INSTITUTE FOR DEFENSE ANALYSES

# 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


Francisco L. Loaiza-Lemos, *Project Leader*

Dale Visser


February 24, 2017


Approved for public release; distribution is unlimited.



IDA Document D-8345

Log: H 2017-000092

Copy



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

*The Institute for Defense Analyses is a non-profit corporation that operates three federally funded research and development centers to provide objective analyses of national security issues, particularly those requiring scientific and technical expertise, and conduct related research on other national challenges.*

**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**

The IDA team acknowledges the comments and suggestions provided by the task sponsor Mr. Bruce Haberkamp, and by David A. Wheeler

**For more information:**

Francisco L. Loaiza-Lemos, Project Leader
floaiza@ida.org, 703-845-6876

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

**Copyright Notice**

© 2017 Institute for Defense Analyses
4850 Mark Center Drive, Alexandria, Virginia 22311-1882 • (703) 845-2000.

This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227-7013 (a)(16) [Jun 2013].

INSTITUTE FOR DEFENSE ANALYSES

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

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 first deliverable under the project description and addresses the study's objective of assessing the maturity and applicability of graph database technology as a viable materiel solution that reflects legacy system realities, and yet can effectively and efficiently deliver the needed *at-rest* and *in-motion* force structure products for the planned DFS portal. Specifically, the objective of the first deliverable is to develop objective metrics and scenarios to test the ability of graph databases to represent and generate force structure data products.

## Background

The Department of Defense (DoD), through its Global Force Management Data Initiative (GFM DI), continues to work toward 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.[1] Currently, within the Army, the data needed to generate *at-rest* and *in-motion* force structure products resides in a large number of legacy systems that use relational database technology. In order to reuse these data sources to power the planned Army Dynamic Force Structure (DFS) Portal, the data must be in a format that supports easy manipulation and integration.

One approach is to recast the legacy source data into a format that is consistent with the GFM DI specifications before it is uploaded into the planned DFS portal.[2] An alternative that offers the potential for reducing the initial cost of converting to the intermediate XML-based GFM DI representation is to recast the legacy source data in the form of Resource Description Framework (RDF) triples, load them into a graph database, and then use the triples directly for the generation of the force structure products. To assess the maturity and applicability of the above-mentioned approach using graph database technology and RDF triples, one needs to define the types of scenarios in which the graph

---

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

[2] http://www.dtic.mil/whs/directives/corres/pdf/826003m_vol1.pdf

database approach would be applied, and the objective metrics to use when comparing it vis-à-vis a traditional approach based on relational database technology.

## Document Structure

This document is organized as follows:

1. Section 1 discusses the rationale for considering a graph database approach to the collection and manipulation of legacy source data, and the advantages and risks associated with using data lakes containing RDF triples versus a traditional approach using relational database technologies.
2. Section 2 documents scenarios likely to occur and discusses how the graph database approach can be employed in each of them.
3. Section 3 addresses the types of objective metrics to be employed when evaluating the maturity and applicability of the graph database approach.
4. Section 4 provides the current set of conclusions and recommendations for this phase of the study.
5. Appendix A contains an initial survey of proprietary and open source graph database implementations.
6. Appendix B contains quantitative data loading performance and scalability results for a subset of the graph databases being considered during this phase of the study. Additional testing is envisioned to continue during subsequent phases of the study.
7. Appendix C contains a number of Python and Java scripts used in the testing of a subset of the graph databases being assessed. The code is licensed for free reuse, and it is intended to help other groups in their evaluations.

## Scope

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 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.

Because of the sensitivity associated with many of the legacy data sources, some of which are classified, the examples used in the description of the graph database approach do not use actual legacy source data. Instead they are based on realistic but nevertheless

synthetic data. The IDA team felt that the use of notional data was appropriate because, as will be shown in the main body of this document, the transformation of data from a relational database representation into RDF triples that are stored in a graph database does not depend on the nature of the tables and columns implemented in the physical schema of the relational database. 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.

Due to the fact that essentially all the Army legacy data pertinent to the generation of force structure products resides in relational databases, the IDA team did not perform a comparison among all possible alternatives to the relational paradigm, but concentrated on a scenario involving the migration from the as-is relational database state of affairs to a to-be end state that uses graph database technologies.

Finally, although the values one would obtain for many of the objective metrics discussed in this document depend directly on the chosen hardware, and possibly the specialized software that often is employed to power a high-traffic portal, the IDA team did not explore this aspect in this phase of the study since that would require acquiring special purpose hardware and software, and 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 potential benefits and risks associated with the use of a graph database approach?
- Do any of the risks identified rise to the level of a "show stopper" for the approach under consideration?
- What are the most likely types of scenarios where the graph database approach would be used?
- Which objective metrics should be considered when assessing the maturity and applicability of the graph database approach as a possible materiel solution for the planned Army DFS portal?
- 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:

- A substantial number of offerings, both proprietary and open source, are available for graph database implementations. Some of these implementations are quite robust, have strong user base support, and have been in existence for quite some time.
- A graph database approach can work in all scenarios in which legacy source data must be transformed into a common representation that is easy to load and manipulate for the purpose of generating force structure products. However, the degree of effort is arguably the lowest where the legacy relational database can be programmatically accessed. Intermediate data dumps in the form of raw text files, XML instance documents, or CSV files add complexity to the approach. This in turn may also increase the risk.
- The objective metrics developed in this phase of the study provide a good road map for the evaluation of proprietary and open source graph database implementations. However, specialized testing with software and hardware specifically designed to power high-traffic portals should be conducted before the final determination regarding whether or not to adopt a graph database approach.

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

- Continue the evaluation of proprietary and open source graph database implementations, with particular emphasis on their performance for loading and retrieving data in a high traffic portal, as well as the robustness of their application program interfaces (APIs), specifically with respect to their support for commonly used scripting languages, e.g., Python, Java, etc.
- 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

1. Rationale for Considering a Graph Database Approach ........................................ 1-1
   A. Data Interoperability in the Context of Legacy Data Sources......................... 1-1
   B. Graph Databases Envisioned Approach ......................................................... 1-4
   C. Benefits and Risks .......................................................................................... 1-6
2. Scenarios for Graph Database Use ......................................................................... 2-1
   A. Summary of Scenarios...................................................................................... 2-1
3. Objective Evaluation Metrics .................................................................................. 3-1
   A. Types of Metrics Applicable to the Selection of a Graph Database................ 3-1
   B. Example Evaluation ......................................................................................... 3-3
4. Conclusions and Recommendations ....................................................................... 4-1
   A. Conclusions ..................................................................................................... 4-1
   B. Recommendations ........................................................................................... 4-1

Appendix A Survey of Graph Database Implementations............................................ A-1
Appendix B Sample Quantitative Results for Scalability of Data Loading....................B-1
Appendix C Sample Code Used for Testing Scalability of Data Loading .....................C-1
References.......................................................................................................................R-1
Acronyms and Abbreviations ...................................................................................... AA-1

**Figures and Tables**

Figure 1-1. The Inherent Lack of Data Interoperability of the Physical Schemata in Relational Databases ........................................................................................ 1-1
Figure 1-2. The Inherent Data Interoperability of Graph Databases Storing RDF Triples ............................................................................................................... 1-3
Figure 1-3. An Approach for Moving Legacy Source Data to the Target Army Force Structure Data Lake ........................................................................................ 1-4
Figure 1-4. A Service-Oriented Architecture Based on a Graph Database Implementation ................................................................................................ 1-5
Figure 1-5. Key Implementation Steps for a Graph Database Solution Architecture ..... 1-6
Figure 2-1. Summary of Possible Scenarios Regarding Conversion and Loading of Legacy Source Data into a Target Graph Database ........................................... 2-1

Table 1-1. Benefits and Risks for Key Implementation Steps........................................ 1-7
Table 3-1. Summary of Applicable Objective Metrics .................................................. 3-1
Table 3-3. Notional Evaluation Example for Graph Database Implementations ........... 3-4

# 1. Rationale for Considering a Graph Database Approach

## A. Data Interoperability in the Context of Legacy Data Sources

Figure 1-1 provides a quick summarization of the data interoperability problem in the context of relational databases.

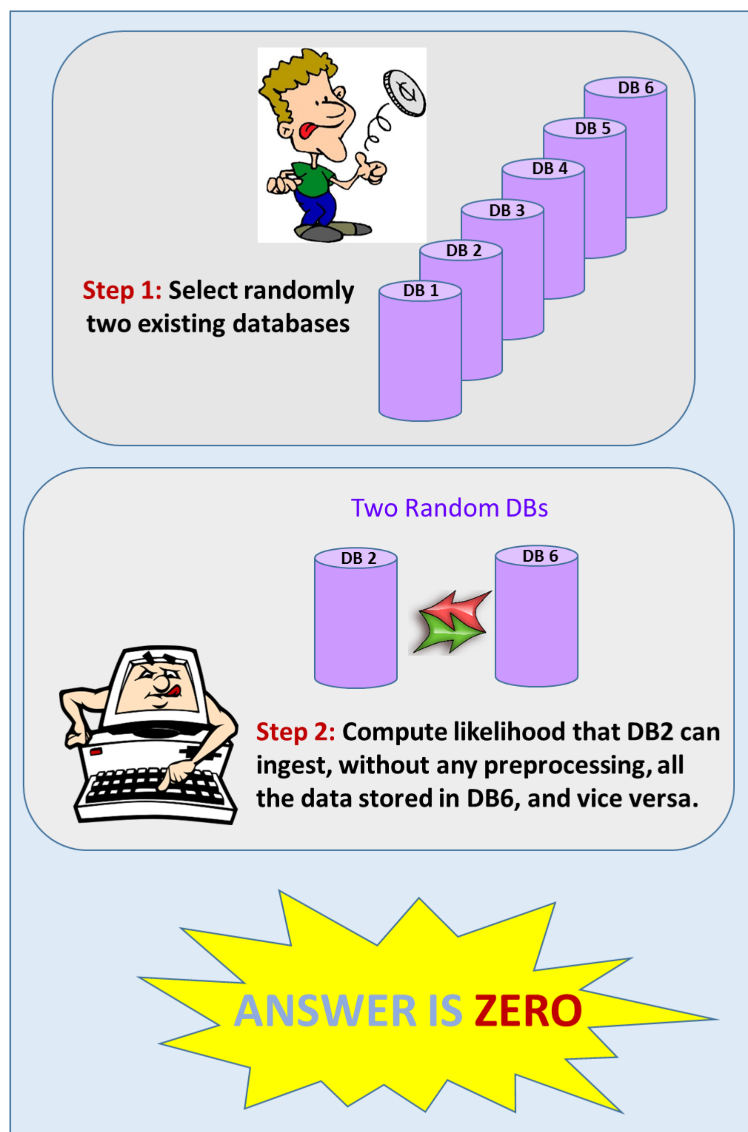


**Figure 1-1. The Inherent Lack of Data Interoperability of the Physical Schemata in Relational Databases**

The cartoon in Figure 1-1 highlights the fact that relational databases implement physical schemata, which are seldom derived from a common information model. Because of this, the concepts that the tables and columns capture do not use the same labels or the same syntactic constraints. A column for the name of a piece of materiel may be specified as a character string with maximum length of 50 characters in one system and of 255 characters in another. Names of individuals may be broken into three or more components, such as first name, middle name, last name, while in another relational database, a single column is expected to capture the entire name string without regard to how the components may be ordered.

Even when there is agreement on the structuring of a person's name as composed of three subelements, the labels may be **fName**, **mName**, and **lName** in one implementation, **firstNm**, **middleNm**, and **lastNm** in another, and **firstName**, **middleName**, and **lastName** in a third. And each implementation may have, as noted above, different constraints regarding its maximum length.

Given this state of complete freedom when choosing labels and modeling concepts with varying degrees of decomposition and syntactic constraints, it is arguably fair to state that if one were to pick two databases at random, the likelihood of being able to take data from any table in the source database and insert it into any other of the available tables within the target database would be essentially zero. This is true even when dealing with databases that support the same functional domain, unless of course an effort has been made to harmonize their logical schemata before they are implemented.

A completely different situation arises in the case of data stores that use a non-relational paradigm. Specifically, when one considers a graph database designed to store Resource Description Framework (RDF) triples, which is one of the growing number of alternatives to the relational database paradigm that are generally grouped under the label of NoSQL databases, the likelihood of being able to take a set of RDF triples from one randomly selected graph database and load it successfully into another graph database is 100%.

This surprising outcome has a rather simple explanation. When data is represented in terms of RDF triples, its form is always the same, namely, **<subject><predicate><object>**. So it really does not matter whether the source table is intended to store materiel data or planning data, or something else. Each field in each of the tables in a relational database will be rendered in the form of a triple in a graph database. And as a result, the source graph database will have a set of triples and the target graph database will be designed to load and store anything that conforms to the **<subject><predicate><object>** format.

Figure 1-2 summarizes the previous discussion for No-SQL solutions such as graph databases storing RDF triples – also referred to as *RDF triple stores* in the literature. Here, contrary to the case of the randomly selected relational databases discussed before, it is

arguably fair to state that if one were to pick two RDF triple stores at random, the likelihood of being able to take a set of RDF triples from one of them and insert it in the target graph database would be 100%. And as noted above, this is true even when the graph databases are storing RDF triples from entirely different functional domains.[3]

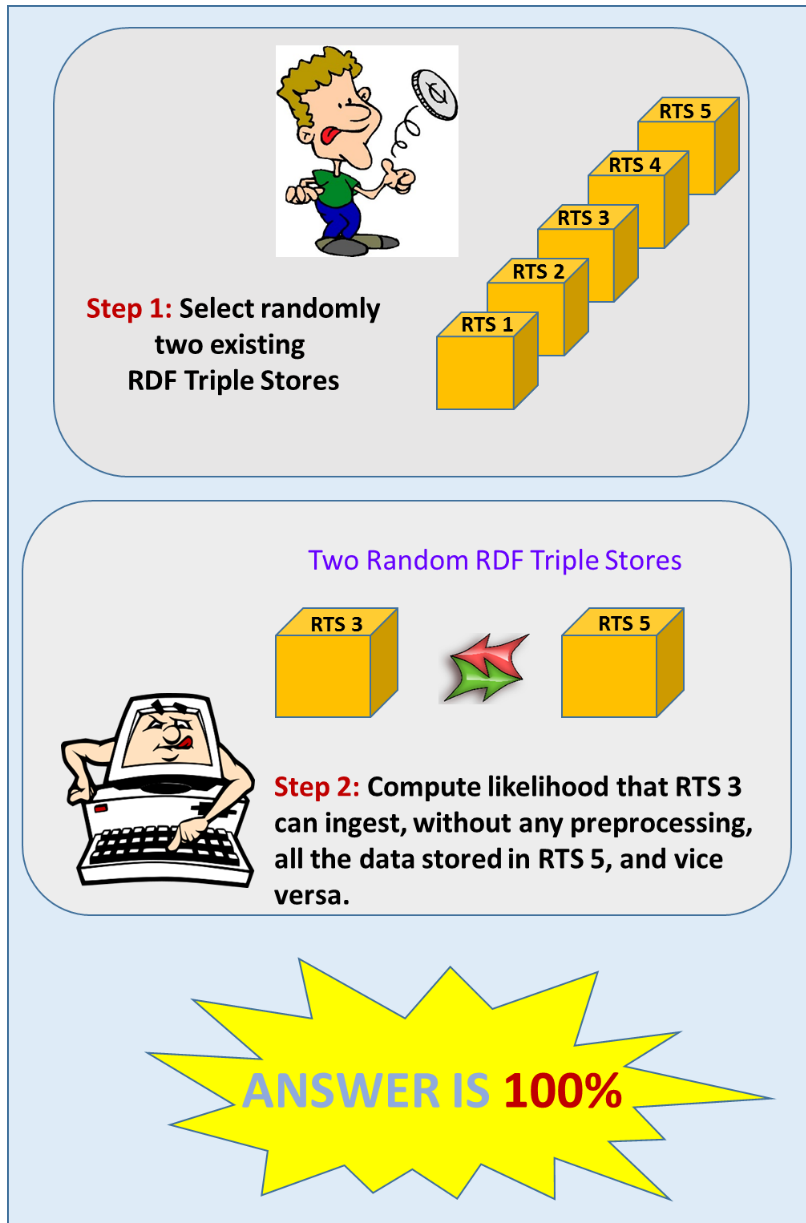


**Figure 1-2. The Inherent Data Interoperability of Graph Databases Storing RDF Triples**

---

[3] The adoption of a data representation based on RDF triples, in combination with graph databases for their storage, eliminates the low-level data interoperability problem that plagues the relational database implementations -- specifically the costly and time-consuming extraction, transformation, and loading (ETL) cycle that must occur every time that one attempts to reuse data resident in relational legacy systems. It does not automatically resolve all semantic interoperability problems.

## B. Graph Databases Envisioned Approach

The implementation of a graph database approach to power the planned Army DFS portal would require the conversion of the legacy source data into a format that can be loaded into a graph database. Figure 1-3 depicts one of the alternatives. Those legacy source relational databases whose data is considered necessary for the production of force structure products would convert their data into RDF triples (notionally depicted as sticks attached to the source databases). The resulting abstract graphs would then be loaded into a target graph database, which would constitute a RDF data lake for Army force structure data.

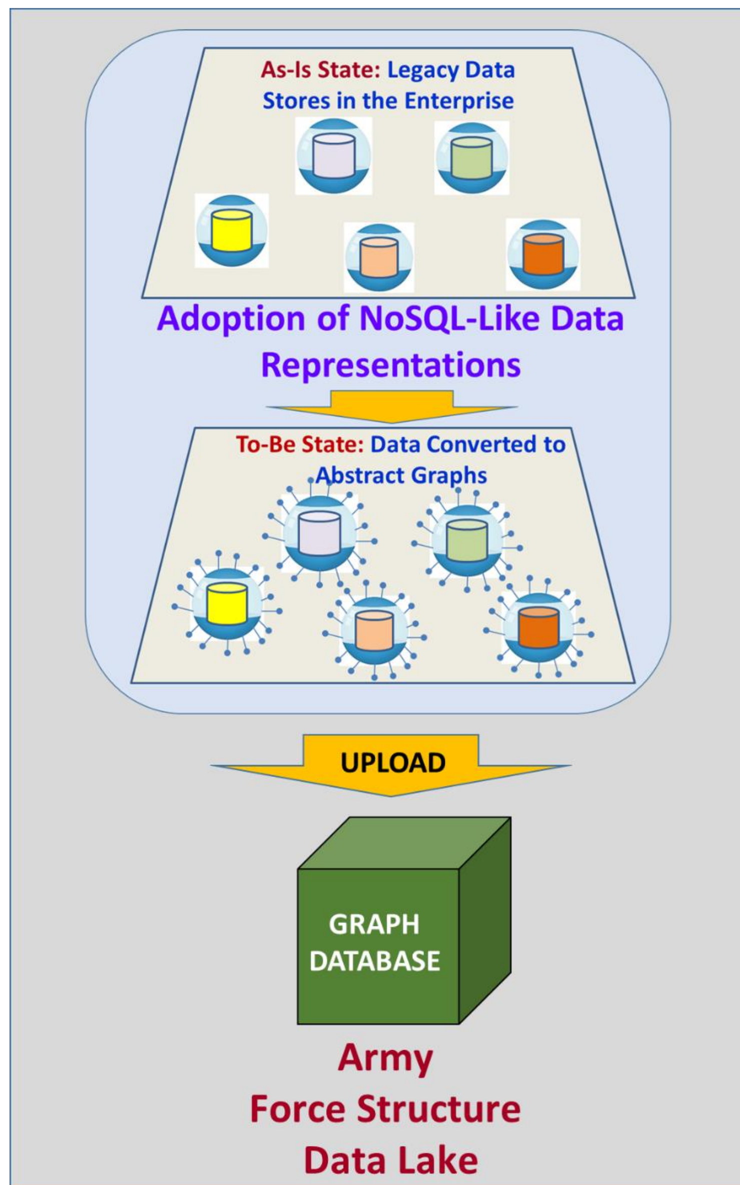


**Figure 1-3. An Approach for Moving Legacy Source Data to the Target Army Force Structure Data Lake**

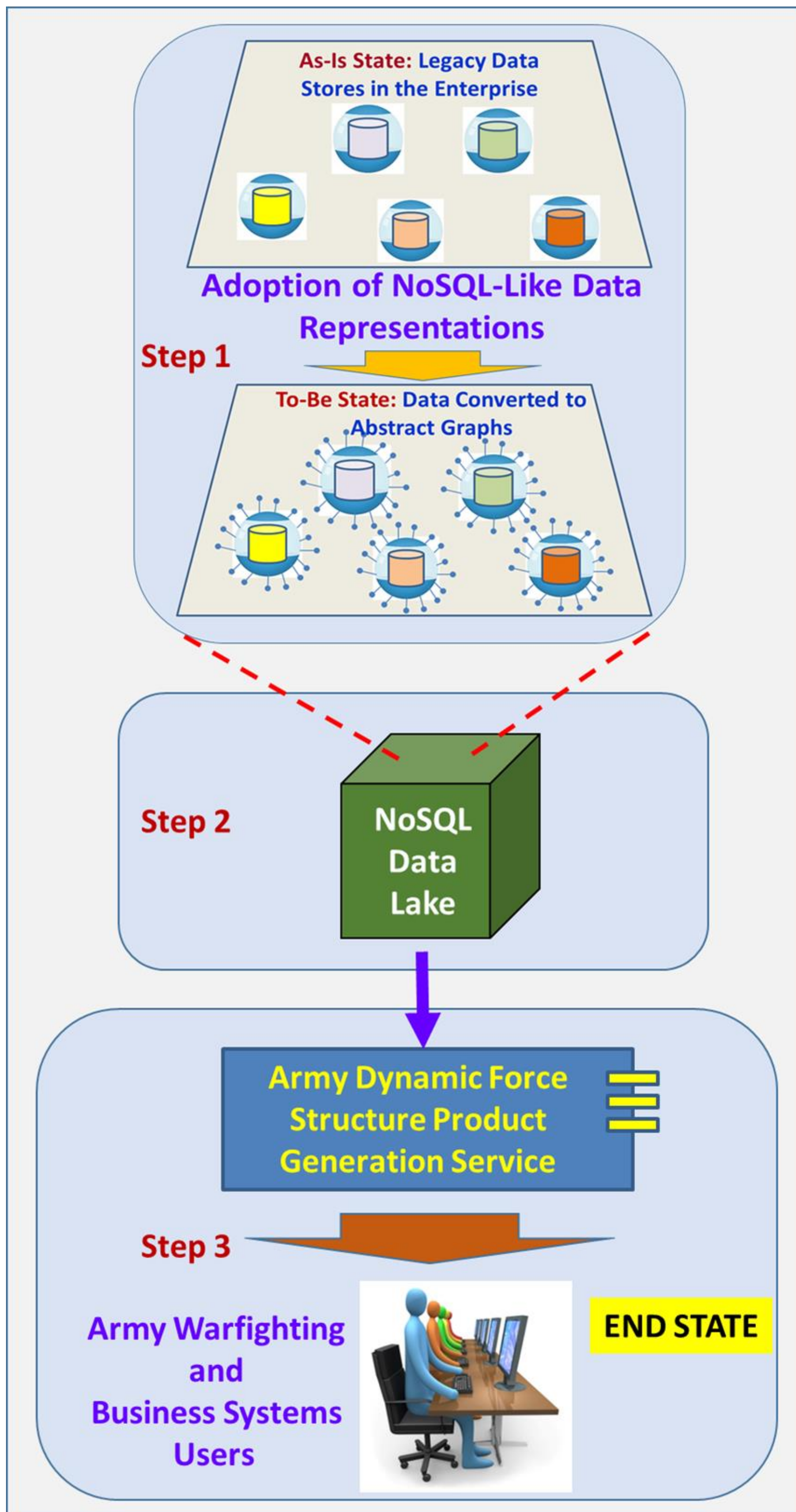


**Figure 1-4. A Service-Oriented Architecture Based on a Graph Database Implementation**

Once a repository of all source data needed to generate force structure products is in place, one could proceed to implement a service-oriented architecture solution as shown schematically in Figure 1-4. In this notional end state, users from the warfighting and the business mission areas would be able to obtain the necessary force structure products from a DFS portal implementation that uses a graph database as its backend data store.

## C. Benefits and Risks

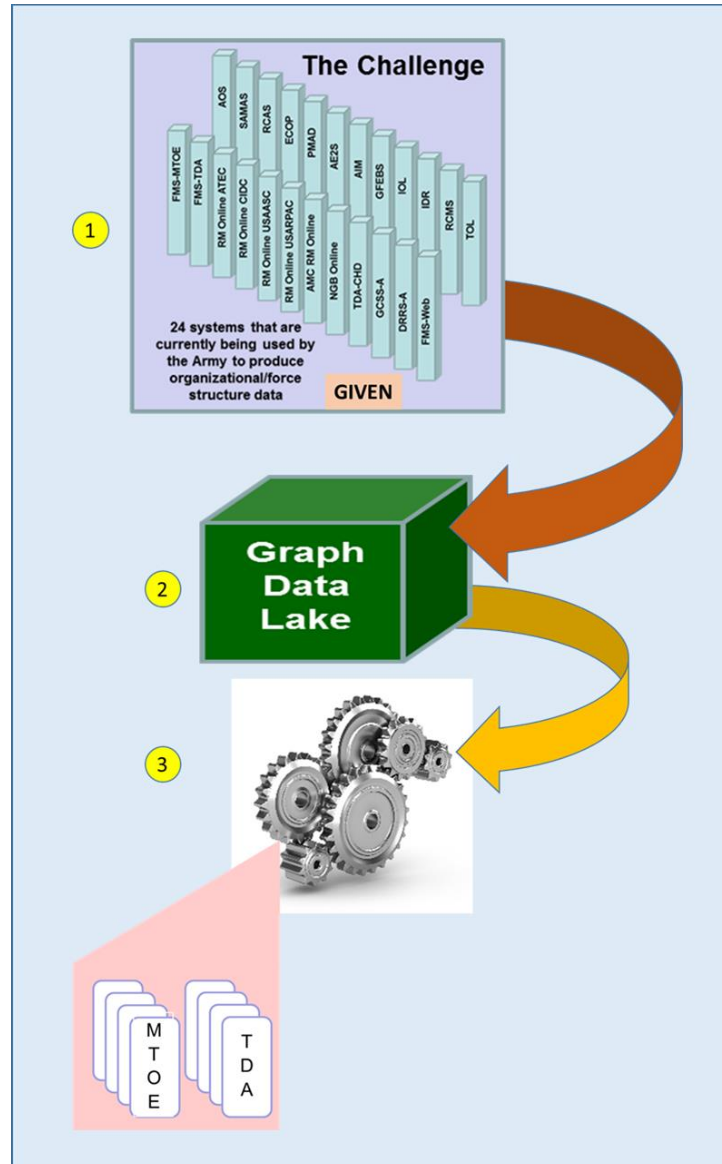


**Figure 1-5. Key Implementation Steps for a Graph Database Solution Architecture**

Figure 1-5 highlights the three key implementation steps that adopting a solution architecture using graph databases for the planned Army DFS portal would entail, namely, the conversion and loading of the legacy source data (Step 1); the update and maintenance

of the data once it is collected in the graph data lake (Step 2); and the actual generation of force structure products such as modified tables of organization and equipment (MTO&E), and tables of distributions and allowances (TDAs). Each step has benefits, but also risks that need to be considered. We provide a summary of the key considerations for each one of the steps (see Table 1-1).

**Table 1-1. Benefits and Risks for Key Implementation Steps**

| Benefits | Risks |
|---|---|
| **Step 1: Migration of Legacy Source Data** | |
| • Conversion of legacy source data from relational databases to RDF triples is independent of physical schema specifications, which saves money and effort.[4]<br>• Coding required to implement either generation of intermediate files for subsequent upload, or direct injection of the legacy source data into the target graph database uses well-documented libraries; the associated learning curve is not steep.<br>• Loading of intermediate files into target graph database can leverage vendor tools, and thus would not require additional coding. | • API support for scripting languages, such as Python or Java, is not uniformly robust. Acceptability of a process based on the programmatic execution of structured query language (SQL) SELECT queries against a legacy relational database followed by the conversion into triples and their injection into the target graph database in a manner that comports with the operational concept of the planned DFS portal has to be decided on a case-by-case basis.<br>• Availability and sufficiency of libraries to connect, add and remove RDF triples programmatically within a graph database varies from implementation to implementation and has to be decided on a case-by-case basis.<br>• Vendor tools that allow manual upload of intermediate files containing RDF triples vary in performance and ease of use. In some cases, file size may be an issue, requiring the partitioning of source data files into small chunks, which increases time and cost to upload data from a source relational database. |
| **Step 2: Maintenance and Update of Source Data in Graph Database Repositories** | |
| **Benefits** | **Risks** |
| • A standardized language is already supported in most graph databases that implement Create, Retrieve, Update, Delete (CRUD) operations, namely, SPARQL.[5] This means that if a graph database has a good implementation of the | • SPARQL is not as robust as SQL. It does not support all the advanced features provided in SQL. There is currently no possibility of writing procedures similar to those available in MSSQL Server,[6] Oracle, [7] or PostgreSQL.[8] Complex manipulations that would be |

[4] The cost and time associated with the generation of new code needed to use the data in the form of RDF triples need to be accounted for in the overall evaluation; however this cost and time investment would apply equally to any scenario for which the end state is to be based on a technology other than graph databases.

[5] The name is a recursive acronym for "SPARQL Protocol and RDF Query Language."

[6] Transact-SQL (T-SQL).

[7] Procedural Language/SQL (PL/SQL).

[8] Procedural Language/pgSQL (PL/pgSQL)

| Step 1: Migration of Legacy Source Data | |
|---|---|
| **Benefits** | **Risks** |
| SPARQL language, then SPARQL queries from any other graph database can be reused. This saves time and money and reduces dependency on a specific vendor. | implementable in an SQL script may require additional in-house coding using either Java or some other language supported by the specific graph database implementation.<br>• Expertise with SPARQL and the associated user knowledge base is less robust than with the SQL procedural languages. The need for experienced SPARQL programmers when the candidate pool is already small to begin with, may increase operational costs.<br>• Depending on the graph database the CRUD implementation may not be mature enough, which means that after multiple insert, delete, and update operations, the graph database may experience notable performance degradation. In other words, depending on the choice made, additional maintenance steps may be required. |
| **Step 3: Generation of Force Structure Products** | |
| **Benefits** | **Risks** |
| • The structure and content of typical force structure products, such as modified tables of organization and equipment (MTO&E), and tables of distributions and allowances (TDAs), is well established. | • Depending of the complexity of the force structure products, additional in-house coding may be required to generate them if in a single pass the SPARQL queries cannot retrieve the data in the form needed.<br>• Depending on the graph database implementation, there may not be programmatic support for scripting languages such as Python or any of the other currently used scripting languages, but only for Java. This should be taken into consideration if the required expertise level in Java is high. |
| • Generally speaking, the adoption of graph database technology for the management of all information resources at the Army enterprise level would represent a massive paradigm shift whose key transformational characteristic would be the complete elimination of the costly and extremely laborious extraction, transformation and loading (ETL) cycle that is associated with database-to-database migration and consolidation. | • Whereas in relational databases it is by now straightforward to restrict access to either tables or even specific columns in a table, this is not something that is readily supported out of the box in graph databases. The implementation of access controls may not be trivial and will require in-house coding. |
| • Use of RDF triples to represent force structure data makes immediately possible the leveraging of associated semantic technologies, such as automated reasoning and inferencing. This in turn could position the Army enterprise for the expeditious adoption of technologies related to artificial intelligence (AI). | • All these technologies require in-house expertise which may not be readily available, or planned for as part of the modernization efforts for the Army enterprise. Without such anticipatory planning the advantages may not materialize. |

# 2. Scenarios for Graph Database Use

## A. Summary of Scenarios

Figure 2-1 shows schematically the scenarios that are likely to apply when moving legacy source data resident in relational databases into a target graph database.

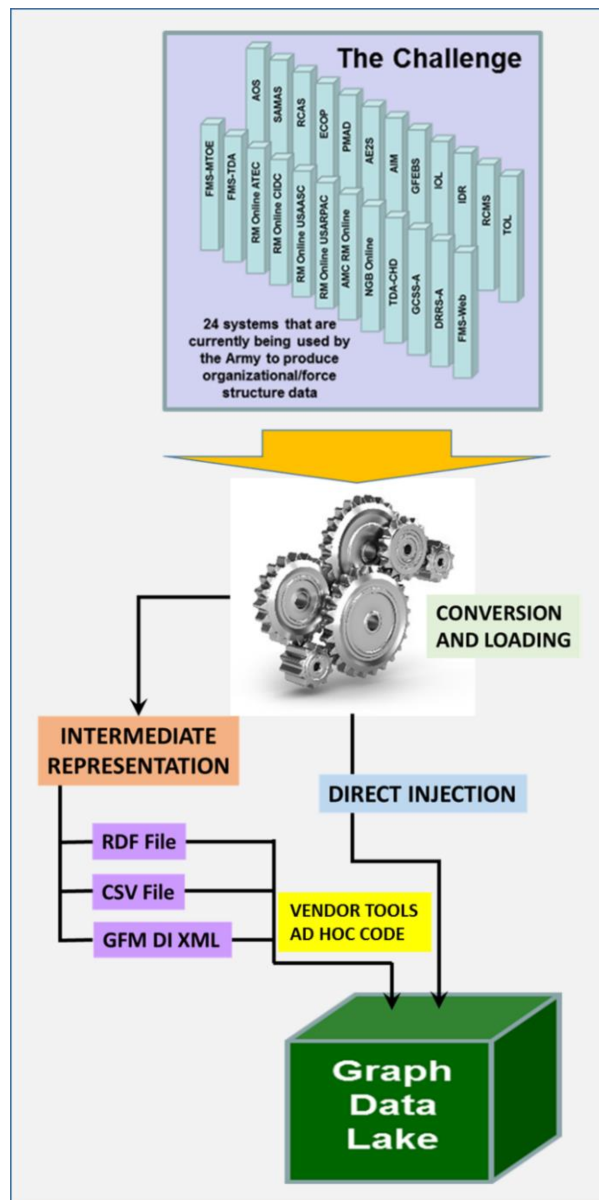


**Figure 2-1. Summary of Possible Scenarios Regarding Conversion and Loading of Legacy Source Data into a Target Graph Database**

As shown, the set of situations breaks into two main categories. On the left side of Figure 2-1, is the category corresponding to the production of *intermediate representations*, which are then subsequently loaded into the target database. Based on the review of the proprietary and open source graph database implementations (see Annex A for more details), some of them are capable of ingesting files containing RDF triples in any of the standard serializations currently in use, such as RDF/XML, Turtle, N3, etc. In other cases, file upload expects the data to be prepared in commonly used, but nevertheless specific formats, such as CSV, which then can be processed using vendor-provided tools. This is the case, for example, for the NEO4J graph database.

Finally, there are already Army information systems that participate in the GFM DI process and serve as sources for the Army Organization Server. In those cases, the data needed may already be formatted in GFM DI conformant XML. This scenario would be, from the point of view of the production of the force structure products, the easiest since the GFM DI structures already support the definition of MTOEs, TDAs, etc. In this case, however, there would be a need to write code to parse the XML files, convert their content to RDF/XML, and then either use vendor tools or attempt to use existing APIs to inject the triples directly into the target graph database.

The simplest scenario is shown on the right side of Figure 2-1 under *direct injection*. Here, the assumption is that both the legacy source relational database and the target graph database can be manipulated programmatically. Most relational database implementations, proprietary and open source, have very robust support for one or more programming languages. In most cases Java libraries support the execution of SQL SELECT queries. A growing number of open source relational databases such as MySQL and PostgreSQL also have good libraries for Python. MSSQL Server traditionally supports only .NET, so there may be a need to mix languages, but most scripting languages can execute compiled code written in a different language (i.e., a Python script calling a compiled .NET C# module that talks to MSSQL Server). It should also be noted that there has been some movement within Microsoft to provide a native implementation of MSSSQL Server in Linux platforms.[9]

From the previous discussion it is, therefore, fair to say that there does not seem to be any technical barrier to using a graph database as a repository for the legacy source data needed to generate force structure products.

---

[9] https://www.microsoft.com/en-us/sql-server/sql-server-vnext-including-Linux

# 3. Objective Evaluation Metrics

## A. Types of Metrics Applicable to the Selection of a Graph Database

Table 3-1 presents a summarization of the applicable objective metrics that could be applied to determine whether or not a given graph database implementation should be considered as a viable materiel solution for the planned Army DFS portal. It should be noted that these metrics will need to be considered both in the aggregate and in light of the envisioned concept of operations for the portal. Some features offered by a graph database may make it more appealing than a competing option if the subpar performance according to one of the metrics is associated with an activity that according to the concept of operations only happens very rarely or just once. For example, spending a day performing the initial data load may be acceptable if this will happen only once and the user friendliness of the GUI offered by the graph database under consideration facilitates the day-to-day operations, as compared to very fast upload times offered by an alternative graph database that offers only a very primitive GUI that makes the management of the graph database extremely laborious.

**Table 3-1. Summary of Applicable Objective Metrics**

| **Metric 1: Ability to Scale up during Upload Operations** | |
|---|---|
| **Measurement** | **Discussion** |
| • Time that it takes for a given graph database to load a given number of RDF triples.<br>• For small datasets the unit of measure is in milliseconds (ms).<br>• For large datasets the unit of measure may be in minutes or hours. | • Use of this metric should be combined with the actual concept of operations before making a final determination regarding the suitability of a graph database implementation as the choice for the planned Army DFS portal.<br>• For example, if the expectation is that large data sets generated out of legacy source relational databases are to be expected just for the initial load into the graph database repository, and afterwards, either no more datasets are expected from the source legacy relational databases, or only small datasets, representing updates or deletions, will need to be processed, then the cost in terms of time spent to effectuate the load is something that may be amortized over time, and other characteristics of the graph database may be more relevant, e.g., very user-friendly interface, clean implementation of the SPARQL specifications, robust APIs for one or more scripting languages to support programmatic manipulation of graph database content, which may be needed to support the generation of force structure products in multiple formats. |

| Metric 2: Response Time for Data Retrieval | |
|---|---|
| **Measurement** | **Discussion** |
| • Time that it takes for a given graph database to execute a query to retrieve RDF triples.<br>• For interactive applications the rule of thumb is that the retrieval times, plus the time it takes to refresh the graphic user interface (GUI) should never exceed 1 second. | • The expectation is that simple queries should have response times that are essentially constant, irrespective of the actual number of RDF triples stored in the graph database.<br>• For very large datasets (i.e., 1TB or more), the response for simple queries tends to be on the order of 2 seconds.[10]<br>• However, either partitioning the datasets into multiple graph database instances and then federating them or optimizing the hardware may be adequate to avoid an unacceptable performance degradation in the query response time. |
| **Metric 3: Provided Support Tools** | |
| **Measurement** | **Discussion** |
| • Availability (yes/no) and percentage of operations pertaining to graph database maintenance and update covered by the tools offered in a given implementation. These operations generally can be run from the command line. | • Although implementing GUIs as web applications is the current trend and it facilitates the management of a graph database, there are occasions where support for said GUIs may not be readily available, e.g., when working in a trimmed-down server configuration with no desktop support. In these cases, the availability of tools that can be run from the command line is essential. |
| **Metric 4: Graphic User Interface (GUI) Usability** | |
| **Measurement** | **Discussion** |
| • Availability (yes/no) of GUI for writing and executing queries in a widely supported query language,<br>• Availability (yes/no) for namespace management,<br>• Availability (yes/no) of GUI feature for paging through large query results,<br>• Availability (yes/no) of GUI feature for saving queries for later reuse,<br>• Availability (yes/no) of GUI feature for creating federated graphs from existing ones saved,<br>• Availability (yes/no) of APIs to third-party high-performance applications for the storage/persistence of graphs. | • A user-friendly interface to write and execute queries in SPARQL is essential for testing and development work. This includes management of namespaces.<br>• Similarly, the ability to save the queries and to examine the potentially very large sets that a query returns in chunks whose size is controlled by the user is a very desirable feature in a GUI.<br>• Setting up federated graphs via an intuitive GUI further enhances the value of this capability whenever it is offered by the specific graph database.<br>• Lastly, availability of multiple choices for persistence is essential to prevent accidental data loss, and for ensuring good performance when dealing with large data sets. |

[10] https://www.w3.org/wiki/LargeTripleStores; https://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VOSArticleLUBMBenchmark

| Metric 5: Code Reusability | |
|---|---|
| **Measurement** | **Discussion** |
| • Percentage of code developed in a different environment that can be reused in a given target graph database. | • In complex applications, over time, the volume of code developed to support its functionality becomes both large and quite valuable. Being able to reuse previously developed code in a different graph database implementation would eliminate a possible hurdle when considering changing the backend graph database to improve overall performance or to take advantage of novel features offered by another implementation. |
| **Metric 6: Standards Implementation** | |
| **Measurement** | **Discussion** |
| • Number of standards supported and number of components in those standards implemented. | • Graph databases that do not support widely used standards, such as SPARQL for queries or a Java API for programmatic manipulation, may require potentially a higher investment to achieve optimal operational performance, because the knowledge base is smaller, which in turn may make the learning curve steeper. In addition, any investment in gaining familiarity in the operation of a graph database that does not implement industry standards is non-transferable, and reduces flexibility when considering migration to a different vendor. (See also the comments in Metric 5 above). |
| **Metric 7: Robust Support for Persistence Options** | |
| **Measurement** | **Discussion** |
| • Number of types of persistence options offered and of choices for backend storage. | • Although high-performance input/output can be achieved with straight text storage to disc or using binary encoding, the ability to store data using a relational database engine as backend storage is quite desirable because of its stability, scalability, and extensive knowledge base available. Thus, graph databases that support the use of MongoDB, PostgreSQL, MySQL, etc., are better suited implementations at the enterprise-level than those that do not. |

## B. Example Evaluation

As noted in the preceding section, the application of the objective metrics for the purpose of selecting a particular graph database implementation to act as the backend data store for the future DFS portal should take into consideration its concept of operations.

In this section it is assumed that large data sets from the 24 legacy relational databases (see Figure 2-1 above) will be ingested only once, and all subsequent data additions will be comparatively small.

**Table 3-2. Notional Evaluation Example for Graph Database Implementations**

| | GRAPH DATABASE IMPLEMENTATION | |
|---|---|---|
| **METRICS** | **AllegroGraph** | **Neo4j** |
| Upload Performance | 10 | 10 |
| Query Response Time (Average) | 10 | 10 |
| Support Tools | 10 | 3 |
| GUI Usability | 10 | 10 |
| Code Reusability | 10 | 3 |
| Standards Implementation | 10 | 3 |
| Persistence Options | 10 | 3 |
| | **70** | **42** |

Table 3-2 shows a notional evaluation example that uses the **AllegroGraph** and the **Neo4j** implementations to demonstrate the use of the objective metrics discussed in the preceding chapter. The model assigns a maximum value of 10 to each of the metrics to indicate that the criterion is perfectly satisfied and a value of 0 if the implementation fails to satisfy the metric entirely. The descriptions of these two graph database implementations contained in Appendix A have been used for the assignment of the values shown.

For the **Upload Performance** both AllegroGraph and Neo4j are given a 10 because both have very robust data ingestion capabilities, and, under the assumed operational concept, their capabilities to ingest legacy source data would be well within the expected range.

Similarly, for **Query Response Time (Average)** and **GUI Usability** both graph database implementations satisfy the criteria and receive a 10. However, with regard to **Support Tools**, **Code Reusability**, **Standards Implementation**, and **Persistence Options** there is a marked difference. AllegroGraph has a very mature set of APIs that enable the user to develop scripts to handle most of the workflow procedures. In addition, AllegroGraph has a very clean implementation of SPARQL, which is the most common query language for RDF triple stores, and it also can use a very large number of backend applications, including high-power engines such as MongoDB. By contrast Neo4j, being the more recent implementation, does not offer the same robustness with regard to APIs. It also uses its own query language Cypher, as opposed to SPARQL, and does not offer as many options for integration with backend data storage applications.

Computing the totals, it appears that for handling RDF triples in an environment that potentially may require backend storage applications capable of handling very large volumes, and where code reuse and robust API support is deemed to be important, the AllegroGraph implementation is preferable to Neo4j.

# 4. Conclusions and Recommendations

## A. Conclusions

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

- A substantial number of offerings, both proprietary and open source, are available for graph database implementations. Some of these implementations are quite robust, have strong user base support, and have been in existence for quite some time.
- A graph database approach can work in all scenarios in which legacy source data must be transformed into a common representation that is easy to load and manipulate for the purpose of generating force structure products. However, the degree of effort is arguably the lowest where the legacy relational database can be programmatically accessed. Intermediate data dumps in the form of raw text files, XML instance documents, or CSV files add complexity to the approach. This in turn may also increase the risk.
- The objective metrics developed in this phase of the study provide a good road map for the evaluation of proprietary and open source graph database implementations. However, specialized testing with software and hardware specifically designed to power high-traffic portals should be conducted before the final determination regarding whether or not to adopt a graph database approach.

## B. Recommendations

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

- Continue the evaluation of proprietary and open source graph database implementations, with particular emphasis on their performance for loading and retrieving data in a high traffic portal, as well as the robustness of their application program interfaces (APIs), specifically with respect to their support for commonly used scripting languages, e.g., Python, Java, etc.
- 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

In the last decade, the popularity of graph databases has continued to increase in proportion to the growth in demand for ways to manipulate data that does not necessarily need to be cast in the rigid schemata of relational databases. Whereas a few years ago only a handful of choices was available, there is now a substantial number of offerings, in both the proprietary and the open source world.[11]

This appendix provides a brief overview of some of those offerings. The main purpose 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. Among the most important characteristics to focus on are 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

Graph databases employ a directed property graph data model whereby information is represented by nodes, edges, and properties. Nodes are entities that one wishes to keep track of, analogous to a row or relation in a relational database. Edges represent relationships between nodes, and in a graphical visualization are the lines that connect any two nodes. Properties are metadata pertaining to nodes, though in many instances, graph database products also allow edges to have property metadata.

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

RDF databases are also known as RDF stores or triple stores, and are a kind of graph database. The basic unit of data storage is a statement triple, analogous in natural language

---

[11] http://www.predictiveanalyticstoday.com/top-graph-databases/

to a declarative statement, composed of a subject (node), a predicate (edge) and an object (node). 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.[12]

The following sections provide descriptions for some of the most commonly used graph databases available today.

## 3. Eclipse RDF4J

RDF4J is a mature, open source RDF triple store implemented in Java. It was originally developed under the name Open RDF Sesame. It runs in both Linux and Windows and has a very user-friendly interface for managing the creation, and deletion of repositories, as well as file uploads and exports. It also support graphs federation, and has command-line tools that permit the management of repositories without the need for the web-based interface.

### a. API Support

#### i. Java

In the RDF+Java world, RDF4J defines one of two de facto standard API sets, the other de facto standard being Apache Jena, which is discussed in the next section. RDF4J has four publicly exposed Java APIs:

- RDF Model – which is defined as "the basic building blocks for manipulating RDF data in Java."[13]
- Repository – which is defined as the "central access point for RDF4J-compatible RDF databases (a.k.a. triple stores), as well as for SPARQL endpoints."[14] This includes the ability to query repositories with SPARQL or SeRQL languages.

[12] https://www.w3.org/TR/sparql11-query/

[13] http://docs.rdf4j.org/programming/#_the_rdf_model_api

[14] http://docs.rdf4j.org/programming/#_the_repository_api

- Rio (“RDF I/O”) – which is a serialization/deserialization capability for RDF data using several standard serialization formats.
- SAIL – which exists at a layer below the above APIs and is a “collection of interfaces designed for low-level transactional access to RDF data.”[15] SAIL is the typical glue point to using RDF4J with arbitrary storage backends.

In addition, there are some useful implementations, including:

- A repository optimized for in-memory usage,
- A “native” disk-based repository,
- A federation repository capable of exposing the union of separate repositories as a single repository,
- An RDF Schema inferencing capability.

By itself, RDF4J is not designed to scale to very large datasets, but scalable proprietary products exist that utilize RDF4J’s API and core utilities. Ontotext GraphDB and Stardog, described later in this document, are two such examples.

#### ii. Representational State Transfer (ReST)

The RDF4J ReST application program interface (API) is compliant with the W3C SPARQL 1.1 Protocol Recommendation. In addition, it provides access to a variety of RDF4J server resources. See http://docs.rdf4j.org/rest-api/ for details.

#### iii. Documentation and Support

Manuals, developer tutorials, and API documentation are all available at http://docs.rdf4j.org/. Being a fully open source project, RDF4J offers no paid support option. However, e-mail lists, an IRC channel, as well as a bug tracker are all documented at http://rdf4j.org/support/.

#### iv. Documentation and Support

- HTTP: Compiled libraries, Executables, Source;
- Maven : Download just the libraries you need for your project[16];
- Git: Source code access.

---

[15] http://docs.rdf4j.org/sail/

[16] Maven is a very popular tool among Java developers for managing library dependencies, whereby all dependencies (compile-time, run-time, testing only, etc.) are specified in configuration files and are pulled from a public repository (or optionally an organization internal repository) as needed when software is built from source code.

## 4. Apache Jena

"A free and open source Java framework for building Semantic Web and Linked Data applications."[17] In the RDF+Java world, Jena is the main competitor to RDF4J.

### a. APIs

It has its own set of competing "standard" Java APIs:

- RDF API – Equivalent to RDF4J RDF Model + Rio. (part of jena-core module),
- ARQ – SPARQL query engine (jena-arq module),
- Ontology API – RDFS and OWL support (part of jena-core module),
- Inference API – Includes OWL and RDFS reasoners; supports creating own inference rules.

In addition, there are some useful implementations:

- TDB – a triple store implementation that supports the other Java APIs,
- Fuseki – "expose your triples as a SPARQL endpoint accessible over HTTP."[18]

### b. Documentation and Support

Tutorials and documentation are available from the homepage.[19] Community support options may also be found there (http://jena.apache.org/help_and_support/), including e-mail lists, an IRC channel, and StackOverflow.

### c. Download Options

- HTTP: Compiled libraries, Executables, Source;
- Maven: Download just the libraries you need for your project;
- Git: Source code access.

The SDK download includes a full suite of command-line utilities for interacting with RDF data.

## 5. AllegroGraph

AllegroGraph is a well-known RDF database capable of scaling up to large numbers of RDF triples (>>$10^{12}$). Unlike most of the other triple store implementations,

---

[17] http://jena.apache.org/

[18] http://jena.apache.org/

[19] http://jena.apache.org/tutorials/ and http://jena.apache.org/documentation/

AllegroGraph is written in Franz Lisp.[20] It is not open source, but offers a free edition for testing and familiarization with the tool that limits the triple store to 5 million triples. A developer edition is similarly limited to 10 million triples, while the enterprise edition has no limit on triples. AllegroGraph provides many command-line and client tools[21] as add-ons to the core triple store product.

**a. API Support**

AllegroGraph primarily supports a REST API, including a SPARQL endpoint, for which they make many language bindings available: Java (both Sesame and Jena variants), Python, C#, Ruby, Clojure, Scala, Lisp and Perl.

**b. 3rd Party Integrations**

- Apache Hadoop/Cloudera,
- MongoDB,
- Apache Solr,
- Top Braid Composer.

**c. Documentation and Support**

Available at http://franz.com/agraph/support/documentation/<version>/. Technical support is available via e-mail (support@franz.com).[23] In addition, an issue tracker is available for bug reports and feature requests.

**d. Download Options**

AllegroGraph components can be downloaded from http://franz.com/. These components include Core AllegroGraph with Java and Python clients; Additional language bindings; Gruff – a visualization tool for RDF triples; AGWebView; an interface to TopBraid Composer; Example code; and extensive documentation.

---

[20] A Lisp system written at UC Berkeley by the students of Professor Richard J. Fateman, based largely on Maclisp and distributed with the Berkeley Software Distribution (BSD) for the Digital Equipment Corp (DEC) VAX. It contains a number of features such as a foreign function interface, which allows interoperation with other languages at the binary level and makes it suitable for application development.

[21] e.g., Gruff, a triplestore browser application: http://franz.com/agraph/gruff/ and AGWebView: a web browser server: http://franz.com/agraph/agwebview/

[22] http://allegrograph.com/support/

[23] http://allegrograph.com/support/

## 6. Ontotext GraphDB

Ontotext is an EU company 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 the "Workbench."

### a. APIs and Integrations

GraphDB exposes both RDF4J- and Jena-compatible Java APIs. It also exposes a SPARQL HTTP endpoint for querying over a REST interface. Lucene, Solr, and Elasticsearch connectors are available for fee-based editions.

### b. Documentation

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

### c. Download Options

The free edition is downloadable at http://info.ontotext.com/graphdb-free-ontotext with registration. 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.

## 7. Stardog

Stardog is a proprietary RDF triple store that also supports semantic features, as well as graph traversal computation (described below). It is made available in Community, Developer, and Enterprise editions. Community is free to use, but limited in terms of the number of databases, statements per database, connections, users, and roles. Developer, as its name implies, is used for development and is licensed per developer on unlimited machines, but it may not be used in production. Enterprise is licensed per machine, but otherwise unrestricted, and includes support via direct telephonic access to technical staff.

### a. APIs, Query Languages, and Graph Traversal

The Stardog HTTP protocol includes the SPARQL protocol and is documented at http://docs.stardog.apiary.io/. Java programming against Stardog is available via a "native" SNARL API (deprecated), as well as RDF4J and Jena APIs. Unless you are already using SNARL, RDF4J is the Stardog-recommended API to use.

In addition to the SPARQL query language, Stardog supports graph traversal computation via the TinkerPop[24] framework. Graph traversal can be thought of as an extension to the query concept. In graph traversal, the traversal engine traverses the graph as specified, optionally performing useful computation as it traverses the data, e.g., computing a metric of how closely two people are connected in a social graph.

### b. Documentation and Support

The Stardog manual is available at http://docs.stardog.com/. Free support is available via an online forum, and a chat site. The Developer Edition comes with e-mail support, while the Enterprise edition adds access to technical support staff via telephonic communication.

### c. Download Options

All versions of Stardog are downloadable at http://stardog.com/#download, with activation past a 30-day evaluation period needed for the fee-licensed versions.

## 8. Neo4j

Neo4j utilizes a "native property graph" model, centered on the idea of both nodes and their relationships having properties. It is available both as an open source-licensed "community edition" and as a fully supported "enterprise edition." It provides support for ACID[25] transactions.

### a. API Support, Query Languages, and Third-Party Integrations

Neo4j supports programmatic manipulations primarily in Java, but it also provides official support for several other popular language bindings: C#, Python, and JavaScript. Ruby, PHP, R, Go, and others are supported at the community level.

Neo4j does not offer support for SPARQL. Instead, it uses another declarative query language called Cypher. Its syntax is designed to be intuitive to read, with match patterns resembling ASCII-art graph diagrams. In 2015, Neo Technology introduced an open source

[24] https://tinkerpop.apache.org/

[25] https://en.wikipedia.org/wiki/ACID

project, openCypher,[26] to increase adoption of and standardization of Cypher across graph database vendors.

Web development framework support is provided for popular frameworks like Spring Data, Django ORM, JDBC, and others. In addition, integrations are provided with software such as MongoDB, Cassandra, ElasticSearch, and Spark/GraphX.

A REST API is also provided (see https://neo4j.com/docs/rest-docs/current/). On top of that, analogous to SPARQL HTTP endpoints on RDF products, Neo4j can expose a "transactional Cypher HTTP endpoint" that allows a series of Cypher statements to be executed within the context of an ACID transaction.[27]

### b. Documentation and Support

There are an active StackOverflow community and e-mail lists for general support. Neo Technology offers online and in-person "Graph Academy" training courses. There are books in print, as well as free e-books about Neo4j.

### c. Download Options

- HTTP: Community edition software,
- Git: Open source collaboration is available at https://github.com/neo4j,
- Hosted Neo4j: See, for example, https://www.graphstory.com/.

## 9. ArangoDB

ArangoDB is open source, and it appears to be designed primarily around a "collection" data model. However it supports a graph data model as well, including traversals.[28] It can be configured to run in clusters for high availability or scalability.

### a. APIs and Query Languages

ArangoDB exposes an HTTP API (https://docs.arangodb.com/3.1/HTTP/). All language drivers rely on this HTTP API.

- Java – Synchronous (https://github.com/arangodb/arangodb-java-driver) and Asynchronous (https://github.com/arangodb/arangodb-java-driver-async) variants;
- JavaScript, is the main supported driver (https://github.com/arangodb/arangojs);

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

[27] http://neo4j.com/docs/developer-manual/3.0/http-api/#http-api-transactional

[28] https://docs.arangodb.com/3.1/Manual/Graphs/

- PHP (https://github.com/arangodb/arangodb-php);
- Unofficial community drivers, e.g., Python (https://github.com/tariqdaouda/pyArango).

ArangoDB uses its own SQL-inspired declarative query language, called the ArangoDB Query Language (AQL).[29] It is a pure data manipulation language (DML) supporting CRUD operations on collection instances. AQL cannot be used for database creation/deletion or administration (i.e., it has no data definition language (DDL) components).

In addition to the above, ArangoDB, offers Foxx, a JavaScript "microservice framework," whereby the administrator may expose additional HTTP endpoints on the ArangoDB server, running with "native in-memory" access to the server process.

### b. Documentation and Support

A manual (https://docs.arangodb.com/3.1/Manual/) and various community mechanisms are available:

- Issue tracker (https://github.com/ArangoDB/ArangoDB/issues),
- StackOverflow tag (http://stackoverflow.com/questions/tagged/arangodb),
- ArangoDB Cookbook (https://docs.arangodb.com/3.1/cookbook/),
- Google Group (https://groups.google.com/forum/#!forum/arangodb).

### c. Download Options

Many download options are available at https://www.arangodb.com/download/, including source code, Windows installer, cloud containers, Docker, and package formats for various Linux package managers.

## 10. Bitsy

Bitsy is a graph database implementation designed to process graph data structures in memory, although the data is persisted in the form of readable text files. It is designed to support the Blueprints API, which means it can be used as the data store for software designed to use TinkerPop-compatible backends.[30] It appears to be primarily intended as an embedded application database with an approach reminiscent of relational database implementations such as Sqlite.

---

[29] https://docs.arangodb.com/3.1/AQL/

[30] https://bitbucket.org/lambdazen/bitsy/wiki/Home

### a. APIs

Bitsy is primarily designed to be used in a Java virtual machine (JVM) environment, which implies support for Java or any of a number of languages that run on the JVM, such as Groovy or Scala.

### b. Documentation and Support

A wiki is the sole source of documentation.[30] Bitsy has a single author, with Twitter handle @lambdazen. The instructions for getting started using Maven work, although the API documentation is lacking. One needs to read through the Wiki and examine the Bitsy source before one can get productive.

### c. Downloading

Java JAR files are downloadable from the download page.[31] They are also accessible via the Maven tool.

## 11. Blazegraph

Blazegraph claims to be a high-performance graph database with RDF/SPARQL and TinkerPop API support. It can be run as an embedded application server or in high availability and cluster configurations. It is available for download and can be used under the GPLv2 open source license. Fee-based licenses for single users as well as at the enterprise level are also available.

### a. APIs and Query Languages

Blazegraph exposes many useful APIs:

- REST:
  - NanoSparqlServer – a lightweight API for RDF;
  - A full SPARQL endpoint, plus additional resources for managing Blazegraph features, like multi-tenancy, transaction management, backup, bulk data load, et al.
- Java:
  - A wrapper API for client access to NanoSparqlServer;
  - Multi-Tenancy API;
  - Wrapper for full REST API;

[31] https://bitbucket.org/lambdazen/bitsy/downloads

- The wrapper may be used as a triple store with the RDF4J API, i.e., has an implementation of the RDF4J Repository API.

- TinkerPop3:
  - Graph traversals using TinkerPop API (Groovy, Java, et al);
  - Python and .NET;
  - Wrappers for full REST API (although these have gaps compared to the Java wrapper).

As mentioned above, Blazegraph supports TinkerPop3 graph traversal and the SPARQL query language, including many useful extensions, such as full text search and geospatial search.

### b. Support and Documentation

Standard commercial licensing cost is presently $2,000/server/year, or half that for "startups, non-profits, and research." Enterprise licensing is also available. Support plans seem to be sold separately. "Community support" is freely available to all, which includes a mailing list and online issue tracker. "Developer support" is available, which includes all manner of consultation related to design and development using Blazegraph (2 seats, 7 days for $750 to 1 year for $24,995). "Production" support is available on an annual basis ($9,995/server/year), and includes emergency patches.

An online user manual is available at https://wiki.blazegraph.com/.

### c. Download Options

The Debian installation package was successfully downloaded and tried on Ubuntu 16.04. After installing the package, the server was launched with "sudo blazegraph start" and the web interface was viewable at http://localhost:9999/blazegraph/. RPM, JAR, WAR, and .tar.gz downloads are also available.

## 12. Cayley

"Cayley is an open-source graph inspired by the graph database behind Freebase and Google's Knowledge Graph." As such, Cayley can be used as an in-memory graph database, or be configured to use one of several data storage back ends: LevelDB, Bolt, MongoDB or PostgreSQL. The graph data uses RDF-like datatypes, with the basic unit of storage being a quad (RDF triple + graph ID). However, it appears that there are fewer restrictions on what types of data may be subjects (originating node) and predicates (edge).

### a. APIs and Query Languages

Cayley is a graph database implementation written in the programming language Go. Its APIs can support querying and an HTTP endpoint. As such, it can be used either as a Go library or as a graph database application. It also offers a RESTful API, a web application that utilizes that API, as well as a REPL. The RESTful API and REPL both support two query languages. The first uses JavaScript object, reminiscent of the Gremlin graph query language. The other language is a simplified version of the deprecated Freebase Metaweb Query Language (MQL).[32] There are community-provided language bindings for several languages such as Clojure, JavaScript/Node.js, Ruby, PHP, Python, .NET, Rust, and Haskell.

### b. Documentation and Support

Documentation is provided within the source repository.[33] Support is community-based, starting with the homepage (https://cayley.io/), a Discourse site, a Twitter account, and an IRC channel.

### c. Download Options

Source is available from the GitHub project, and binaries are downloadable from a web page that tracks the version releases.[34]

## 13. DataStax Enterprise Graph

DataStax Enterprise (DSE) is a proprietary database product built on top of Apache Cassandra.[35] This statement is important to understanding DSE. Apache Cassandra is an open source database designed from the ground up to be linearly and dynamically scalable across multiple peer-to-peer nodes (a.k.a. partitions), and also to allow specifying where data should live geographically. A central concept in defining Cassandra schema is to appropriately define key spaces and configure a data replication strategy so that data is distributed across the cluster in a uniform and performant pattern. Cassandra by itself does not support graph data. DSE Graph is a supported extension of DSE that enables working with graph data within DSE.

---

[32] https://github.com/nchah/freebase-mql

[33] https://github.com/cayleygraph/cayley/tree/master/docs

[34] https://github.com/cayleygraph/cayley/releases

[35] https://cassandra.apache.org/

### a. APIs and Query Languages

DSE supports multiple programming languages (C++, .NET, Clojure, Go, Java, Node.js, PHP, Python, Ruby) with libraries called drivers. These are more sophisticated than traditional RDBMS drivers, in that they are aware of DSE cluster states, and use that information to access the cluster in the most efficient way possible.

DSE Graph supports the Apache TinkerPop 3.0 graph computing framework, including the Gremlin language, and also provides a gremlin-console tool.

### b. Support and Documentation

DSE offers three levels of subscription support[36]: Basic, Standard, and Max. Basic is targeted at small do-it-yourself (DIY) projects. Standard provides many of the enterprise-level features needed for scaled deployment. Max is targeted for high-end power users of DSE, e.g., intelligent, data-intensive cloud applications. DSE Graph is only supported via upgrades planned for the Standard or Max levels.

For trial usage and testing, i.e., not-production use, DSE can be used for free for up to 6 months.

Many learning resources are included within DSE and on the DataStax website, including online courses, tutorials, and in-person training at https://academy.datastax.com/. Full documentation, including for the drivers, is available at https://docs.datastax.com/.

### c. Download Options

DSE is easily downloaded for a variety of platforms from the DataStax website. Registration is required. DSE is, at its core, implemented in Java.

## 14. Dgraph

Dgraph is an open source graph database, designed for "scalable, distributed, low latency and high throughput." It is implemented in the programming language Go, and development continues at the Dgraph company under the support of venture capital funding. According to their own documentation,[37] it is not ready for production systems. The current version is 0.4.4, and they state that 1.0 will be the first production-ready version. Prior to that, any new version update may introduce compatibility-breaking changes. Nonetheless, they have made some interesting design choices in pursuit of the virtues stated above.

---

[36] http://www.datastax.com/products/subscriptions

[37] https://wiki.dgraph.io/Dgraph#Status

It supports an RDF-like full property graph schema in which nodes may hold URI-like IDs, primitive data, or "objects," which are analogous to JSON objects, and which fulfill the notion of nodes containing metadata.

### a. APIs and Query Languages

According to the production-ready roadmap,[38] Java and Go "response drivers" should exist, but their site does not offer any other documentation of these drivers or what they are. The documentation only discusses the HTTP API, which uses Facebook's GraphQL[39] syntax. TinkerPop and Gremlin support are planned for after the 1.0 release.

### b. Documentation and Support

Official documentation for Dgraph lives at https://wiki.dgraph.io/. The home page also links to community discussions hosted on Discourse and Slack. Source code access and an issue tracker are provided at https://github.com/dgraph-io/dgraph/.

### c. Download Options

The homepage contains a prominent download link. The easiest way to get up and running is to use the first option presented, which is to download their installation shell script and run it. For security reasons one should first inspect the script before running it. Manually downloading the tarball can be accomplished as follows:

```
wget https://github.com/dgraph-io/dgraph/releases/download/v0.4.4/dgraph-
linux-amd64-v0.4.4.tar.gz
```

In the above command, the string "v0.4.4" should be replaced with the one corresponding to the latest version. Manual installation instructions are given for various Linux versions and Mac. Docker images are also available and are described on the same page.

## 15. FlockDB

FlockDB is an open source, distributed, fault-tolerant graph database from Twitter designed for storing adjacency lists.[40] The adjacency-list emphasis means it is not designed for efficiently traversing graphs. It does have the stated goals of high throughput of CRUD operations, complex set arithmetic queries, fast paging through millions of query results, and horizontal scaling and online data migrations.

---

[38] https://github.com/dgraph-io/dgraph/issues/1

[39] See https://facebook.github.io/graphql/ and https://wiki.dgraph.io/Queries_and_Mutations

[40] https://github.com/twitter/flockdb

### a. APIs and Query Languages

A Thrift definition file is included, which allows the easy generation of clients in all the languages that Apache Thrift[41] supports, namely, C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml, Delphi, et al. An enhanced Ruby client library is also available, which adds some "syntactic sugar" on top of the Thrift interface. The demo outlined in the documentation shows how to populate data, and perform simple-to-complex adjacency queries.[42]

### b. Documentation and Support

Limited documentation is available with the source code. Community support is available via Internet Relay Chat (IRC), an e-mail list, and the Twitter hashtag, #flockdb.

### c. Downloading

The source code (mostly Scala-based) is downloadable at https://github.com/twitter/flockdb/releases, and may be built using freely available tools.

## 16. GraphDB

GraphDB comes from Microsoft, and is perhaps best described as a freely available .NET toolkit for building an in-memory graph data store. GraphDB provides a distributed memory storage capability, which uses a message passing framework, to accomplish storing and querying data distributed across the RAM of many servers.

### a. APIs and Query Languages

Schemata are defined using the Trinity Specification Language (TSL). Data is manipulated using .NET-compatible languages like C#, typically leveraging Language Integrated Query (LINQ), an SQL-like syntax extension available within C#, as well as other .NET languages.

### b. Documentation and Support

An online manual is available at https://www.graphengine.io/docs/manual/. No paid, commercial support is available, but free support is available from the project contacts, and via StackOverflow.[43]

---

[41] https://thrift.apache.org/

[42] https://github.com/twitter/flockdb/blob/master/doc/demo.markdown

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

### c. Downloading

The recommended way to get a copy of this graph database is by downloading the GraphDB Visual Studio Extension (Visual Studio 2012+). Methods for doing this are described on the Downloads page,[44] as is how to get the GraphDB Core library via the NuGet package manager.

## 17. Grapholytic

Grapholytic is a SaaS offering from the German company MIOsoft. Their website offers free demonstrations of graph manipulation utilizing the Cypher language, which may indicate that Neo4j forms part of their implementation. Not much else can be gleaned from their website. It is reasonable to assume that it uses the MIOvantage software product broadly described on their U.S. website.[45]

## 18. GraphBase

GraphBase is a product from an Australian company of the same name. It bills itself as a "2nd generation" graph DBMS. The learning materials on their website claim that graphs in the "Graph Simple Form" are encouraged/enforced by GraphBase. This form seems to place a few restrictions: only one edge between any two nodes, no edges connecting a node to itself, no properties on edges (only on nodes), and only four types of edges (some directed, some not).

Multiple editions are available. "Agility Edition" is freely available and downloadable. Commercially licensed "Enterprise Edition" and "i6" are also available. The i6 version appears to be a product for monitoring data streams in real time.

### a. APIs and Query Languages

Both a Java API and REST API are provided. An RDF+ extension is provided, which provides the Jena API.

### b. Documentation and Support

No explicit link to documentation is offered on their website. It is hard to find much on their website, though. They claim online training is available, for individuals or teams.[46]

---

[44] https://www.graphengine.io/download.html

[45] https://www.miosoft.com/products/miovantage/

[46] http://graphbase.net/Training.html

### c. Downloading

Downloads of the free trial version or the agility edition are supposed to be available with site registration.[47] However, the IDA team was unsuccessful in using the provided download links after registration. Subsequent requests via e-mail for support with the software download went unanswered.

[47] http://graphbase.net/TryNow.html

# Appendix B
# Sample Quantitative Results for Scalability of Data Loading

## 1. Introduction

This appendix describes scalability tests for a small subset of the graph database implementations covered in Appendix A above. The IDA study team plans to continue the testing of proprietary and open source graph database offerings in the subsequent phases of the study. Although the tests described here are not exhaustive, they nevertheless provide a quantitative metric regarding the effort necessary to migrate legacy relational database content to a graph data lake, in a manner consistent with the portal architecture solution described in Section 1.B (See Figure 1-4).

Since the actual time necessary to load data into a given graph database implementation will depend in large part on the hardware that is used to host it, and comparison of available hardware choices for the data store backend is not part of the assessment covered during this phase of the study, the majority of the statistics presented are in the form of normalized charts, where the base case – for example loading 10K triples into the graph database being tested – represent the normalized time unit, and all subsequent loads, i.e., 20K, 40K, etc., are referenced to that base case. These metrics provide a good indication regarding performance degradation in graph database implementations as a function of pre-existing data load, which is a recurring cost, as opposed to the one-time migration of legacy source data, which, even if initially onerous, is amortized over the life-cycle of the implemented solution architecture.

## 2. Relational Database Baseline

Relational databases constitute the default data store backend choice for the vast majority of web applications that offer dynamic content and support user interactivity. During this phase of the study, the IDA team tested the scalability of data upload in relational databases using two open source relational database systems, namely, MySQL and PostgreSQL.

### a. Statistics for the MySQL Case

The basic test consisted of measuring the time that it would take to load a series of computer-generated records into a single test table. The complexity of the table structure itself was deemed to be of secondary importance when measuring how fast one can load

millions of records into the table, as compared to factors such as the presence of indices. The relatively simple structure of the test table **PERSON** is shown in Table B-1.

**Table B-1. Structure of the Test Table PERSON**

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

A sample of the data loaded into the **PERSON** table is shown in Table B-2.

**Table B-2. Sample Data Stored in the PERSON Test Table**

| perID | fname | lname | dob |
|---|---|---|---|
| 1000000005 | MARIKO | LLOF857804344 | 2159-5-13 |
| 1000000010 | STEPHANIE | PROP208452903 | 2189-10-3 |
| 1000000015 | CHARLIE | AMVZ270803024 | 2125-6-30 |
| 1000000020 | LUCIO | BNCH68188979 | 2153-5-23 |
| 1000000025 | RENAY | XILI143950779 | 2167-5-8 |
| 1000000030 | LASONYA | PXDN559333784 | 2128-6-7 |
| 1000000035 | VERNA | JXXE272799784 | 2198-2-26 |
| 1000000040 | LOREN | PLDM515826885 | 2181-1-19 |
| 1000000045 | ELVA | BLBN795672406 | 2144-11-29 |
| 1000000050 | JEREMIAH | CVGK497719544 | 2171-8-10 |

To avoid any possibility of inadvertently creating personally identifiable information (PII) during the testing, the last name assigned to each of the records is a combination of randomly chosen letters followed by a number generated with a random number generator. Similarly, the birth dates correspond to events taking place in the next century, which also excludes the possibility of their being associated with the birthday of any individual who is alive at the time of the writing of this report.

We note that since the point of the exercise was not to create a data set that could be used for evaluating other aspects of the relational engines, such as date functions supported by MySQL and other relational database engines, the dates were randomly generated, with months taking values between 1 and 12, and days taking values between 1 and 31, irrespective of whether some combinations were improper (e.g., February 31 or November 31).

The last point to note is that in MySQL offers two choices for the table template one can use when creating a new table in a database. The older and simpler template known as

MYISAM, which does not support referential integrity constraints, and INNODB, which does and requires additional metadata. With loads of up to 1 million records, the difference in performance was small (26.713747 seconds required to generate and load 1 million records using the MYISAM template, versus 27.337114 seconds required to generate and load 1 million records using the INNODB template, which is approximately 2% slower). To minimize the run time associated with each of the record sets, the IDA team selected the MYISAM template.

Figure B-1 shows the statistics obtained when programmatically generating and loading a series of datasets beginning at 1 million and ending at 128 million records. As the graphic shows, relational engines have an almost constant rate, so that if it takes 42.93 seconds to load 1 million records, it will take roughly 128 times longer to load 128 million records (i.e., 5448.16 seconds).

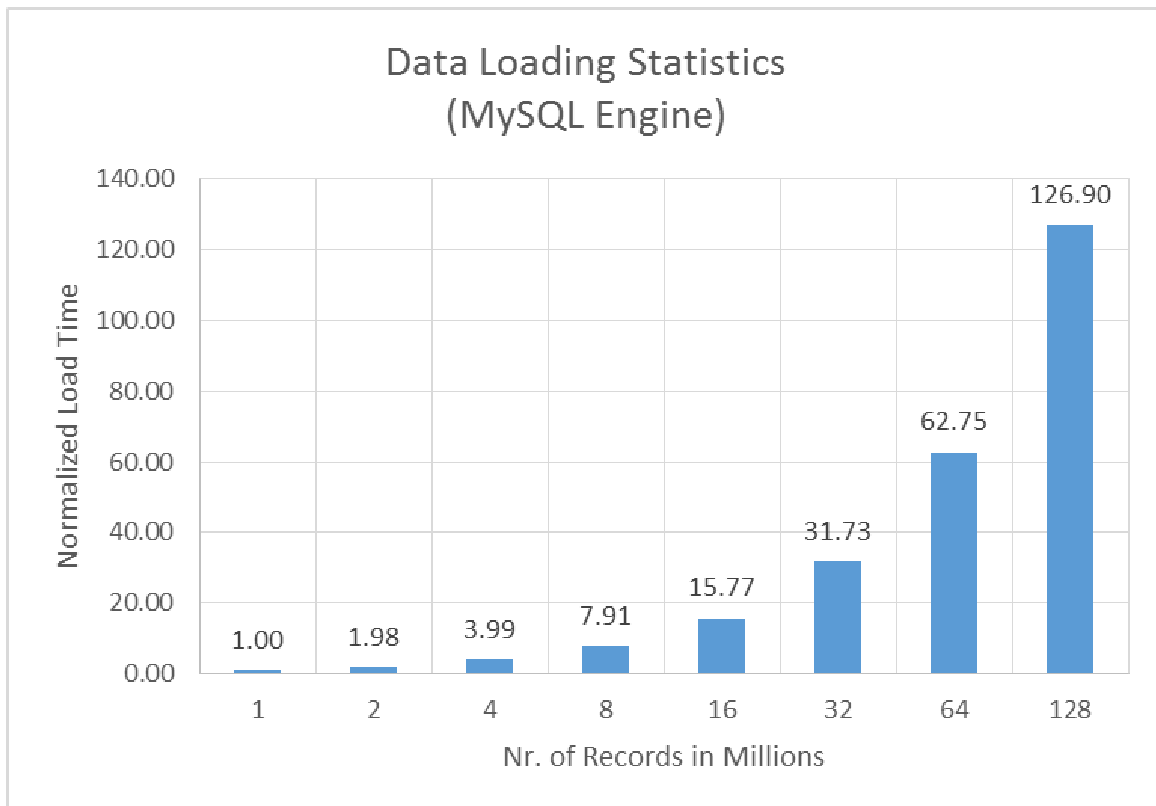


**Figure B-1. Statistics for Programmatic Data Loading into the Test Table PERSON[48]**

The indexing of a column such as **lname** in the test table **PERSON** can be accomplished with the statement:

```
CREATE INDEX lname_idx ON PERSON(lname);
```

[48] The reference unit of time for this test corresponds to 42.93 seconds.

There is a visible performance degradation as one crosses from 4 million to 8 million records

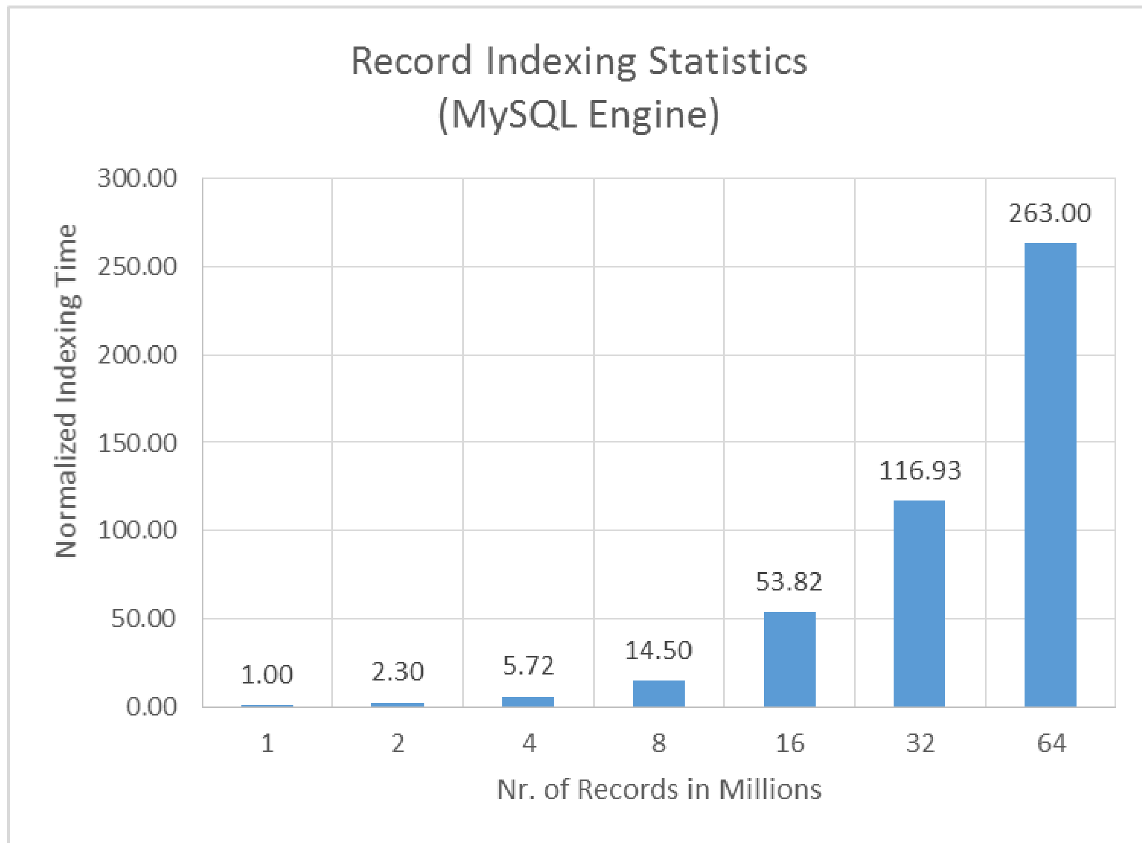


**Figure B-2. Statistics for Adding an Index to a Column in the Test Table PERSON**[49]

The performance degradation for the indexing operation becomes quite glaring when going from 64 million to 128 million, where the time required is almost tenfold than what it takes to index 64 million records in the test table **PERSON**. This behavior is likely to be caused by the configuration of the virtual machine in which the tests were conducted, e.g., the number of cores available, the amount of RAM, etc., and highlights the need to size adequately the server on which the backend data store is going to reside to avoid unacceptably slow response times.

[49] The reference unit of time for this test corresponds to 4.33 seconds.

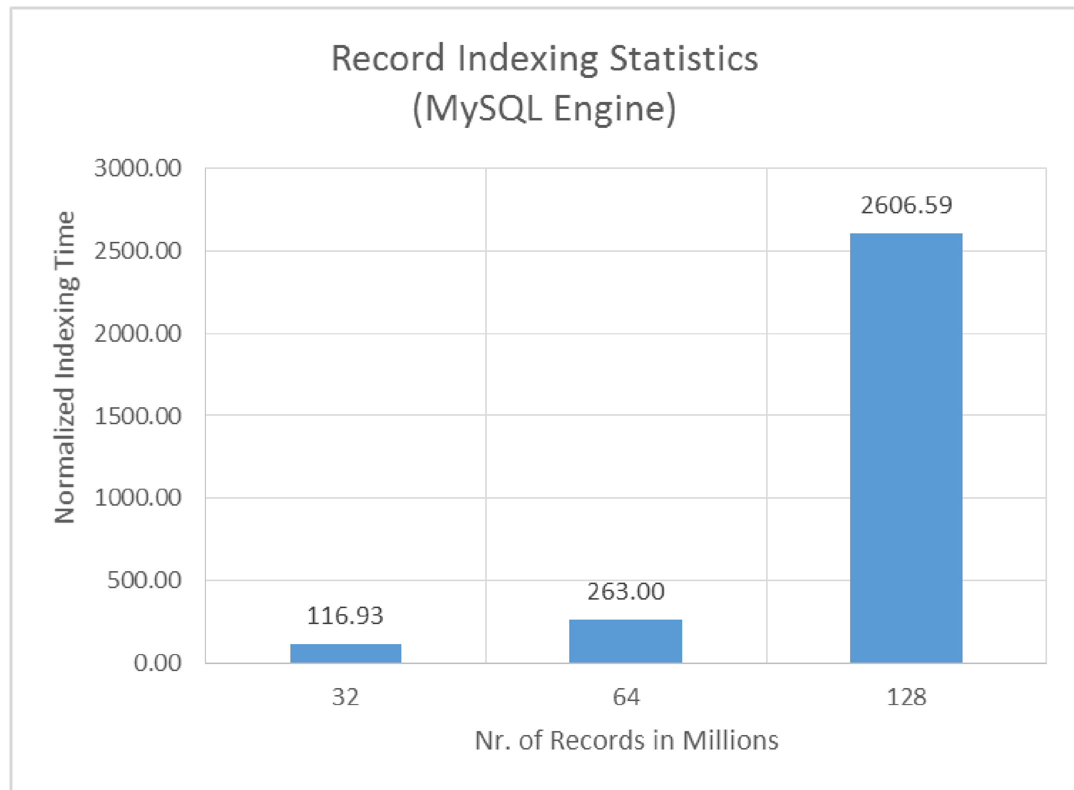


**Figure B-3. Visible Performance Degradation for the Indexing of a Single Column in the Test Table PERSON**

Simple queries such as finding whether there are records with duplicate values for the **lname** column in the test table **PERSON** can be accomplished with the following query:

```
SELECT lname, COUNT( lname ) x FROM person GROUP BY lname HAVING x >1;
```

Figure B-4 below shows the statistics associated with the above query, which reflects an almost linear correlation between the size of the set and the response time of the query, namely, that doubling the number of records doubles the execution time as one progresses through the series from 1 million to 128 million records.

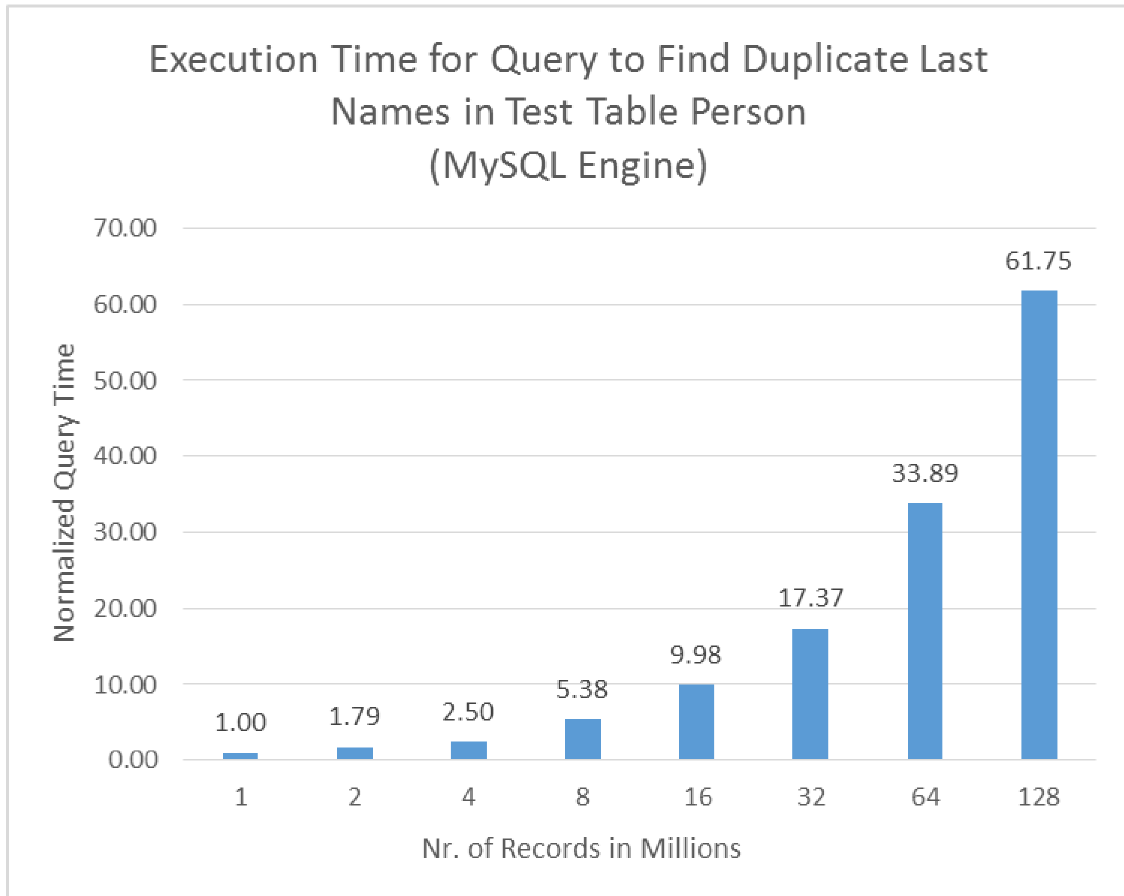


**Figure B-4. Statistics for Execution Time of Query to Find Duplicate Last Names in the Test Table PERSON**[50]

There is relatively little performance degradation when retrieving records filtered by a specific value of the **lname** column. Figure B-5 shows the statistics when performing a query of the type shown below, as one progresses through the series, beginning with 1 million records in the test table **PERSON** and ending at 128 million records.

```
SELECT * FROM person WHERE lname='YTIB540047024';
```

[50] The reference unit of time for this test corresponds to 1.17 seconds.

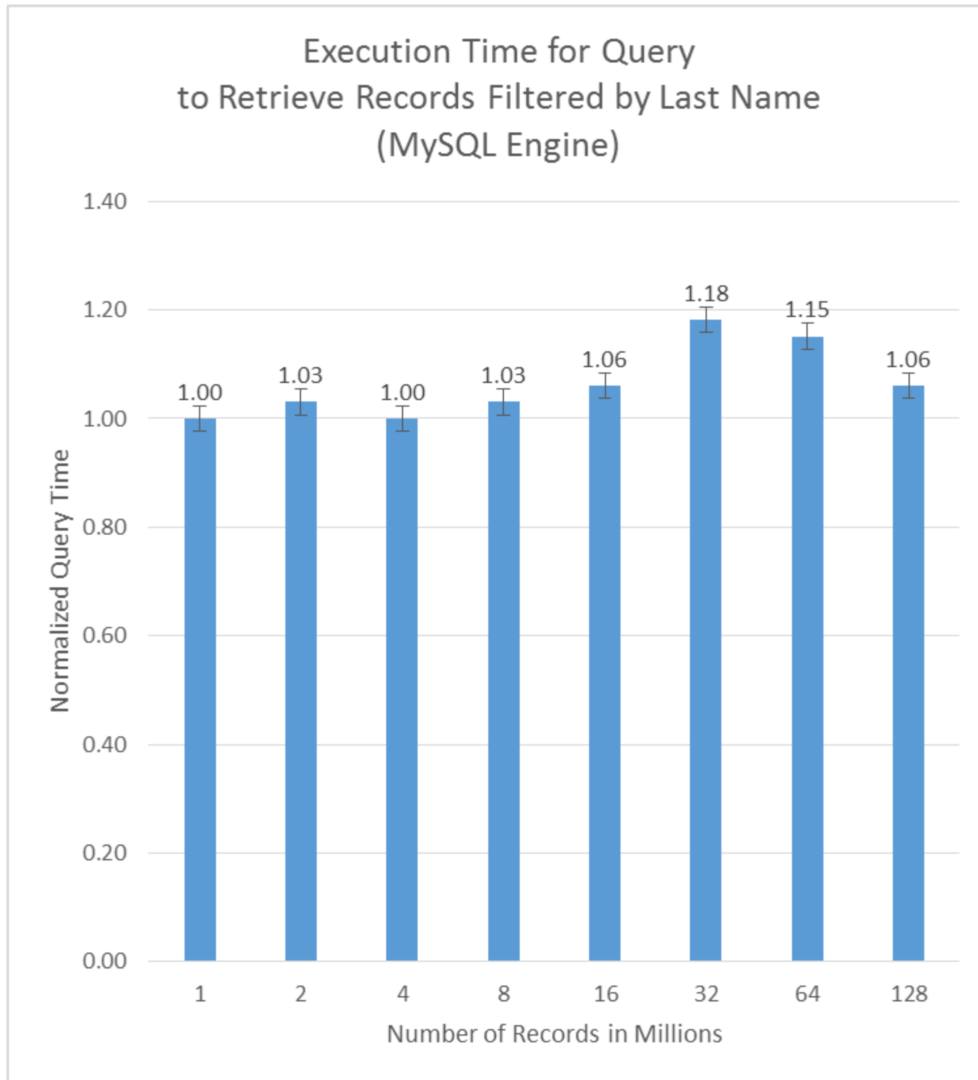


**Figure B-5. Statistics for Execution Time of Query to Retrieve a Specific Record in theTest Table PERSON**[51]

## 3. RDF4J Performance

As noted in Appendix A above, RDF4J is not designed to support very large data sets by itself. However, both the very user-friendly interface offered by the Workbench web interface and the libraries available to develop applications for managing the content of the triple store make this implementation quite appealing.

The availability of good APIs for programmatic manipulation of MySQL, PostgreSQL, and other relational databases, as well as RDF4J itself, makes the possibility of moving records directly from a legacy source database to a target graph database that

[51] The reference unit of time for this test corresponds to 0.33 seconds.

uses RDF4J very appealing. Although the actual times for loading the records is much slower than when creating and loading records into a relational database, nevertheless, the time needed for the record injection behaves linearly, i.e., doubling the number of triples doubles the time needed to put the data in the RDF4J graph database.[52]

Note that although the total times for the programmatic loading of records shown in Figure B-6 include not only the time to execute the SQL query, but also the time to convert each record into the corresponding RDF triple. Those times compare fairly well with what one obtains when loading 1 million triples (approx. 333K records from the test table **PERSON**) using the RDF4J console, namely 85,919 ms, which correspond to 2578 ms for 30K triples.

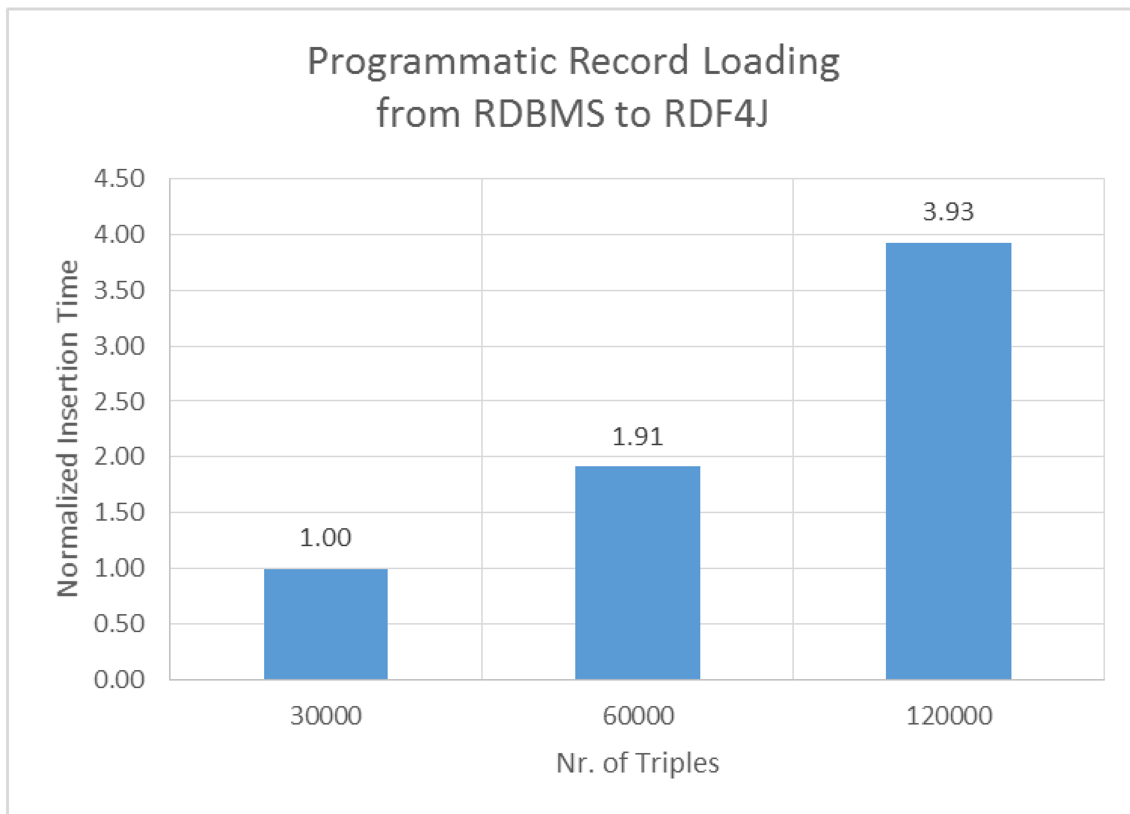


**Figure B-6. Statistics for Loading Programmatically Into RDF4J Records from the Test Table PERSON Using MySQL Server**

Programmatic injection into RDF4J of data already formatted as RDF triples is also possible, and it scales well, as shown in Figure B-7. One should carefully assess whether development of in-house code versus use of the vendor-provided tools is justified, because most likely, the latter may already be utilizing the same libraries that ad-hoc code requires,

---

[52] The unit of reference for this test is 2710 ms.

and thus there may be little gain in terms of performance. On the other hand, if one needs to support additional data manipulations, the in-house development may be not only desirable, but unavoidable.

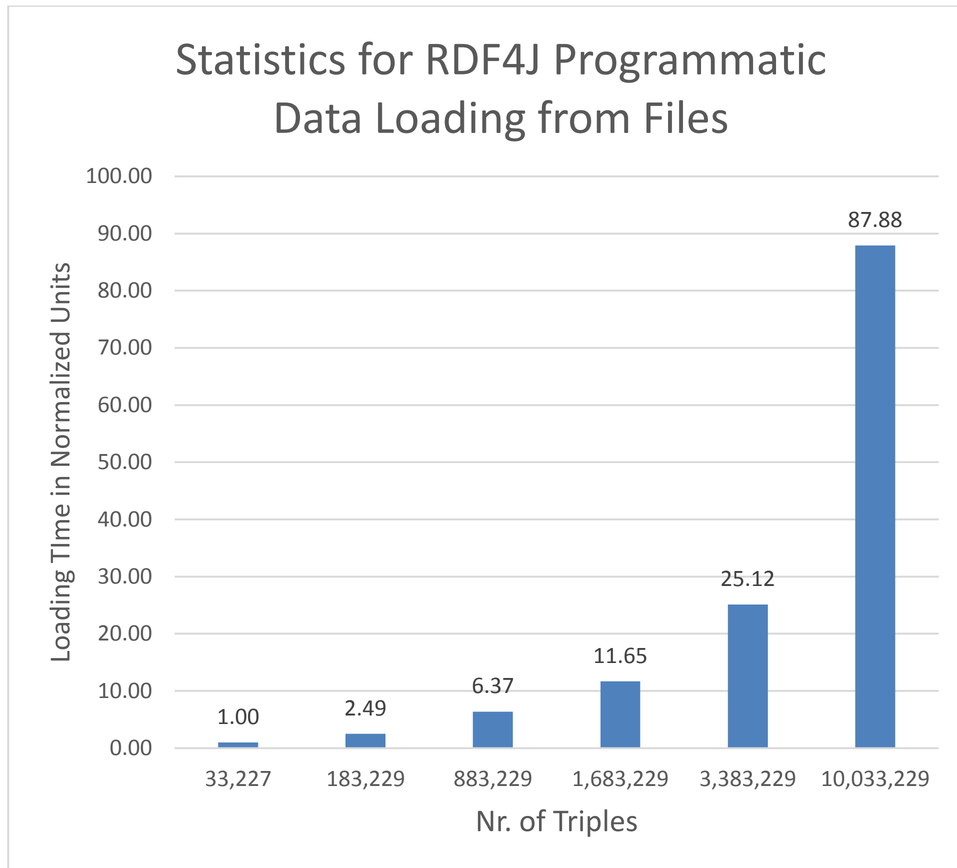


**Figure B-7. Statistics for Programmatic Data Loading using Preformatted Data Files**[53]

Additional optimizations may be required to achieve performances that can support heavy traffic usage. We note as an example the striking difference in execution times when the query contains a filter on the first name (**fname**) versus the last name (**lname**). The only visible difference between the values of those two predicates is that **fname** is pure alpha, whereas **lname** is alpha-numeric.

[53] The reference unit time is 4.9 seconds.

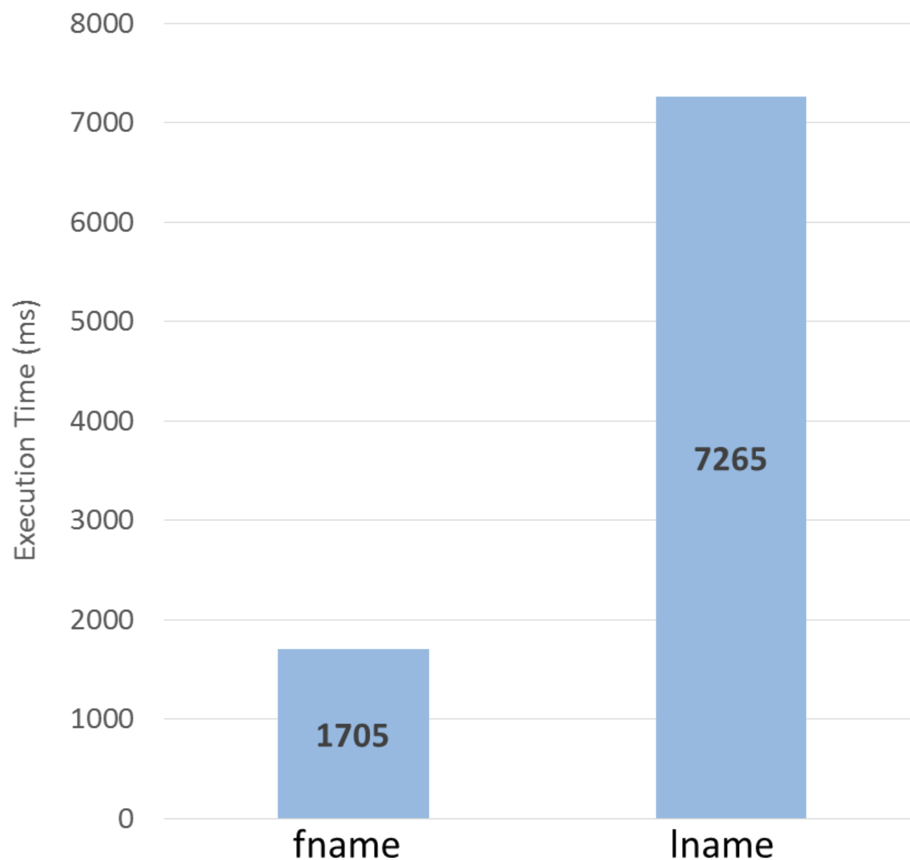


**Figure B-8. Performance Difference For RDF Data Retrieval in RDF4J When Filtering on the First Name or the Last Name**

Below is the query filtering on first name (**fname**):

```
PREFIX usaper: <http://Army.gov/Portal/ForceStructure/Person/meta#>
select ?person ?fname ?lname ?dob
where {?person usaper:lname ?lname .
       ?person usaper:fname ?fname .
       ?person usaper:dob ?dob .
    filter(?fname = "HELEN")}
```

The corresponding query filtering on last name (**fname**) is:

```
PREFIX usaper: <http://Army.gov/Portal/ForceStructure/Person/meta#>
select ?person ?fname ?lname ?dob
where {?person usaper:lname ?lname .
       ?person usaper:fname ?fname .
       ?person usaper:dob ?dob .
    filter(?lname = "ODXS882914495")}
```

## 4. Neo4j Performance

The discussion presented in Section 3.B highlighted some of the shortcomings of this graph database implementation. However, given the popularity of this application and the active user base, it may reach sufficient critical mass to either entice its developers to offer a SPARQL implementation or for its query language CYPHER to gain sufficient acceptance among the new generation of graph databases that it may not represent too much of a risk using Neo4j in the future.

Figure B-9 shows how the Neo4j scales up when loading data formatted as CSV files.[54] As the values in the graphic show, the behavior is fairly linear Loading 10K triples takes about 10 times longer than loading 1K, and so on.

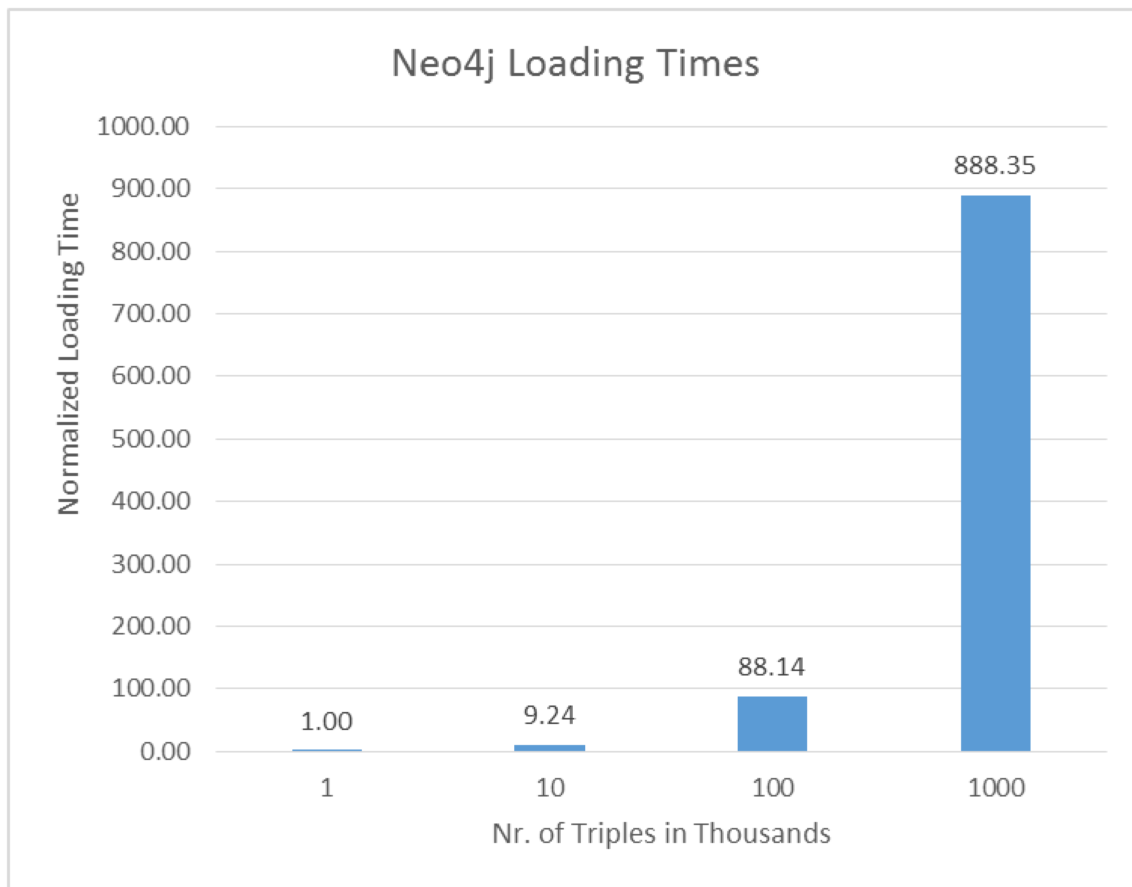


**Figure B-9. Statistics for Loading of Data into Neo4j**

[54] The reference unit time is 1.010533 seconds.

## 5. Stardog Performance

Stardog is a proprietary graph database implementation that leverages the RDF4J APIs. It claims much higher scalability and a robust set of inference capabilities. Testing yielded a hard-earned lesson that bulk-loading large quantities of RDF data into a Stardog database all at once should be done at the time of database initialization. If one attempts to bulk-load data into a live Stardog DB, the transactional system bogs down past a few million triples and always causes a heap error on the server process before 24 million triples can be loaded.

That said, the better method to use is to ingest the RDBMS data using an RDBMS-to-RDF converter script and output it into a set of serialized RDF files that are large, but not too large. These files can be placed on a filesystem accessible to the Stardog server, and the server can then be requested to initialize a new database using the set of RDF files. The IDA team found that files of around 300k to 400k triples each were ideal for loading throughput.

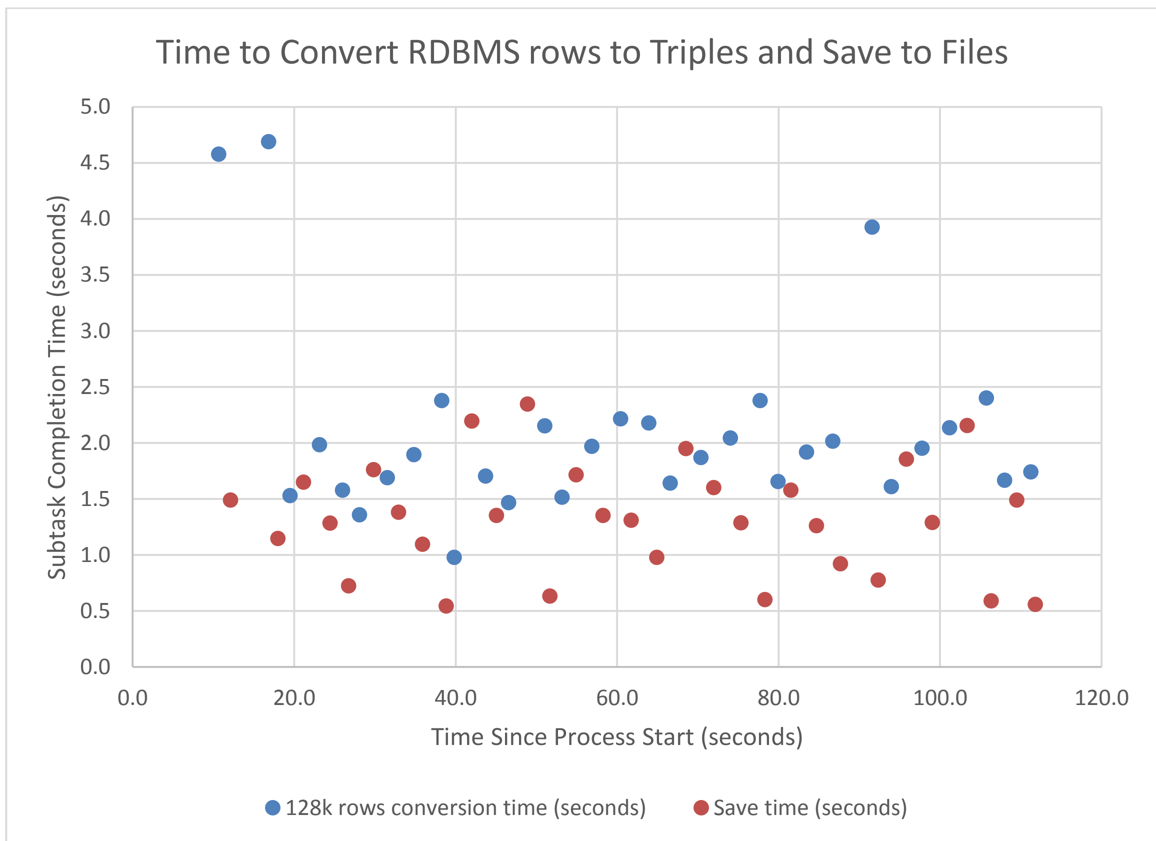


**Figure B-10. Stardog Statistics for Programmatic Conversion and Loading of Data Stored in the Test Table PERSON**

The graph shows the rate at which 4 million DB rows (12 million RDF statements in this case) could be loaded into Stardog in this fashion. The server reported that the triples were loaded in 4 minutes 29 seconds, corresponding to an overall load rate of approximately 44,600 triples per second.

## 6. Summary

Although the graph databases selected during this phase of the study are not faster in terms of loading data than are commonly used relational databases (e.g., MySQL, PostgreSQL), their performance does not appear to deteriorate as the size of the datasets increases, and in some cases their performance is comparable.

As we noted above, depending on whether or not one expects very large data sets to be constantly loaded into the graph database, the cost of the initial load may be amortized over the life cycle and therefore does not matter much. Almost without exception, the graph databases tested performed well when the data loads were small (between 10K and 100K triples). In scenarios for which mostly updates on that order of magnitude are expected, the cost of data loading should not be considered the determining factor for whether or not to adopt this technology.

Query retrieval times also show little impact as a function of data set size. As with relational databases, if the graph database implementation offers the possibility of indexing the triples and using other optimization techniques, the query retrieval times may be adequate to support heavy traffic, although additional testing, and more specifics concerning the actual use cases that need to be supported will be required.

# Appendix C
# Sample Code Used for Testing Scalability of Data Loading

## 1. Introduction

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. In addition, the code snippets also give an objective baseline regarding how the IDA team conducted the tests documented in Appendix B above so that they can be improved on or replaced if found insufficient or inadequate.

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.[55] The Institute for Defense Analyses, however, retains the copyright of all the code contained in this appendix.


```
# 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.
```


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

## 2. Code to Convert Relational Database Content to RDF

The Python script below shows how to use the Python RDFLIB library for the purpose of generating test RDF triples. The program connects to a relational database where data to be converted may reside in one or more tables. The SQL SELECT query retrieves the dataset, and the code loops through it adding every new triple to the graph. When that process terminates, the program invokes the serialization method supported by the RDFLIB library. Since this is a very simple harness, the print statement dumps the results to standard output, where they can be redirected to a text file for persistence and reuse. The generated RDF triples can be used to test the speed and scalability of a graph database.

```
    # 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
#

from rdflib import Graph, Literal, BNode, Namespace, RDF, URIRef, XSD
import MySQLdb as mdb

g = Graph()

n = Namespace("http://Army.gov/Portal/ForceStructure/Person/meta#")

# Replace <user name here> and <password string here> in the line below
# with the applicable entries for your MySQL server implementation

con = mdb.connect('localhost', '<user name here>', '<password string here>', 'portal02')

with con:
```

```
    cur = con.cursor()
    cur.execute("SELECT * FROM person01a")

    rows = cur.fetchall()

    for row in rows:
      perLabel =  URIRef("http://Army.gov/Portal/ForceStructure/Person/meta#"+"PER01-" + str(row[0]))
      g.add((perLabel,n.fname,Literal(str(row[1]),datatype=XSD.string)))
      g.add((perLabel,n.lname,Literal(str(row[2]),datatype=XSD.string)))
      g.add((perLabel,n.dob,Literal(str(row[3]),datatype=XSD.date)))

# Change the value of the format to generate other serializations
# such as RDF-XML, Turtle, etc.

print( g.serialize(format='nt') )
```

## 3. Code to Populate a Relational Database Test Table for Baseline Performance Measurements

The Python script below shows how to use the Python **MySQLdb** and the **random** libraries to generate and load into the test table **PERSON** sets of records ranging in size from 1 million to 128 million records.[56]

```
     # 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
```

[56] See Appendix B.2 above for a description of the test table **PERSON**.

#

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

from random import *
import MySQLdb as mdb
import time
fn = []
alphabet = ['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','X','Y','Z']

f = open('FirstNames.txt', 'r')

counter01 = 0

for line in f:
   columns = line.split()
   fn.append(columns[0])
   counter01 += 1
print counter01

con = mdb.connect('localhost', 'root', '2MctlMctl!', 'portal02');

idcounter = 1000000005

with con:
   cur = con.cursor()
   t0 = time.clock()

   for j in range(0,128000000):
      fname = ''
      lname = ''
      dob = ''
      N01 = randrange(0, len(alphabet))
      N02 = randrange(0, len(alphabet))
      N03 = randrange(0, len(alphabet))
      N04 = randrange(0, len(alphabet))
#   print first,second,third
      fname = fn[randrange(0,counter01)]
      lname = alphabet[N01] + alphabet[N02] + alphabet[N03] + alphabet[N04] +
str(randrange(10000001, 900000001))
      year = str(randint(2100,2200))
      month = str(randint(1,12))
      day = str(randint(1,31))
      dob = year+'-'+month+'-'+day
      rec = "INSERT INTO person08a VALUES(" + str(idcounter) + ",'" + fname + "','" + lname + "','" +
dob +"')"
      cur.execute(rec)
      idcounter += 5

   print time.clock() - t0, "seconds required to generate and load 128M records (myisam) --person08a"
```

## 4. Code to Convert Relational Database Content to CSV

As noted in Appendix A, some graph databases such as Neo4j are designed to ingest CSV files instead of files that use any of the multiple serializations available for RDF triples. The Python script below essentially accomplishes the same task as the program in Section 2 above, but the output used the comma delimited format.

```
    # 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
#
    import csv
    import MySQLdb as mdb

# Replace <user name here> and <password string here> in the line below
# with the applicable entries for your MySQL server implementation


    con = mdb.connect('localhost', '<user name here>', '<password string here>', 'portal02'))

    with con:
      cur = con.cursor()

      cur.execute("SELECT * FROM person01a")

      rows = cur.fetchall()

      with open('person01a.csv','wb') as csvfile:

        mywriter = csv.writer(csvfile,delimiter=',',quotechar='"',quoting=csv.QUOTE_ALL)

        for row in rows:
```

```
            perid = "PER01a" + str(row[0])

            fname = str(row[1])

            lname = str(row[2])

            dob = str(row[3])

        mywriter.writerow([perid,fname,lname,dob])
```

## 5. Code to Inject Programmatically Content into Neo4j

Besides the method for loading content into Neo4j using its command line utility to ingest CSV files, it is also possible to load that content programmatically by directly connecting to the source relational database and then invoking the methods of the **py2neo** library. The Python script below shows the use of the **cypher.execute** method.

```
    # 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
#
    import MySQLdb as mdb
    from py2neo import Graph,Node
    import time

    graph = Graph()

    cypher = graph.cypher
```

```
con = mdb.connect('localhost', 'root', '2MctlMctl!', 'portal02');

with con:
   cur = con.cursor()

   cur.execute("SELECT * FROM person01a")

   rows = cur.fetchall()

   t0 = time.clock()


   for row in rows:

      perid = "PER01a" + str(row[0])

      fname = str(row[1])

      lname = str(row[2])

      dob = str(row[3])

# version 3.0 of py2neo now uses Graph.run({query})

      cypher.execute("CREATE (a {perid:{a},fname:{b},lname:{c},dob:{d}})",
          a=perid,b=fname,c=lname,d=dob)

print time.clock() - t0, "seconds required to load 1000K records into neo4j server"
```

## 6. Code to Extract a Subset of Lines

During the testing, relatively large text files were generated containing 1 million lines or more. It later became quite clear that most word processors available in the various Linux distributions, such as gedit, leafpad, etc., are not well suited for manipulating files of that size. For example, a simple cut-and-paste operation to extract the first million lines out of a file containing 10 time bigger becomes nearly impossible when using a text processor. The same operation completes in the blink of an eye if carried out programmatically. The Python script below extracts the number of lines controlled by the **if** condition testing the value of the variable **counter**.

```
# 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
#
text_file = open('person01a01.nt', "w")
counter = 0
with open('person01a.nt') as f:
for line in f:
if(counter > 1000000):
break
text_file.write(line)
counter = counter + 1
```

## 7. Code to Programmatically Move Data from a MySQL Database into an RDF4J Graph Database

The first step in using graph databases as the backend data store of the DFS portal is the migration of the legacy source data to the graph repository. Ideally, one would like to be able to access the legacy database programmatically, retrieve the records from the pertinent tables via SQL SELECT statements, convert each record into a corresponding set of RDF triples, and finally inject them directly into the graph database. There are very robust and stable Java APIs for both MySQL and RDF4J, and the example code below highlights the various components needed to implement the type of data migration discussed above. It should be noted that this code is not optimized.

```
# 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.#


/*
 * A simple application to:
 * (1) read records from a source DB (implmented in MySQL)
 * (2) convert them into RDF triples
 * (3) and finally insert them into a target RDF4J triple store
 *
 * The parameters used for connecting to the MySQL
 * are read from an external ConfigFile.txt
 *
 * The executable jar has been produced using NetBeans 8.02
 *
 */
package readbyline;

/**
 *
 * @author Dr. Francisco Loaiza -- Institute for Defense Analyses
 */

// libraries for reading external text files
import java.io.*;
import java.util.Scanner;
import java.io.FileNotFoundException;
// libraries to connect to MySQL
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.ResultSet;
// libraries to connect to RDF4J
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.RepositoryConnection;
import org.eclipse.rdf4j.model.IRI;
import org.eclipse.rdf4j.model.ValueFactory;
import org.eclipse.rdf4j.repository.http.HTTPRepository;
// end of imports

public class ReadByLine {
public final static boolean DEBUGMODE=true;
public final static String myParams[] = new String[8];
    /**
     * @param args the command line arguments
     */
```

```
public ReadByLine() throws FileNotFoundException
  {

      String myComment = "#";
      Scanner linReader = new Scanner(new File("ConfigFile.txt"));
      Integer j = 0;
      while (linReader.hasNext())
      {
         String temp = linReader.nextLine();
         int pos = temp.indexOf(myComment);
         if(pos < 0)
         {
          myParams[j] = temp;
          j = j + 1;
         }
      }
      linReader.close();
  }
  public static void main(String args[])  throws FileNotFoundException
  {
      new ReadByLine();
      //for (String param : myParams) {
      //System.out.println(param);
      //}
      String url = myParams[0];
      //System.out.println(url);
      String table = myParams[1];
      //System.out.println(table);
      String user = myParams[2];
      //System.out.println(user);
      String password = myParams[3];
      //System.out.println(password);
      String lowPK = myParams[4];
      //System.out.println(lowPK);
      String highPK = myParams[5];
      //System.out.println(highPK);
      String style = myParams[6];
      //System.out.println(style);
      String target = myParams[7];
      //System.out.println(target);

      Connection conn = null;
      java.sql.Statement stmt = null;
      ResultSet rs = null;

      String rdf4jServer = "http://localhost:8080/rdf4j-server/";
      String repositoryID = myParams[7];

      String namespace = "http://Army.gov/Portal/ForceStructure/Person/meta#";
```

```
        Repository repo = new HTTPRepository(rdf4jServer, repositoryID);

        repo.initialize();

        RepositoryConnection connecto = repo.getConnection();

        ValueFactory f = connecto.getValueFactory();
        IRI context1 = f.createIRI("http://usPortal/demoContext1");
        try { conn =  DriverManager.getConnection(url, user, password);

          stmt = conn.createStatement();
          String SQLstring = "SELECT * FROM " + table + " WHERE perID >=" + lowPK + " AND perID
<= " + highPK;
          rs = stmt.executeQuery(SQLstring);

          System.out.println("\n");

           while(rs.next()) {
           //System.out.println(rs.getString(1) + " " + rs.getString(2) + " " + rs.getString(3) + " " +
rs.getString(4));

           String persona = "Person" + rs.getString(1);
           IRI person = f.createIRI(namespace + persona);

           IRI fname = f.createIRI(namespace + "fname");
           IRI lname = f.createIRI(namespace + "lname");
           IRI dob = f.createIRI(namespace + "dob");

           connecto.add(person, fname, f.createLiteral(rs.getString(2)),context1);
           connecto.add(person, lname, f.createLiteral(rs.getString(3)),context1);
           connecto.add(person, dob, f.createLiteral(rs.getString(4)),context1);

          }

          }catch (SQLException ex) {
      // handle any errors
           System.out.println("SQLException: " + ex.getMessage());
           System.out.println("SQLState: " + ex.getSQLState());
           System.out.println("VendorError: " + ex.getErrorCode());
          }
          finally {
      // it is a good idea to release
      // resources in a finally{} block
      // in reverse-order of their creation
      // if they are no-longer needed

          if (rs != null) {
          try {
             rs.close();
            } catch (SQLException sqlEx) { } // ignore
```

```
        rs = null;
       }

       if (stmt != null) {
       try {
           stmt.close();
          } catch (SQLException sqlEx) { } // ignore

       stmt = null;
   }
  }

  }
}
```

Most of the parameters needed for running the script, namely, the name of the MySQL server, the source table, the components of the connect string, and the values of the primary keys that correspond to the subset of source data that one wants to move into the graph database, are kept in a text file called **ConfigFile.txt**. An example of this file is provided below.

```
# param 01: Name of the source DB
jdbc:mysql://localhost:3306/portal02?useSSL=false
#
# param 02: Name of the source table
person01a
#
# param 03: Name of the DB user
<your db user name here>
#
# param 04: Password
<your db password here>
#
# param 05: lower value of the primary key
1000350005
#
# param 06: upper value of the primary key
1000750000
#
# param 07: serialization style
N3
#
# param 08: RDF4J target repository name
dem02
```

## 8. Code to Connect to PostgreSQL from Python

This Python module is used by subsequent Python code as a utility for connecting to PostgreSQL using the SQLAlchemy library.

```
# 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 Dr. Dale Visser -- Institute for Defense Analyses
 */


"""
Helper for connecting SQLAlchemy to a database.
"""
from os.path import basename

from sqlalchemy import create_engine


def connect(host, password, path_or_name, protocol, username):
    """
    Establish the database connection.
    :param protocol: 'postgresql' or 'sqlite'
    :param username: username for connecting to the database server, if
    applicable
    :param password: password for connecting to the database server, if
    applicable
    :param host: host of database server, if applicable
    :param path_or_name: for SQLite, the relative or absolute pathname of
    the database file; for PostgreSQL the name of the database
    :return: a tuple with the database name and connection object; in that order
    """
    if protocol == 'postgresql':
```

```
        db_name = path_or_name
    elif protocol == 'sqlite':
        db_name = basename(path_or_name)
    else:
        raise ValueError('Expected protocol to be postgresql or sqlite.')
    credentials = '{}:{}@{}'.format(username, password, host) \
        if (username and password and host) else ''
    uri = '{}://{}/{}'.format(protocol, credentials, path_or_name)
    engine = create_engine(uri)
    return db_name, engine
```

## 9. A command-line runnable Python Script for Generating a Large PostgreSQL "users" database

The script depends on the "begins"[57] and the "SQLAlchemy"[58] packages. It also uses a large plain-text file with a first name (all caps) per line.

```
# 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 Dr. Dale Visser -- Institute for Defense Analyses
 */


#!/usr/bin/env python3
"""
Utility script for setting up a database with a 'lots_of_users' table with as
many rows as one wants.
```

[57] https://pypi.python.org/pypi/begins

[58] http://www.sqlalchemy.org/

```
"""
from sys import stderr
from sqlalchemy import Table, Column, Integer, String, MetaData, select, func
from sqlalchemy.exc import ProgrammingError, IntegrityError
import begin
from connect import connect
from random import choice, randint
import time
from string import ascii_uppercase

CHUNK_SIZE = 100000
ENGINE = None
SIZE = CHUNK_SIZE


def define_table():
    """
    Generate the lots_of_users table object and associated metadata object.
    :return: a tuple with the metadata object as it's first element, and the
    table object as its second element
    :rtype tuple
    """
    metadata = MetaData()
    users = Table('lots_of_users', metadata,
                  Column('id', Integer, primary_key=True),
                  Column('first_name', String(16), nullable=False),
                  Column('last_name', String(60), nullable=False),
                  Column('date_of_birth', String(10), nullable=False))
    return metadata, users


def rand_letters(length):
    return ''.join([choice(ascii_uppercase) for _ in range(length)])


def rand_date():
    year = str(randint(2100, 2200))
    month = str(randint(1, 12))
    day = str(randint(1, 31))
    return '-'.join([year, month, day])


def read_in_first_names():
    with open('FirstNames.txt', 'r') as f:
        first_names = [line.split()[0] for line in f]
    print(len(first_names), "first names were read in.")
    return first_names


@begin.subcommand
def create():
```

```
    """
    Create a table and populate it with a large number of random users.
    :return: table reference
    :rtype Table
    """
    t0 = time.clock()
    print('Creating {} entries in lots_of_users.'.format(SIZE))
    metadata, users = define_table()
    metadata.create_all(ENGINE)
    first_names = read_in_first_names()
    generated = 0
    while generated < SIZE:
        chunk_size = min(SIZE - generated, CHUNK_SIZE)
        end = generated + chunk_size - 1
        print('Generating entries {} to {}...'.format(generated, end), end='')
        entries = [{'id': generated + i, 'first_name': choice(first_names),
                    'last_name': rand_last_name(),
                    'date_of_birth': rand_date()}
                   for i in range(chunk_size)]
        print(' adding entries to database table.')
        try:
            ENGINE.execute(users.insert(entries))
        except IntegrityError:
            print('Key exists. Drop table first.', file=stderr)
        generated += chunk_size
    print('Done.')
    print(time.clock() - t0, "seconds required to generate and load", SIZE,
          "records.")
    return users


def rand_last_name():
    return rand_letters(4) + str(randint(10000001, 900000001))


@begin.subcommand
def drop():
    """
    Drop the table if it exists.
    """
    _, users = define_table()
    try:
        users.drop(ENGINE)
    except ProgrammingError:
        print('Table already not in database.', file=stderr)


@begin.subcommand
def count():
    """
    Count the number of rows in the table.
```

```
    """
    _, users = define_table()
    try:
        for row in ENGINE.execute(select([func.count(users.c.id)])):
            print(row[0])
    except ProgrammingError:
        print('Need to create table first!', file=stderr)


@begin.start
@begin.convert(size=int)
def run(path_or_name='postgres', protocol='postgresql', username='postgres', password=None,
db_host='localhost', size=128000000):
    # """
    # Program entry point.
    # :param path_or_name: required parameter defining path to a SQLite
    # database file, or the name of a database within a PostgreSQL server
    # :param protocol: optional parameter; 'postgresql' or 'sqlite'; defaults
    # to 'postgresql'
    # :param username: optional parameter; needed for protocol=='postgresql'
    # defaults to 'postgres'
    # :param password: optional parameter; needed for protocol=='postgresql'
    # :param db_host: optional parameter; needed for protocol=='postgresql';
    # defaults to 'localhost'
    # :param size: number of rows to generate
    # """
    # pylint: disable=too-many-arguments, global-statement
    global ENGINE, SIZE
    SIZE = size
    _, ENGINE = connect(host=db_host, password=password,
                        path_or_name=path_or_name, protocol=protocol,
                        username=username)
```

## 10. GraphBuilder helper code for sql2rdf script

The script depends on the “rdflib”[59] and “SQLalchemy” libraries. The program is capable of inspecting a database from a SQLAlchemy connection and then mapping its tables and rows into a clean set of RDF statements ready for use in a graph database.

```
# 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 Dr. Dale Visser -- Institute for Defense Analyses
 */


"""
Helper constants and class for taking data from an RDBMS and generating RDF
triples.
"""
from os.path import split, join
from collections import OrderedDict
from rdflib import Graph, Literal, Namespace
from sqlalchemy import select, MetaData


def resource(label, index):
    """
    Generate a resource URI.
    :param label: the type of resource
    :param index: a unique index number of the resource
    :return: a constructed URI for the resource
    """
```

[59] https://pypi.python.org/pypi/rdflib

```
    return RESOURCE_BASE['{}_{}'.format(label, index)]


def make_predicate(verb):
    """
    Generate a predicate for a given verb.
    :param verb: verb expressed by predicate.
    :return: URI to be used as predicate for the given verb phrase.
    """
    return PREDICATE_BASE[verb]

RESOURCE_BASE = Namespace('http://ida.org/resource/')
PREDICATE_BASE = Namespace('http://ida2.org/predicate/')
HAS_NAME = make_predicate('hasName')
HAS_TABLE = make_predicate('hasTable')
HAS_RECORD = make_predicate('hasRecord')


class RDBMSGraphBuilder(object):
    """
    Helper class for taking a database's structure and data, and building an
    RDF graph from it.
    """

    identifier = 'temp_sqlite_disk_graph'

    def __init__(self, db_index, db_name, engine):
        self.graph = Graph()
        self.db_resource = resource('db', db_index)
        self.num_statements = 0
        self.add(self.db_resource, HAS_NAME, Literal(db_name))
        self.engine = engine
        self.metadata = MetaData()
        self.metadata.reflect(engine)

    def add(self, subject, predicate, object_value):
        """
        Helper for triples insertion.
        """
        self.graph.add((subject, predicate, object_value))
        self.num_statements += 1

    def add_table(self, table_index, table):
        """
        Add a table resource to the database resource
        :param table_index: a unique number associated with the table
        :param table: SQLAlchemy Table object
        :return: tuple with the table resource URI, and a list of predicate
        URIs to use for the various table columns
        """
        columns = [c.name for c in table.columns]
```

```
    column_predicates = [make_predicate(c) for c in columns]
    table_resource = resource('table', table_index + 1)
    self.add(self.db_resource, HAS_TABLE, table_resource)
    self.add(table_resource, HAS_NAME, Literal(table.name))
    return table_resource, column_predicates

def add_record(self, table_resource, table_name, row_number):
    """
    Add a row record into a table.
    :param table_resource: the table resource URI
    :param table_name: the table name
    :param row_number: the row number
    :return: the resource URI to use for the table row
    """
    row_resource = resource(table_name, row_number)
    self.add(table_resource, HAS_RECORD, row_resource)
    return row_resource

def add_value(self, row_resource, column_predicate, column_value):
    """
    Add a column value to a row.
    :param row_resource: the resource URI for the table row
    :param column_predicate: the column predicate
    :param column_value: the value (a number, boolean or string)
    """
    self.add(row_resource, column_predicate, Literal(column_value))

def process_tables(self):
    """
    Actually generate the RDF triples.
    """
    for table_index, table in enumerate(self.metadata.sorted_tables):
        table_resource, column_predicates = self.add_table(table_index,
                                                           table)
        select_stmt = select([table])
        result = self.engine.execute(select_stmt)
        for row_number, row in enumerate(result):
            row_resource = self.add_record(table_resource, table.name,
                                           row_number + 1)
            for predicate, value in OrderedDict(
                    zip(column_predicates, row)).items():
                self.add_value(row_resource, predicate, value)

def process_tables_incrementally(self, out_path,
                                 serializer='turtle',
                                 statements_per_file=100000):
    """
    Generate a series of files containing the RDF statements mapped from
    the database tables. Prints status messages as it goes along.
    :param out_path: the name of the file to output the serialized RDF
    graph to
```

```
        :param serializer: may be 'xml', 'pretty-xml', 'n3', 'nt', 'turtle',
        or 'trix'; defaults to 'turtle'
        :param statements_per_file: Approximate number of statements per
        file; defaults to 100,000
        """
        file_number = 0
        for table_index, table in enumerate(self.metadata.sorted_tables):
            table_resource, column_predicates = self.add_table(table_index,
                                                               table)
            result = self.engine.execute(select([table]))
            print('Ingesting table: {}'.format(table.name))
            for row_number, row in enumerate(result):
                row_resource = self.add_record(table_resource, table.name,
                                               row_number + 1)
                for predicate, value in OrderedDict(
                        zip(column_predicates, row)).items():
                    self.add_value(row_resource, predicate, value)
                    if len(self.graph) >= statements_per_file:
                        self.serialize_increment(file_number, out_path,
                                                 serializer)
                        file_number += 1
        if len(self.graph) > 0:
            self.serialize_increment(file_number, out_path, serializer)

    def serialize_increment(self, file_number, out_path, serializer):
        """
        Helper method for serializing incremental RDF files. Prints out a
        statement explaining what it's doing.
        :param file_number: starts at zero, increases from there
        :param out_path: original destination file name
        :param serializer: may be 'xml', 'pretty-xml', 'n3', 'nt', 'turtle',
        or 'trix'; defaults to 'turtle'
        """
        path_base, path_filename = split(out_path)
        segment_filename = '{}_{}'.format(str(file_number),
                                          path_filename)
        partial_file = join(path_base, segment_filename)
        print('  Writing {} RDF statements to {}'.format(len(self.graph),
                                                        partial_file))
        self.serialize(partial_file, serializer)
        self.graph = Graph()  # free up memory

    def serialize(self, out_path, serialization_format='turtle'):
        """
        Generate a serialized RDF file at the given path.
        :param out_path: where to write a file with a serialized
        representation of the graph.
        :param serialization_format: 'turtle', 'xml', 'n3', 'turtle', 'nt',
        'pretty-xml', or trix'; the default is 'turtle'
        """
        with open(out_path, 'wb') as out_file:
```

```
out_file.write(self.graph.serialize(format=serialization_format))
```

## 11. Sql2py runnable script

The script depends on the “connect” and “graph_builder” modules described above. The prerequisites for using the script are:

1. Python 3
2. A working database server to connect to. At present, it can be either SQLite (local only) or PostgreSQL (local or remote).
3. The following Python packages:
4. SQLAlchemy
5. rdflib
6. rdflib-sqlalchemy[60]
7. begins

The IDA team found the use of Anaconda[61] and the `conda` tool to perform software development in Python quite adequate. Since the last three packages are available only on PyPI, shell, commands to create the environment look something like this:

```
$ conda create -n my_env python=3.5 sqlalchemy
$ source activate my_env
$ pip install rdflib rdflib-sqlalchemy begins
```

### How to Use

`sql2rdf.py` is executable and has a help function:

```
$ ./sql2rdf.py --help
usage: sql2rdf.py [-h] [--protocol PROTOCOL] [--username USERNAME] [--
password PASSWORD] [--db-host DB_HOST]
                  [--serializer SERIALIZER]
                  PATH_OR_NAME INDEX OUT_PATH

positional arguments:
  PATH_OR_NAME
  INDEX
  OUT_PATH

optional arguments:
  -h, --help            show this help message and exit
  --protocol PROTOCOL   (default: postgresql)
  --username USERNAME, -u USERNAME
                        (default: postgres)
  --password PASSWORD, -p PASSWORD
```

[60] https://pypi.python.org/pypi/rdflib-sqlalchemy

[61] https://www.continuum.io/downloads

```
                        (default: None)
  --db-host DB_HOST, -d DB_HOST
                        (default: localhost)
  --serializer SERIALIZER, -s SERIALIZER
                        (default: turtle)
```

Other allowed values for `--serializer` are 'xml', 'pretty-xml', 'n3', 'nt', or 'trix'. `--protocol` can also be 'sqlite'. In that case, don't include `--username`, `--password`, or `--db-host`.

EXAMPLE: Suppose you have a local PostgreSQL instance name 'postgres', a user named db_guy, with password Pazzw0rd:

```
$ ./sql2rdf.py --username db_guy --password Pazzw0rd --serializer
pretty-xml postgres 3 db_3.xml
Ingesting data into graph.
That took 1287 ms.
Total RDF statements: 33227
Serializing graph to pretty-xml file: /tmp/out3.xml
That took 3792 ms.
Done. Total time elapsed = 5079 ms.
```


```
# 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 Dr. Dale Visser -- Institute for Defense Analyses
 */
```

```
#!/usr/bin/env python3
"""
Driver for converting RDBMS data to serialized RDF.
"""

from math import ceil
```

```
from time import perf_counter

import begin

from connect import connect
from graph_builder import RDBMSGraphBuilder


def milliseconds(initial, final):
    """
    Gives milliseconds between two performance counters.
    :param initial: initial timestamp
    :param final: final timestamp
    :return: difference in milliseconds, rounded up
    """
    return ceil(1000*(final - initial))


# pylint: disable=too-many-arguments
@begin.start
@begin.convert(index=int)
def run(path_or_name, index, out_path, protocol='postgresql',
        username='postgres', password=None, db_host='localhost',
        serializer='turtle'):
    """
    Defines the command-line interface entry point to this script.
    :param path_or_name: required parameter defining path to a SQLite
    database file, or the name of a database within a PostgreSQL server
    :param index: required parameter defining what the index number of the
    database should be in the RDF statements that are output
    :param out_path: required parameter giving the name of the file to output
    the serialized RDF graph to
    :param protocol: optional parameter; 'postgresql' or 'sqlite'; defaults
    to 'postgresql'
    :param username: optional parameter; needed for protocol=='postgresql'
    defaults to 'postgres'
    :param password: optional parameter; needed for protocol=='postgresql'
    :param db_host: optional parameter; needed for protocol=='postgresql';
    defaults to 'localhost'
    :param serializer: optional parameter; may be 'xml', 'pretty-xml', 'n3',
    'nt', 'turtle', or 'trix'; defaults to 'turtle'
    """
    init = perf_counter()
    print('RDF statements will be written in {} format.'.format(
        serializer))
    db_name, engine = connect(host=db_host, password=password,
                              path_or_name=path_or_name, protocol=protocol,
                              username=username)
    builder = RDBMSGraphBuilder(db_index=index, db_name=db_name,
                                engine=engine)
    builder.process_tables_incrementally(out_path, serializer)
```

```
graph_done = perf_counter()
print('Total RDF statements: {}'.format(builder.num_statements))
print('Done. That took {} ms.'.format(milliseconds(init, graph_done)))
```

## 12. Sql2Rdf.java

This code depends on the RDF4J SDK collection of jar files,[62] as well as slf4j-nop-1.7.10.jar[63] and postgresql-0.4.1212.jar.[64]

```
# 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.#


/*
 * A simple application to:
 * <ol>
 * <li>read records from a source DB (implmented in PostgreSQL)</li>
 * <li>convert them into RDF triples</li>
 * <li>and finally insert them into a target RDF4J triple store</li>
 * </ol>

 * @author Dr. Francisco Loaiza -- Institute for Defense Analyses
 * @author Dale Visser -- Institute for Defense Analyses
 */
package rdf4j.timer;
```

[62] http://rdf4j.org/download/

[63] https://www.slf4j.org/manual.html and https://mvnrepository.com/artifact/org.slf4j/slf4j-nop

[64] https://jdbc.postgresql.org/download.html

```
import java.io.File;
import java.io.FileNotFoundException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import org.eclipse.rdf4j.model.IRI;
import org.eclipse.rdf4j.model.Model;
import org.eclipse.rdf4j.model.ModelFactory;
import org.eclipse.rdf4j.model.ValueFactory;
import org.eclipse.rdf4j.model.impl.LinkedHashModelFactory;
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.RepositoryConnection;
import org.eclipse.rdf4j.repository.RepositoryException;
import org.eclipse.rdf4j.repository.manager.RepositoryManager;
import org.eclipse.rdf4j.repository.manager.RepositoryProvider;

public class Sql2Rdf implements Runnable {

    private static final String[] KEYS = {"db_url", "table", "user",
        "password", "lower_limit", "upper_limit", "format", "repo_name"};
    private final Map<String, String> myParams = new HashMap<>();

    public Sql2Rdf() {
        String myComment = "#";
        try (final Scanner lineReader = new Scanner(new File("/home/dale/"
                + "NetBeansProjects/rdf4j-timer/src/rdf4j/timer/"
                + "ConfigFile.txt"))) {
            Integer j = 0;
            while (lineReader.hasNext()) {
                String line = lineReader.nextLine();
                int pos = line.indexOf(myComment);
                if (pos < 0) {
                    this.myParams.put(KEYS[j], line);
                    j = j + 1;
                }
            }
//          System.out.println(this.myParams);
        } catch (FileNotFoundException fnf) {
            System.out.println("Couldn't find configuration file.");
            System.out.println(fnf.getMessage());
        }
    }

    public String get(String key) {
        return myParams.get(key);
    }
```

```
private static final String RDF4J_SERVER
      = "http://localhost:8080/rdf4j-server/";
private static final File RDF4J_LOC
      = new File("/usr/share/tomcat8/.RDF4J/server");

@Override
public void run() {
   String table = this.get("table");
   String id_lower_limit = this.get("lower_limit");
   String id_upper_limit = this.get("upper_limit");
   String SQLstring = "SELECT * FROM " + table + " WHERE id >="
         + id_lower_limit + " AND id <= " + id_upper_limit;
   System.out.println("SQL Query: " + SQLstring);
   String repositoryID = this.get("repo_name");
   RepositoryManager manager =
         RepositoryProvider.getRepositoryManager(RDF4J_SERVER);
   manager.initialize();
   Repository repo = manager.getRepository(repositoryID);
   repo.initialize();
   try (RepositoryConnection rdf4j_connection = repo.getConnection()) {
      System.out.println("Connected to rdf4j succesfully.");
      try (Connection jdbc_connection = this.connectToJdbc()) {
         System.out.println("Connected to JDBC successfully.");
         try (Statement stmt = jdbc_connection.createStatement()) {
            try (ResultSet results = stmt.executeQuery(SQLstring)) {
               System.out.println("Ready to add query results.");
               addResultsToRepository(results, rdf4j_connection);
            }
         }
      } catch (SQLException ex) {
         System.out.println("SQLException: " + ex.getMessage());
         System.out.println("SQLState: " + ex.getSQLState());
         System.out.println("VendorError: " + ex.getErrorCode());
      }
   } catch (RepositoryException ex) {
         ex.printStackTrace();
   }
}

private Connection connectToJdbc() throws SQLException {
   return DriverManager.getConnection(this.get("db_url"), this.get("user"),
         this.get("password"));
}

private static final String NAMESPACE
      = "http://Army.gov/Portal/ForceStructure/Person/meta#";
private static final String CONTEXT = "http://usPortal/demoContext1";
private static final int CHUNK_SIZE = 5000;
private static final ModelFactory MODEL_FACTORY
      = new LinkedHashModelFactory();
```

```
    private static void addResultsToRepository(final ResultSet results,
            final RepositoryConnection rdf4j_connection)
            throws RepositoryException, SQLException {
        ValueFactory factory = rdf4j_connection.getValueFactory();
        Model model = MODEL_FACTORY.createEmptyModel();
        IRI context = factory.createIRI(CONTEXT);
        IRI fname = factory.createIRI(NAMESPACE, "fname");
        IRI lname = factory.createIRI(NAMESPACE, "lname");
        IRI dob = factory.createIRI(NAMESPACE, "dob");
        int count = 0;
        while (results.next()) {
            IRI person = factory.createIRI(NAMESPACE,
                    "Person" + results.getString(1));
            addToModel(model, factory, person, fname, results, 2, context);
            addToModel(model, factory, person, lname, results, 3, context);
            addToModel(model, factory, person, dob, results, 4, context);
            count++;
            if (0 == count % CHUNK_SIZE) {
                System.out.println(count + " rows so far. About to add " +
                        model.size() + " statements to repository.");
                rdf4j_connection.add(model);
                model.clear();
            }
        }
        System.out.println("# of results processed: " + count);
    }

    private static void addToModel(Model model, ValueFactory valueFactory,
            IRI person, IRI predicate, final ResultSet results, int column,
            IRI context) throws SQLException {
        model.add(
                person,
                predicate,
                valueFactory.createLiteral(results.getString(column)),
                context);
    }

    public static void main(String args[]) {
        Sql2Rdf sql2rdf = new Sql2Rdf();
        long startTime = System.currentTimeMillis();
        sql2rdf.run();
        long stopTime = System.currentTimeMillis();
        System.out.println("Elapsed time: " + (stopTime - startTime) + " ms");
    }

}
```

Below is an example of the **ConfigFile.txt** that can be used to pass the various parameters needed by the program:

```
# param 01: Name of the source DB
jdbc:postgresql://localhost:5432/postgres
#
# param 02: Name of the source table
lots_of_users
#
# param 03: Name of the DB user
postgres
#
# param 04: Password
P@zzw0rd
#
# param 05: lower value of the primary key
0
#
# param 06: upper value of the primary key
7999999
#
# param 07: serialization style
N3
#
# param 08: RDF4J target repository name
dem04
```

## 13. Java program for timing a SPARQL Query

This program can be used to time the execution of SPARQL queries passed to the RDF4J graph database server programmatically. The SPARQL string is fully modifiable. The code has been tested with several and different schemata and queries.

```
    # 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 Dale Visser -- Institute for Defense Analyses
 */
```

```
package rdf4j.timer;

import org.eclipse.rdf4j.query.TupleQuery;
import org.eclipse.rdf4j.query.TupleQueryResult;
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.RepositoryConnection;
import org.eclipse.rdf4j.repository.manager.RepositoryManager;
import org.eclipse.rdf4j.repository.manager.RepositoryProvider;

/**
 *
 * @author Dale Visser
 */
public class CountDaleSparqlTimer {

    private static final String RDF4J_SERVER
            = "http://localhost:8080/rdf4j-server/";
    private static final String REPO_ID = "dem02";

    public static void main(String[] args) {
        RepositoryManager manager
                = RepositoryProvider.getRepositoryManager(RDF4J_SERVER);
        manager.initialize();
        Repository repo = manager.getRepository(REPO_ID);
        repo.initialize();
        try (RepositoryConnection conn = repo.getConnection()) {
            long startTime = System.currentTimeMillis();
            TupleQuery query = conn.prepareTupleQuery(
                    "SELECT (COUNT(?user) AS ?count) \n"
                    + "WHERE {\n"
                    + "  ?user <http://Army.gov/Portal/ForceStructure/Person/meta#fname> \"DALE\" .\n"
                    + "}");
            try (TupleQueryResult result = query.evaluate()) {
                while (result.hasNext()) {
                    System.out.println("Count: "
                            + result.next().getValue("count"));
                }
            }
            long stopTime = System.currentTimeMillis();
            System.out.println();
            System.out.println("Elapsed time: " + (stopTime - startTime) + " ms");
        }
    }
}
```

## 14. Maven Project File for Stardog Test Code

For testing Stardog, the IDA team leveraged Maven to easily pull in all dependencies and to facilitate the "build and run" cycle when working from the command-line. Note that for the last goal in the Maven file, the "copy dependencies" Maven plugin is included in the configuration, so that one can automatically have all the jar files needed for running the code in the lib/ folder.

```
# 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 Dr. Dale Visser -- Institute for Defense Analyses
 */


<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ida.org.stardogtimer</groupId>
  <artifactId>stardog-timer</artifactId>
  <packaging>jar</packaging>
  <version>0.1-SNAPSHOT</version>
  <name>stardog-timer</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  </properties>
  <repositories>
    <repository>
      <id>stardog-public</id>
      <url>http://maven.stardog.com</url>
```

```xml
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>com.complexible.stardog</groupId>
    <artifactId>client-http</artifactId>
    <version>4.2</version>
    <type>pom</type>
  </dependency>
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.4.1212</version>
  </dependency>
  <dependency>
     <groupId>org.slf4j</groupId>
     <artifactId>slf4j-simple</artifactId>
     <version>1.7.22</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
    <plugin>
      <artifactId>maven-dependency-plugin</artifactId>
      <executions>
        <execution>
          <phase>install</phase>
          <goals>
            <goal>copy-dependencies</goal>
          </goals>
          <configuration>
            <outputDirectory>${project.build.directory}/lib</outputDirectory>
          </configuration>
        </execution>
      </executions>
    </plugin>
```

```
    </plugins>
  </build>
</project>
```

## 15. StardogSql2SerializedRDF.java

This code is the first of two tasks tested against the Stardog graph database. This first task consisted of taking a PostgreSQL database of known schema, translating the data into RDF, and serializing the RDF into a collection of large, *but not too large*, TriG[65] files. The second task is covered in the next section.

Here is the command used to run the program once built. Note the directive to the JRE to use increased memory for the Java heap.

```
$ java -Xmx1600m -Dorg.slf4j.simpleLogger.logFile="test.log" -
Dorg.slf4j.simpleLogger.showDateTime=true -cp "target/stardog-timer-
0.1-SNAPSHOT.jar:target/lib/*"
ida.org.stardogtimer.StardogSql2SerializedRDF
```

```
    # 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 Dr. Dale Visser -- Institute for Defense Analyses
     */
package ida.org.stardogtimer;

import com.complexible.common.openrdf.model.Models2;
import com.complexible.common.rdf.model.Values;
import com.complexible.stardog.StardogException;
```

[65] https://en.wikipedia.org/wiki/TriG_(syntax)

```
import org.openrdf.model.IRI;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import static java.lang.System.out;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import org.openrdf.model.Model;
import org.openrdf.rio.RDFFormat;
import org.openrdf.rio.RDFHandlerException;
import org.openrdf.rio.Rio;
import org.slf4j.LoggerFactory;
import org.slf4j.Logger;

public class StardogSql2SerializedRDF implements Runnable {

    private static final Logger LOGGER = LoggerFactory.getLogger(StardogSql2SerializedRDF.class);

    private static final String[] KEYS = {"db_url", "table", "user",
        "password", "lower_limit", "upper_limit", "format", "repo_name"};
    private final Map<String, String> myParams = new HashMap<>();

    public StardogSql2SerializedRDF() {
        String myComment = "#";
        try (final Scanner lineReader = new Scanner(new File("/home/dale/"
                + "Documents/stardog-timer/ConfigFile.txt"))) {
            Integer j = 0;
            while (lineReader.hasNext()) {
                String line = lineReader.nextLine();
                int pos = line.indexOf(myComment);
                if (pos < 0) {
                    this.myParams.put(KEYS[j], line);
                    j = j + 1;
                }
            }
//            System.out.println(this.myParams);
        } catch (FileNotFoundException fnf) {
            System.out.println("Couldn't find configuration file.");
            System.out.println(fnf.getMessage());
        }
    }

    public String get(String key) {
        return myParams.get(key);
    }
```

```
private static final String STARDOG_SERVER = "http://192.168.59.104:5820/";

@Override
public void run() {
   String table = this.get("table");
   String id_lower_limit = this.get("lower_limit");
   String id_upper_limit = this.get("upper_limit");
   String SQLstring = "SELECT * FROM " + table + " WHERE id >="
         + id_lower_limit + " AND id <= " + id_upper_limit;
   LOGGER.info("SQL Query: {}", SQLstring);
   try (java.sql.Connection jdbc_connection = this.connectToJdbc()) {
      LOGGER.info("Connected to JDBC successfully.");
      try (Statement stmt = jdbc_connection.createStatement()) {
         try (ResultSet results = stmt.executeQuery(SQLstring)) {
            LOGGER.info("SQL query executed. Ready to "
                  + "add results to Stardog.");
            writeResults(results);
         }
      }
   } catch (SQLException ex) {
      LOGGER.info("SQLException: {}\nSQLState: {}\nVendorError: {}",
            ex.getMessage(), ex.getSQLState(), ex.getErrorCode());
   }
}

private java.sql.Connection connectToJdbc() throws SQLException {
   return DriverManager.getConnection(this.get("db_url"), this.get("user"),
         this.get("password"));
}

private static final String NAMESPACE
      = "http://Army.gov/Portal/ForceStructure/Person/meta#";
private static final String CONTEXT = "http://usPortal/demoContext1";
private static final int CHUNK_SIZE = 128 * 1024;

private static void writeResults(final ResultSet results)
      throws SQLException {
   LOGGER.info("About to process results {} rows at a time.", CHUNK_SIZE);
   IRI context = Values.iri(CONTEXT);
   IRI fname = Values.iri(NAMESPACE, "fname");
   IRI lname = Values.iri(NAMESPACE, "lname");
   IRI dob = Values.iri(NAMESPACE, "dob");
   int count = 0;
   Model model = Models2.newModel();
   try {
      while (results.next()) {
         IRI person = Values.iri(NAMESPACE,
               "Person" + results.getString(1));
         add(model, person, fname, results, 2, context);
         add(model, person, lname, results, 3, context);
```

```
            add(model, person, dob, results, 4, context);
            count++;
            if (0 == count % CHUNK_SIZE) {
                writeTriplesAndClear(count, model);
            }
        }
        writeTriplesAndClear(count, model);
    } catch (StardogException se) {
        LOGGER.info("StardogException occured. count = {}", count);
        se.printStackTrace(System.out);
    } catch (IOException ioe) {
        LOGGER.info("IOException: {}", ioe.getMessage());
        ioe.printStackTrace(System.out);
    }
    LOGGER.info("# of results processed: {}", count);
}

private static void writeTriplesAndClear(int count, Model model) throws RDFHandlerException,
IOException {
    LOGGER.info("{} rows so far. Saving to disk.", count);
    try (FileOutputStream fos = new FileOutputStream(
            "/home/dale/out" + count + ".trig")) {
        Rio.write(model, fos, RDFFormat.TRIG);
    }
    LOGGER.info("Save completed.");
    model.clear();
}

private static void add(Model model,
        IRI person, IRI predicate, final ResultSet results, int column,
        IRI context) throws SQLException {
    model.add(person, predicate, Values.literal(results.getString(column)),
            context);
}

public static void main(String args[]) {
    StardogSql2SerializedRDF sql2rdf = new StardogSql2SerializedRDF();
    long startTime = System.currentTimeMillis();
    sql2rdf.run();
    long stopTime = System.currentTimeMillis();
    LOGGER.info("Elapsed time: {} ms", (stopTime - startTime));
}

}
```

Below is an example of a **ConfigFile.txt** file that gets read by the above code:

```
# param 01: Name of the source DB
jdbc:postgresql://localhost:5432/postgres
#
# param 02: Name of the source table
```

```
lots_of_users
#
# param 03: Name of the DB user
postgres
#
# param 04: Password
P@zzw0rd
#
# param 05: lower value of the primary key
0
#
# param 06: upper value of the primary key
3999999
#
# param 07: serialization style
N3
#
# param 08: Stardog target repository name
new_users_db
```

## 16. StardogTrig2Db

This code is for the 2nd measured Stardog operation, which is taking a collection of Trig files stored in a filesystem accessible to the Stardog server process, and initializing a new Stardog database with them. The reason this approach is taken, is that it was learned through trial and error, and then much reading of the Stardog forums, that Stardog's transaction system prevents the ingestion of large quantities of data all at once on a live database.

Here is an example of how this code is launched from the command-line, to produce output usable for extracting performance data:

```
$ java -Dorg.slf4j.simpleLogger.logFile="test.log" -
Dorg.slf4j.simpleLogger.showDateTime=true -cp "target/stardog-timer-
0.1-SNAPSHOT.jar:target/lib/*" ida.org.stardogtimer.StardogTrig2Db
```

```
# 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 Dr. Dale Visser -- Institute for Defense Analyses
         */
package ida.org.stardogtimer;

import com.complexible.stardog.api.admin.AdminConnection;
import com.complexible.stardog.api.admin.AdminConnectionConfiguration;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import org.slf4j.LoggerFactory;
import org.slf4j.Logger;

public class StardogTrig2Db implements Runnable {

    private static final Logger LOGGER
            = LoggerFactory.getLogger(StardogTrig2Db.class);

    private static final String[] KEYS = {"db_url", "table", "user",
        "password", "lower_limit", "upper_limit", "format", "repo_name"};
    private final Map<String, String> myParams = new HashMap<>();

    public StardogTrig2Db() {
        String myComment = "#";
        try (final Scanner lineReader = new Scanner(new File("/home/dale/"
                + "Documents/stardog-timer/ConfigFile.txt"))) {
            Integer j = 0;
            while (lineReader.hasNext()) {
                String line = lineReader.nextLine();
                int pos = line.indexOf(myComment);
                if (pos < 0) {
                    this.myParams.put(KEYS[j], line);
                    j = j + 1;
                }
            }
        } catch (FileNotFoundException fnf) {
            System.out.println("Couldn't find configuration file.");
            System.out.println(fnf.getMessage());
        }
```

```
    }

    public String get(String key) {
        return myParams.get(key);
    }

    private static final String STARDOG_SERVER = "http://192.168.59.104:5820/";

    @Override
    public void run() {
        String repositoryID = this.get("repo_name");
        try (AdminConnection connection
                = AdminConnectionConfiguration.toServer(STARDOG_SERVER)
                .credentials("admin", "admin").connect()) {
            LOGGER.info("AdminConnection obtained.");
            Path folder = new File("/home/dale/Documents/trig").toPath();
            Path[] trig_files = Files.list(folder)
                    .filter(p -> p.toFile().isFile()).toArray(Path[]::new);
            LOGGER.info("About to create DB and load {} TRIG files.",
                    trig_files.length);
            connection.disk(repositoryID).create(trig_files);
        } catch (IOException ioe) {
            LOGGER.info("IOException: {}", ioe.getMessage());
            ioe.printStackTrace(System.out);
        }
    }

    public static void main(String args[]) {
        StardogTrig2Db sql2rdf = new StardogTrig2Db();
        long startTime = System.currentTimeMillis();
        sql2rdf.run();
        long stopTime = System.currentTimeMillis();
        LOGGER.info("Elapsed time: {} ms", (stopTime - startTime));
    }

}
```

# 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* |
|---|---|---|---|---|---|---|
| Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.** | | | | | | |
| 1. REPORT DATE (DD-MM-YY)<br>24-02-2017 | | | 2. REPORT TYPE<br>Final | | | 3. DATES COVERED (From – To) |
| 4. TITLE AND SUBTITLE<br>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 | | | | | | 5a. CONTRACT NUMBER<br>HQ0034-14-D-0001<br>5b. GRANT NUMBER<br>5c. PROGRAM ELEMENT NUMBERS |
| 6. AUTHOR(S)<br>Francisco L. Loaiza-Lemos, Dale Visser | | | | | | 5d. PROJECT NUMBER<br>BC-5-4277<br>5e. TASK NUMBER<br>5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES<br>Institute for Defense Analyses<br>4850 Mark Center Drive<br>Alexandria, VA 22311-1882 | | | | | | 8. PERFORMING ORGANIZATION REPORT NUMBER<br>D-8345<br>H 2017-000092 |
| 9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)<br>Mr. Bruce Haberkamp<br>Army CIO/G-6 (SAIS-AOD)<br>5850 23rd Street, Bldg. 220, Ft. Belvoir, Virginia 20060-5832 | | | | | | 10. SPONSOR'S / MONITOR'S ACRONYM<br>SAIS-AOD<br>11. SPONSOR'S / MONITOR'S REPORT NUMBER(S) |
| 12. DISTRIBUTION / AVAILABILITY STATEMENT<br>Approved for public release; distribution is unlimited. | | | | | | |
| 13. SUPPLEMENTARY NOTES<br>Project Leader: Francisco L. Loaiza-Lemos | | | | | | |
| 14. ABSTRACT<br>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 goal of this phase of the study is to develop objective metrics and scenarios to test the ability of graph databases to represent and generate force structure data products. | | | | | | |
| 15. SUBJECT TERMS<br>Graph database, Resource Description Framework (RDF), Not only SQL (NoSQL), relational database, RDF triple, RDF triple store, Application Program Interface (API), data interoperability, 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<br>Mr. Bruce Haberkamp | |
| a. REPORT<br>Unclassified | b. ABSTRACT<br>Unclassified | c. THIS PAGE<br>Unclassified | Unlimited | 98 | 19b. TELEPHONE NUMBER (Include Area Code)<br>703-545-1464 | |