INSTITUTE FOR DEFENSE ANALYSES

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

Francisco L. Loaiza-Lemos, *Project Leader*

Dale Visser

Russell J. Smith

March 30, 2018

Approved for public release; distribution is unlimited.

IDA Document
D-8980

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

About This Publication

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

Acknowledgments
Steven P. Wartik

For more information:

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

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

INSTITUTE FOR DEFENSE ANALYSES

IDA Document D-8980

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

Francisco L. Loaiza-Lemos, *Project Leader*

Dale Visser

Russell J. Smith

Executive Summary

This document was prepared by the Institute for Defense Analyses (IDA) in support of the FY16 Army analysis "*Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure (DFS) Portal Implementation*."

This document constitutes the final report under the project description and addresses the analysis' objective of assessing the maturity and applicability of graph database technology as a practicable materiel solution that reflects legacy system realities and that can effectively and efficiently deliver the needed *at-rest* and *in-motion* force structure products for the planned Army DFS portal.

Specifically, the final report provides a summary of the technical assessments conducted during the execution of the project, the associated conclusions and recommendations, and a short exploration of additional technologies and data representations that may be appropriate for the implementation of the planned Army DFS Portal. The IDA team applied rapid prototyping techniques as part of the continuing evaluation of technologies covered during the analysis. The team used data collected during those activities to continue maturing the decision process needed to determine the best-of-breed options. The assessments leverage the metrics elaborated in preceding phases of the analysis, which were documented in the previous three deliverables. [1,2,3]

Background

The final phase of the analysis is aligned with the goals and objectives of the Department of Defense (DoD) as expressed in its Global Force Management Data Initiative (GFM DI), whereby DoD is seeking the standardization of all authorized force structure data so that it can be understandable to, and usable by, both warfighting and business

[1] IDA Document D-8345, *Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure (DFS) Portal Implementation: Part 1, Preliminary Characterization of Data Sources, Representation Options, Test Scenarios and Objective Metrics*, F. Loaiza, D. Visser, February 24, 2017.

[2] The second deliverable was IDA Document D-8516, *Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure (DFS) Portal Implementation: Part 2, Technical Feasibility, Affordability, and Architecture Integration Options,* F. Loaiza, D. Visser, June 1, 2017.

[3] The third deliverable was IDA Document D-8852, *Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure (DFS) Portal Implementation: Part 3, Risks, Mitigation Approach, and Roadmap*, F. Loaiza, R. Smith, December 29, 2017.

systems across the DoD Enterprise.[4] As noted in the previous deliverables under this project, the challenge in all of the related activities is the harmonization of data that currently resides in a large number of relational legacy systems so that it can be readily used in the generation of *at-rest* and *in-motion* force structure products. An additional challenge is to identify those technologies that can readily deliver the type of performance needed to implement interactive web-based solutions.

The main motivation for exploring graph database technology has been its potential for cost reduction along with the procedural simplicity of an approach that directly recasts the source data from legacy relational systems in the form of Resource Description Framework (RDF) triples,[5] collects them in a graph data store, and then uses the triples to generate the force structure products. However, as has been noted in the previous phases of the analysis, without special-purpose hardware the performance of standard graph database implementations may not be adequate for interactive human-in-the-loop implementations. In this final phase of the analysis, therefore, we explored a thread already initiated in the third deliverable—the use of *key:value* pair representations, such as the one used in JavaScript Object Notation (JSON) serializations,[6] in combination with search engines that use a record-level inverted index approach, such as the open-source, enterprise search platform Apache Solr,[7] for the purpose of assessing the feasibility of achieving industry-standard data retrieval times—i.e., less than a second per query—even in the absence of special-purpose hardware and software and handling large data volumes.

Document Structure

This document is organized as follows:

1. Section 1 presents a summary of the analytical results from the previous three deliverables, together with the main conclusions and recommendations that still remain valid at the end of the study.
2. Section 2 documents the use of record-level inverted index search engines, such as Apache Solr, to power the fast data retrieval operations needed in the planned Army DFS Portal implementation.
3. Section 3 provides the final set of conclusions and recommendations for the whole study.

---

4 http://www.prim.osd.mil/init/init_osdmanpower.html

5 https://www.w3.org/RDF/

6 "The JSON Data Interchange Format" (PDF). ECMA International. October 2013. Retrieved 23 September 2016.

7 http://lucene.apache.org/solr/

4. Appendix A covers in detail the results obtained when using JSON-style *key:value pair* data representations, coupled with the inverted index search engine Apache Solr. Specifically, the discussion highlights the potential offered by this combination to achieve substantial improvements in the efficiency of data access and retrieval when dealing with highly nested data as it exists in force structure representations. The experimental results employ the sample data from the "six degrees of separation" (SDOS) use case that the IDA team used in earlier phases of the study to stress the performance of open source and commercial implementations.
5. Appendix B contains Python and Hypertext Preprocessor (PHP) scripts that were used for the tests described in Appendix A. The code is licensed for free reuse, and it is intended to help other groups in their evaluations.

Scope

As in the previous deliverables, 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, 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 both be generated and would be accessible as inputs for subsequent manipulations required by the graph database approach.

Although other serializations are currently supported by Apache Solr, the conceptual simplicity of the JSON implementation, as well as the ease with which it can be transformed into other representations, such as RDF triples, or even SQL statements, together with its broad acceptance in the context of web application development, made this an optimal choice for assessing the feasibility of the approach. The IDA team, therefore, feels that no loss of generality is incurred by narrowing the scope of the assessment to just one implementation of the *key:value* pair data representation.

As noted in the preceding phases of the study, all tests were performed using off-the-shelf, standard computer equipment. The assessment of potential benefits associated with specialized hardware and software alternatives for use in combination with graph databases was deemed outside of the scope of the analysis, mainly because so many other components of the full-solution architecture for the planned Army DFS portal are still undefined.

Finally, the data sets used in the analysis, as well as the reported performance of the various applications tested, are intended only to demonstrate the feasibility of the proposed approaches for how to leverage the technologies being assessed—e.g., graph databases, alternate data representations, inverted index search engines—and should not be interpreted as reference performance benchmarks for actual implementation.

Analytical Approach

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

- What are the lessons learned from the previous phases of the analysis and how can they help inform the decision process for determining the optimal mix needed to implement the planned Army DFS portal?
- How can other data representations of the RDF triples content be leveraged to implement a semantic layer that aids in the harmonization of data from multiple disparate sources?
- What additional technologies can be brought to bear so that the potential benefits associated with the use of graph databases will not be negated by poor data access and retrieval performance, specifically, in the context of interactive web applications?
- What additional steps should be taken to facilitate the adoption of graph databases as part of the overall solution architecture supporting the Army DFS Portal?
- What are the enterprise-wide implications for the Army of adopting a graph database approach?

Conclusions and Recommendations

Based on the analytical work performed during the final phase of the study, the IDA team concluded the following:

- Search engines – such as Apache Solr – that leverage inverted index data structures and can store data expressed in the form of *key:value* pairs have achieved a high degree of maturity and acceptance in the commercial world. Their ability to handle data volumes of the magnitude expected for the planned Army DFS portal is adequate, even when the Solr server runs on standard hardware.
- Interactive web applications that use an Apache Solr server loaded with RDF triples converted into JSON documents made up of *key:value* pairs are feasible, and the data retrieval performance observed in preliminary tests is consistently well under one second per query, thus easily satisfying industry standards for interactive web application development and use.

For the final phase of the study, the recommendations are as follows:

- The solution architecture for the Army DFS portal should include technologies such as inverted index search engines, coupled with data representations compatible with such engines, to minimize the risk of the poor data retrieval performance associated with most graph database implementations.
- Processes for transforming force structure data from one representation into another should be identified early in the planning for the Army DFS to ensure that each of the technologies included in the solution architecture can be optimally exploited.
- Simulation techniques should be employed to assess the impact of hybrid solution architectures on the concept of operations of the Army DFS portal.

Contents

Figures and Tables

1. Summary of Previous Conclusions and Recommendations

This chapter reviews the conclusions and recommendations made during the preceding phases of the analysis. The details of each of the underlying analytical results encapsulated in the summarizations provided here are available in the respective deliverables.[8]

A. First Deliverable: Preliminary Characterization of Data Sources, Representation Options, Test Scenarios, and Objective Metrics

Table 1-1 summarizes the conclusions and recommendations made during the first phase of the analysis. The first phase was concentrated on discovering the potential benefits and risks associated with the use of a graph database approach for the planned Army DFS portal. The analytical results documented possible risks and whether any of them could rise to the level of a "show stopper" for the approach under consideration. The IDA team also attempted to identify the most likely types of scenarios in which the graph database approach could be used.

Table 1-1. Summary of Conclusions and Recommendations for the First Deliverable

Conclusions	
Subject	**Comment**
• Availability and Maturity of Graph Database Implementations	• 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.
• Applicable Scenarios for Graph Database Use	• 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.
• Fitness of objective Metrics	• The objective metrics developed in this phase of the study provide a good road map for evaluating 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 on whether or not to adopt a graph database approach.

[8] See footnotes 1, 2 and 3.

Recommendations	
Subject	**Comment**
• Good Performance and Robust APIs	• Both proprietary and open source graph database implementations need to be thoroughly evaluated with respect to their performance for loading and retrieving data in a high-traffic portal and to the robustness of their application program interfaces (APIs), specifically with respect to their support for commonly used scripting languages, e.g., Python, Java, etc.
• Flexible and Broad Risk Mitigation	• Risk mitigation strategies must be developed to cope with all potential risks that may arise from the adoption of graph database technology as a materiel solution for the planned Army DFS portal. In some cases, and for specific purposes, the solution architecture may require a mixture of technologies that better cope with the known weaknesses of the current graph database implementations.

Finally, the analysis explored applicable objective metrics that should be considered when assessing the maturity and applicability of the graph database approach for an implementation of the planned Army DFS portal. As noted in the recommendations, under the assumption that the majority of the applications to be developed in support of the planned Army DFS portal would include interactive web applications, the IDA team recommended emphasizing both the performance characteristics offered by the available implementations and developing a broad and flexible risk mitigation strategy.

B. Second Deliverable: Technical Feasibility, Affordability, and Architecture Integration Options

Table 1-2 summarizes the recommendations made after the second phase of the analysis. The second phase was concentrated on understanding whether or not the data structures used by the Army legacy relational databases would be easily re-expressed as subgraphs, so that their data content could then be placed in RDF triple stores – one of the most common implementations of the graph database paradigm.

As part of the analysis, the IDA team explored the choices available for developing a time- and cost-efficient data conversion process from relational tables into RDF triples for the data resident in Army legacy relational data stores. The team also identified choices available for representing Army force structure legacy data in the form of RDF triples that would negatively affect data access and retrieval performance. Finally, the IDA team investigated the options for a solution architecture that can support the integration of graph databases into the mix of technologies needed to implement the planned Army DFS Portal.

Table 1-2. Summary of Conclusions and Recommendations for the Second Deliverable

Subject	Comment
Conclusions	
Subject	**Comment**
• RDF triples data representation capabilities	• All data structures likely to be found in pertinent Army legacy relational data stores – namely, those containing the source force structure data needed to populate the planned Army DFS Portal – can be re-expressed in a straightforward manner using RDF triples. The difference in the degree of complexity of the transformation chosen for the relational data structures obeys strategic considerations, such as reuse and expansion of the data to satisfy novel and emerging uses.
• Re-expressing the semantics of relational data stores in graph databases	• A "semantic layer" in the form of an appropriately sized ontology is quite useful for organizing the resources in an RDF triple store in the same way that data is bundled in relational data stores under the concept of a "table." The semantic layer could also be used to retain traceability back to the data sources.
• Enhancing data retrieval performance	• Certain types of data structures common in relational data stores can lead to very poor data retrieval performance – such as in the canonical example of multiple layers of node dependencies found in networks which has been popularized under the rubric of "six degrees of separation." Pre-filtering and the use of materialized views essentially eliminate the performance issue in the relational stores, although they reduce flexibility and add complexity to the physical schema. Similar approaches can also be used to improve the performance of RDF triple stores, but the downside implications may be handled more elegantly through judicious use of federated triple stores and special-purpose hardware and software.
• Impact of DFS portal concept of operations on technologies mix.	• The solution architecture options that can support the integration of graph databases in the mix of technologies needed to implement the planned Army DFS Portal are generally satisfactory, but a final determination of optimal choices will require the inclusion and analysis of the concept of operations for the planned DFS portal and the timelines associated with the key Army information systems.
• Impact of DFS portal concept of operations on choice of graph database implementation	• The selection of best-of-breed options may be more sensitive to the concept of operations for the planned DFS portal than to factors of size, scalability, and data retrieval performance.
Recommendations	
Subject	**Comment**
• Evaluation of graph databases implementations, as well as other NoSQL options	• Continue the evaluation of available graph database implementations, both proprietary and open source, and expand the scope to include other promising NoSQL choices.
• Use of rapid prototyping techniques to identify optimal choices	• Continue using rapid prototyping techniques to collect performance statistics that can inform both the selection process of the optimal graph database implementation and its integration into the mix of technologies needed to implement the planned Army DFS Portal.

C. Third Deliverable: Risks, Mitigation Approach, and Roadmap

Table 1-3 summarizes the recommendations made during the third phase of the analysis. The third phase of the analysis was concentrated on obtaining a more definitive understanding of the main technical risks associated with the use of graph databases as part of the technology mix supporting the Army DFS Portal. The IDA team also sought to define the mitigation approaches that would best preserve the potential benefits associated with graph databases while minimizing the unavoidable technical risks associated with the graph database technology. Finally, the analysis explored options regarding available implementation roadmaps that would be most appropriate in light of all of the risks and alternatives, as well as key steps needed to facilitate the adoption of graph databases as part of the overall solution architecture supporting the Army DFS Portal.

Table 1-3. Summary of Conclusions and Recommendations for the Third Deliverable

Conclusions	
Subject	**Comment**
• Graph databases main technical risk	• As briefly noted in the first and second deliverables, the main risk associated with the adoption of graph databases when compared to relational data stores in the context of massive graphs is their inferior performance with respect to data retrieval and complex query execution. For interactive applications, any data storage and retrieval technology that requires more than one or two seconds to deliver the answer is unlikely to be a strong contender in the solution architecture that supports those use cases.
• Emerging risk mitigation alternatives	• Some proprietary graph database solutions for “big data” are reaching a sufficient level of maturity to be competitive with relational data stores in terms of performance. Specifically, the combination of graph databases and frameworks for distributed storage and processing, such as Apache Hadoop and Apache Spark, make it possible to efficiently partition very large datasets to compensate for any slowdowns caused by the size of the graphs.
• Additional ways of capturing relational data store semantics	• The idea of a “semantic layer” for organizing the resources in an RDF triple store can be readily implemented using alternative data representations that are not only closely related to the graph formalism – and, therefore, can be readily converted back and forth – but that also can be directly processed using a programming language (e.g., Prolog).
• Graph databases as the best alternative to costly and time consuming Extraction, Transformation and Loading (ETL)	• The key rationale for using graph databases is mainly to enable the cost-effective handling of legacy data, bypassing the laborious and expensive extraction, transformation, and loading (ETL) associated with traditional approaches, and said rationale is supported by all the findings obtained so far.
Recommendations	
Subject	**Comment**
• Additional exploration of alternatives	• Conduct additional comparisons regarding the use of other programming languages and data representations for the purpose of implementing a “semantic layer” as part of the graph database solution.
• Explore “big data” options	• Explore applicable emerging “big data” solutions with regard to their applicability in a future implementation of the Army DFS Portal.

2. Leveraging *key:value* Pair Representations to Improve Search Performance

A. Background

As was noted in the previous deliverables, [9] one of the main concerns associated with the adoption of graph database technology is that most available implementations tend to be slower than standard relational database engines. This can become problematic when manipulating large data sets via interactive solutions that use a graph database as their main back-end data store. In all fairness, it should be noted that there are well-documented approaches to boosting the performance of graph databases, ranging from using specialized hardware with large numbers of cores per CPU and big RAM capacity to using software platforms such as Hadoop, which allow data sets with hundreds of billions of subgraphs to be optimally partitioned into large numbers of smaller but very efficient computing nodes that can then operate in parallel.

We also noted in the preceding analytical results that although the data representation via RDF triples is quite appealing due to its being a well-established standard with broad support, not only in the commercial but also in the academic world, other data representations, such as the clauses used in Prolog's knowledgebases and the serialization employed in JavaScript Object Notation (JSON) and JSON for Linked Data (JSON-LD), also offer capabilities that may be optimal for solving some types of similar problems. That these representations can be easily transformed into one another should also be kept in mind, since the adoption of RDF as the baseline will not, therefore, automatically negate the ability to use alternative data representations that may be best suited for specific processes.

B. The Inverted Index Data Structure

With the preceding background in mind, the IDA team conducted a short review of leading search engine technologies that have achieved substantial market penetration and a high degree of maturity, to assess whether they would be suitable for integration into the solution architecture for the Army DFS portal—specifically, the IDA team sought to determine whether any of these search engines would be suitable for situations involving

[9] See Footnotes [1], [2] and [3] above.

very large datasets that must be interactively searched and manipulated, that is, where query response times must be less than two seconds.

The ever-growing volume of unstructured data has created a demand for algorithms that can quickly identify content related to a specific keyword or concept. Document search engines can use either forward indexing—an approach that maps the content of a document to the set of words that appears in it—or inverted indexing—an approach that combines all the lists of words extracted from the individual documents into a master list and then links each word to every document it appears in. [10]

Searching on the basis of a forward-indexing scheme requires searching each list of words extracted from the respective documents to find out whether it contains the keyword being used in the search. For large collections of documents and large lists of words per document, the search time can become prohibitively long. In addition, this type of engine does not automatically keep track of previous results but traverses each list anew for every query.

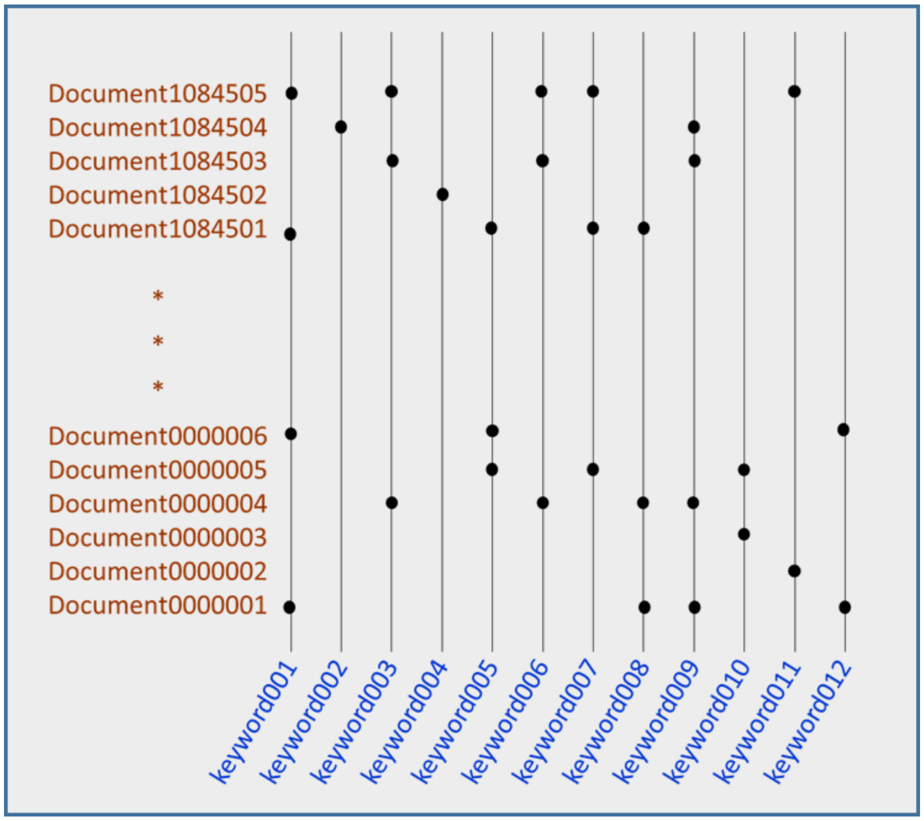
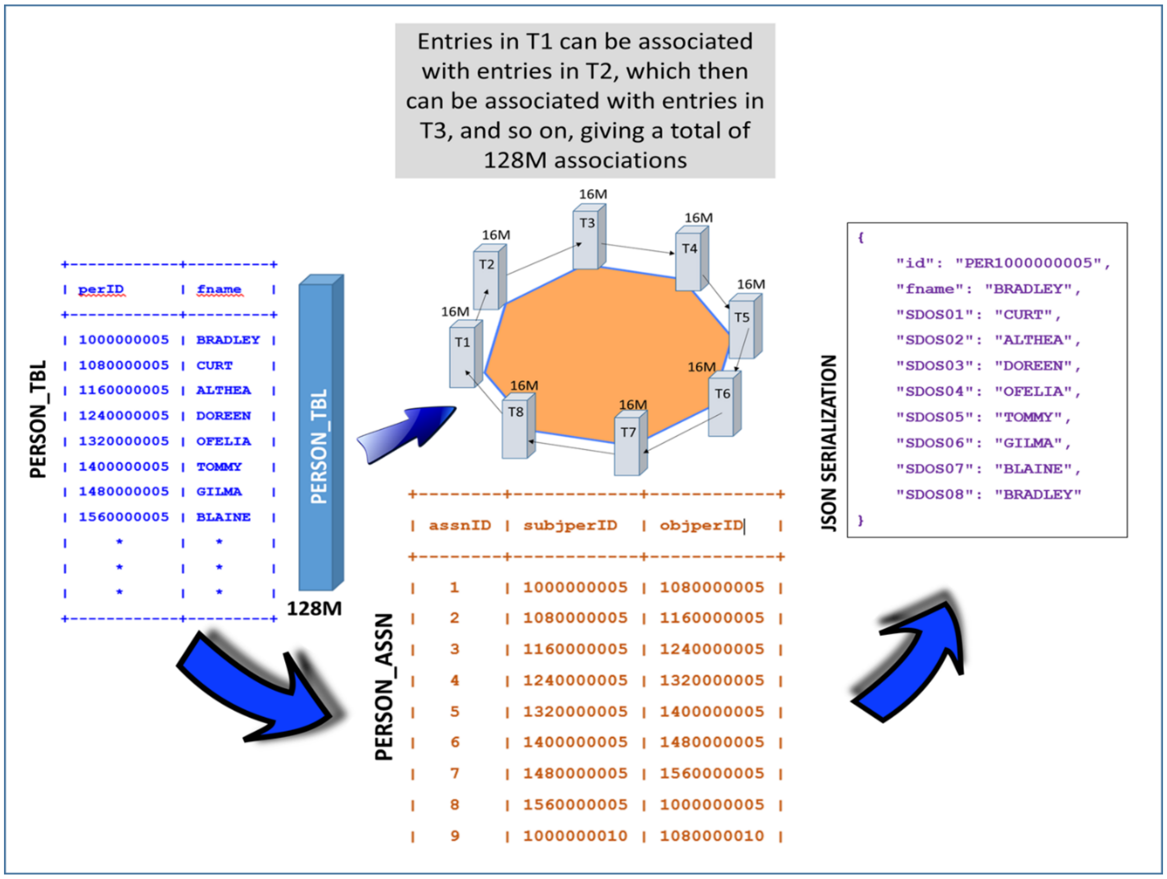


Figure 2-1. Notional Depiction of an Inverted Indexing Approach

[10] The master list is normally scrubbed so that words with high frequency but little semantic content, e.g., **it**, **in**, **the**, **a**, **as**, **for**, **that**, etc., are removed. Further conditioning of the master list may be applied to enhance the ability to retrieve pertinent documents using this approach.

An inverted index algorithm, on the other hand, will be much more efficient for this type of search. This can be easily understood using the notional depiction given in Figure 2-1 of what an inverted index set looks like. As shown therein, given a set of keywords—which usually are automatically generated from the content of each document, as was mentioned above—the engine associates to each keyword a list of the documents in which the keyword appears. The number of document entries linked to each keyword will be, on average, much smaller than the total number of documents being searched. When one needs to find which documents contain a given keyword, the engine only needs to fetch the set of documents already associated with the keyword and display it. Because each list of documents is a set in the mathematical sense, it is also easy to see how using set intersection, set union, set difference, etc., allows the engine to retrieve the list of documents that contain all the keywords given in the query (set intersection) or any keyword in the list of keywords (set union), etc.

C. The “Six Degrees of Separation” (SDOS) Use Case in JSON


Entries in T1 can be associated with entries in T2, which then can be associated with entries in T3, and so on, giving a total of 128M associations
16M
T1
T2
T3
T4
T5
T6
T7
T8
PERSON_TBL
PERSON_TBL
128M
PERSON_ASSN
JSON SERIALIZATION

```
+------------+---------+
| perID      | fname   |
+------------+---------+
| 1000000005 | BRADLEY |
| 1080000005 | CURT    |
| 1160000005 | ALTHEA  |
| 1240000005 | DOREEN  |
| 1320000005 | OFELIA  |
| 1400000005 | TOMMY   |
| 1480000005 | GILMA   |
| 1560000005 | BLAINE  |
|     .      |    .    |
|     .      |    .    |
|     .      |    .    |
+------------+---------+
```

```
+--------+------------+------------+
| assnID | subjperID  | objperID   |
+--------+------------+------------+
|   1    | 1000000005 | 1080000005 |
|   2    | 1080000005 | 1160000005 |
|   3    | 1160000005 | 1240000005 |
|   4    | 1240000005 | 1320000005 |
|   5    | 1320000005 | 1400000005 |
|   6    | 1400000005 | 1480000005 |
|   7    | 1480000005 | 1560000005 |
|   8    | 1560000005 | 1000000005 |
|   9    | 1000000010 | 1080000010 |
```

```
{
    "id": "PER1000000005",
    "fname": "BRADLEY",
    "SDOS01": "CURT",
    "SDOS02": "ALTHEA",
    "SDOS03": "DOREEN",
    "SDOS04": "OFELIA",
    "SDOS05": "TOMMY",
    "SDOS06": "GILMA",
    "SDOS07": "BLAINE",
    "SDOS08": "BRADLEY"
}
```

Figure 2-2. Representation of the “Six Degrees of Separation” (SDOS) Use Case as JSON Documents

During the preceding analytical activities conducted in the course of the project, the IDA team used the "six degrees of separation" (SDOS) use case as a good way to compare the performance of various alternative technologies against the traditional relational data store model. [11]

Figure 2-2 schematically shows the conversion into a set of JSON documents of the SDOS example data originally prepared using a relational database implementation consisting of just two tables, namely, **PERSON_TBL** and **PERSON_ASSN**. The **PERSON_TBL** was loaded with 128 million records of notional personnel information. [12] To create the SDOS scenario, the set was conceptually divided into 8 subsets of 16 million records each (shown in the figure as the blocks labeled T1, T2, . . . , T8). This then allowed for the creation of person associations between instances of the respective T blocks that were captured in the **PERSON_ASSN** table.

For simplicity the associations (represented in the figure as the arrows connecting blocks T1 through T8) were constructed by picking a record in any of the eight blocks, but linking it to an instance of person residing in the immediately following block. For example, if one begins with an instance in the T1 block, then the association in the **PERSON_ASSN** table would link said instance of person to an instance of person in the T2 block. In the context of the SDOS use case, this association is given the semantics of being the first degree of separation, namely, **SDOS01**, for the instance of person in the T1 block.

The record in the T2 block can then be similarly linked to a record in the T3 block to create the second degree of separation, namely, **SDOS02**, with respect to the instance of person selected in the T1 block. Following this approach one can create any number of degrees of separation for the 16 million instances of person in the T1 block. For the technical reasons mentioned below, the IDA team chose to make the last association, i.e., **SDOS08**—which in the example under discussion is the one that links instances in the T8 block—a link back to the record in the starting block. In the example under discussion, this would be the instance of person in the T1 block. This allows for easy checking of any violation of the construction pattern described above, since a listing of the instances of person representing the first eight degrees of separation would always have to start and end with the same record chosen from the starting block.

The **SDOS01** through **SDOS08** associations in the **PERSON_ASSN** table correspond to a set of 16 million subgraphs, each containing eight nodes and eight edges that capture the semantics of the SDOS use case, namely, that **person X *"knows"* person Y**, and **person Y**

[11] Although the original problem was restricted to six degrees of separation, we use the term, and hence the acronym SDOS, to refer to the general type of graphs, with edges representing the predicate "knows" and vertices representing instances of person.

[12] To eliminate unnecessary clutter in the figure, only the key (**perID**) and the person's first name (**fname**) fields in the **PERSON_TBL** are shown.

***“knows”* person Z**, etc. It is now fairly straightforward to rewrite the preceding statements as *key:value* pairs of the type used in JSON serializations, which is exactly the end state shown on the right hand of Figure 2-2.

Each such JSON document is constructed by defining an identifier key called **“id”** whose value is made up of the prefix **“PRE”** and the value of **perID** used in the relational data store for that instance of person. The JSON document also contains a *key:value* pair made up of the key **“fname”**, whose value is the string corresponding to the first name of the person in the block that starts the graph—in the example under discussion, this would be the T1 block. Finally, the serialization contains eight *key:value* pairs that capture the degrees of separation between the person instance in the starting block (e.g., T1) and the person instances in blocks sequentially following it (e.g., T2, T3, . . . , T8). As shown in the figure these are the keys **“SDOS01”**, **“SDOS02”**, . . . , **“SDOS08”**, and in our example their values are the respective first name entries for each of the person instances residing in blocks T1, T2, . . . , T8. [13]

In similar fashion one can construct JSON documents for the subgraphs that start with instances of person from the T2 block instead of the T1 block. And after those are built, one can continue with the remaining blocks that make up the **PERSON_TBL**, until 128 million JSON documents have been constructed that capture all the associations contained in the **PERSON_ASSN** table and express the SDOS semantics **person X *“knows”* person Y**. Note that by construction, the associations—and, therefore, the resulting graphs—are directed, going always from a lower numbered block to the immediately following higher numbered one. Although one can interpret the relationship **“knows”** as being commutative (if person X “knows” person Y, then person Y “knows” person X) the sample data considered here does not use this generalization.

[13] The JSON documents produced via the approach described in this section are partially analogous to a materialized view that conflates the **PERSON_TBL** and **PERSON_ASSN** tables into a single flat table reflecting every possible association among the instances of **PERSON_TBL**. See footnote 2 for details of a programmatic approach to build the SDOS01 case as a materialized view.

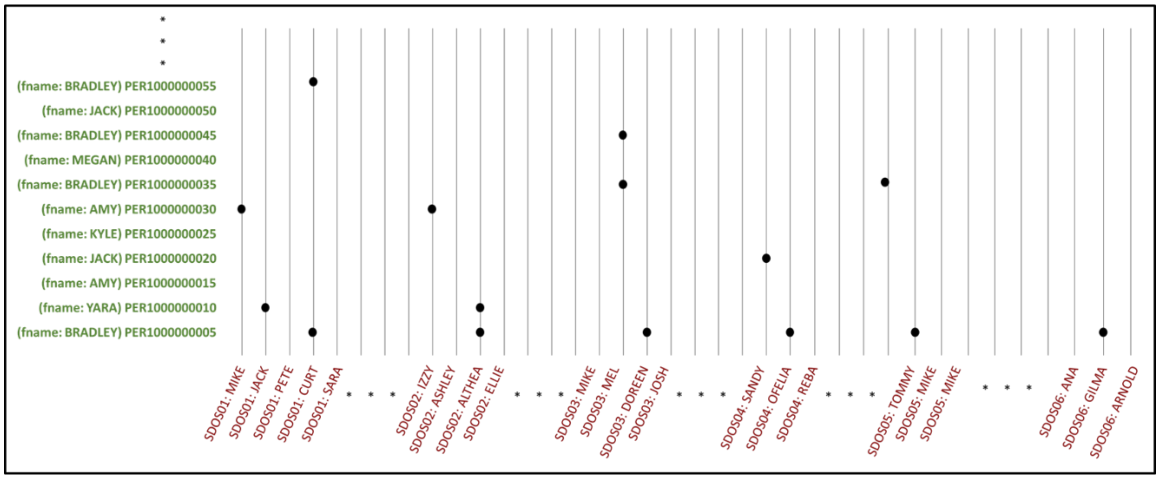

(fname: BRADLEY) PER1000000055
(fname: JACK) PER1000000050
(fname: BRADLEY) PER1000000045
(fname: MEGAN) PER1000000040
(fname: BRADLEY) PER1000000035
(fname: AMY) PER1000000030
(fname: KYLE) PER1000000025
(fname: JACK) PER1000000020
(fname: AMY) PER1000000015
(fname: YARA) PER1000000010
(fname: BRADLEY) PER1000000005
SDOS01: MIKE
SDOS01: JACK
SDOS01: PETE
SDOS01: CURT
SDOS01: SARA
SDOS02: IZZY
SDOS02: ASHLEY
SDOS02: ALTHEA
SDOS02: ELLIE
SDOS03: MIKE
SDOS03: MEL
SDOS03: DOREEN
SDOS03: JOSH
SDOS04: SANDY
SDOS04: OFELIA
SDOS04: REBA
SDOS05: TOMMY
SDOS05: MIKE
SDOS05: MIKE
SDOS06: ANA
SDOS06: GILMA
SDOS06: ARNOLD

Figure 2-3. Notional Inverted Index Set for the SDOS Use Case

Figure 2-3 shows a notional depiction of an inverted index set for JSON documents that represent the SDOS use case as described in the preceding paragraphs. The figure shows how readily one can find all the documents with the *key:value* pair **fname : BRADLEY** that also contain the *key:value* pair **SDOS01 : CURT**, namely, the documents with **id : 1000000005** and **id : 1000000055**. A more quantitative description of how well the Apache Solr search engine scales up is presented in the following section.

D. Example of Scalability and Performance – Apache Solr

The Apache Solr search engine is a top-performing implementation of the inverted index search algorithm and, therefore, was selected to test the applicability of this technology for use in the Army DFS portal. The testing was done with the JSON serializations for the SDOS use case discussed in the preceding section. As with previous experiments, the IDA team used a Dell PC with 16GB of RAM and a 5th generation Intel Core i7 processor with two cores. The Apache Solr version used was 7.2.1, and the operating system was Linux Centos 7.

1. Data Loading Performance

Figure 2-4 shows the time in milliseconds it took to successively load 246MB datasets, each containing 1 million JSON documents, each document corresponding to a subgraph with 8 nodes. As shown therein, it takes about a minute and three seconds on average to load a document of that size and complexity.

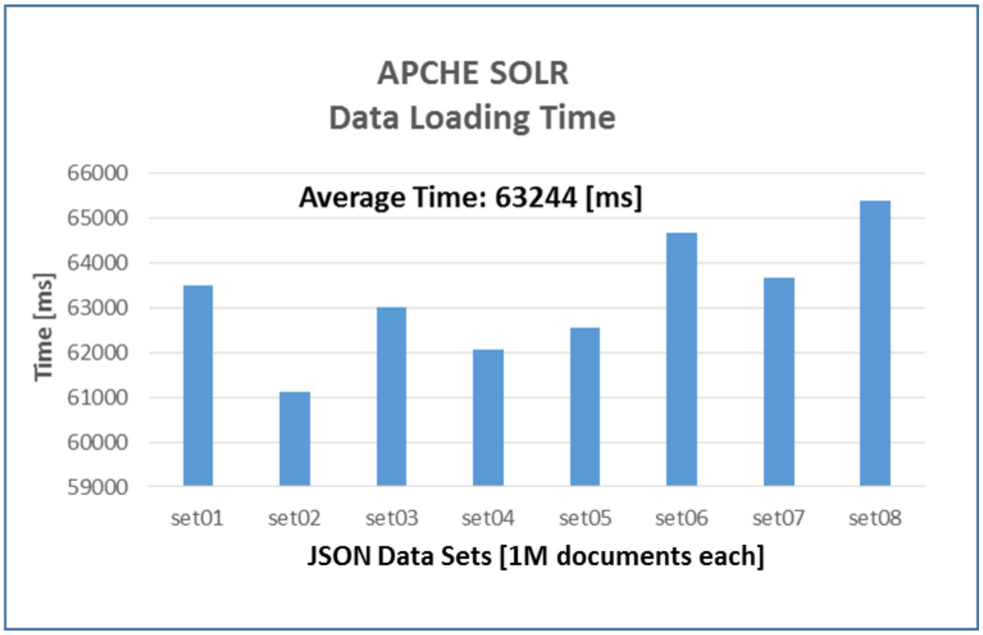


Figure 2-4. Example of Time Required to Load Data Sets into Apache Solr Server

2. Data Retrieval Performance

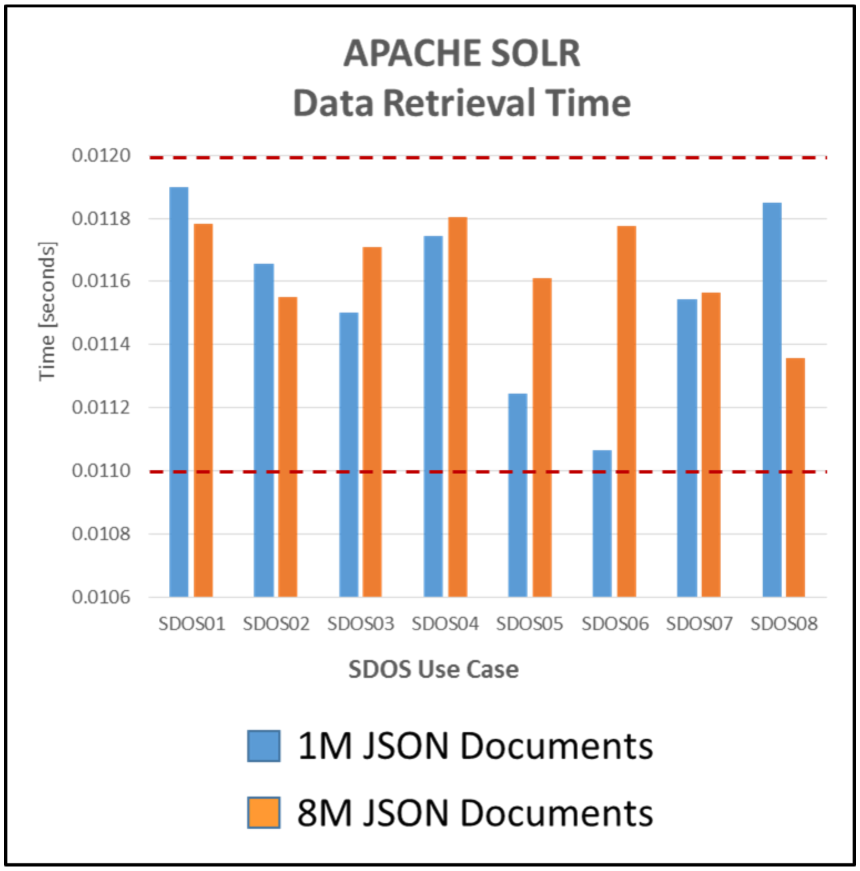


Figure 2-5. Time to Access and Retrieve Data for the "TYLER knows DORIAN" Query

Figure 2-5 shows the retrieval times obtained with a web application written in hypertext preprocessor (PHP) that uses the Solarium API. The application must load the libraries, connect to the Solr search engine, and then output the number of documents found and the statistics for the search. The figure shows the results for 1 million and 8 million JSON documents loaded into the Solr server.

As can be seen in Figure 2-5, on average the data is retrieved in under **12 milliseconds**, with very little variability associated with the type of query (e.g., **SDOS01** vs. **SDOS08**). This is a much better performance than when using SPARQL queries executed in RDF triple stores such as RDF4J and AllegroGraph, or when using unification queries with a Prolog knowledgebase or using recursive SQL queries against a relational data store using **PERSON_TBL** and the **PERSON_ASSN** tables. In any of those implementations, the observed retrieval times tend to be in the tens or even hundreds of seconds.

As we showed in the previous analytical results, for relational databases it is possible to obtain similar data retrieval performance by converting the **PERSON_TBL** and the **PERSON_ASSN** tables into a materialized view of any specific query. Once the results are persisted in the new table, there is no need to traverse the association table thousands or even millions of times to answer questions such as "**How many instances of individuals named 'X' know individuals named 'Y'?**"

However, the effort involved in the definition and execution of said materialized views for a relational data store implementation of the SDOS use case with data volumes of 128 million or larger is substantial once the degree of separation is greater than 3. Furthermore, in order to be practical, the materialized view would need to be built with complete generality rather than for a single specific case—such as **X = 'TAYLOR'** and **Y = 'DORIAN'** in the above query—unless one wants to build potentially a couple of millions of them to cover each possible combination of first names and degrees of separation. In contrast, the JSON representation of the SDOS use case data in combination with an engine that uses the inverted index algorithm, such as Apache Solr, appears to provide a very elegant solution.

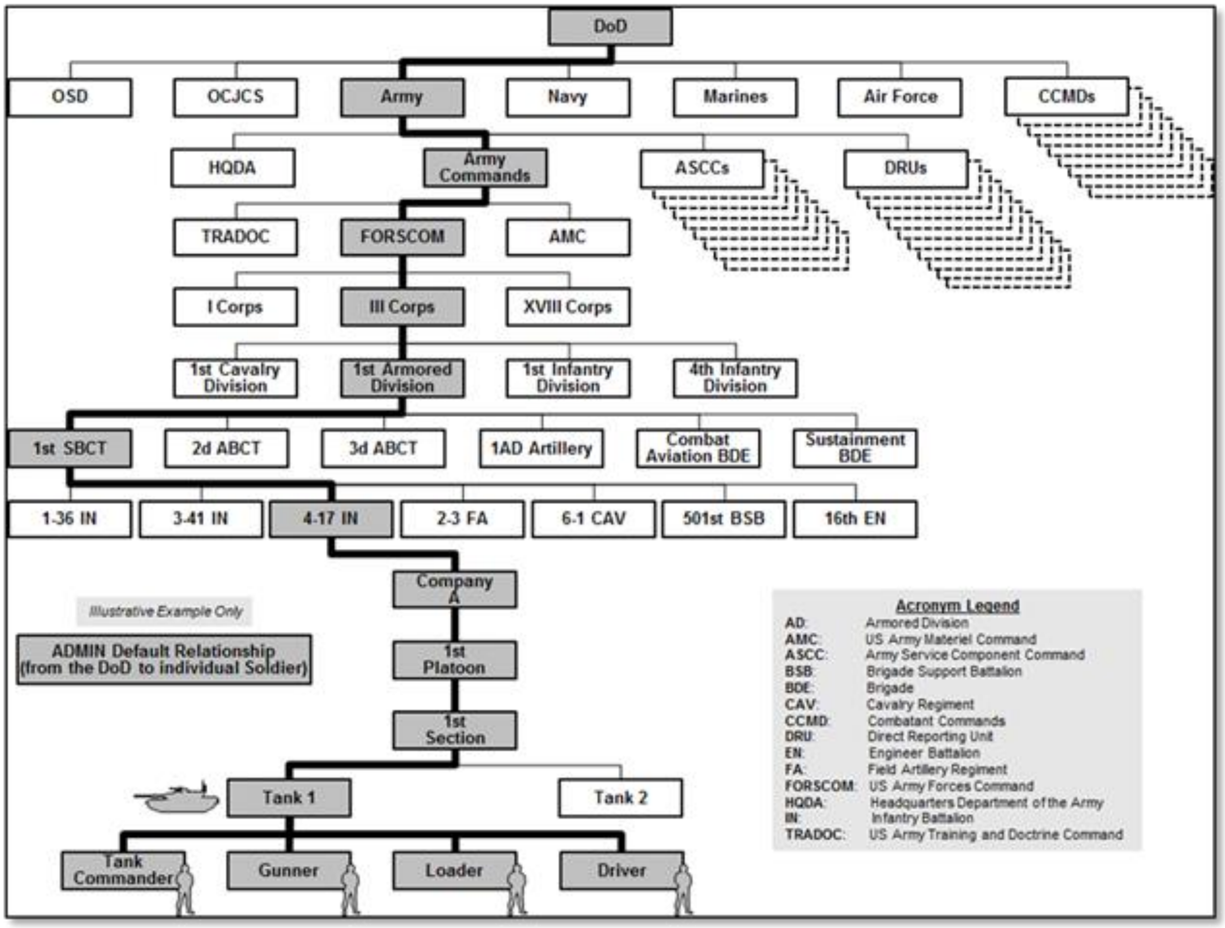
E. Force Structure Data as a Variant of the SDOS Use Case

The Army Data Strategy [14] establishes five strategic level data goals, that is, to make all operationally relevant data *visible*, *accessible*, *understandable*, *trusted*, and *interoperable* (VAUTI). Force structure data is operationally relevant, and, therefore, should be fully interoperable across all Army mission areas.

[14] *Army Data Strategy*, Information Architecture Division, Army Architecture Integration Center, HQDA CIO/G-6, Version 1.0, February 2016.

To that effect, the Army has endorsed the activities of the Global Force Management Data Initiative (GFM DI).[15] The GFM DI data strategy calls for the establishment and population of Organizational Servers (Org Servers) from which users may obtain and exchange authorized force structure data. When using relational data stores, the strategy foresees the use of DoD-wide, standardized and unambiguous identifiers, known as Force Management Identifiers (FMIDS) and defined as part of the Organizational and Force Structure Construct (OFSC).[16] This approach is intended to support both stable nodes, known as organizational elements (OEs), and dynamic links among OEs which make the Dynamic Force Structure Representation (DFSR).

Figure 2-6 shows a notional example encompassing Army command and/or administrative control requirements that can be represented using the type of relations defined as part of the OFSC specification.



Acronym Legend

AD:	Armored Division
AMC:	US Army Materiel Command
ASCC:	Army Service Component Command
BSB:	Brigade Support Battalion
BDE:	Brigade
CAV:	Cavalry Regiment
CCMD:	Combatant Commands
DRU:	Direct Reporting Unit
EN:	Engineer Battalion
FA:	Field Artillery Regiment
FORSCOM:	US Army Forces Command
HQDA:	Headquarters Department of the Army
IN:	Infantry Battalion
TRADOC:	US Army Training and Doctrine Command

Figure 2-6. Notional Example of Force Structure Data

[15] *Global Force Management Data Initiative*, DoD Instruction (DoDI) 8260.03, February 19, 2014.

[16] *Organizational and Force Structure Construct (OFSC) for Global Force Management (GFM)*, DoD Instruction (DoDI) 8260.03, August 23, 2006.

This type of data can be readily loaded into the physical schema of a GFM-DI-compliant Organization Server. Since, as we showed in a previous phase of the study, any type of data resident in a relational data store can be serialized in the form of RDF triples, and once in that form, it in turn can then be transformed into JSON documents, the GFM DI force structure data can be loaded into in an Apache Solr server enabling the creation of interactive web applications that can operate well within the expected performance limits, i.e., less than 1-second query execution time.

3. Conclusions and Recommendations

A. Conclusions

Based on the analytical work during the final phase of the study, the IDA team concluded the following:

- Search engines—such as Apache Solr—that leverage inverted index data structures and can store data expressed in the form of *key:value* pairs have achieved a high degree of maturity and acceptance in the commercial world. Their ability to handle data volumes of the same magnitude as that expected for the planned Army DFS portal is adequate, even when the Solr server runs on standard hardware.
- Interactive web applications that use an Apache Solr server loaded with RDF triples converted into JSON documents made up of *key:value* are feasible, and the data retrieval performance observed in preliminary tests is consistently well under one second per query.

B. Recommendations

For the final phase of the study, the recommendations are as follows:

- The solution architecture for the Army DFS portal should include technologies such as inverted index search engines, coupled with data representations compatible with such engines, to minimize the risk of the poor data retrieval performance associated with most graph database implementations.
- Processes for transforming force structure data from one representation into another should be identified early in the planning for the Army DFS to ensure that each of the technologies included in the solution architecture can be optimally exploited.
- Simulation techniques should be employed to assess the impact of hybrid solution architectures on the concept of operations of the Army DFS portal.

Appendix A. Combining *Key:Value* Pair Data Representations with Fast Inverted Index Search Engines

1. Introduction

As was mentioned in the main portion of this document, the Apache Solr engine is a very fast inverted index search engine that has been optimized to handle large collections of documents expressed in the form of *key:value* pairs. In a previous deliverable[17] it was shown that data serialized as RDF triples can be readily transformed into JSON documents that use *key:value* pairs to capture content.

This annex shows an approach based on the combination of the two above-mentioned technologies that could be used to mitigate the risk of the poor data retrieval performance exhibited by graph databases when processing deeply nested SPARQL queries, such as those encountered in the "six degrees of separation" (SDOS) use case. To be able to quantify the advantages offered by the proposed approach and compare them with results obtained in the previous phases of the study, the IDA team serialized as JSON documents the same sample data that was stored in the tables **Person** and **PersonAssociation** within a MySQL server.[18]

2. Apache Solr Server Performance for the SDOS Use Case

The following sections show the substantial improvement in data retrieval time that can be obtained when using inverted index search engines, such as Apache Solr, to solve the SDOS use case.

a. Loading the JSON Documents into the Apache Solr Server

The JSON documents generated by the Python script described in the preceding section can be loaded into the Apache Solr server from the command line using curl. The example curl command shown below loads a file named **SDOS128_01f.json** into the Solr server.

[17] See footnote 2 above.

[18] See footnotes 2 and 3 above

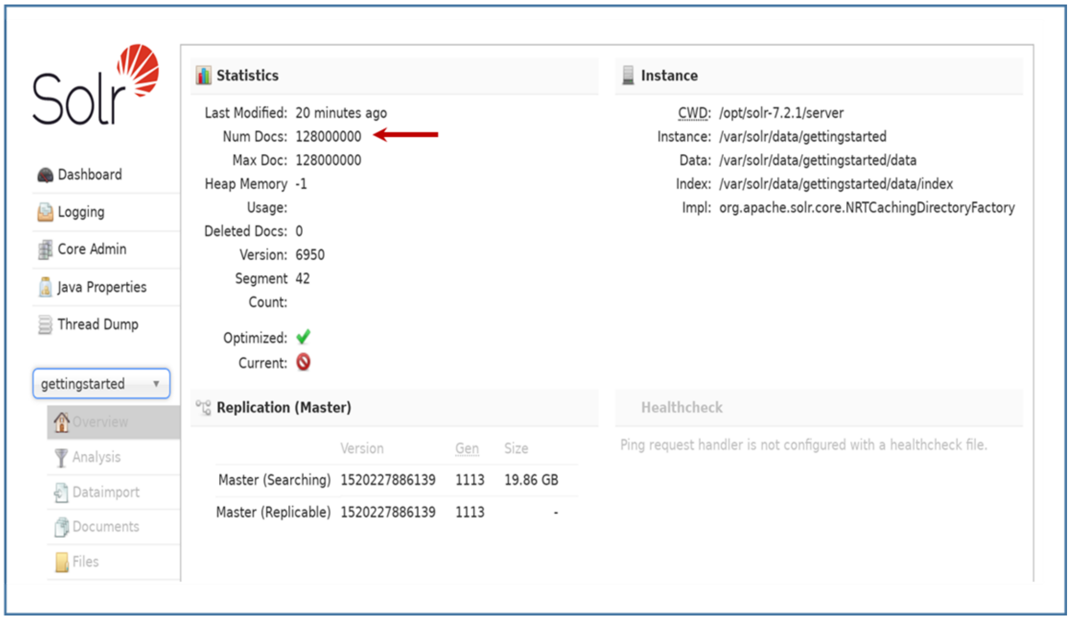
```
curl  'http://localhost:8983/solr/gettingstarted/update?commit=true'  --
data-binary @./SDOS128_01f.json -H 'Content-type:application/json'
```

When the upload is completed, the Solr server will respond indicating that the document has been successfully loaded. The entry **QT** entry gives the time in milliseconds it took the Apache Solr server to process the file.

```
{
  "responseHeader":{
    "status":0,
    "QTime":61490}}
```

On a standard Dell PC tower with a two core Intel i7 chip and 16GB of random access memory (RAM), it takes on average 60 to 70 seconds to process a 246MB file.

On a computer running Apache Solr server a user-friendly web interface called the administrator console can be accessed by default at http://localhost:8983/solr. The administrator console allows for the quick inspection of the data stored in the server. Figure A-1 shows the status of the Solr server after all 128 million JSON documents have been loaded.



Statistics

Last Modified: 20 minutes ago
Num Docs: 128000000
Max Doc: 128000000
Heap Memory Usage: -1
Deleted Docs: 0
Version: 6950
Segment Count: 42
Optimized: ✓
Current: ⊘

Instance

CWD: /opt/solr-7.2.1/server
Instance: /var/solr/data/gettingstarted
Data: /var/solr/data/gettingstarted/data
Index: /var/solr/data/gettingstarted/data/index
Impl: org.apache.solr.core.NRTCachingDirectoryFactory

Replication (Master)

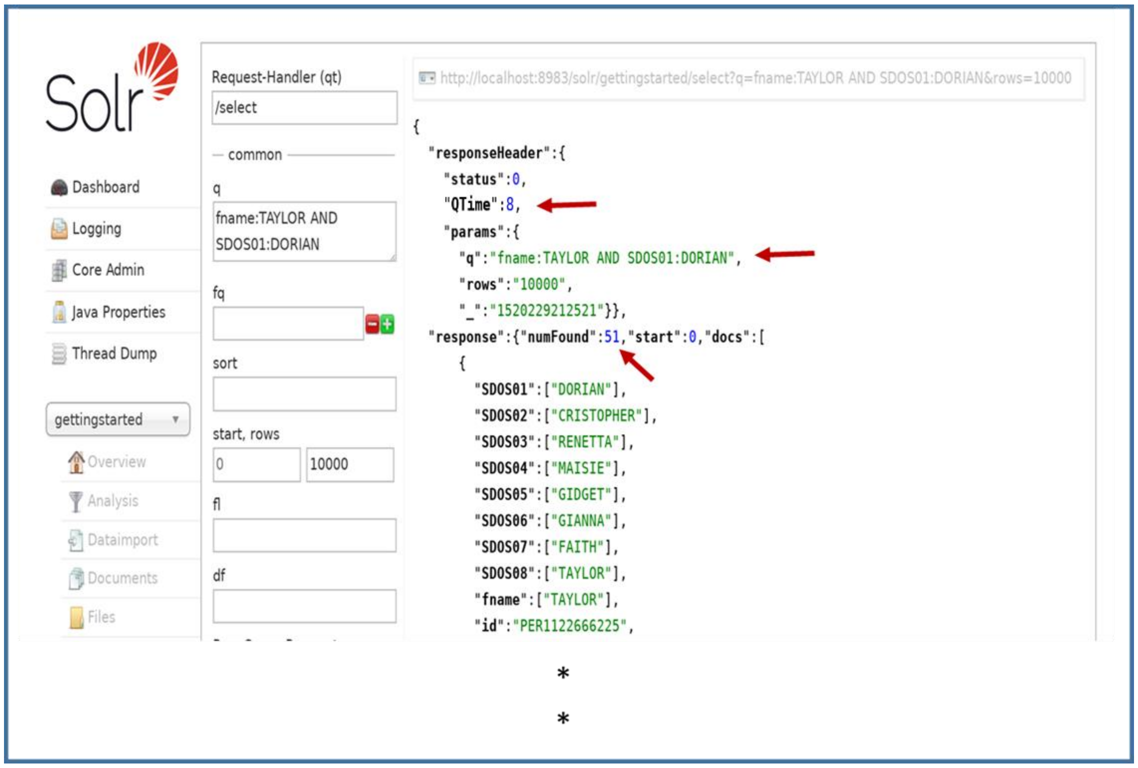
	Version	Gen	Size
Master (Searching)	1520227886139	1113	19.86 GB
Master (Replicable)	1520227886139	1113	-

Healthcheck

Ping request handler is not configured with a healthcheck file.

Figure A-1. Screen Capture of the Apache Solr Server Admin Interface with All 128 Million JSON Documents Loaded

b. Querying the Apache Solr Server



Request-Handler (qt): /select

q: fname:TAYLOR AND SDOS01:DORIAN

start, rows: 0 10000

```
http://localhost:8983/solr/gettingstarted/select?q=fname:TAYLOR AND SDOS01:DORIAN&rows=10000

{
  "responseHeader":{
    "status":0,
    "QTime":8,
    "params":{
      "q":"fname:TAYLOR AND SDOS01:DORIAN",
      "rows":"10000",
      "_":"1520229212521"}},
  "response":{"numFound":51,"start":0,"docs":[
      {
        "SDOS01":["DORIAN"],
        "SDOS02":["CRISTOPHER"],
        "SDOS03":["RENETTA"],
        "SDOS04":["MAISIE"],
        "SDOS05":["GIDGET"],
        "SDOS06":["GIANNA"],
        "SDOS07":["FAITH"],
        "SDOS08":["TAYLOR"],
        "fname":["TAYLOR"],
        "id":"PER1122666225",
```

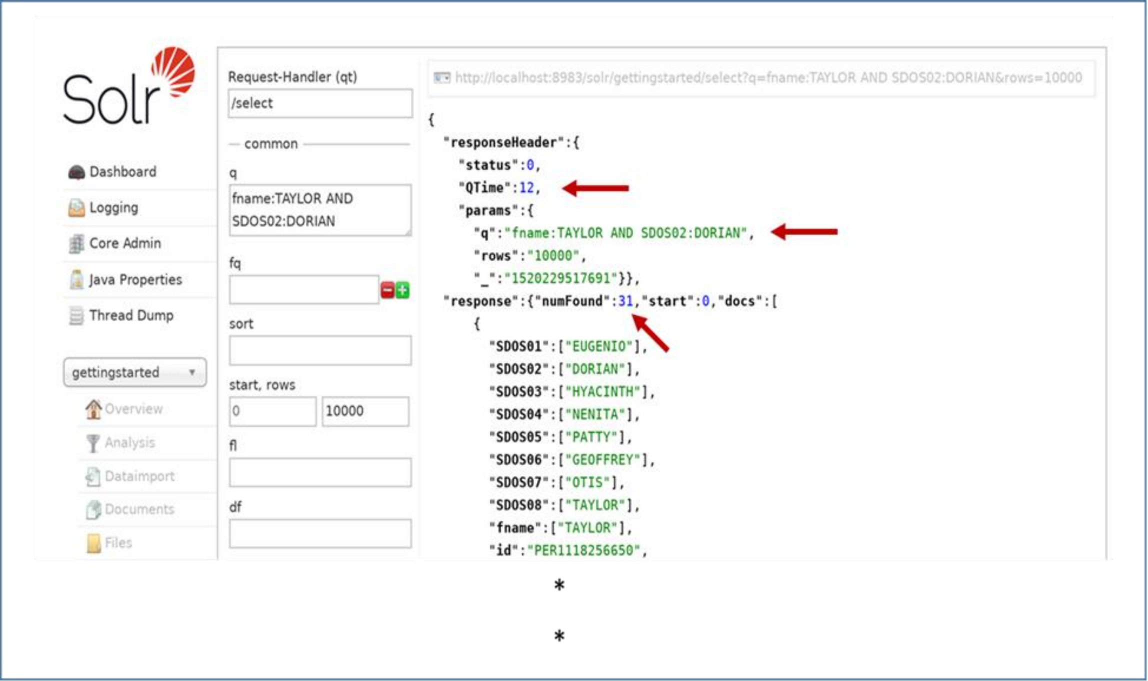
*

*

Figure A-2. Results for the SDOS01 Query Using the Solr Admin Interface

Once all the SDOS data has been loaded into the server, one can also use the administrator console to query the database. Figure A-2 the status of the console after executing the query entered in the textbox labeled **q** using the standard Solr query syntax. In the example shown, the query asks for all the JSON documents that satisfy the condition that their **fname** key has the value **"TAYLOR"** and the **SDOS01** key has the value **"DORIAN"**. The Apache Solr engine inspects all 128 million JSON documents and retrieves those instances that satisfy the condition. As shown in the figure, the **response** block shows that there are **51** such documents in the database. The (query time) **QT** entry in the **responseheader** block indicates that it took the Solr server **38 milliseconds** to process the query.

Figure A-3 shows the results for the SDOS02 query. The query inspects again all 128 million JSON documents and retrieves those instances that satisfy the condition that the **fname** key has the value **"TAYLOR"** and the **SDOS02** key has the value **"DORIAN"**. The **response** block shows that there are **31** such documents. The **QT** entry indicates that it took the Solr server **12 milliseconds** to process the query.



Request-Handler (qt): /select

common

q: fname:TAYLOR AND SDOS02:DORIAN

fq:

sort:

start, rows: 0 10000

fl:

df:

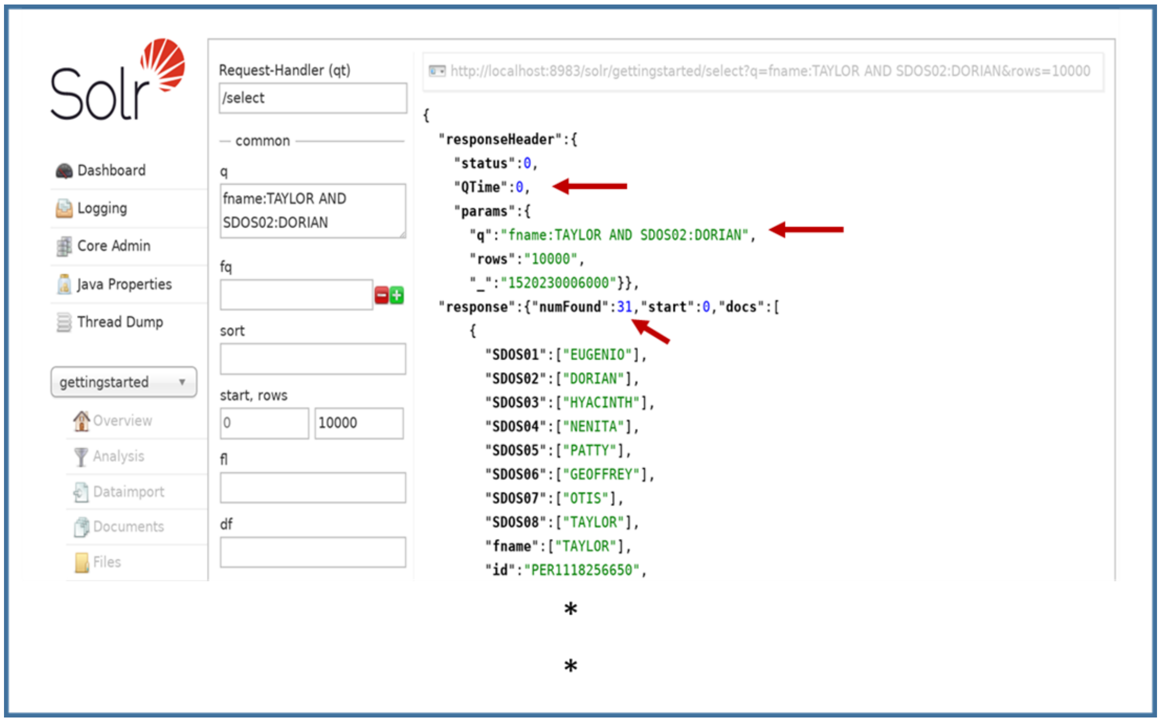
http://localhost:8983/solr/gettingstarted/select?q=fname:TAYLOR AND SDOS02:DORIAN&rows=10000

```
{
  "responseHeader":{
    "status":0,
    "QTime":12,
    "params":{
      "q":"fname:TAYLOR AND SDOS02:DORIAN",
      "rows":"10000",
      "_":"1520229517691"}},
  "response":{"numFound":31,"start":0,"docs":[
      {
        "SDOS01":["EUGENIO"],
        "SDOS02":["DORIAN"],
        "SDOS03":["HYACINTH"],
        "SDOS04":["NENITA"],
        "SDOS05":["PATTY"],
        "SDOS06":["GEOFFREY"],
        "SDOS07":["OTIS"],
        "SDOS08":["TAYLOR"],
        "fname":["TAYLOR"],
        "id":"PER1118256650",
                      *

                      *
```

Figure A-3. Results for the SDOS02 Query Using the Solr Admin Interface

The Apache Solr server caches the results of the queries it executes. When executing again the same **SDOS02** query described above, the **QT** drops down to less than 1 millisecond (displayed as a **QT** value of **0** milliseconds) as shown in Figure A-4.



Request-Handler (qt): /select

common

q: fname:TAYLOR AND SDOS02:DORIAN

fq:

sort:

start, rows: 0 10000

fl:

df:

http://localhost:8983/solr/gettingstarted/select?q=fname:TAYLOR AND SDOS02:DORIAN&rows=10000

```
{
  "responseHeader":{
    "status":0,
    "QTime":0,
    "params":{
      "q":"fname:TAYLOR AND SDOS02:DORIAN",
      "rows":"10000",
      "_":"1520230006000"}},
  "response":{"numFound":31,"start":0,"docs":[
      {
        "SDOS01":["EUGENIO"],
        "SDOS02":["DORIAN"],
        "SDOS03":["HYACINTH"],
        "SDOS04":["NENITA"],
        "SDOS05":["PATTY"],
        "SDOS06":["GEOFFREY"],
        "SDOS07":["OTIS"],
        "SDOS08":["TAYLOR"],
        "fname":["TAYLOR"],
        "id":"PER1118256650",
                      *

                      *
```

Figure A-4. Retrieval Time for Cached Queries in the Apache Solr Server

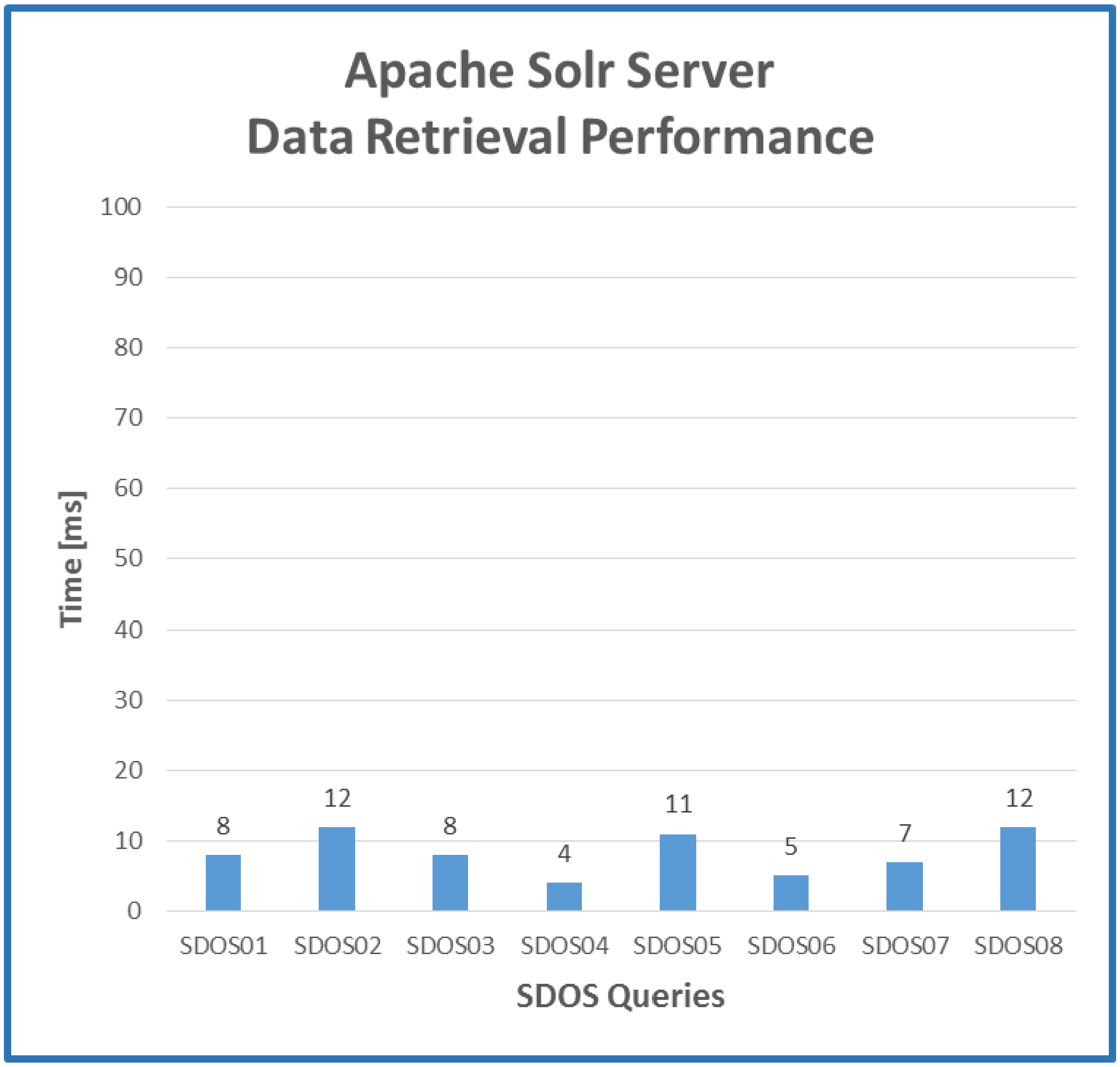


Figure A-5. Apache Solr Server Data Retrieval Times for the SDOS Use Case

Figure A-5 summarizes the data retrieval performance of the Apache Solr server for all eight degrees of separation. The results show vastly improved performance when compared to the data retrieval times obtained using a Prolog knowledgebase loaded with only eight million clauses corresponding to the **SDOS01** through **SDOS08** associations for the first million entries in the T1 block (See Figure A-6). A full discussion of the use of Prolog is contained in the third deliverable. [19]

[19] See footnote 3.

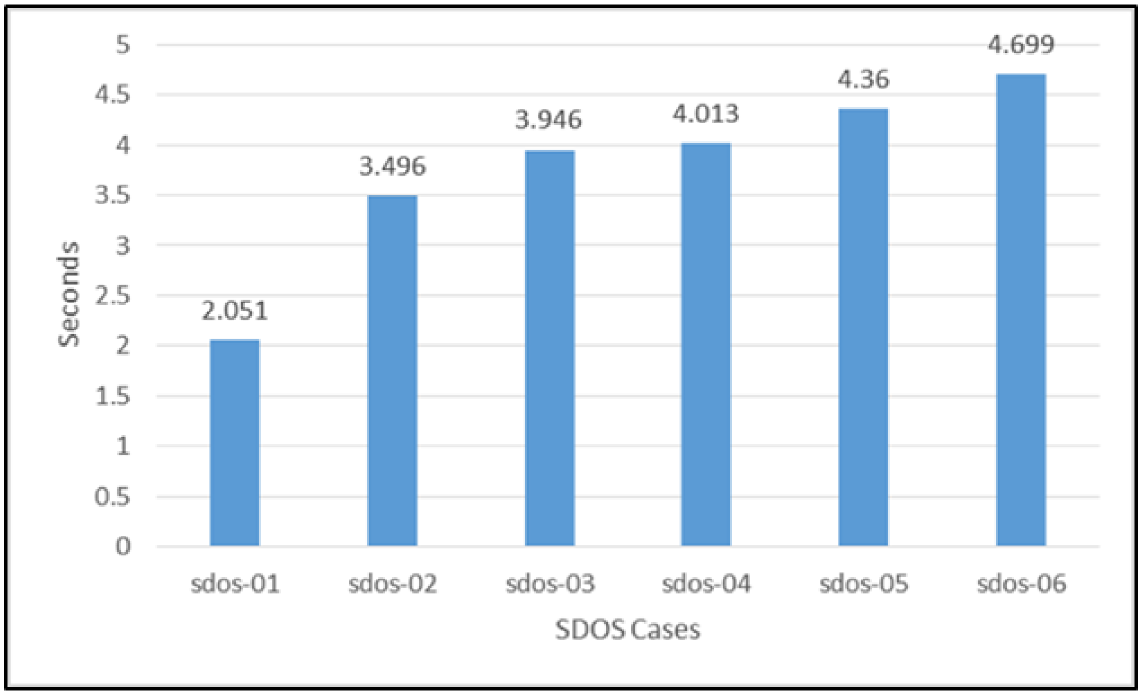


Figure A-6. Prolog Data Retrieval Times for the SDOS Use Case and 8 Million Clauses

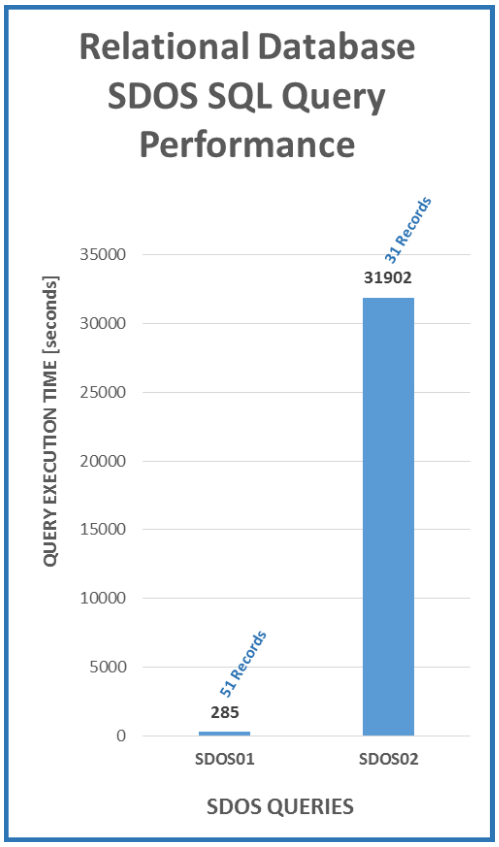


Figure A-7. SQL Query Data Retrieval Times for the SDOS Use Case using 128 Million Entries

As shown in Figure A-7, the performance of relational databases for the SDOS use case is even poorer than that shown by Prolog knowledgebases. Preliminary testing conducted during the second phase of the study using a MySQL server implementation showed execution times that were completely inadequate for the development of web-based applications.[20] As shown in the figure, the retrieval of the 51 records corresponding to the SDOS01 query took 285 seconds in MySQL compared to just 8 milliseconds using Solr. The SDOS02 query took over a thousand times longer to complete.

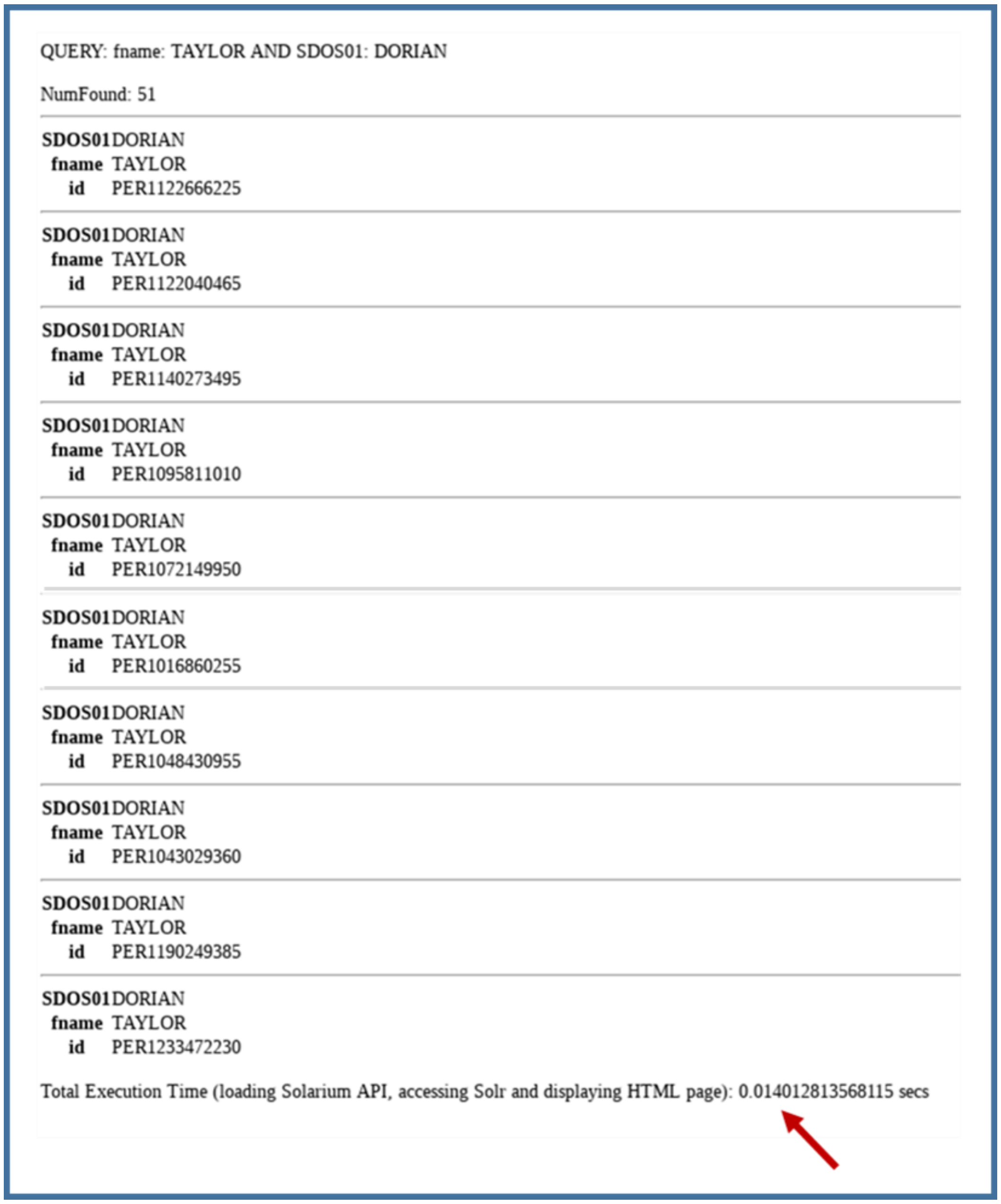
3. Using the Apache Solr Server for Web Application Development

The preceding section showed how a change in data representation—i.e., from RDF triples to JSON *key:value* pairs—can be combined with search engines optimized for the selected data representation—e.g., Apache Solr—to bring the data retrieval times well within the limits needed for interactive web applications. This section shows a basic example that uses the server-side scripting language PHP in combination with the Apache Solr server operating as the backend data store. The web application uses Solarium, a very user-friendly Solr client library for PHP that can be used to pass queries to the Solr server and then retrieve and display the results.[21]

[20] See footnote 2 above.

[21] https://solarium.readthedocs.io/en/latest/

a. Basic Web Application Interface



```
QUERY: fname: TAYLOR AND SDOS01: DORIAN

NumFound: 51

SDOS01 DORIAN
 fname TAYLOR
    id PER1122666225

SDOS01 DORIAN
 fname TAYLOR
    id PER1122040465

SDOS01 DORIAN
 fname TAYLOR
    id PER1140273495

SDOS01 DORIAN
 fname TAYLOR
    id PER1095811010

SDOS01 DORIAN
 fname TAYLOR
    id PER1072149950

SDOS01 DORIAN
 fname TAYLOR
    id PER1016860255

SDOS01 DORIAN
 fname TAYLOR
    id PER1048430955

SDOS01 DORIAN
 fname TAYLOR
    id PER1043029360

SDOS01 DORIAN
 fname TAYLOR
    id PER1190249385

SDOS01 DORIAN
 fname TAYLOR
    id PER1233472230

Total Execution Time (loading Solarium API, accessing Solr and displaying HTML page): 0.014012813568115 secs
```

Figure A-8. Web Applications Using the Solr Server as Backend Data Store

Figure A-8 shows a basic web application built with PHP that queries the Solr server and displays the results for the **SDOS01** query. The number of documents found is the same as what was found using the Solr administrative console, but now the retrieval time includes the time that it takes to load the Solarium API libraries, as well as the time that it takes to render the HTML results. In spite of all this, the performance appears to be well within the limit of requiring less than 1 second.

b. Web Application Statistics

Figure A-9 summarizes the performance observed in a very basic Web application using PHP and the Solr server as the backend data store.

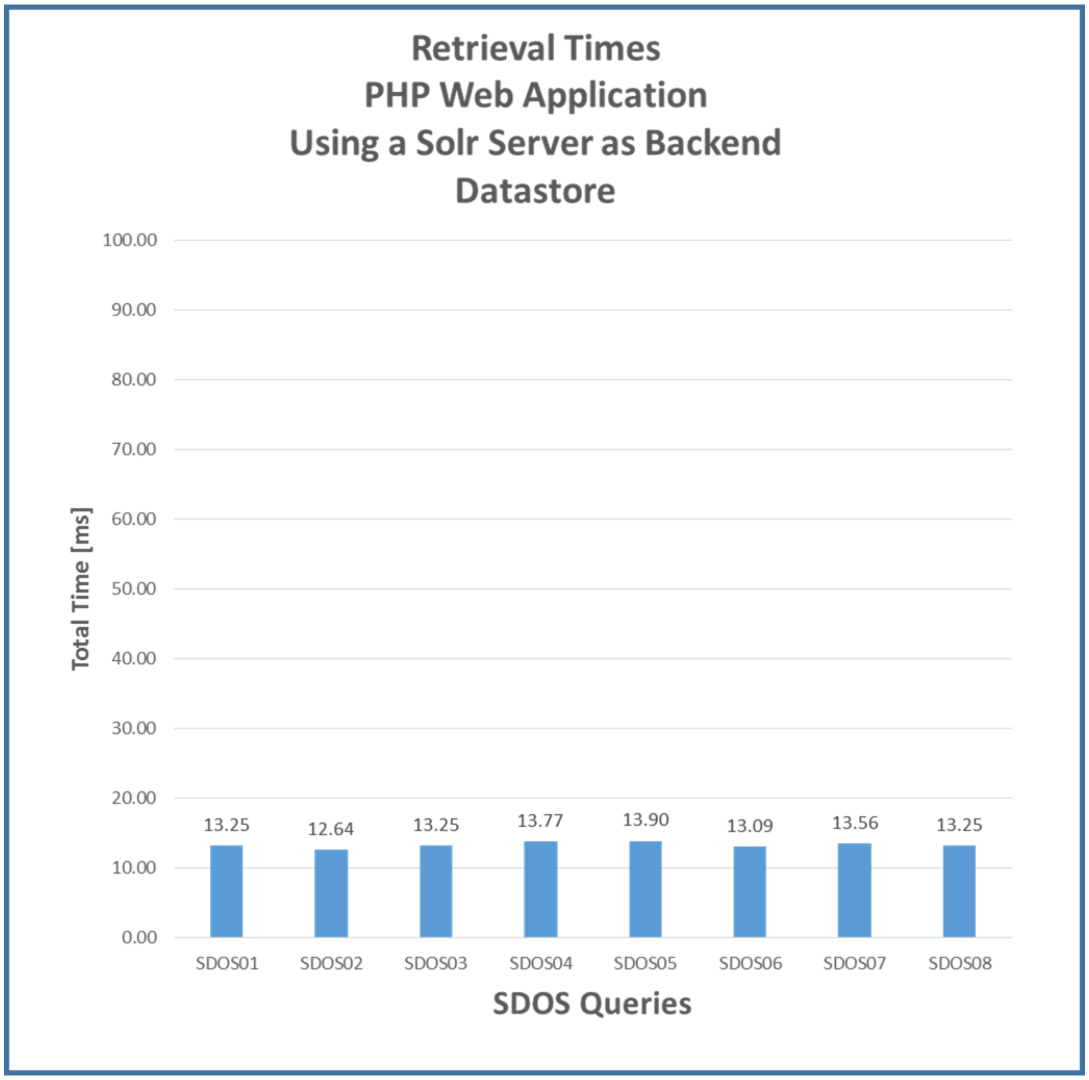


Figure A-9. Data Retrieval Performance for Web Applications Using the Solr Server as Backend Data Store

As can be seen in Figure A-9, the application is well within the limits desired for interactive applications, which would arguably support the claim that performance degradation due to added web application functionality could be mitigated through the use of more powerful hardware.

Appendix B.
Sample Code Used for Testing Solutions Based on the Apache Solr Search Engine

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 database manipulation using inverted index search engines, such as Apache Solr.

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

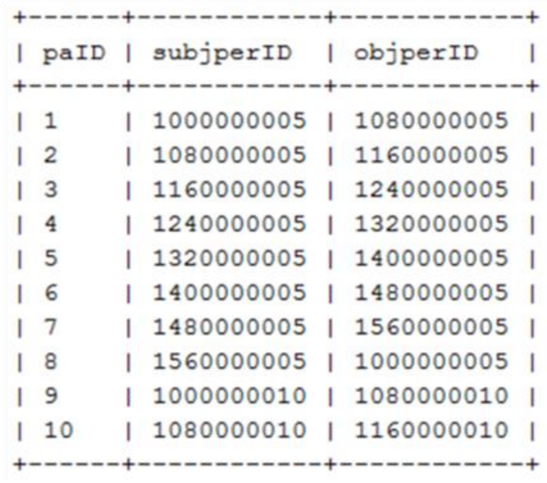
[22] https://opensource.org/licenses/MIT

A. Preparation of the JSON Documents

1. Python Scripts to Generate the JSON Documents

The 128 million entries in the **Person** table are notionally divided into 8 blocks of 16 million records each. The **PersonAssociation** table contains links corresponding to eight degrees of separation expressed in the form of records listing the key of the subject person (**subjperID**) and its associated object person (**objperID**). The **PersonAssociation** table is constructed so that the first degree of separation (**SDOS01**) is built between records from the T1 block referencing records from the T2 block. Similarly, the second degree of separation (**SDOS02**) is built by associating the preceding instances from the T2 block to instances in the T3 block. The same pattern is used to create all eight degrees of separation used to test both relational data store implementations as well as their alternatives.

The first Python script listed below traverses all 16 million records from the T1 block stored in the **PersonAssociation** table. For each entry, it fetches the associated identifiers from the remaining T2 through T8 blocks to create the respective **SDOS01** through **SDOS08** associations. Figure B-1 shows a snippet of the resulting sorted table containing the first 10 entries.



```
+------+------------+------------+
| paID | subjperID  | objperID   |
+------+------------+------------+
| 1    | 1000000005 | 1080000005 |
| 2    | 1080000005 | 1160000005 |
| 3    | 1160000005 | 1240000005 |
| 4    | 1240000005 | 1320000005 |
| 5    | 1320000005 | 1400000005 |
| 6    | 1400000005 | 1480000005 |
| 7    | 1480000005 | 1560000005 |
| 8    | 1560000005 | 1000000005 |
| 9    | 1000000010 | 1080000010 |
| 10   | 1080000010 | 1160000010 |
+------+------------+------------+
```

Figure B-1. Snippet of the Sorted PersonAssociation Table

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


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

import MySQLdb as mdb

startVal = 1000000005
counter = 1

A = []
B = []

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

with con:

    for j in range(16000000):

        cur = con.cursor()

# ---------------- first record ------------------------

#  This code uses the alias perAssn3 for the original PERSON_ASSN table and persAssn2 for the resorted
variant of the PERSON_ASSN table.

#       print "First Record"

        sqlStr = "SELECT subjperID,objperID FROM perAssn3 WHERE subjperID =" + str(startVal)
#       print sqlStr
```

```
        cur.execute(sqlStr)
        row = cur.fetchone()
        subj = int(row[0])
        obj  = int(row[1])

        sqlStr = "INSERT INTO persAssn2 VALUES(" + str(counter) + "," + str(subj) + "," + str(obj) + ")"

#       print sqlStr

        cur.execute(sqlStr)

        counter = counter + 1

# ---------------- second record ------------------------

#       print "Second Record"

        sqlStr = "SELECT subjperID,objperID FROM perAssn3 WHERE subjperID =" + str(obj)
#       print sqlStr

        cur.execute(sqlStr)
        row = cur.fetchone()
        subj = int(row[0])
        obj  = int(row[1])

        sqlStr = "INSERT INTO persAssn2 VALUES(" + str(counter) + "," + str(subj) + "," + str(obj) + ")"

#       print sqlStr

        cur.execute(sqlStr)

        counter = counter + 1

# ---------------- third record ------------------------

#       print "Third Record"

        sqlStr = "SELECT subjperID,objperID FROM perAssn3 WHERE subjperID =" + str(obj)
#       print sqlStr

        cur.execute(sqlStr)
        row = cur.fetchone()
        subj = int(row[0])
        obj  = int(row[1])

        sqlStr = "INSERT INTO persAssn2 VALUES(" + str(counter) + "," + str(subj) + "," + str(obj) + ")"

#       print sqlStr

        cur.execute(sqlStr)
        counter = counter + 1

# ---------------- fourth record ------------------------

#       print "Fourth Record"

        sqlStr = "SELECT subjperID,objperID FROM perAssn3 WHERE subjperID =" + str(obj)
#       print sqlStr

        cur.execute(sqlStr)
        row = cur.fetchone()
        subj = int(row[0])
        obj  = int(row[1])

        sqlStr = "INSERT INTO persAssn2 VALUES(" + str(counter) + "," + str(subj) + "," + str(obj) + ")"

#       print sqlStr

        cur.execute(sqlStr)

        counter = counter + 1

# ---------------- fifth record ------------------------

#       print "Fifth Record"

        sqlStr = "SELECT subjperID,objperID FROM perAssn3 WHERE subjperID =" + str(obj)
#       print sqlStr

        cur.execute(sqlStr)
        row = cur.fetchone()
        subj = int(row[0])
        obj  = int(row[1])

        sqlStr = "INSERT INTO persAssn2 VALUES(" + str(counter) + "," + str(subj) + "," + str(obj) + ")"

#       print sqlStr

        cur.execute(sqlStr)

        counter = counter + 1

# ---------------- sixth record ------------------------

#       print "Sixth Record"

        sqlStr = "SELECT subjperID,objperID FROM perAssn3 WHERE subjperID =" + str(obj)
#       print sqlStr

        cur.execute(sqlStr)
        row = cur.fetchone()
        subj = int(row[0])
        obj  = int(row[1])

        sqlStr = "INSERT INTO persAssn2 VALUES(" + str(counter) + "," + str(subj) + "," + str(obj) + ")"

#       print sqlStr

        cur.execute(sqlStr)

        counter = counter + 1

# ---------------- seventh record ------------------------

#       print "Seventh Record"

        sqlStr = "SELECT subjperID,objperID FROM perAssn3 WHERE subjperID =" + str(obj)
#       print sqlStr

        cur.execute(sqlStr)
        row = cur.fetchone()
        subj = int(row[0])
        obj  = int(row[1])

        sqlStr = "INSERT INTO persAssn2 VALUES(" + str(counter) + "," + str(subj) + "," + str(obj) + ")"

#       print sqlStr

        cur.execute(sqlStr)

        counter = counter + 1

# ---------------- eigth record ------------------------

#       print "Eigth Record"

        sqlStr = "SELECT subjperID,objperID FROM perAssn3 WHERE subjperID =" + str(obj)
#       print sqlStr

        cur.execute(sqlStr)
        row = cur.fetchone()
        subj = int(row[0])
        obj  = int(row[1])

        sqlStr = "INSERT INTO persAssn2 VALUES(" + str(counter) + "," + str(subj) + "," + str(obj) + ")"

#       print sqlStr

        cur.execute(sqlStr)

        counter = counter + 1
```

```
# ------------------------------------------------------

    con.commit()

    startVal = startVal + 5

con.close()
```

Once the **PersonAssociation** table has been sorted in the manner indicated above, one can use the Python script listed below to generate JSON documents with the *key:value* pairs, as shown in Figure B-2.

```
{
    "id": "PER1000000005",
    "fname": "BRADLEY",
    "SDOS01": "CURT",
    "SDOS02": "ALTHEA",
    "SDOS03": "DOREEN",
    "SDOS04": "OFELIA",
    "SDOS05": "TOMMY",
    "SDOS06": "GILMA",
    "SDOS07": "BLAINE",
    "SDOS08": "BRADLEY“
}
```

Figure B-2. Example of a JSON Document Containing All the *Key:Value* Pairs Needed for the SDOS Use Case Test

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

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

import MySQLdb as mdb
import json

con = mdb.connect('localhost', <user>, <password>, <db>)

with con:

    cur = con.cursor()

    localDict = {}              # create an empty dictionary

# This script generates JSON documents for the first million of records in the T1 block
# To generate all JSON for the 16 million one simply needs to change the range in
# increments of 8 million, e.g., (8000000,16000000,8) for the second million,
# (16000000,24000000,8) for the third, and so on.

    for jj in range(0,8000000,8):

#-------------------- FIRST RECORD ------------------------------------------

        mycntr = jj + 1


#  This code uses the alias persAssn2 for the original PERSON_ASSN table.


# get the first pair of person-person association
        sqlStr01 = "SELECT subjperID,objperID FROM persAssn2 WHERE assnID =" + str(mycntr)

        cur.execute(sqlStr01)

        row01 = cur.fetchone()

        subj = int(row01[0])
        obj  = int(row01[1])

# create a string in the form of "PER1000000005"
        perID = "PER" + str(subj)

# get the fname for that entry in the person08a Tbl
        sqlStr02 = "SELECT fname FROM person08a WHERE perID =" + str(subj)

        cur.execute(sqlStr02)

        row02 = cur.fetchone()

# place the fname in the variable fname
```

```
        fname = str(row02[0])


# add items to the dictionary

        localDict['id'] = perID
        localDict['fname'] = fname

#--------------------- SDOS01 ----------------------------------------------

# get the name of the associated person
        sqlStr03 = "SELECT fname FROM person08a WHERE perID =" + str(obj)

        cur.execute(sqlStr03)

        row03 = cur.fetchone()

# place the fname in the variable fname
        fname = str(row03[0])
        localDict['SDOS01'] = fname

#--------------------- SDOS02 ----------------------------------------------

        mycntr = jj + 2
        sqlStr04 = "SELECT objperID FROM persAssn2 WHERE assnID =" + str(mycntr)
        cur.execute(sqlStr04)

        row04 = cur.fetchone()

        obj  = int(row04[0])

        sqlStr03a = "SELECT fname FROM person08a WHERE perID =" + str(obj)
        cur.execute(sqlStr03a)
        row03a = cur.fetchone()

# place the fname in the variable fname

        fname = str(row03a[0])
        localDict['SDOS02'] = fname

#--------------------- SDOS03 ------------------------------------------

        mycntr = jj + 3
        sqlStr05 = "SELECT objperID FROM persAssn2 WHERE assnID =" + str(mycntr)
        cur.execute(sqlStr05)

        row05 = cur.fetchone()

        obj  = int(row05[0])

        sqlStr04a = "SELECT fname FROM person08a WHERE perID =" + str(obj)
        cur.execute(sqlStr04a)
        row04a = cur.fetchone()

# place the fname in the variable fname
        fname = str(row04a[0])
```

```
    localDict['SDOS03'] = fname

#--------------------- SDOS04 --------------------------------------------

    mycntr = jj + 4
    sqlStr06 = "SELECT objperID FROM persAssn2 WHERE assnID =" + str(mycntr)
    cur.execute(sqlStr06)

    row06 = cur.fetchone()

    obj  = int(row06[0])

    sqlStr05a = "SELECT fname FROM person08a WHERE perID =" + str(obj)
    cur.execute(sqlStr05a)
    row05a = cur.fetchone()

# place the fname in the variable fname
    fname = str(row05a[0])
    localDict['SDOS04'] = fname

#--------------------- SDOS05 --------------------------------------------

    mycntr = jj + 5
    sqlStr07 = "SELECT objperID FROM persAssn2 WHERE assnID =" + str(mycntr)
    cur.execute(sqlStr07)

    row07 = cur.fetchone()

    obj  = int(row07[0])

    sqlStr06a = "SELECT fname FROM person08a WHERE perID =" + str(obj)
    cur.execute(sqlStr06a)
    row06a = cur.fetchone()

# place the fname in the variable fname
    fname = str(row06a[0])
    localDict['SDOS05'] = fname

#--------------------- SDOS06 --------------------------------------------

    mycntr = jj + 6
    sqlStr08 = "SELECT objperID FROM persAssn2 WHERE assnID =" + str(mycntr)
    cur.execute(sqlStr08)

    row08 = cur.fetchone()

    obj  = int(row08[0])

    sqlStr07a = "SELECT fname FROM person08a WHERE perID =" + str(obj)
    cur.execute(sqlStr07a)
    row07a = cur.fetchone()

# place the fname in the variable fname
    fname = str(row07a[0])
    localDict['SDOS06'] = fname
```

```
#--------------------- SDOS07 -----------------------------------------

       mycntr = jj + 7
       sqlStr09 = "SELECT objperID FROM persAssn2 WHERE assnID =" + str(mycntr)
       cur.execute(sqlStr09)

       row09 = cur.fetchone()

       obj  = int(row09[0])

       sqlStr08a = "SELECT fname FROM person08a WHERE perID =" + str(obj)
       cur.execute(sqlStr08a)
       row08a = cur.fetchone()

# place the fname in the variable fname
       fname = str(row08a[0])
       localDict['SDOS07'] = fname

#--------------------- SDOS08 -----------------------------------------

       mycntr = jj + 8
       sqlStr10 = "SELECT objperID FROM persAssn2 WHERE assnID =" + str(mycntr)
       cur.execute(sqlStr10)

       row10 = cur.fetchone()

       obj  = int(row10[0])

       sqlStr09a = "SELECT fname FROM person08a WHERE perID =" + str(obj)
       cur.execute(sqlStr09a)
       row09a = cur.fetchone()

# place the fname in the variable fname
       fname = str(row09a[0])
       localDict['SDOS08'] = fname

       print json.dumps(localDict, ensure_ascii=False,sort_keys=True,indent=4, separators=(',', ': ')),","

       localDict = {}
```

B. Example PHP Web Application

The PHP script shown below uses the Solarium libraries to interface with the Apache Solr server, pass the query and render the results in HTML.

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



<?php

$time_start = microtime(true);

require(__DIR__.'/init.php');

htmlHeader();

// create a client instance
$client = new Solarium\Client($config);

// get a select query instance
$query = $client->createSelect();

// create a filterquery
$query->createFilterQuery('sdos01')->setQuery('fname:TAYLOR AND SDOS01:DORIAN');

echo '<p>QUERY: fname: TAYLOR AND SDOS01: DORIAN</p>';

// this executes the query and returns the result
$resultset = $client->execute($query);

// display the total number of documents found by solr
echo 'NumFound: '.$resultset->getNumFound();
// show documents using the resultset iterator
foreach ($resultset as $document) {

    echo '<hr/><table>';
    foreach ($document as $field => $value) {
        if (is_array($value)) {
            $value = implode(', ', $value);
        }
        if($field =='id' OR $field == 'fname' OR $field == 'SDOS01'){
        echo '<tr><th>' . $field . '</th><td>' . $value . '</td></tr>';
        }
    }

    echo '</table>';
}
```

```
$time_end = microtime(true);

$time = $time_end - $time_start;

echo '<p>Total Execution Time (loading Solarium API, accessing Solr and displaying HTML page): '. $time .' secs </p>
';

htmlFooter();
```

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

Apache Solr, A Practical Approach to Enterprise Search, 1st Edition, by Dikshant Shahi, Apress (2015) https://www.amazon.com/Apache-Solr-Practical-Approach-Enterprise/dp/1484210719/ref=sr_1_4?s=books&ie=UTF8&qid=1520362788&sr=1-4

Acronyms and Abbreviations

AI	artificial intelligence
API	Application Program Interface
AQL	ArangoDB Query Language
AWS	Amazon Web Services
CDS	Cross-domain solution
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
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
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
OE	organizational element
OWL	Web Ontology Language
PHP	hypertext preprocessor
PII	personally identifiable information
PKB	Prolog Knowledge Base
RDF	Resource Description Framework
ReST	Representational State Transfer
SaaS	software as a service
SPARQL	A recursive acronym for SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
TB	Terabyte
TDA	Table of Distributions and Allowances
TSL	Trinity Specification Language
XML	Extensible Markup Language

REPORT DOCUMENTATION PAGE						*Form Approved OMB No. 0704-0188*

1. REPORT DATE (DD-MM-YY) 30-03-18	2. REPORT TYPE Final	3. DATES COVERED (From – To)
4. TITLE AND SUBTITLE Assessment of Graph Databases as a Viable Materiel Solution for the Army's Dynamic Force Structure (DFS) Portal Implementation: Final Report		5a. CONTRACT NUMBER HQ0034-14-D-0001 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBERS
6. AUTHOR(S) Francisco L. Loaiza-Lemos, Dale Visser, Russell J. Smith		5d. PROJECT NUMBER BC-5-4277 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES Institute for Defense Analyses 4850 Mark Center Drive Alexandria, VA 22311-1882		8. PERFORMING ORGANIZATION REPORT NUMBER D-8980
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Mr. Bruce Haberkamp Army CIO/G-6 (SAIS-AOD) 5850 23rd Street, (Building 220, Room 236, Module A), Ft. Belvoir, Virginia 20060-5832		10. SPONSOR'S / MONITOR'S ACRONYM SAIS-AOD 11. SPONSOR'S / MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		
13. SUPPLEMENTARY NOTES Project Leader: Francisco L. Loaiza-Lemos		

14. ABSTRACT

This document summarizes the conclusions and recommendations made during the three preceding phases of the study and examines the maturity and applicability of two additional technologies, namely, (1) inverted index search engines, such as the open source implementation provided by Apache Solr; and (2) *key:value* pair data representations, such as the one used in JSON document serializations, for re-expressing the results of highly recursive SQL queries typically employed in large relational force structure data stores. The document specifically describes preliminary results showing how the combination of the two technologies mentioned above can be used to build interactive web applications that can be reliably keep the total data access and retrieval time of at < 1s per query. Since graphs expressed in the form of RDF triples can be easily transformed into JSON documents and vice versa, the employment of these two technologies as part of the solution architecture for the planned Army Dynamic Force Structure portal would be complementary to the use of a graph database baseline. Rapid prototyping techniques have been applied as part of the continuing evaluation of described technologies that could complement a graph database implementation. The assessments presented in this document leverage the metrics elaborated in previous reports provided to the sponsor.

15. SUBJECT TERMS

Inverted index search engines, key:value pair, Apache Solr, JSON, recursive SQL query, JSON, force structure, relational data store, web application, interactive web application

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Mr. Bruce Haberkamp
a. REPORT	b. ABSTRACT	c. THIS PAGE	Unlimited	52	19b. TELEPHONE NUMBER (Include Area Code) 703-545-1464