentation/. Support is available via e-mail, Twitter, and Stack Overflow.

a. API Support

GraphDB exposes both RDF4J- and Jena-compatible Java APIs. It also exposes a SPARQL HTTP endpoint for querying over a REST interface.

b. Query Languages

GraphDB supports SPARQL 1.1 queries, including:

- SPARQL 1.1 Update,
- SPARQL 1.1 Federation,

[80] Description of visualization and figures taken from http://graphdb.ontotext.com/documentation/free/exploring-data.html.

- SPARQL 1.1 Graph Store HTTP Protocol,
- Geo-spatial indexing and GeoSPARQL.

c. Third-Party Integrations

A Lucene connector for full-text search is available on all editions. The Enterprise version (see below) adds Solr and Elasticsearch connectors.

d. Download and Licensing Options

The free edition is downloadable at http://info.ontotext.com/graphdb-free-ontotext with registration. Supported editions are in two flavors: Standard (GraphDB SE) and Enterprise (GraphDB EE).[81] GraphDB SE removes the limitation of only two concurrent queries that the free edition has and provides a commercial SLA. GraphDB SE can be licensed on a perpetual model or annual subscription model:

Model	One-time cost ($/core)	Annual Cost ($/core)
SE Perpetual	2100	420+ (depends on SLA)
SE Annual Subscription	n/a	1050+ (depends on SLA)

GraphDB EE is a superset of SE, adding Solr and ElasticSearch connectors for full text indexing and search, and support for replication clusters. These clusters support simultaneous load, query, and inferencing over 10^{10}+ triples, dynamic configuration of

[81] GraphDB comes in two flavors – Standalone server (GraphDB-SE) and HA Cluster version (GraphDB-Enterprise) The pricing policy counts the number of CPU-Cores used – for the Enterprise version only worker nodes are counted. We work with three pricing models: perpetual (with annual maintenance), annual subscription and managed service.

Perpetual Model:

1 CPU-Core of GDB-Enterprise is priced at 4,100 USD.

1 CPU-Core of GDB-SE is priced at 2,100 USD.

The annual maintenance is calculated on 20% of the license cost per annum We offer SLAs on top of that based on your specific requirements that may include specific response/resolution times, 24/7 etc.

Annual subscription model:

1 Core of GDB-Enterprise is priced at 2,050 USD per annum.

1 Core of GDB-SE is priced at 1,050 USD per annum.

This model includes maintenance already.

SLAs may be signed on top as well.

Managed Service:

We can offer you a complete service that includes monitoring, 99%+ availability and simplifies your process but completely removing devops responsibilities on your end.

I can provide you with an offer based on your specific requirements.

cluster and individual nodes, failover, and load balancing. The replication cluster architecture (see Figure A-8) has:

1. A non-data-storing master node, which is the front end, managing and coordinating incoming requests,
2. Multiple large worker nodes, which store the data and perform all reads and writes.

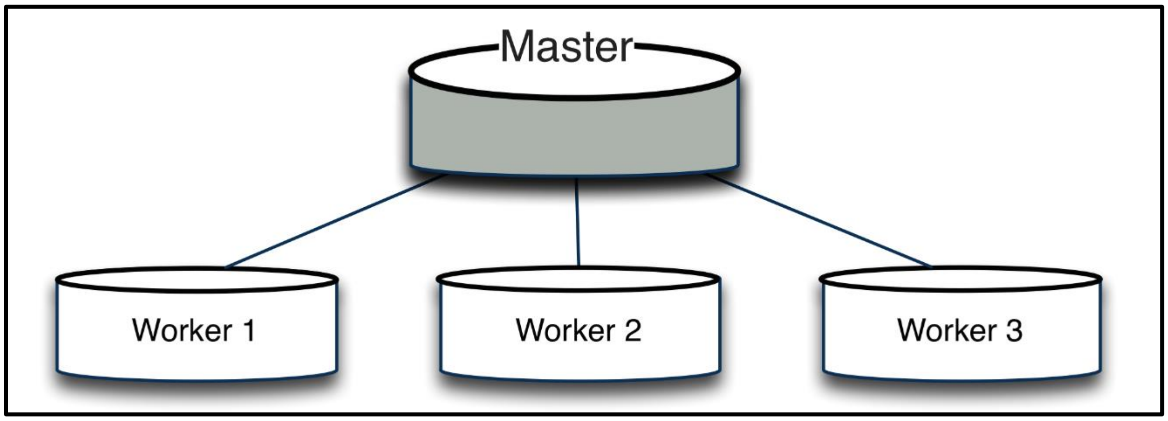


Figure A-8. Replication Cluster Architecture in GraphDB-EE

Pricing for GraphDB EE is offered with the same options as SE, except that the per-core cost is for worker nodes only (not master nodes):

Model	One-time cost ($/core)	Annual Cost ($/core)
EE Perpetual	4100	820+ (depends on SLA)
EE Annual Subscription	n/a	2050+ (depends on SLA)

A cloud edition is available, for hosting on Amazon Web Services (AWS). In addition, a software-as-a-service (SaaS) offering is available, also hosted on AWS, and includes monitoring and a 99% availability guarantee.

28. MongoDB

MongoDB is one of the most popular NoSQL implementations. It is referred to in its documentation as a “document database.” This does not mean that it is optimized for storing what a user typically thinks of as documents, e.g., Word, Excel and PDF files. It actually means that data is stored in a format very closely related to JavaScript Object Notation (JSON). JSON is one of the de facto standards for exchanging structured data in web applications and web services. A single JSON document contains a set of fields and values. Document schema are flexible in that the values can themselves be embedded documents.[82] Documents with a shared schema are organized together as collections. Table A-1[83] gives a mapping between RDBMS and MongoDB terms:

Table A-1. Concept Mapping Between Relational and Key-Value Data Stores

RDBMS Term/Concept	*MongoDB Term/Concept*
Database	Database
Table	Collection (or "document collection")
Row	Document
Index	Index
Join (1:1 or 1:many relationships)	Typically: Embedded documents
Join (many:many or complex structures such as graphs)	Typically: Document reference fields or $lookup API function

The document DB design trades away some of the full complex join functionality available in RDBMS systems.[84] The primary historical design consideration behind the RDBMS was optimizing for minimum storage, when storage was very expensive. MongoDB instead optimizes for fast query times. With appropriate design of schema, most queries should be accomplished with only a single disk read.

For applications, like Dynamic Force Structure solutions, which at present must perform complex joins among a number of different RDBMS systems, MongoDB has experience in providing solutions. Whitepapers on proven methodologies are available for:

- Migration of existing data from RDBMS systems to MongoDB,
- Combining existing data sources to load data into MongoDB "views" that look like regular document collections, and that can be queried for particular purposes.[85]

MongoDB claims superb scalability in three dimensions: cluster (# of nodes), performance (IOPs[86] while maintaining strict latency SLAs), and data (# documents per DB). MongoDB is capable of sharding, i.e., splitting data across multiple nodes for

[82] Up to a current limit of 16 MB per document.

[83] Adapted from "RDBMS to MongoDB Migration Guide." https://www.mongodb.com/collateral/rdbms-mongodb-migration-guide

[84] The joins that are possible in MongoDB are analogous to left outer joins in RDBMS parlance.

[85] "10-Step Methodology to Creating a Single View of your Business." https://www.mongodb.com/use-cases/single-view

[86] I/O (read/write) operations per second.

efficient access. Ops Manager can be leveraged for deploying clusters and scaling those clusters out and up as needs increase.[87]

The latest version of MongoDB is 3.4.2, and as of 3.4, support is built in for graph traversals, assuming the schema have been set up in an appropriate manner to support it.

a. API Support

MongoDB Compass is a freely available GUI tool for data exploration and analysis. MongoDB also provides the *mongo shell*, an interactive JavaScript interface for scriptable command line management of MongoDB instances. Native MongoDB data manipulation and queries are calls in the mongo shell API. Below are a few examples of different calls available:

- db.collection.insert()
- db.collection.find()
- db.collection.update()
- db.collection.remove(), db.collection.deleteMany()
- db.orders.aggregate(), db.orders.mapReduce()

The above calls, and many more, are chainable and composable to perform many types of query operations and data aggregations.

b. Programming Language Support

Official MongoDB drivers are available for many common programming languages,[88] which allow access to mongo shell capability from a preferred development language. Popular community-provided drivers are available for Go and Erlang, with a host of other languages also having community support.[89]

c. Download and Licensing Options

MongoDB Community Server is freely downloadable and is released under an AGPL license. Support is community-based only for this version. For small MongoDB installations, one can purchase support on an annual basis, which is called MongoDB Professional. MongoDB, Inc., also offers MongoDB Enterprise Advanced (EA). MongoDB EA provides a higher SLA than MongoDB professional, as well as a host of exclusive proprietary features:

[87] https://www.mongodb.com/mongodb-scale

[88] C, C++11, C#, Java, Node.js, Perl, PHP, Python, Motor (Python asynchronous), Ruby, Scala

[89] https://docs.mongodb.com/ecosystem/drivers/community-supported-drivers/

- Ops Manager or Cloud Manager Premium (management interface and monitoring dashboard, gets installed on a separate machine from MongoDB instances);
- Authentication and Authorization support (Red Hat, Kerberos, LDAP);
- Auditing;
- Additional storage options (encrypted, in-memory);
- Platform Certifications (Windows, various popular Linux distributions);
- Private, On-Demand training;
- Emergency Patches;
- Commercial License, Warranty and Indemnification.

The MongoDB EA annual cost is $11,999 per server, up to 250 GB RAM. For a 500 GB RAM server, the cost would be $24k/year.

MongoDB, Inc. also provides a SaaS implementation called MongoDB Atlas, hosted on Amazon's cloud. The IDA team inquired about whether this cloud implementation had obtained FedRAMP certification and got the impression that MongoDB, Inc., has not pursued certification, although they did point out that Amazon has FedRAMP certification. Further investigation of the Atlas implementation may be warranted if a cloud component is part of the solution architecture for the planned Army DFS Portal. Some information is already available at https://www.mongodb.com/cloud. Three cost/feature tiers are described there: Free, Essential (starting at $.08/hour per dedicated instance), and Professional (enterprise-grade, which is the only option that includes support). A price estimation tool is available at https://www.mongodb.com/cloud/atlas/pricing, which lets one consider various provisions of RAM and storage.

d. Documentation and Customer Support

Documentation is available at https://docs.mongodb.com/ for the MongoDB Server, MongoDB Atlas, MongoDB Ops Manager, MongoDB Compass, the various drivers and connectors, and the MongoDB Cloud Manager. The Cloud Manager is a tool for managing, monitoring and backing up a MongoDB infrastructure. Free community support is available from a variety of sources, including events, webinars, and user groups. See https://www.mongodb.com/community for links and details.

Professional, Enterprise Advanced, and "Professional Atlas Cloud" SKUs provide fixed levels of 24/7 call-in and on-site support. In addition, paid consultation services are available for assistance in setup, application development, administration, and operations. See https://www.mongodb.com/products/consulting for details.

e. Third Party Integrations

A variety of third-party integrations are documented on the MongoDB website at https://docs.mongodb.com/ecosystem/tools/. A Hadoop integration is available that allows MongoDB to be a source or sink for Hadoop operations. Red Hat Enterprise Linux Identity Management integration is available. Integrations with various third-party administration and monitoring tools are available. HTTP and REST interfaces to MongoDB may be created using various third-party tools. There's even support in Wireshark for monitoring MongoDB's over-the-wire protocols. In addition, there are connectors for the open source cluster computing framework, Apache Spark, and RDBMS-focused Business Intelligence tools such as Tableau.

29. Azure DocumentDB

Azure DocumentDB is a NoSQL database-as-a-service available only through the Azure cloud. A new version of this implementation was announced at Microsoft's annual developer conference in May 2017.[90] A DocumentDB database[91] stores JSON-formatted data documents organized in collections without the need to define explicit schema (see Figure A-9). It provides automatically indexing and replicating across all selected Azure regions. All storage is on SSDs. Attachment of data blobs, such as media files, to documents is supported.[92]

[90] https://arstechnica.com/information-technology/2017/05/azure-adds-mysql-postgresql-and-a-way-to-do-cloud-computing-outside-the-cloud/

[91] In DocumentDB parlance, a "database" is a logical container of users and collections.

[92] Attachments may be within DocumentDB (2 GB limit/account) or on a remote media store.

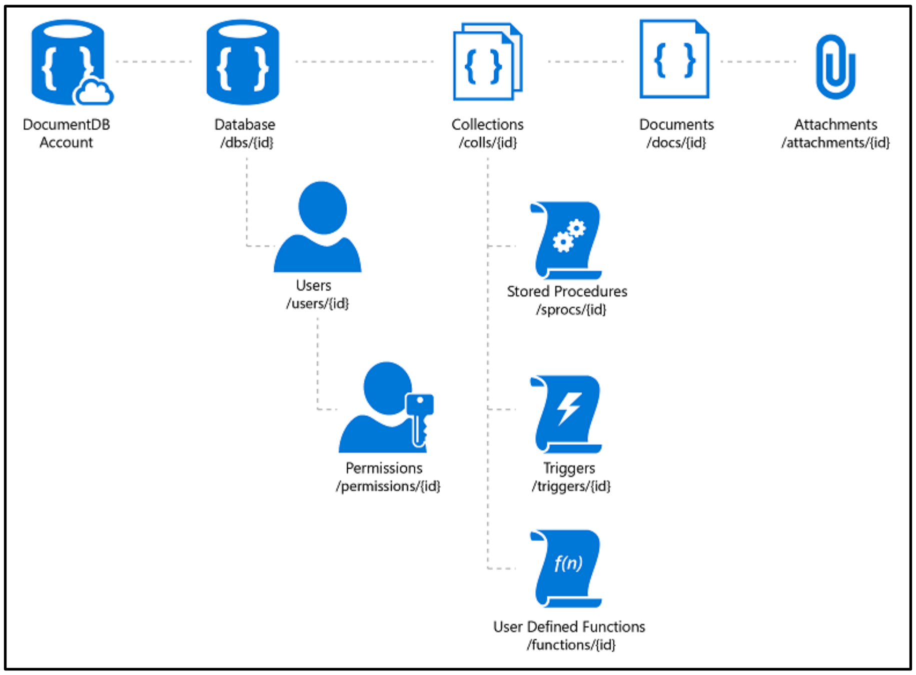

DocumentDB Account
Database /dbs/{id}
Collections /colls/{id}
Documents /docs/{id}
Attachments /attachments/{id}
Users /users/{id}
Stored Procedures /sprocs/{id}
Permissions /permissions/{id}
Triggers /triggers/{id}
User Defined Functions /functions/{id}

Figure A-9. Schematic Depiction of Data Supported by Azure DocumentDB

If DocumentDB sounds a bit like MongoDB, that is no accident. In fact, it can be configured to provide a MongoDB API emulation so that it can work with applications designed for MongoDB.

a. API Support

DocumentDB primarily has a REST API supporting CRUD operations and the submission of queries. API bindings exist for the following languages: .NET, Node.js, Java, JavaScript, and Python. There is also a server-side JavaScript API for creating stored procedures, triggers,[93] and user-defined functions[94] associated with document collections, analogous to the role of T-SQL in stored procedures on SQL Server.

b. Query Languages

DocumentDB offers a SQL-like query language,[95] which includes geospatial indexing and querying. A demo page is available at https://www.documentdb.com/sql/demo to try it out. While DocumentDB indexes all documents by default, the indexing can be turned

[93] Triggers allow the creation of server-side logic to occur when certain conditions occur. Say, a data canonicalization routine that runs when a record is added.

[94] User-defined functions allow database administrators to extend the DocumentDB query language.

[95] Syntax documentation available at https://msdn.microsoft.com/library/azure/dn782250.aspx

off for certain documents, and the indexing policy (consistent versus lazy) can be set as needed.

c. Third-Party Integrations

DocumentDB exposes a MongoDB emulation called *DocumentDB: API for MongoDB,* which allows it to be used in place of MongoDB in apps using the MongoDB API. The API documentation describes the rich variety of bindings that exist for MongoDB. Any of these may also be used with the DocumentDB's emulator. However, user comments at Microsoft's site imply that the emulation is not perfect and some options on some API bindings are not available.[96]

d. Download and Licensing Options

DocumentDB may be previewed and tried for free by signing up for a 1-month free trial of Azure at https://azure.microsoft.com/en-us/free/. From the Azure Portal (https://portal.azure.com/), which is where one manages all of one's Azure services and deployments, a new DocumentDB account and database(s) may be spun up by clicking on NoSQL (DocumentDB) on the left-hand menu. Figure A-10 gives a high-level view of the account management supported by Azure DocumentDB.

[96] https://docs.microsoft.com/en-us/azure/documentdb/documentdb-protocol-mongodb

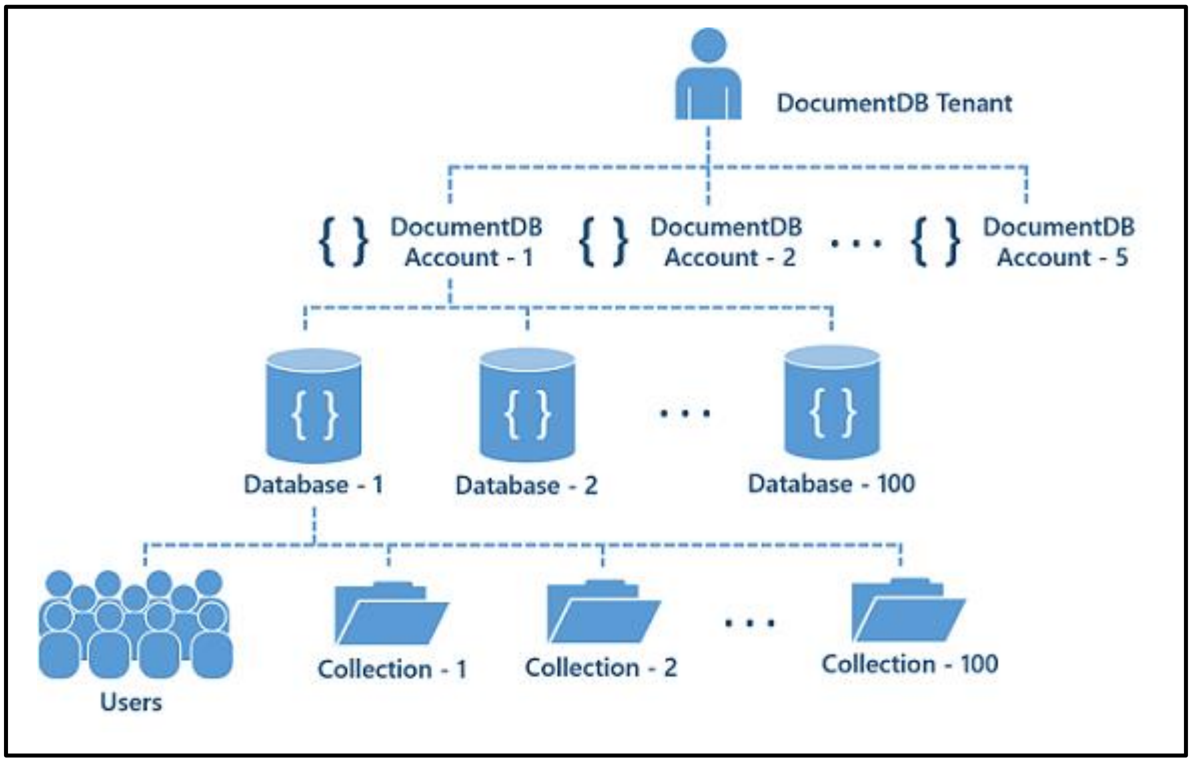


Figure A-10. Account Management in Azure DocumentDB

Once one is using normal paid Azure services, a basic level of DocumentDB service in a single region costs as follows:

Unit	Price
SSD Storage (per GB)	$0.25 per GB per month
Reserved RUs[97] per second (per 100 RUs, 400 RUs minimum)	$.008/hr per 100 reserved RU/second (therefore $0.032/hour minimum)

A good alternative to using a free trial is downloading the free DocumentDB emulator,[98] which allows one to develop and test applications on a local machine.

[97] A Request Unit (RU) is a measure of throughput used as a proxy for CPU, I/O and memory usage. One RU corresponds to the throughput of one read operation on a 1 KB document.

[98] https://aka.ms/documentdb-emulator

Appendix B
Sample Code Used for Testing Conversion of Legacy Relational Data to RDF Triples

The code examples included in this section are provided primarily to facilitate the development of assessment tests similar to those described in this document for graph databases not covered specifically during this phase of the study.

To eliminate barriers to the reuse of an entire snippet or a portion thereof all the code examples are released under the MIT license shown below.[99] The Institute for Defense Analyses, however, retains the copyright of all the code contained in this appendix.

[99] https://opensource.org/licenses/MIT

A. Materialized View of the PersonAssociation Table

As briefly discussed in Section 2B.4 above, the performance of relational databases can be substantially improved by caching the data obtained through the execution of a costly SELECT query so that it is no longer necessary to repeat the query every time that its data is needed.

1. SQL Queries for Manual Generation of the SDOS-1 View

The SQL queries listed below show a step-by-step construction of the tables that are used to create a materialized view corresponding to the subset of relationships from the PersonAssociation table that connect instances of Person that have a first name equal to "TAYLOR" with those that have a first name equal to "DORIAN." Some of the intermediate queries shown here need not be explicitly executed in a separate manner. However, since conflating intermediate steps can make the review of the logic being implemented harder, the IDA team chose to spell out all the steps.

```
--
-- STEP 1 :: Partition the Person Table into 8 separate tables of 16M records each
--

INSERT INTO T0
 SELECT perID,fname,lname,dob
 FROM person
 WHERE perID > 1000000000 AND
       perID < 1080000005;

-- Query OK, 16000000 rows affected (43.61 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0


INSERT INTO T1
 SELECT perID,fname,lname,dob
 FROM person
 WHERE perID > 1080000000 AND
       perID < 1160000005;

-- Query OK, 16000000 rows affected (44.67 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0


INSERT INTO T2
 SELECT perID,fname,lname,dob
 FROM person
 WHERE perID > 1160000000 AND
       perID < 1240000005;

-- Query OK, 16000000 rows affected (44.39 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0

INSERT INTO T3
 SELECT perID,fname,lname,dob
 FROM person
 WHERE perID > 1240000000 AND
       perID < 1320000005;

-- Query OK, 16000000 rows affected (58.21 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0

INSERT INTO T4
 SELECT perID,fname,lname,dob
 FROM person
 WHERE perID > 1320000000 AND
       perID < 1400000005;

-- Query OK, 16000000 rows affected (46.53 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0

INSERT INTO T5
 SELECT perID,fname,lname,dob
 FROM person
 WHERE perID > 1400000000 AND
       perID < 1480000005;

-- Query OK, 16000000 rows affected (44.54 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0

INSERT INTO T6
 SELECT perID,fname,lname,dob
 FROM person
 WHERE perID > 1480000000 AND
       perID < 1560000005;


INSERT INTO T7
 SELECT perID,fname,lname,dob
 FROM person
 WHERE perID > 1560000000 AND
       perID < 1640000005;

-- Query OK, 16000000 rows affected (44.63 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0

=================================================

--
```

-- STEP 2 :: Partition the PersonAssociation table into 8 separate tables of 16M each --

```
create table T0T1(
                      subjperID DECIMAL(10,0),
                      objperID  DECIMAL(10,0),
                      PRIMARY KEY(subjperID,objperID)
                     )ENGINE=MyISAM;

create table T1T2(
                      subjperID DECIMAL(10,0),
                      objperID  DECIMAL(10,0),
                      PRIMARY KEY(subjperID,objperID)
                     )ENGINE=MyISAM;

create table T2T3(
                      subjperID DECIMAL(10,0),
                      objperID  DECIMAL(10,0),
                      PRIMARY KEY(subjperID,objperID)
                     )ENGINE=MyISAM;

create table T3T4(
                      subjperID DECIMAL(10,0),
                      objperID  DECIMAL(10,0),
                      PRIMARY KEY(subjperID,objperID)
                     )ENGINE=MyISAM;

create table T4T5(
                      subjperID DECIMAL(10,0),
                      objperID  DECIMAL(10,0),
                      PRIMARY KEY(subjperID,objperID)
                     )ENGINE=MyISAM;

create table T5T6(
                      subjperID DECIMAL(10,0),
                      objperID  DECIMAL(10,0),
                      PRIMARY KEY(subjperID,objperID)
                     )ENGINE=MyISAM;

create table T6T7(
                      subjperID DECIMAL(10,0),
                      objperID  DECIMAL(10,0),
                      PRIMARY KEY(subjperID,objperID)
                     )ENGINE=MyISAM;

create table T7T0(
                      subjperID DECIMAL(10,0),
                      objperID  DECIMAL(10,0),
                      PRIMARY KEY(subjperID,objperID)
```

```
                    )ENGINE=MyISAM;


INSERT INTO T0T1
 SELECT subjperID,objperID
 FROM PersonAssociation
 WHERE subjperID > 1000000000 AND
       subjperID < 1080000005;

-- Query OK, 16000000 rows affected (25.61 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0


INSERT INTO T1T2
 SELECT subjperID,objperID
 FROM PersonAssociation
 WHERE subjperID > 1080000000 AND
       subjperID < 1160000005;

-- Query OK, 16000000 rows affected (27.92 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0


INSERT INTO T2T3
 SELECT subjperID,objperID
 FROM PersonAssociation
 WHERE subjperID > 1160000000 AND
       subjperID < 1240000005;

-- Query OK, 16000000 rows affected (28.12 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0


INSERT INTO T3T4
 SELECT subjperID,objperID
 FROM PersonAssociation
 WHERE subjperID > 1240000000 AND
       subjperID < 1320000005;


-- Query OK, 16000000 rows affected (28.11 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0


INSERT INTO T4T5
 SELECT subjperID,objperID
 FROM PersonAssociation
 WHERE subjperID > 1320000000 AND
```

```
       subjperID < 1400000005;

-- Query OK, 16000000 rows affected (28.44 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0


INSERT INTO T5T6
 SELECT subjperID,objperID
 FROM PersonAssociation
 WHERE subjperID > 1400000000 AND
       subjperID < 1480000005;

-- Query OK, 16000000 rows affected (28.14 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0


INSERT INTO T6T7
 SELECT subjperID,objperID
 FROM PersonAssociation
 WHERE subjperID > 1480000000 AND
       subjperID < 1560000005;

-- Query OK, 16000000 rows affected (27.87 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0


INSERT INTO T7T0
 SELECT subjperID,objperID
 FROM PersonAssociation
 WHERE subjperID > 1560000000;

-- Query OK, 16000000 rows affected (26.30 sec)
-- Records: 16000000  Duplicates: 0  Warnings: 0
```

```
--
-- STEP 3 :: Filtering records for source and destination
--         according to the restriction chosen
--         (e.g., source person has fname = 'TAYLOR' and
--         target person has fname = 'DORIAN'

create table TAYLORT0(
                     perID decimal(10,0) PRIMARY KEY,
                     fname varchar(50),
                     lname varchar(100),
                     dob varchar(10)
                    )ENGINE=MyISAM;

create table TAYLORT1(
                     perID decimal(10,0) PRIMARY KEY,
                     fname varchar(50),
                     lname varchar(100),
                     dob varchar(10)
                    )ENGINE=MyISAM;

create table TAYLORT2(
                     perID decimal(10,0) PRIMARY KEY,
                     fname varchar(50),
                     lname varchar(100),
                     dob varchar(10)
                    )ENGINE=MyISAM;

create table TAYLORT3(
                     perID decimal(10,0) PRIMARY KEY,
                     fname varchar(50),
                     lname varchar(100),
                     dob varchar(10)
                    )ENGINE=MyISAM;

create table TAYLORT4(
                     perID decimal(10,0) PRIMARY KEY,
                     fname varchar(50),
                     lname varchar(100),
                     dob varchar(10)
                    )ENGINE=MyISAM;

create table TAYLORT5(
                     perID decimal(10,0) PRIMARY KEY,
                     fname varchar(50),
                     lname varchar(100),
                     dob varchar(10)
                    )ENGINE=MyISAM;
```

```
create table TAYLORT6(
                    perID decimal(10,0) PRIMARY KEY,
                    fname varchar(50),
                    lname varchar(100),
                    dob varchar(10)
                   )ENGINE=MyISAM;

create table TAYLORT7(
                    perID decimal(10,0) PRIMARY KEY,
                    fname varchar(50),
                    lname varchar(100),
                    dob varchar(10)
                   )ENGINE=MyISAM;

======================================================

create table DORIANT0(
                    perID decimal(10,0) PRIMARY KEY,
                    fname varchar(50),
                    lname varchar(100),
                    dob varchar(10)
                   )ENGINE=MyISAM;

create table DORIANT1(
                    perID decimal(10,0) PRIMARY KEY,
                    fname varchar(50),
                    lname varchar(100),
                    dob varchar(10)
                   )ENGINE=MyISAM;

create table DORIANT2(
                    perID decimal(10,0) PRIMARY KEY,
                    fname varchar(50),
                    lname varchar(100),
                    dob varchar(10)
                   )ENGINE=MyISAM;

create table DORIANT3(
                    perID decimal(10,0) PRIMARY KEY,
                    fname varchar(50),
                    lname varchar(100),
                    dob varchar(10)
                   )ENGINE=MyISAM;

create table DORIANT4(
                    perID decimal(10,0) PRIMARY KEY,
                    fname varchar(50),
                    lname varchar(100),
                     dob varchar(10)
                    )ENGINE=MyISAM;

create table DORIANT5(
                     perID decimal(10,0) PRIMARY KEY,
                     fname varchar(50),
                     lname varchar(100),
                     dob varchar(10)
                    )ENGINE=MyISAM;



create table DORIANT6(
                     perID decimal(10,0) PRIMARY KEY,
                     fname varchar(50),
                     lname varchar(100),
                     dob varchar(10)
                    )ENGINE=MyISAM;

create table DORIANT7(
                     perID decimal(10,0) PRIMARY KEY,
                     fname varchar(50),
                     lname varchar(100),
                     dob varchar(10)
                    )ENGINE=MyISAM;


============ POPULATE THE TABLES ======================

INSERT INTO TAYLORT0

  SELECT perID,fname,lname,dob
  FROM T0
  WHERE fname = 'TAYLOR';

-- Query OK, 9007 rows affected (3.39 sec)
-- Records: 9007  Duplicates: 0  Warnings: 0


INSERT INTO TAYLORT1

  SELECT perID,fname,lname,dob
  FROM T1
  WHERE fname = 'TAYLOR';

-- Query OK, 8736 rows affected (3.37 sec)
-- Records: 8736  Duplicates: 0  Warnings: 0
```

```
INSERT INTO TAYLORT2

  SELECT perID,fname,lname,dob
  FROM T2
  WHERE fname = 'TAYLOR';

-- Query OK, 8836 rows affected (3.32 sec)
-- Records: 8836  Duplicates: 0  Warnings: 0


INSERT INTO TAYLORT3

  SELECT perID,fname,lname,dob
  FROM T3
  WHERE fname = 'TAYLOR';

-- Query OK, 8879 rows affected (3.42 sec)
-- Records: 8879  Duplicates: 0  Warnings: 0


INSERT INTO TAYLORT4

  SELECT perID,fname,lname,dob
  FROM T4
  WHERE fname = 'TAYLOR';

-- Query OK, 9137 rows affected (3.40 sec)
-- Records: 9137  Duplicates: 0  Warnings: 0


INSERT INTO TAYLORT5

  SELECT perID,fname,lname,dob
  FROM T5
  WHERE fname = 'TAYLOR';

-- Query OK, 9057 rows affected (3.32 sec)
-- Records: 9057  Duplicates: 0  Warnings: 0


INSERT INTO TAYLORT6

  SELECT perID,fname,lname,dob
  FROM T6
  WHERE fname = 'TAYLOR';

-- Query OK, 8925 rows affected (3.36 sec)
-- Records: 8925  Duplicates: 0  Warnings: 0


INSERT INTO TAYLORT7

  SELECT perID,fname,lname,dob
  FROM T7
  WHERE fname = 'TAYLOR';

-- Query OK, 9016 rows affected (3.41 sec)
-- Records: 9016  Duplicates: 0  Warnings: 0

==============================================

INSERT INTO DORIANT0

   SELECT perID,fname,lname,dob
   FROM T0
   WHERE fname = 'DORIAN';

-- Query OK, 8801 rows affected (3.10 sec)
-- Records: 8801  Duplicates: 0  Warnings: 0


INSERT INTO DORIANT1

   SELECT perID,fname,lname,dob
   FROM T1
   WHERE fname = 'DORIAN';

-- Query OK, 8951 rows affected (1.46 sec)
-- Records: 8951  Duplicates: 0  Warnings: 0


INSERT INTO DORIANT2

   SELECT perID,fname,lname,dob
   FROM T2
   WHERE fname = 'DORIAN';

-- Query OK, 9040 rows affected (1.47 sec)
-- Records: 9040  Duplicates: 0  Warnings: 0



INSERT INTO DORIANT3

   SELECT perID,fname,lname,dob
   FROM T3
   WHERE fname = 'DORIAN';

-- Query OK, 8840 rows affected (1.46 sec)
-- Records: 8840  Duplicates: 0  Warnings: 0


INSERT INTO DORIANT4

   SELECT perID,fname,lname,dob
   FROM T4
   WHERE fname = 'DORIAN';

-- Query OK, 8980 rows affected (1.47 sec)
-- Records: 8980  Duplicates: 0  Warnings: 0


INSERT INTO DORIANT5

   SELECT perID,fname,lname,dob
   FROM T5
   WHERE fname = 'DORIAN';

-- Query OK, 9039 rows affected (1.46 sec)
-- Records: 9039  Duplicates: 0  Warnings: 0


INSERT INTO DORIANT6

   SELECT perID,fname,lname,dob
   FROM T6
   WHERE fname = 'DORIAN';

-- Query OK, 8957 rows affected (1.46 sec)
-- Records: 8957  Duplicates: 0  Warnings: 0


INSERT INTO DORIANT7

   SELECT perID,fname,lname,dob
   FROM T7
   WHERE fname = 'DORIAN';

-- Query OK, 8985 rows affected (1.46 sec)
-- Records: 8985  Duplicates: 0  Warnings: 0

====================================================
--
-- STEP 4 :: create the sdosOne table and populate it
--

CREATE TABLE sdosOne(
                      subjperID decimal(10,0),
                      objperID decimal(10,0),
                      PRIMARY KEY(subjperID,objperID)
                     )ENGINE=MyISAM;

-- Query OK, 0 rows affected (0.04 sec)

INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT0 as a,
      DORIANT1 as b,
      T0T1 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;

-- Query OK, 8 rows affected (13.46 sec)
-- Records: 8  Duplicates: 0  Warnings: 0

--

INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT1 as a,
      DORIANT2 as b,
      T1T2 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;

-- Query OK, 6 rows affected (13.24 sec)
-- Records: 6  Duplicates: 0  Warnings: 0

--
```

```
INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT2 as a,
      DORIANT3 as b,
      T2T3 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;

-- Query OK, 5 rows affected (13.25 sec)
-- Records: 5  Duplicates: 0  Warnings: 0

--

INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT3 as a,
      DORIANT4 as b,
      T3T4 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;

-- Query OK, 10 rows affected (13.20 sec)
-- Records: 10  Duplicates: 0  Warnings: 0

--

INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT4 as a,
      DORIANT5 as b,
      T4T5 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;
```

```
-- Query OK, 6 rows affected (13.18 sec)
-- Records: 6  Duplicates: 0  Warnings: 0

--

INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT5 as a,
      DORIANT6 as b,
      T5T6 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;

-- Query OK, 5 rows affected (13.17 sec)
-- Records: 5  Duplicates: 0  Warnings: 0

--

INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT6 as a,
      DORIANT7 as b,
      T6T7 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;

-- Query OK, 3 rows affected (13.20 sec)
-- Records: 3  Duplicates: 0  Warnings: 0

--

INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT7 as a,
      DORIANT0 as b,
      T7T0 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;

-- Query OK, 8 rows affected (13.15 sec)
-- Records: 8  Duplicates: 0  Warnings: 0

====================================================
```

--
-- STEP 5 :: Find all the instances of source person named 'TAYLOR'
--   who know an instance of target person named 'DORIAN'
--

```
SELECT a.perID as person00,
        a.fname as fname00,
        b.perID as person01,
        b.fname as fname01

 FROM person08a as a,
      person08a as b,
      sdosZero as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY person00;
```

OUTPUT OF SELECT QUERY FOR STEP 5

```
+------------+---------+------------+---------+
| person00   | fname00 | person01   | fname01 |
+------------+---------+------------+---------+
| 1004976325 | TAYLOR  | 1084976325 | DORIAN  |
| 1016860255 | TAYLOR  | 1096860255 | DORIAN  |
| 1027603495 | TAYLOR  | 1107603495 | DORIAN  |
| 1038392570 | TAYLOR  | 1118392570 | DORIAN  |
| 1043029360 | TAYLOR  | 1123029360 | DORIAN  |
| 1048430955 | TAYLOR  | 1128430955 | DORIAN  |
| 1072149950 | TAYLOR  | 1152149950 | DORIAN  |
| 1079501795 | TAYLOR  | 1159501795 | DORIAN  |
| 1083783770 | TAYLOR  | 1163783770 | DORIAN  |
| 1088888525 | TAYLOR  | 1168888525 | DORIAN  |
| 1095811010 | TAYLOR  | 1175811010 | DORIAN  |
| 1122040465 | TAYLOR  | 1202040465 | DORIAN  |
| 1122666225 | TAYLOR  | 1202666225 | DORIAN  |
| 1140273495 | TAYLOR  | 1220273495 | DORIAN  |
| 1190249385 | TAYLOR  | 1270249385 | DORIAN  |
| 1203953995 | TAYLOR  | 1283953995 | DORIAN  |
| 1207403270 | TAYLOR  | 1287403270 | DORIAN  |
| 1209078120 | TAYLOR  | 1289078120 | DORIAN  |
| 1233472230 | TAYLOR  | 1313472230 | DORIAN  |
| 1247940635 | TAYLOR  | 1327940635 | DORIAN  |
| 1249164380 | TAYLOR  | 1329164380 | DORIAN  |
| 1251558130 | TAYLOR  | 1331558130 | DORIAN  |
| 1258145160 | TAYLOR  | 1338145160 | DORIAN  |
| 1271797260 | TAYLOR  | 1351797260 | DORIAN  |
| 1273994435 | TAYLOR  | 1353994435 | DORIAN  |
| 1283096850 | TAYLOR  | 1363096850 | DORIAN  |
| 1297638350 | TAYLOR  | 1377638350 | DORIAN  |
| 1307518160 | TAYLOR  | 1387518160 | DORIAN  |
| 1309076190 | TAYLOR  | 1389076190 | DORIAN  |
| 1323205235 | TAYLOR  | 1403205235 | DORIAN  |
| 1326333740 | TAYLOR  | 1406333740 | DORIAN  |
| 1349861265 | TAYLOR  | 1429861265 | DORIAN  |
| 1360127715 | TAYLOR  | 1440127715 | DORIAN  |
| 1396712980 | TAYLOR  | 1476712980 | DORIAN  |
| 1398762465 | TAYLOR  | 1478762465 | DORIAN  |
| 1400596635 | TAYLOR  | 1480596635 | DORIAN  |
| 1404557310 | TAYLOR  | 1484557310 | DORIAN  |
| 1418384575 | TAYLOR  | 1498384575 | DORIAN  |
| 1432621910 | TAYLOR  | 1512621910 | DORIAN  |
| 1447515160 | TAYLOR  | 1527515160 | DORIAN  |
| 1530830930 | TAYLOR  | 1610830930 | DORIAN  |
| 1531651245 | TAYLOR  | 1611651245 | DORIAN  |
| 1559036900 | TAYLOR  | 1639036900 | DORIAN  |
| 1582673590 | TAYLOR  | 1022673590 | DORIAN  |
| 1607800590 | TAYLOR  | 1047800590 | DORIAN  |
| 1610257485 | TAYLOR  | 1050257485 | DORIAN  |
| 1611331480 | TAYLOR  | 1051331480 | DORIAN  |
| 1618627300 | TAYLOR  | 1058627300 | DORIAN  |
| 1618801440 | TAYLOR  | 1058801440 | DORIAN  |
| 1638764355 | TAYLOR  | 1078764355 | DORIAN  |
| 1639182220 | TAYLOR  | 1079182220 | DORIAN  |
+------------+---------+------------+---------+
-- 51 rows in set (0.00 sec)
```

2. Python Scripts to Generate the Materialized view Programmatically

The steps described in the preceding section can be executed programmatically. The Python script below calls 10 subprograms that carry out the creation of the empty intermediate tables, their subsequent population, and finally the generation of the materialized view for the SDOS-1 case discussed in the main part of this document.

Running all the programs takes a little over 12 minutes, but all the queries that are made using the materialized view thereafter run in under one hundredth of a second. With only three attributes characterizing the instances of the Person table, this may seem like a large investment. However, with a richer set of attributes one could run a large number of combinations and the cost of the queries would be minimal. For example one could find all the instances of “TAYLOR” that are born in February and know instances of “DORIAN” that live in Seattle (assuming that the records contain that information). But no matter what combination one thinks of, the time to retrieve the data using the materialized view will be under one hundredth of a second. This arguably provides justification for the initial investment in creating the materialized view.

a. The Top Level Program

```
# Copyright ©2017. The Institute for Defense Analyses (IDA).
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do
 # so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or substantial
# portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
# SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE
# BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH,
# THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.#

# Author: Francisco Loaiza, Ph.D., J.D.
#       Institute for Defense Analyses
#       Alexandria, Virginia, USA
#

#!/usr/bin/python
# -*- coding: utf-8 -*-

import os
import time

t0 = time.time()
```

```
print t0," start time"

import CreateTSeries
import CreateT2TSeries
import CreateSDOSSeries
import CreateTaylorSeries
import CreateDorianSeries
import PopulateTSeries
import PopulateT2TSeries
import PopulateTaylorSeries
import PopulateDorianSeries
import PopulateSDOSOne

print time.time() - t0, "seconds to complete SDOS-0"

#
# 730.946802855 seconds to complete SDOS-0 (or 12.18 min)
#
```

b. Database Preparation

The two Python scripts shown in this section create the intermediate tables that will store the instances of Person that have fname = “TAYLOR” and fname = “DORIAN” respectively.

```
# Author: Francisco Loaiza, Ph.D., J.D.
#         Institute for Defense Analyses
#         Alexandria, Virginia, USA
#
#!/usr/bin/python
# -*- coding: utf-8 -*-

# CreateTSeries.py

import MySQLdb as mdb
import time

# 0.040752 time to create T0 through T7 tables using a PC with 2 cores
```

```
t0 = time.clock()

con = mdb.connect(‘localhost’, ‘<the user>‘, ‘<the password>‘, ‘<the database>‘);

with con:

   cur = con.cursor()
   cur.execute("DROP TABLE IF EXISTS T0")
   cur.execute("""
   CREATE TABLE T0(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS T1")
   cur.execute("""
CREATE TABLE T1(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS T2")
   cur.execute("""
CREATE TABLE T2(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS T3")
   cur.execute("""
CREATE TABLE T3(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS T4")
   cur.execute("""
CREATE TABLE T4(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS T5")
   cur.execute("""
CREATE TABLE T5(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;

""")

   cur.execute("DROP TABLE IF EXISTS T6")
   cur.execute("""
CREATE TABLE T6(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS T7")
   cur.execute("""
CREATE TABLE T7(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

print time.clock() - t0, "time to create T0 through T7 tables"

con.close()
```

```
# Copyright ©2017. The Institute for Defense Analyses (IDA).
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do
 # so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or substantial
# portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
# SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE
# BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH,
# THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.#

# Author: Francisco Loaiza, Ph.D., J.D.
#        Institute for Defense Analyses
#        Alexandria, Virginia, USA
#

#!/usr/bin/python
# -*- coding: utf-8 -*-

#CreateT2TSeries.py

import MySQLdb as mdb
import time

t0 = time.clock()

con = mdb.connect('localhost', '<the user>', '<the password>', '<the database>');

with con:

    cur = con.cursor()

    cur.execute("DROP TABLE IF EXISTS T0T1")
    cur.execute("""create table T0T1(
                    subjperID DECIMAL(10,0),
                    objperID  DECIMAL(10,0),
                    PRIMARY KEY(subjperID,objperID)
                   )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS T1T2")
    cur.execute("""create table T1T2(
                    subjperID DECIMAL(10,0),
                    objperID  DECIMAL(10,0),
                    PRIMARY KEY(subjperID,objperID)
                   )ENGINE=MyISAM;
""")
```

```
    cur.execute("DROP TABLE IF EXISTS T2T3")
    cur.execute("""create table T2T3(
                   subjperID DECIMAL(10,0),
                   objperID  DECIMAL(10,0),
                   PRIMARY KEY(subjperID,objperID)
                  )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS T3T4")
    cur.execute("""create table T3T4(
                   subjperID DECIMAL(10,0),
                   objperID  DECIMAL(10,0),
                   PRIMARY KEY(subjperID,objperID)
                  )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS T4T5")
    cur.execute("""create table T4T5(
                   subjperID DECIMAL(10,0),
                   objperID  DECIMAL(10,0),
                   PRIMARY KEY(subjperID,objperID)
                  )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS T5T6")
    cur.execute("""create table T5T6(
                   subjperID DECIMAL(10,0),
                   objperID  DECIMAL(10,0),
                   PRIMARY KEY(subjperID,objperID)
                  )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS T6T7")
    cur.execute("""create table T6T7(
                   subjperID DECIMAL(10,0),
                   objperID  DECIMAL(10,0),
                   PRIMARY KEY(subjperID,objperID)
                  )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS T7T0")
    cur.execute("""create table T7T0(
                   subjperID DECIMAL(10,0),
                   objperID  DECIMAL(10,0),
                   PRIMARY KEY(subjperID,objperID)
                  )ENGINE=MyISAM;
""")

print time.clock() - t0, "time to create T2T tables"

con.close()
```

```
# Copyright ©2017. The Institute for Defense Analyses (IDA).
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do
 # so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or substantial
# portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
# SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE
# BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH,
# THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.#

# Author: Francisco Loaiza, Ph.D., J.D.
#         Institute for Defense Analyses
#         Alexandria, Virginia, USA

#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb
import time

# CreateSDOSSeries

t0 = time.clock()

con = mdb.connect(‘localhost’, ‘<the user>‘, ‘<the password>‘, ‘<the database>‘);

with con:

    cur = con.cursor()

    cur.execute("DROP TABLE IF EXISTS sdosZero")
    cur.execute("""CREATE TABLE sdosZero(
                  subjperID decimal(10,0),
                  objperID decimal(10,0),
                  PRIMARY KEY(subjperID,objperID)
                 )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS sdosOne")
    cur.execute("""CREATE TABLE sdosOne(
                  subjperID decimal(10,0),
                  objperID decimal(10,0),
                  PRIMARY KEY(subjperID,objperID)
                 )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS sdosTwo")
    cur.execute("""CREATE TABLE sdosTwo(
                    subjperID decimal(10,0),
                    objperID decimal(10,0),
                    PRIMARY KEY(subjperID,objperID)
                   )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS sdosThree")
    cur.execute("""CREATE TABLE sdosThree(
                    subjperID decimal(10,0),
                    objperID decimal(10,0),
                    PRIMARY KEY(subjperID,objperID)
                   )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS sdosFour")
    cur.execute("""CREATE TABLE sdosFour(
                    subjperID decimal(10,0),
                    objperID decimal(10,0),
                    PRIMARY KEY(subjperID,objperID)
                   )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS sdosFive")
    cur.execute("""CREATE TABLE sdosFive(
                    subjperID decimal(10,0),
                    objperID decimal(10,0),
                    PRIMARY KEY(subjperID,objperID)
                   )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS sdosSix")
    cur.execute("""CREATE TABLE sdosSix(
                    subjperID decimal(10,0),
                    objperID decimal(10,0),
                    PRIMARY KEY(subjperID,objperID)
                   )ENGINE=MyISAM;
""")

print time.clock() - t0, "time to create sdosZero through sdosSix tables"

con.close()
```

```
# Author: Francisco Loaiza, Ph.D., J.D.
#         Institute for Defense Analyses
#         Alexandria, Virginia, USA

#!/usr/bin/python
# -*- coding: utf-8 -*-

# CreateTaylorSeries.py

import MySQLdb as mdb
import time

t0 = time.clock()

con = mdb.connect('localhost', '<the user>', '<the password>', '<the database>');

with con:

    cur = con.cursor()
    cur.execute("DROP TABLE IF EXISTS TAYLORT0")
    cur.execute("""CREATE TABLE TAYLORT0(perID DECIMAL(10,0) PRIMARY KEY,
                fname VARCHAR(50),
                lname VARCHAR(100),
                dob   VARCHAR(10)
               )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS TAYLORT1")
    cur.execute("""CREATE TABLE TAYLORT1(perID DECIMAL(10,0) PRIMARY KEY,
                fname VARCHAR(50),
                lname VARCHAR(100),
                dob   VARCHAR(10)
               )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS TAYLORT2")
    cur.execute("""CREATE TABLE TAYLORT2(perID DECIMAL(10,0) PRIMARY KEY,
                fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS TAYLORT3")
   cur.execute("""CREATE TABLE TAYLORT3(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS TAYLORT4")
   cur.execute("""CREATE TABLE TAYLORT4(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS TAYLORT5")
   cur.execute("""CREATE TABLE TAYLORT5(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;

""")

   cur.execute("DROP TABLE IF EXISTS TAYLORT6")
   cur.execute("""CREATE TABLE TAYLORT6(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS TAYLORT7")
   cur.execute("""CREATE TABLE TAYLORT7(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

print time.clock() - t0, "time to create TAYLORT0 through TAYLORT7 tables"

con.close()
```

```
# Copyright ©2017. The Institute for Defense Analyses (IDA).
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do
 # so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or substantial
# portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
# SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE
# BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH,
# THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.#

# Author: Francisco Loaiza, Ph.D., J.D.
#       Institute for Defense Analyses
#       Alexandria, Virginia, USA

#!/usr/bin/python
# -*- coding: utf-8 -*-

# CreateDorianSeries.py

import MySQLdb as mdb
import time

t0 = time.clock()

con = mdb.connect('localhost', '<the user>', '<the password>', '<the database>');

with con:

    cur = con.cursor()
    cur.execute("DROP TABLE IF EXISTS DORIANT0")
    cur.execute("""CREATE TABLE DORIANT0(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS DORIANT1")
    cur.execute("""CREATE TABLE DORIANT1(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

    cur.execute("DROP TABLE IF EXISTS DORIANT2")
    cur.execute("""CREATE TABLE DORIANT2(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS DORIANT3")
   cur.execute("""CREATE TABLE DORIANT3(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS DORIANT4")
   cur.execute("""CREATE TABLE DORIANT4(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS DORIANT5")
   cur.execute("""CREATE TABLE DORIANT5(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;

""")

   cur.execute("DROP TABLE IF EXISTS DORIANT6")
   cur.execute("""CREATE TABLE DORIANT6(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

   cur.execute("DROP TABLE IF EXISTS DORIANT7")
   cur.execute("""CREATE TABLE DORIANT7(perID DECIMAL(10,0) PRIMARY KEY,
            fname VARCHAR(50),
            lname VARCHAR(100),
            dob   VARCHAR(10)
           )ENGINE=MyISAM;
""")

print time.clock() - t0, "time to create DORIANT0 through DORIANT7 tables"

con.close()
```

c. Populating the New Tables

Once all the intermediate tables have been created, they can be loaded with the pertinent subsets from the Person table and the PersonAssociation table respectively.

```
# Copyright ©2017. The Institute for Defense Analyses (IDA).
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do
 # so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or substantial
# portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
# SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE
# BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH,
# THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.#

# Author: Francisco Loaiza, Ph.D., J.D.
#        Institute for Defense Analyses
#        Alexandria, Virginia, USA

#!/usr/bin/python
# -*- coding: utf-8 -*-

#PopulateTSeries.py

import MySQLdb as mdb
import time

# 0.040752 time to create T0 through T7 tables using a PC with 2 cores

t0 = time.clock()

con = mdb.connect(‘localhost’, ‘<the user>‘, ‘<the password>‘, ‘<the database>‘);

with con:

    cur = con.cursor()

    cur.execute("""
INSERT INTO T0
 SELECT perID,fname,lname,dob
 FROM person08a
 WHERE perID > 1000000000 AND
       perID < 1080000005;
""")

    cur.execute("""
INSERT INTO T1
 SELECT perID,fname,lname,dob
 FROM person08a
 WHERE perID > 1080000000 AND
       perID < 1160000005;
""")

    cur.execute("""
INSERT INTO T2
 SELECT perID,fname,lname,dob
 FROM person08a
 WHERE perID > 1160000000 AND
       perID < 1240000005;
""")

    cur.execute("""
INSERT INTO T3
 SELECT perID,fname,lname,dob
 FROM person08a
 WHERE perID > 1240000000 AND
       perID < 1320000005;
""")

    cur.execute("""
INSERT INTO T4
 SELECT perID,fname,lname,dob
 FROM person08a
 WHERE perID > 1320000000 AND
       perID < 1400000005;
""")

    cur.execute("""
INSERT INTO T5
 SELECT perID,fname,lname,dob
 FROM person08a
 WHERE perID > 1400000000 AND
       perID < 1480000005;
""")

    cur.execute("""
INSERT INTO T6
 SELECT perID,fname,lname,dob
 FROM person08a
 WHERE perID > 1480000000 AND
       perID < 1560000005;
""")

    cur.execute("""
INSERT INTO T7
 SELECT perID,fname,lname,dob
 FROM person08a
 WHERE perID > 1560000000 AND
       perID < 1640000005;
""")

print time.clock() - t0, "time to populate T0 through T7 tables"

con.close()
```

```
# Author: Francisco Loaiza, Ph.D., J.D.
#       Institute for Defense Analyses
#       Alexandria, Virginia, USA


#!/usr/bin/python
# -*- coding: utf-8 -*-

# PopulateT2TSeries.py

import MySQLdb as mdb
import time

# 0.040752 time to create T0 through T7 tables using a PC with 2 cores

t0 = time.clock()

con = mdb.connect(‘localhost’, ‘<the user>‘, ‘<the password>‘, ‘<the database>‘);

with con:

    cur = con.cursor()

    cur.execute("""INSERT INTO T0T1
 SELECT subjperID,objperID
 FROM per08aper08aAssn
 WHERE subjperID > 1000000000 AND
       subjperID < 1080000005;
""")


    cur.execute("""INSERT INTO T1T2
 SELECT subjperID,objperID
 FROM per08aper08aAssn
 WHERE subjperID > 1080000000 AND
       subjperID < 1160000005;
""")
```

```
    cur.execute("""INSERT INTO T2T3
 SELECT subjperID,objperID
 FROM per08aper08aAssn
 WHERE subjperID > 1160000000 AND
       subjperID < 1240000005;
""")


    cur.execute("""INSERT INTO T3T4
 SELECT subjperID,objperID
 FROM per08aper08aAssn
 WHERE subjperID > 1240000000 AND
       subjperID < 1320000005;
""")


    cur.execute("""INSERT INTO T4T5
 SELECT subjperID,objperID
 FROM per08aper08aAssn
 WHERE subjperID > 1320000000 AND
       subjperID < 1400000005;
""")


    cur.execute("""INSERT INTO T5T6
 SELECT subjperID,objperID
 FROM per08aper08aAssn
 WHERE subjperID > 1400000000 AND
       subjperID < 1480000005;
""")


    cur.execute("""INSERT INTO T6T7
 SELECT subjperID,objperID
 FROM per08aper08aAssn
 WHERE subjperID > 1480000000 AND
       subjperID < 1560000005;
""")


    cur.execute("""INSERT INTO T7T0
 SELECT subjperID,objperID
 FROM per08aper08aAssn
 WHERE subjperID > 1560000000;
""")

print time.clock() - t0, "time to populate T2T tables"

con.close()
```

```
# Copyright ©2017. The Institute for Defense Analyses (IDA).
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do
 # so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or substantial
# portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
# SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE
# BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH,
# THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.#

# Author: Francisco Loaiza, Ph.D., J.D.
#       Institute for Defense Analyses
#       Alexandria, Virginia, USA



#!/usr/bin/python
# -*- coding: utf-8 -*-

# PopulateTaylorSeries.py

import MySQLdb as mdb
import time

t0 = time.clock()

con = mdb.connect(‘localhost’, ‘<the user>‘, ‘<the password>‘, ‘<the database>‘);

with con:

    cur = con.cursor()

    cur.execute("""INSERT INTO TAYLORT0

  SELECT perID,fname,lname,dob
  FROM T0
  WHERE fname = 'TAYLOR';
""")


    cur.execute("""INSERT INTO TAYLORT1

  SELECT perID,fname,lname,dob
  FROM T1
  WHERE fname = 'TAYLOR';
""")
```

```
    cur.execute("""INSERT INTO TAYLORT2

  SELECT perID,fname,lname,dob
  FROM T2
  WHERE fname = 'TAYLOR';
""")


    cur.execute("""INSERT INTO TAYLORT3

  SELECT perID,fname,lname,dob
  FROM T3
  WHERE fname = 'TAYLOR';
""")


    cur.execute("""INSERT INTO TAYLORT4

  SELECT perID,fname,lname,dob
  FROM T4
  WHERE fname = 'TAYLOR';
""")


    cur.execute("""INSERT INTO TAYLORT5

  SELECT perID,fname,lname,dob
  FROM T5
  WHERE fname = 'TAYLOR';
""")


    cur.execute("""INSERT INTO TAYLORT6

  SELECT perID,fname,lname,dob
  FROM T6
  WHERE fname = 'TAYLOR';
""")


    cur.execute("""INSERT INTO TAYLORT7

  SELECT perID,fname,lname,dob
  FROM T7
  WHERE fname = 'TAYLOR';
""")

print time.clock() - t0, "time to populate TAYLORT0 through TAYLORT7 tables"

con.close()
```

```
# Copyright ©2017. The Institute for Defense Analyses (IDA).
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do
 # so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or substantial
# portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
# SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE
# BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH,
# THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.#

# Author: Francisco Loaiza, Ph.D., J.D.
#       Institute for Defense Analyses
#       Alexandria, Virginia, USA


#!/usr/bin/python
# -*- coding: utf-8 -*-

# PopulateDorianSeries.py

import MySQLdb as mdb
import time

# 0.040752 time to create T0 through T7 tables using a PC with 2 cores

t0 = time.clock()

con = mdb.connect('localhost', '<the user>', '<the password>', '<the database>');

with con:

    cur = con.cursor()

    cur.execute("""INSERT INTO DORIANT0

  SELECT perID,fname,lname,dob
  FROM T0
  WHERE fname = 'DORIAN';
""")


    cur.execute("""INSERT INTO DORIANT1

  SELECT perID,fname,lname,dob
  FROM T1
  WHERE fname = 'DORIAN';
""")
```

```
    cur.execute("""INSERT INTO DORIANT2

  SELECT perID,fname,lname,dob
  FROM T2
  WHERE fname = 'DORIAN';
""")


    cur.execute("""INSERT INTO DORIANT3

  SELECT perID,fname,lname,dob
  FROM T3
  WHERE fname = 'DORIAN';
""")


    cur.execute("""INSERT INTO DORIANT4

  SELECT perID,fname,lname,dob
  FROM T4
  WHERE fname = 'DORIAN';
""")


    cur.execute("""INSERT INTO DORIANT5

  SELECT perID,fname,lname,dob
  FROM T5
  WHERE fname = 'DORIAN';
""")


    cur.execute("""INSERT INTO DORIANT6

  SELECT perID,fname,lname,dob
  FROM T6
  WHERE fname = 'DORIAN';
""")


    cur.execute("""INSERT INTO DORIANT7

  SELECT perID,fname,lname,dob
  FROM T7
  WHERE fname = 'DORIAN';
""")

print time.clock() - t0, "time to populate DORIANT0 through DORIANT7 tables"

con.close()
```

d. Generating the Materialized View SDOSOne

The final step is to collect into a single table all the subsets of the Person table that satisfy the constraints imposed for the SDOS-1 case.

```
# Author: Francisco Loaiza, Ph.D., J.D.
#        Institute for Defense Analyses
#        Alexandria, Virginia, USA


#!/usr/bin/python
# -*- coding: utf-8 -*-

# PopulateSDOSOne.py

import MySQLdb as mdb
import time

# 0.040752 time to create T0 through T7 tables using a PC with 2 cores

t0 = time.clock()

con = mdb.connect('localhost', '<the user>', '<the password>', '<the database>');

with con:

    cur = con.cursor()

    cur.execute("""INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT0 as a,
      DORIANT1 as b,
      T0T1 as assn01
 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;
""")


   cur.execute("""INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT1 as a,
      DORIANT2 as b,
      T1T2 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;
""")

   cur.execute("""INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT2 as a,
      DORIANT3 as b,
      T2T3 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;

""")

   cur.execute("""INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT3 as a,
      DORIANT4 as b,
      T3T4 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;
""")

   cur.execute("""INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT4 as a,
      DORIANT5 as b,
      T4T5 as assn01
 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;
""")

   cur.execute("""INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT5 as a,
      DORIANT6 as b,
      T5T6 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;
""")

   cur.execute("""INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT6 as a,
      DORIANT7 as b,
      T6T7 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;
""")

   cur.execute("""INSERT INTO sdosOne

 SELECT assn01.subjperID,assn01.objperID

 FROM TAYLORT7 as a,
      DORIANT0 as b,
      T7T0 as assn01

 WHERE assn01.subjperID = a.perID AND
       assn01.objperID  = b.perID

 ORDER BY assn01.subjperID;
""")


print time.clock() - t0, "time to populate sdosOne"

con.close()
```

B. Generation of the RDF Triples

The generation of the RDF triples corresponding to the Person table is straightforward using the Python library RDFLIB. The main reason for including the code here is to highlight the fact that whereas the treatment of small datasets causes no problem during the serialization process, the creation of 640 million triples can exceed the capacity of the RDFLIB buffers that normally cache the graph being created.

The IDA team found that, if the RDF triples were generated in smaller chunks of 5 million triples each, the performance of the code was acceptable. This also helped when reviewing the files containing the triples, because most text editors also have difficulty manipulating files with sizes is in the gigabyte range. On the other hand, files whose size is at most 200 MB can still be manipulated without crashing most text editors.

1. Generation of the RDF Triples for the Person Data Set

```
# Copyright ©2017. The Institute for Defense Analyses (IDA).
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do
 # so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or substantial
# portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
# SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE
# BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH,
# THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.#

# Author: Francisco Loaiza, Ph.D., J.D.
#        Institute for Defense Analyses
#        Alexandria, Virginia, USA

#!/usr/bin/python
# -*- coding: utf-8 -*-

#generateTriples01forSDOS.py

from rdflib import URIRef, BNode, Literal
from rdflib import Namespace
from rdflib import Graph, ConjunctiveGraph
from rdflib.namespace import RDF,XSD

import MySQLdb as mdb

n = Namespace("http://usa/graphportal/resources/personnel#")
```

```
con = mdb.connect(‘localhost’, ‘<the user>‘, ‘<the password>‘, ‘<the database>‘)

with con:

cur = con.cursor()

# first loop runs 128 times

for k in range(128):

cg = ConjunctiveGraph()

g = Graph(cg.store, URIRef("http://usa/graphportal/demo01"))

outfile = "sdos_" + str(k) + ".ttl"

lowerlim = 1000000005 + (k * 1000000 * 5)
upperlim = 1000000005 + ((k + 1) * 1000000 * 5)

print lowerlim,upperlim

#  A query to fetch all the attributes of instances of Person and the perID
# of the instances they are associated with, as specified in the PersonAssociation table.

strSQL = "SELECT a.perID, a.fname,a.lname,a.dob,b.objperID FROM person as a, PersonAssociation as b WHERE
a.perID >= " + str(lowerlim) + " AND a.perID <= " + str(upperlim) + " AND a.perID = b.subjperID"

  cur.execute(strSQL)

  rows = cur.fetchall()

# Second loop serializes the dataset

  for row in rows:
    per01str = "http://usa/graphportal/resources/personnel#" + "PER" + str(row[0])
    subjPer = URIRef(per01str)
    per02str = "http://usa/graphportal/resources/personnel#" + "PER" + str(row[4])
    objPer = URIRef(per02str)
    fname = Literal(row[1], datatype=XSD.string)
    lname = Literal(row[2], datatype=XSD.string)
    dob = Literal(row[3], datatype=XSD.string)
    g.add( (subjPer, RDF.type, n.Person) )
    g.add((subjPer, n.fname, fname))
    g.add((subjPer, n.lname, lname))
    g.add((subjPer, n.dob, dob))
    g.add((subjPer, n.knows, objPer))

# The graph is written to file

print cg.serialize(destination=outfile,format="turtle")
```

2. Generation of Additional RDF Triples for the Person Data Set

An issue that the IDA team wanted to explore was whether the addition of triples that tag a resource with additional metadata would have a positive impact with regard to the retrieval of information stored in the RDF triple store. To test this hypothesis, the resources corresponding to instances of Person with fname equal to either "TAYLOR" or "DORIAN" were explicitly marked as being of type TaylorPerson or DorianPerson. In this phase of the study no attempt was made to create a complete set of statements in the TBox that would give the entire pedigree of the additional classes being used.

```
# Copyright ©2017. The Institute for Defense Analyses (IDA).
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do
 # so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or substantial
# portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
# SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE
# BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH,
# THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.#

# Author: Francisco Loaiza, Ph.D., J.D.
#        Institute for Defense Analyses
#        Alexandria, Virginia, USA

#!/usr/bin/python
# -*- coding: utf-8 -*-

# generateAdditionalTaylorOrDorianPersonType.py

from rdflib import URIRef, BNode, Literal
from rdflib import Namespace
from rdflib import Graph, ConjunctiveGraph
from rdflib.namespace import RDF,XSD

import MySQLdb as mdb

n = Namespace("http://usa/graphportal/resources/personnel#")

cg = ConjunctiveGraph()

g = Graph(cg.store, URIRef("http://usa/graphportal/demo01"))

con = mdb.connect('localhost', '<the user>', '<the password>', '<the database>')
```

```
with con:

    cur = con.cursor()

    cur.execute("""SELECT perID FROM person WHERE fname = 'TAYLOR' limit 80000""")

# change 'TAYLOR' to 'DORIAN' in the preceding query to generate the DorianPerson triples

    rows01 = cur.fetchall()

    for row in rows01:
        per01str = "http://usa/graphportal/resources/personnel#" + "PER" + str(row[0])
        subjPer = URIRef(per01str)
        g.add( (subjPer, RDF.type, n.TaylorPerson) )


print cg.serialize(destination='TaylorPerTypes.ttl',format="turtle")

# change the string for the destination to 'DorianPerTypes.ttl' for the DorianPerson triples
```

References

Although graph database technologies are young as compared to their relational database counterpart, a growing secondary literature is readily available. The following are some of the most recent offerings. The URLs point to Amazon.com where the items can be purchased.

Graph Databases: New Opportunities for Connected Data 2nd Edition, by Ian Robinson, Jim Webber, and Emil Eifrem, published by O'Reilly Media; 2 edition (July 9, 2015). https://www.amazon.com/Graph-Databases-Opportunities-Connected-Data/dp/1491930896/ref=cm_cr_arp_d_product_top?ie=UTF8

Neo4j in Action 1st Edition, by Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner, published by Manning Publications; 1 edition (December 21, 2014).
https://www.amazon.com/Neo4j-Action-Aleksa-Vukotic/dp/1617290769/ref=cm_cr_arp_d_product_top?ie=UTF8

Linked Data: Structured Data on the Web 1st Edition, by David Wood, Marsha Zaidman, Luke Ruth, and Michael Hausenblas, published by Manning Publications; 1 edition (January 24, 2014).
https://www.amazon.com/Linked-Data-David-Wood/dp/1617290394/ref=sr_1_2?s=books&ie=UTF8&qid=1474990762&sr=1-2

Linked Data for Libraries, Archives and Museums: How to Clean, Link and Publish your Metadata, by Seth van Hooland (Author), Ruben Verborgh, published by Amer Library Assn Editions (June 25, 2014).
https://www.amazon.com/Linked-Data-Libraries-Archives-Museums/dp/0838912516/ref=sr_1_1?s=books&ie=UTF8&qid=1474991823&sr=1-1

The Great Cloud Migration: Your Roadmap to Cloud Computing, Big Data and Linked Data, by Michael C. Daconta, published by Outskirts Press (October 11, 2013). https://www.amazon.com/Great-Cloud-Migration-Roadmap-Computing/dp/147872255X/ref=sr_1_1?s=books&ie=UTF8&qid=1474992107&sr=1-1

Information as Product, by Michael C. Daconta, published by Outskirts Press (October 21, 2007).
https://www.amazon.com/Information-as-Product-Michael-Daconta/dp/1432716549/ref=sr_1_2?s=books&ie=UTF8&qid=1474992167&sr=1-2

The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management 1st Edition, by Michael C. Daconta (Author), Leo J. Obrst (Author), Kevin T. Smith, published by Wiley; 1 edition (May 30, 2003). https://www.amazon.com/Semantic-Web-Services-Knowledge-Management/dp/0471432571/ref=sr_1_5?s=books&ie=UTF8&qid=1474992283&sr=1-5

Joe Celko's Complete Guide to NoSQL: What Every SQL Professional Needs to Know about Non-Relational Databases 1st Edition, by Joe Celko, published by Morgan Kaufmann; 1 edition (October 7, 2013). https://www.amazon.com/Celkos-Complete-Guide-NoSQL-Non-Relational-ebook/dp/B00G4N7HPS/ref=dp_kinw_strp_1

Acronyms and Abbreviations

AI	artificial intelligence
API	Application Program Interface
AQL	ArangoDB Query Language
AWS	Amazon Web Services
CRUD	Create, Retrieve, Update, Delete
CSV	Comma Separated Values
DDL	data definition language
DFS	Dynamic Force Structure
DML	data manipulation language
DoD	Department of Defense
DSE	DataStax Enterprise
ETL	Extraction, Transformation and Loading
GFM DI	Global Force Management Data Initiative
GUI	Graphic User Interface
IDA	Institute for Defense Analyses
IRC	Internet Relay Chat
JVM	Java virtual machine
LINQ	Language Integrated Query
MQL	Metaweb Query Language
MTO&E	Modified Table of Organization and Equipment
NoSQL	Not only Structured Query Language
OWL	Web Ontology Language
PII	personally identifiable information

RDF	Resource Description Framework
ReST	Representational State Transfer
SaaS	software as a service
SPARQL	A recursive acronym for SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
TB	Terabyte
TDA	Table of Distributions and Allowances
TSL	Trinity Specification Language
XML	Extensible Markup Language

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE (DD-MM-YY) 01-06-17	2. REPORT TYPE Final	3. DATES COVERED (From – To)
4. TITLE AND SUBTITLE Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure (DFS) Portal Implementation: Part 2, Technical Feasibility, Affordability, and Architecture Integration Options		5a. CONTRACT NUMBER HQ0034-14-D-0001 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBERS
6. AUTHOR(S) Francisco L. Loaiza-Lemos, Dale Visser		5d. PROJECT NUMBER BC-5-4277 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES Institute for Defense Analyses 4850 Mark Center Drive Alexandria, VA 22311-1882		8. PERFORMING ORGANIZATION REPORT NUMBER D-8516 H 2017-000309
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Mr. Bruce Haberkamp Army CIO/G-6 (SAIS-AOD) 5850 23rd Street, Bldg. 220, Ft. Belvoir, Virginia 20060-5832		10. SPONSOR'S / MONITOR'S ACRONYM SAIS-AOD 11. SPONSOR'S / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Project Leader: Francisco L. Loaiza-Lemos

14. ABSTRACT

This document describes an assessment of the maturity and applicability of graph database technology as a viable materiel solution that reflects the realities of legacy systems, and yet can deliver, for the planned DFS portal, effectively and efficiently the needed at-rest and in-motion force structure products. Specifically, the objective of the second deliverable is to determine the technical feasibility, affordability, and necessary architecture integration needed to include graph databases in the mix of technologies needed to support the planned Army DFS Portal. Rapid prototyping techniques have been applied, to stress the implementations of graph databases chosen for the study. Data collected during those activities will inform the determination of best-of-breed options. The assessments leverage the metrics elaborated in the initial phase of the study which have been already documented in the first deliverable.

15. SUBJECT TERMS

Graph database, Resource Description Framework (RDF), Not only SQL (NoSQL), relational database, RDF triple, RDF triple store, Application Program Interface (API), data interoperability, solution architecture, technology integration, data reuse, SPARQL, Structured Query Language (SQL), objective metric, scalability, query response time.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Mr. Bruce Haberkamp
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified	Unlimited	140	19b. TELEPHONE NUMBER (Include Area Code) 703-545-1464