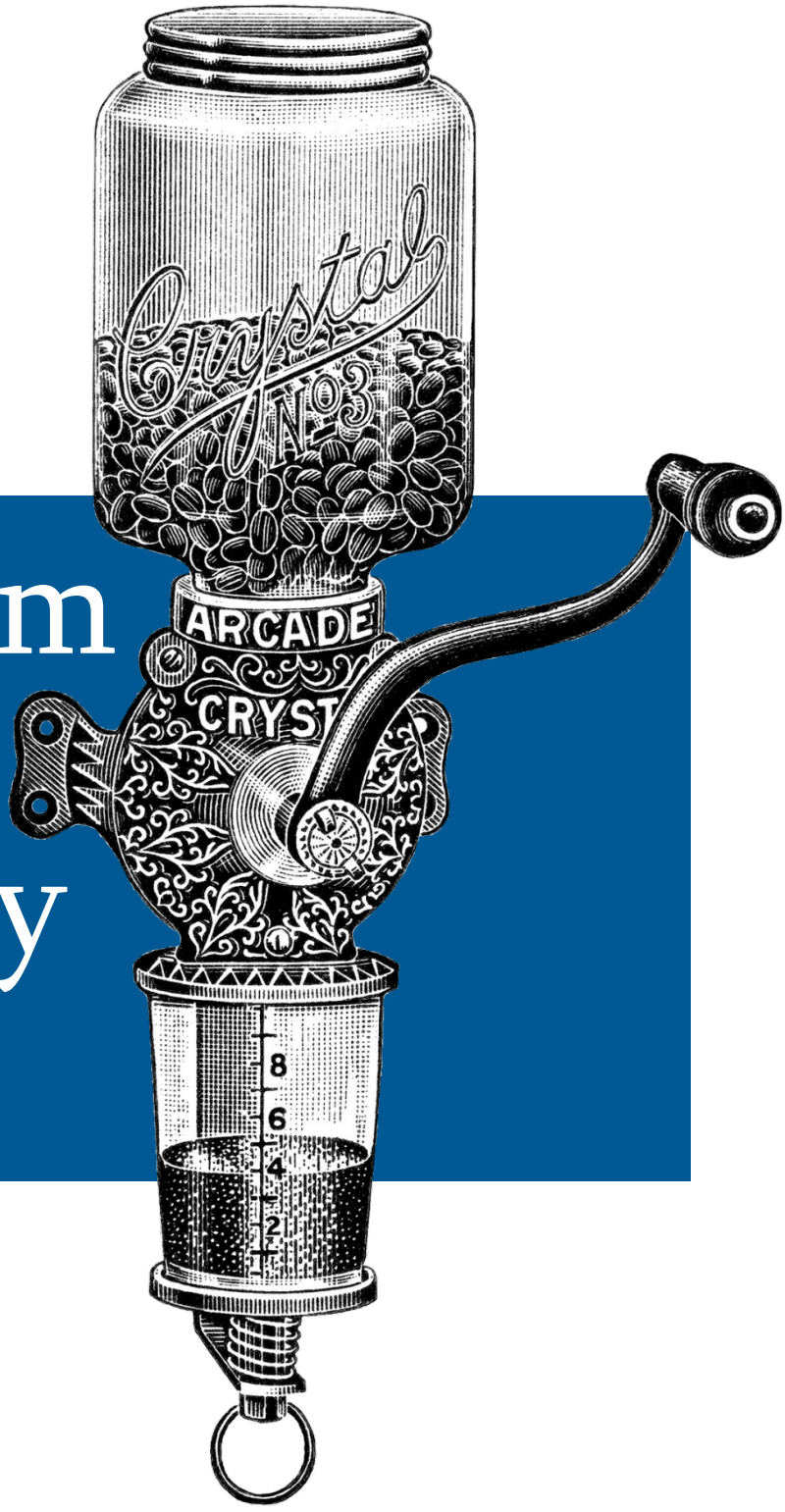
MITRE

A Field Guide

ETL from RDF to Property Graph



Dorian Voegeli

ETL from RDF to Property Graph

Dorian Voegeli

Table of Contents

Preface

Within this preface is general information about this document, such as what the document is about, how to use it, its organization, what technologies it uses for examples, its style conventions, and acknowledgements. This section introduces many terms and acronyms. Please refer to either the introduction, later chapters, or glossary for the definitions of these terms and acronyms.

About

This document is a field guide to extracting data from a database, transforming the data, and loading the data into a database. The term ETL is the shorthand reference to this activity (extract, transform, load). Specifically, this document focuses on the ETL process from an RDF (Resource Description Framework) model to a property graph model.

Usage

This document does not evaluate proprietary database technologies. However, for illustrative purposes, this document will refer to such proprietary technologies. References to any proprietary technologies are not an endorsement.

Though this document briefly introduces the concepts of an RDF model and a property graph model, this document assumes that readers are familiar with the broad terms and concepts about these models.

Organization

Each chapter in this document has items that relate to a problem area. Each item has two parts: a title and a body. The item's title is a phrase that sums up the item. The item's body will usually contain a problem description, a possible solution, or an example.

Technologies

Whenever possible, this document will refer to open standards set forth by a standards committee. If that is not possible, this document will use free and open-source software. Otherwise, this document will defer to proprietary solutions.

AllegroGraph[1] is the technology used to represent an RDF model. For data modeling, it uses the open-standards RDF, RDFS, SKOS, and OWL. For querying, it uses the open-standard SPARQL query language.

Neo4j[2] is the technology used to represent a property graph model. For data modeling, it uses a non-standard representation of nodes, edges, labels, and attributes. For querying, it primarily uses its own proprietary Cypher query language; secondarily, it uses the open-source Gremlin query language. Finally, a plugin provides limited interoperability with the SPARQL query language.[3]

Style Conventions

The following typographical conventions are in use throughout this document:

Italic

Indicates new terms or technology-related terms, such as a URL, email, or filename

`Constant width`

Used for program listings or within paragraphs to refer to programming examples

`Constant width bold`

Designates commands or other text that the user should type

`Constant width italic`

Specifies values supplied by the user or determined by the context

[1] http://franz.com/agraph/allegrograph/
[2] http://neo4j.com/
[3] https://github.com/neo4j-contrib/sparql-plugin/

Tip

Offers a tip or suggestion

Warning

Marks a warning or cautioning

Remember

Signifies an important note

Acknowledgements

Thank you, Kim Halladay, for hacking through a dense jungle of information to clear a path for this document; you have been very helpful in grounding this work. Thank you, Ram Muscu, for offering access to your modeled data and for offering insights into a general framework for organizing this document. Thank you, Bob Daniels, for the big issues related graph databases and for presenting many wonderful, open-ended questions about RDF and property graphs. Thank you, Joe Portner, for your excellent suggestions for, and edits to, the preface. Thank you, Caroline Kennedy, for your countless edits. Thank you, Justin Brunelle and Jackie Morin; for without you two, there would be no total effect that is greater than the sum of either of our individual contributions. Finally, thank you, Elizabeth, my wife.

Disclaimer

The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author.

Introduction

Within this introduction is a brief description of the extract, transform, and load process, as well as a general description of both the resource description framework and the property graph concepts. Concluding this section is a side-by-side comparison between the resource description framework and the property graph databases. This information intends to give the reader a sense of how the resource description framework and the property graph databases relate in respect to the extract, transform, and load process.

Extract, Transform, Load

The following is a summary of the ETL process [1]:

Together, the steps *extract, transform, and load (ETL)* form a single process that describes the procedure of placing a data set from one database into another. In practice, the ETL process is more complex, but in its most basic form, it consists of three steps:

- The *extract* step reads data from a specified source database and extracts a desired subset of data.
- The *transform* step works with the acquired data to convert it to a desired state by using rules or lookup tables or by creating combinations with other data.
- The *load* step writes either all or a subset of changes of a data set to a target database that may or may not have previously existed.

Usually, there are two different motivations for executing the ETL process. The first is to acquire a temporary subset of data, which has several uses, such as for on-demand analysis and for generating reports. The second is to create a permanent data set, which may be the result of a database migration or a data type conversion.

It is important to note that these three steps do not have an order. Rather, the ETL process is an iteration between these steps. In other words, each step will inform the other, so it is normal to constantly switch among each step to complete the process or to complete any step of the process.

Warning

ETL is an iterative process among the steps of extract, transform, and load.

Resource Description Framework

The *Resource Description Framework (RDF*) specifies a language for defining data items and relationships, by using a graph representation, with the intent to scale up and work across the entire Internet [2]. The *World Wide Web Consortium (W3C)*—a standardization committee for web technologies—created and maintains the RDF specification.

Numerous other standards by the W3C support RDF. Its query language is *SPARQL*, which is a recursive acronym that stands for *SPARQL Protocol and RDF Query Language*. RDF has two schema standards, which are RDFS and OWL. *RDFS* stands for *Resource Description Framework Schema*, which provides the ability to organize data items as sets and to define relationships between those sets [2]. *OWL* stands for *Web Ontology Language*, which intends to represent rich and complex knowledge about things, groups of things, and relations between things [3]. Both RDFS and OWL facilitate reasoning engines by adding formal definitions of the meaning of an object and relationship types that *inference* engines can exploit for reasoning.[4]

RDF uses a *triplestore* data model, which breaks down knowledge into statements. A *statement* combines a resource, a property, and a property value; the other name for a statement is a *subject-predicate-object assertion* [4]. For example, "Lucky is the pet of Elizabeth," where "Lucky" is the subject, "pet of" is the predicate, and "Elizabeth" is the object. Ultimately, RDF is about describing subjects by relating them to objects [5].

[4] Suppose that these assertions are in the database, "A veterinarian checks Lucky" and "Animals are checked by veterinarians," but not this assertion, "Lucky is an animal." With inferencing, the query, "show me all of the animals," will return "Lucky." Not only is inference convenient for creating new assertions from existing assertions, but it reduces modeling complexity (so that it improves query accuracy) and reduces storage (so that it increases query speed).

These assertions have the name *triples*. However, it is common to extend the triple into a *quadruple* or *quintuple*. The reason for doing so is to capture metadata about the triple. For instance, these extensions can capture the context of the triple or can hold a global identifier for indexing purposes. In practice, these quadruples and quintuples still use the name triples.

Remember

A triple extended to a quadruple or quintuple still keeps the triple moniker.

Property Graph

A *property graph* is similar to a graph in that it has *edges* and *nodes*, but dissimilar in that the nodes and edges can hold any number of *attributes* in the form of *key-value-pairs* [6]. A node can have a *label* that represents metadata, such as its context, an index, or a constraint. The edge connecting the nodes represents a *relationship* that relates two nodes to each other. A relationship always has a type, a start node, an end node, and a direction [6]. However, despite the explicit direction, navigation between nodes occurs in either direction [6]. Like nodes, relationships can have attributes, which are usually a weight, such as a distance, cost, rating, strength, or time interval [7]. Thus, a property graph is a great candidate for querying information about relationships.

As of this writing, there is not a standard for a property graph model, though efforts to create one are underway by the W3C.[5] The current alternative to a standard is the *Apache Software Foundation's*[6] *TinkerPop*[7], which is an open-source project that provides an entire stack of technologies that offers a standard interface to a graph database [8]. Two important technologies that TinkerPop provides are the *Blueprints API*, which is a framework that provides a set of generic interfaces [9], and the *Gremlin* query language, which is a powerful, domain-specific language designed for traversing graphs [8].

Warning

The property graph model does not have a standardization.

---

[5] https://www.w3.org/community/propertygraphs/
[6] This non-profit corporation supports open-source Apache software projects.
[7] http://tinkerpop.incubator.apache.org/

Comparison

Table 1 compares aspects of RDF and property graphs.[8]

Table 1 – Comparing RDF and Property Graph

	Property Graph	RDF
Features	Inferences and relationships across data Interesting visualizations from nodes and edges	
Data Type	Linked Data	
Data Model	Graph	
Graph Models	Binary, Directed, Multi	
Standardized	No	Yes
Serialization Formats	Vendor-specific	Turtle, N-Triple, N-Quads JSON-LD, RDF/XML
Query Languages	SPARQL-like, Cypher, Gremlin G, GraphLog, SoSQL, BiQL, SNQL	SPARQL, RDQL, Versa RQL, SeRQL, XUL
Metadata	Vertex or Edge Attributes	Reification
Graph Storage	Directed, Labeled, Multigraph, Hypergraph	Triples
Type-centric	Node	Edge
Graph Traversal	Optimized for Graph Traversals	Logarithmic Edge Traversal [10]
Common Perception	Pragmatic Whiteboard to Database	Semantic Web Standardized Knowledge

[8] Refer to the glossary for definitions of acronyms.

Unit One

EXTRACT

Chapter 1

HTTP Interface: Extract

Item 1: HTTP interfaces potentially allow any language to query the RDF database

Even though an RDF database will support different languages, the database may not support the particular programming language that a solution uses. Before discarding the solution due to this reason, check to see if the RDF database supplies an *HTTP interface*, which is a way to communicate with the database via HTTP (Hypertext Transfer Protocol) verbs.

If the RDF database has an HTTP interface, such as a *RESTful API (Representational State Transfer)*[1], then change the solution by converting the direct database calls to HTTP calls. This process may take some time, so decide if rewriting the solution is worth the trouble. Finally, keep in mind that, while an HTTP interface provides a broader choice of languages while implementing a solution, there may be efficiency trade-offs compared to directly accessing the database.

[1] A RESTful API uses an HTTP interface in a way that is more scalable and maintainable.

Warning – Proprietary Solution [11]

AllegroGraph has an extensive RESTful API.[2] All of the database clients that AllegroGraph provides (Sesame, Apache Jena, Python, C#, Clojure, Perl, Ruby, Scala, and Lisp) make calls to its RESTful API.

[2] As of AllegroGraph 5.0.

Chapter 2

Format

Item 2: Serialization formats depend on the access points of the RDF database

An RDF database has several access points for retrieving data. Unfortunately, these different access points do not typically offer the same *serialization* formats for the retrieved data. Serialization is the process of formatting data for either storage or transmission for future reconstruction [12]. To find all of the serialization formats that the RDF database offers, the user must first find all of the access points of the database.

There are usually three different access points for locating a particular serialization format for an ETL solution. The first access point is the database export tool. This route will probably have the least variety because the export tool is more like direct database access than queries to the database. The second access point is database queries. Also limited in variety, this route usually offers different formats than a database export tool. If available, the third access point is an HTTP interface. The user can find the *JSON* (JavaScript Object Notation) format with this route. JSON is a lightweight data-interchange format that is easy for humans to read and write and also easy for machines to parse and generate.

Warning – Proprietary Solution [11]

In AllegroGraph, the HTTP interface offers the following responses listed in Table 2.

Table 2 – AllegroGraph HTTP response codes and serialization formats

HTTP Response Code	Serialization Format
application/rdf+xml	RDF/XML
text/plain	N-triples
text/x-nquads	N-quads
application/trix	TriX
text/rdf+n3	N3
application/json	JSON
application/x-quints+json	JSON w/ unique identifier

Item 3: Determine how the database represents a triple set in JSON

With an HTTP interface, the de facto serialization format is JSON, which is a framework for structuring objects in plain text. Regarding sets of triples, a common format is arrays of string elements, such as:

```
[[“Dorian”, “RECEIVES_TASKS_FROM”, “Kim”, “work”, “32”],
 [“Dorian”, “BRAINSTORMS_WITH”, “Bob”, “work”, “33”]
 [“Dorian”, “CONSULTS_WITH”, “Ram”, “work”, “34”]]
```

The RDF database will rarely use a triple that has only three elements. Instead, the common number of elements in a triple is either four or five, which includes the context and the unique identifier.

Warning – Proprietary Solution [11]

AllegroGraph uses arrays of strings to encode RDF triples as JSON. The strings contain terms in UTF-8 (Unicode Transformation Format) encoded format, which is similar to N-triples that allow non-ASCII (American Standard Code for Information Interchange) characters. Arrays of three elements are triples in the default graph, arrays of four elements add the graph name/context as the fourth element, and arrays of five elements add the triple’s unique identifier as the fifth element.

Item 4: A good starting format is a CSV file for nodes and relationships

CSV (comma-separated values) is a simple format that stores tabular data in plain text, though there is no standard on to how to prepare the structure for importing into a property graph. As a general approach, the user can align the structure of the CSV to the structure of the property graph.

In other words, because a property graph is about nodes and relations, it may be worthwhile to structure the CSV file into nodes and relations. General candidates for nodes are usually subjects or objects, while predicates are candidates for relations. An example is illustrated in Table 3.

Table 3 – Example of CSV structure for nodes and relations

nodes.csv	relations.csv
Dorian Voegeli, Intern	Kim Halladay, TASK_LEADS, Dorian Voegeli, Summer
Kim Halladay, Employee	Dorian Voegeli, REPORTS_TO, Kim Halladay, Summer

Chapter 3

Query

Item 5: Proprietary features can add simple free-text capabilities to SPARQL queries

When there is a need to query *free-text*, which is not an explicit ability of SPARQL, the user should decide if the RDF database has any proprietary features to augment SPARQL queries with simple free-text capabilities. This can help to improve the query process. Free-text is simply unstructured text, such as a person's name.

Warning – Proprietary Solution [11]

AllegroGraph has a feature called a Magic Property, which is a predicate in a SPARQL query that produces bindings by using something other than simple subgraph matching. These extensions give a much richer query environment at the cost of non-portability.

Note that Magic Properties can use patterns with multiple inputs and outputs. SPARQL's list notation provides a syntactic sugar to make this quite readable. Here is an example that looks for text matching `willows` in the free-text index named `titles` and then binds `?book` to the matches it finds:

```
select * {
  ?book fti:match ('willows' 'titles' ) .
}
```

Item 6: Define a SPARQL query in terms of a function, where appropriate

In the case of complicated queries that differ only by a few parameters, it is advisable to find out if the RDF database has a feature to wrap such a query into a function. Though this is a non-standard feature, it will make querying easier and more accurate.

Check if the RDF database is able to bind those complicated queries into some sort of function that accepts parameters. To take this step further, use these functions as parameters to another function, thus allowing for rich, expressive queries.

Warning – Proprietary Solution [11]

AllegroGraph has the SPIN (SPARQL Inferencing Notation) API that defines a function in terms of a SPARQL query. Then, the user can call that function in other SPARQL queries. These SPIN functions can appear in filters so that the user can compute values in `ASSIGN` and `SELECT` expressions.

Item 7: Determine the validity of inferred triples

If an RDF data set incorporates RDFS terminology, then a data base can deduce new triples through inference. The database stores less data with this technique, with the trade-off of recreating the inferred triple when queried.

The main issue with inferred triples is that a query can return triples that are no longer valid. This can happen if there is not a function within the system that will back up and remove triples that were dependent upon the inferred triple.

Warning [13]

When the user removes an explicit statement, the RDF database should find and retract any inferred statements based upon the removed explicit statement. This implies that the entire inferencing process would need to rerun, thus consuming valuable computer resources and time. The lack of support for retraction means that query results can return misleading data. This advanced form of inferencing management is currently uncommon in RDF databases.

Chapter 4

Index

Item 8: Be sure to preserve the triple's unique identifier

Though not part of a standard, it is common for an RDF database to have a *unique identifier* for each triple. This unique identifier has a few uses in an RDF data set that are difficult to replicate in a property graph.[3]

Regardless, saving the unique identifier somewhere in the property graph is good in lieu of any other options. Doing so allows transformation back and forth between an RDF database and a property graph database. This can enable the reuse of queries that rely on a triple's unique identifier, as is the case with reification. RDF is edge-centric, so it may make sense to save the triple as the edge's unique identifier, as it essentially describes an edge, as shown in Figure 1.

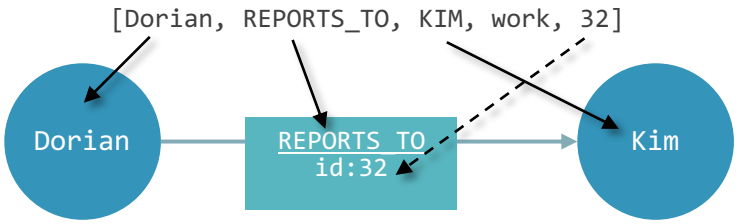


Figure 1 – Preserving the triple's unique identifier

[3] For instance, in reification, the subject has the value of a triple id; thus, the user can make statements about a statement. In a property graph, this amounts to collapsing a node-relation-node into a single node, which a property graph does not support.

Item 9: RDF index can become the graph context

A triple that has four elements is a quadruple. This fourth element can serve several purposes, but, conventionally, it is the *graph context*, which is similar to the concept of a subgraph. Another use for this fourth element is a label or a place to put properties for a triple.

This graph context in RDF can easily convert to the graph context in a property graph, as shown in Figure 2.

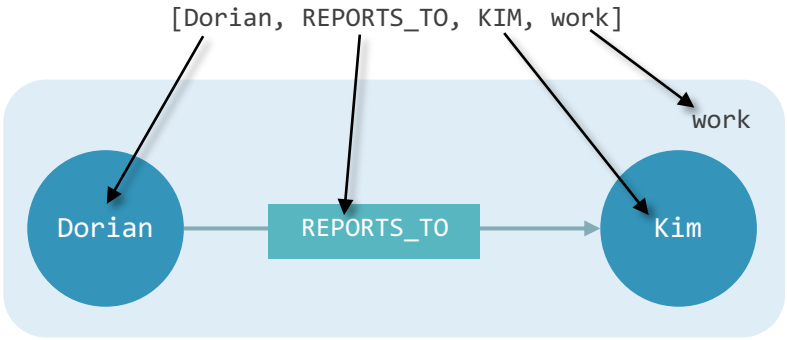


Figure 2 – Fourth element from RDF to the graph context in a property graph

Item 10: Use static analysis features to determine the optimal index

When querying millions of triples, query optimization is important. Using an index is one technique to optimize queries. There may be several ways to index data, but one that is immediately clear is to index on the elements of a triple.[4]

The usual optimization is on one of the five elements of a triple. Some RDF databases are capable of static analysis to decide if the query on the desired index matches the index available in the database. Another optimization is to find what is the best index for the query.

[4] The subject, predicate, object, graph context, and unique identifier.

Warning [11]

Where each line refers to one of the indices required by the query, the line indicates which index the query really wants and which index the query used based on what is available in the database. In AllegroGraph, the line `(desired p.o.s.g.i. optimal 6 actual o.s.p.g.i. suboptimal 4)` means that the query wanted `p.o.s.g.i.` (which would have been optimal), but it got `o.s.p.g.i.` (which was suboptimal). The three possible values are optimal, suboptimal, and full (a full scan).

Item 11: Use unique constraints to ensure unique property values

Sometimes a property is unique, such as an email or identification number. These are similar to a primary key in traditional relational database management systems. However, there is no concept of a primary key in RDF or in property graph. Identify unique elements to use as a *unique constraint*, such as a particular subject or object type, to ensure unique property values.

Warning [6]

In Neo4j, the user can rely on unique constraints to make sure that property values are unique for all nodes with a specific label. Unique constraints do not mean that all nodes must have a unique value for the properties.

Chapter 5

Literal: Extract

Item 12: Consider preserving free-text search capabilities

Free-text search may be an important feature in an RDF database that the user can take advantage of to better process a compound object literal[5]. Sometimes a linked document database stores the free-text or has a free-text search engine attached to it.

The user should decide if the property graph is able to handle free-text queries. Otherwise, the user may have to further break down the free-text into triples or load them into a separate database and link to them. Depending on the situation, linking a free-text search engine to a property graph may be worth the trouble.

Item 13: SPARQL and RDF differ in literal definitions

A *literal* is a data point in a string format, which is simply unstructured text. Despite SPARQL's use for querying RDF, these standards are not completely compatible with each other. For instance, sometimes the way they handle literal data is not compatible. Even within RDF, different specification versions handle literal definitions differently.

[5] A compound object literal is a string that can undergo further decomposition, such as a timestamp.

Warning [5]

No current or planned version of RDF allows a literal to appear in the subject or predicate of a triple. Additionally, predicates cannot be blank nodes. SPARQL does allow these, and if the user were to query an RDF data source with SPARQL, the query would never return predicates with blank nodes.

Chapter 6

Reification: Extract

Item 14: Every reification pattern has its compromises

Dealing with *reification* in RDF is not always straightforward. Reification is the process of making statements about an RDF triple. While the RDF standard includes a section on reification, it is neither elegant nor efficient.

Usually, any method of reification is verbose and inefficient in terms of storage and retrieval [14], and also clutters the data model. For instance, the user can theoretically turn every predicate into a node to attach properties to the predicate. Not only does this make little sense, it also would complicate the relationship names in addition to confusing predicates for either a subject or an object. Figure 3 illustrates this complication.

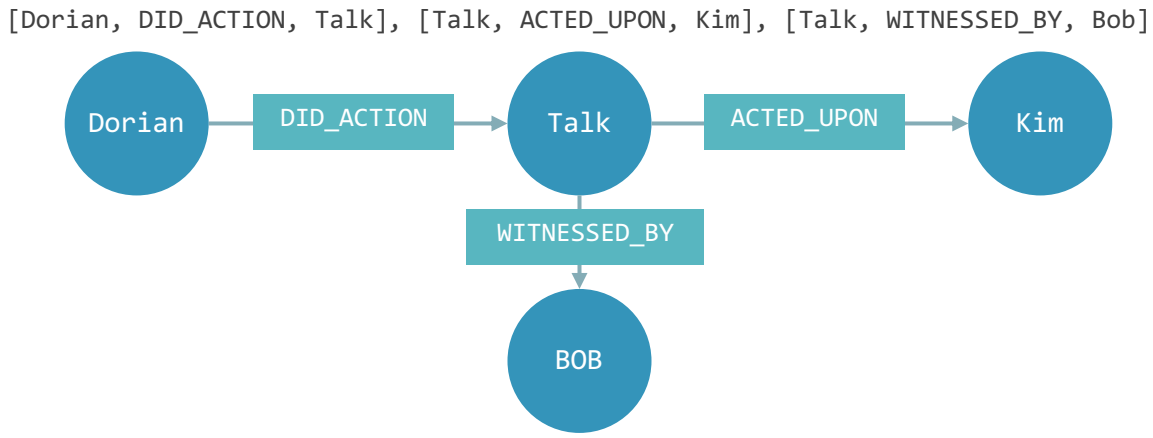


Figure 3 – Reification by converting predicates to nodes

Recording the reification process or searching for clues about the process of reification will help the user to compensate for the added information in the data model when he or she is trying to find the difference between data versus metadata.

Item 15: The triple's unique identifier in the subject or object

Most RDF databases will have a unique identifier for each triple. This unique identifier represents a whole statement, so a triple can use the unique identifier reference in either its subject or object. This reification improves model clarity and reduces the size of the data set.

However, this pattern of reification is difficult to portray in a property graph. It is very tough to transform it without losing the context.[6]

Item 16: Four extra triples that refer to a single triple

A particularly noisy reification pattern is to add four triples that function as a single triple, which is the convention that the RDF standard recommends. This pattern uses four statements that set up a given node as a triple. The four triples represent the statement, subject, predicate, and object. Then, the user can use the node in a triple, thus making statements about the triple. The basic pattern is,

```
[node, TYPE, Statement],
[node, SUBJECT, subject data],
[node, PREDICATE, predicate data],
[node, OBJECT, object data]
```

after which a user can attach statements to the node, like so,

```
[node, PREDICATE, object data]
```

Unfortunately, this convention obscures the data model, as seen in Figure 4.

[6] See Item 36: Triple ID in the subject or object is difficult to transform

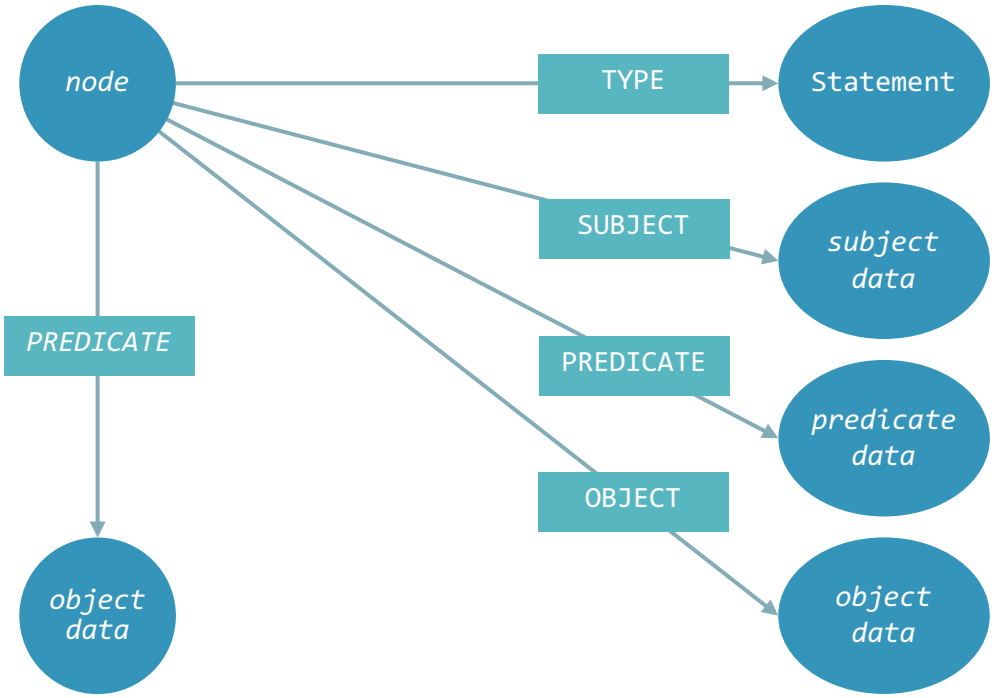


Figure 4 – Graph of reification convention in RDF specification

Item 17: The fourth element is properties on the predicate

It is common for RDF databases to extend a triple into a quadruple. Usually, this fourth element is the graph context. Sometimes, though, it represents properties of the triple.

Warning

Using the fourth element as a placeholder for properties is not the intended purpose of the fourth element, so it will cause confusion for other people working on the data set.

An example is `[Dorian, RECEIVES_TASKS_FROM, Kim, Summer Internship Period]`, where the fourth element `Summer Internship Period` reflects that Dorian receives tasks from Kim for the summer internship period.

Property graphs give a simplified version of this latter form of reification by adding a property holder for every edge and encapsulating edge attributes as key-value pairs. That is, the concept of one level of reification is inherently part a property graph [14].

Unit Two

TRANSFORM

Chapter 7

CSV: Transform

Item 18: Find and remove unintended artifacts

Commonly, CSV will contain *unintended artifacts*, which are characters that prevent the property graph from properly loading the file. Sometimes either an operating system or an RDF database can cause artifacts.

The basic approach to removing artifacts is pattern matching. A user can do this through regular expressions, though he or she should only consider using them after exhausting all other possible options. Despite regular expressions being a good choice for very complex pattern matching, they can fail in very subtle and catastrophic manners.

Warning [6]

Table 4 lists some of Neo4j's recommendations against unintended artifacts in the CSV file when using `LOAD CSV`.

Table 4 – Neo4j's recommendations against unintended artifacts

Artifact	Recommendation
BOM byte order mark (2 UTF-8) bytes at top of file	Remove them
Linked Data BOM byte order mark (2 UTF-8) bytes at top of file	Remove them
Binary zeros or other such non-text-characters	Remove them
Special character in non-quoted text	Quote them out
No unexpected newlines in quoted and unquoted text-fields	Quote text Remove newlines
Stray quotes Standalone double in the middle of non-quoted text Single quote in the middle of non-quoted text Non-escaped quotes in quoted text	Escape them Remove strays
Empty fields	Skip them Default values

Item 19: Guidelines to properly formatting a CSV

Different property graph databases require different formatting for the CSV files they will import, so the user will need to check the documentation of the property graph database. Usually, the RDF database does have a way to format the CSV properly. The transformation process usually offers more control of the formatting process, but first the user will need to decide how well the RDF database can handle formatting during the export process.

After the user finds the formatting guidelines recommended by the property graph, it is advisable that he or she automate the process of formatting a CSV file. A quick solution is a simple search-and-replace feature. Another solution is encapsulating similarly performing types of transformations into a function. Regular expressions are an option that a user should carefully consider before using it, as they work best with regular

languages, which means that, in certain instances, it's impossible to have a good solution.[7] Table 5 lists some common formatting styles.

Warning [6]

Table 5 lists some information from the Neo4j database regarding formatting a CSV file to use with `LOAD CSV`.

Table 5 – Neo4j's recommendations regarding CSV formatting

Guideline
Character encoding is `UTF-8`
Line termination is system dependent, e.g., it is `\n` on UNIX or `\r\n` on Windows
Default field terminator is ,
Character for string quotation is double quote "
Escape character is `\`
Inconsistent line breaks or mixed windows and UNIX line breaks should be made consistent, and it is best to choose UNIX style
Empty fields should have default values or be removed
Fix inconsistent headers (missing, too many columns, different delimiter in header)
Make sure that unusual text is always quoted, such as special characters in non-quoted text
Check the cypher import statement for typos
Labels, property names, and relationship-types are case-sensitive

[7] For instance, when trying to use regular expressions to parse HTML.

Chapter 8

JSON: Transform

Item 20: Make sure that the JSON file is not malformed

JSON has more formatting requirements than CSV, so caution is advisable when doing transformations of JSON-formatted data. It is worthwhile for the user to try to validate the JSON formatting before moving on from the transformation process. It may also be a good idea to run the JSON through a *linter* program[8]. A linter is a program that determines syntax validity.

Remember though, a JSON linter can only detect syntactic errors. It will not detect semantic errors. In other words, it will not help the user meet the formatting requirements for a property graph database. For example, it will not detect missing headers or empty nodes.

Item 21: JSON is usually the default format for HTTP interfaces

When accessing an HTTP interface, especially a RESTful API, the default format will commonly return and accept JSON. Most languages and most databases have some means of correctly parsing the JSON format.

[8] An online JSON linter is at http://jsonlint.com/, though an offline version is better with a workflow.

Regarding RDF and property graphs, there is a new standard emerging called *JSON-LD* (JavaScript Object Notation Linked Data). There are already a few approaches for converting between JSON and RDF.[9] The user needs to find out if his or her property graph can accept JSON-LD and if it is a good fit for his or her ETL process.

[9] The W3C has a standard API for converting between RDF and JSON-LD. The specification for the API is at http://json-ld.org/spec/latest/json-ld-rdf/

Chapter 9

Aggregate

Item 22: Reduce duplicate triples by determining the criteria of duplicate triples

While duplicates are very efficient in some contexts, in others, they can complicate the data model and the data queries. Duplicate triples can lead to inefficiencies that increase storage and retrieval times. The user can remove these duplicate triples after first determining that they are a problem.

Before removing any data, the user must first figure out the criteria for duplicate triples. This may be as simple as defining the criteria as, "duplicate triples have the same subject and predicate." Defining the criteria can also be very complex, such as in the case of object timestamps with the same month, day, and year. These would need a free-text search for a compound literal object.

After determining the criteria for duplicate triples, the user can consider if it is better to remove triples in the RDF database, in the property graph database, or in a separate process. The user can then use an automated process to remove the duplicates.

Item 23: Alignment of data points across data sets

Alignment occurs when separate data sets have common data points [15]. For instance, different databases may have the same person in their data sets. This allows teams to keep up separate data sets, but share common data points, as shown in Figure 5.

The main issue with alignment is keeping the data points synchronized. Synchronization becomes difficult to have if the data points are in different formats, which, in this case, is RDF and property graph. In this case, a simple solution would be to have the aligned data point be read-only. Otherwise, the user will need to create a lightweight, persistent ETL process during the synchronization process.

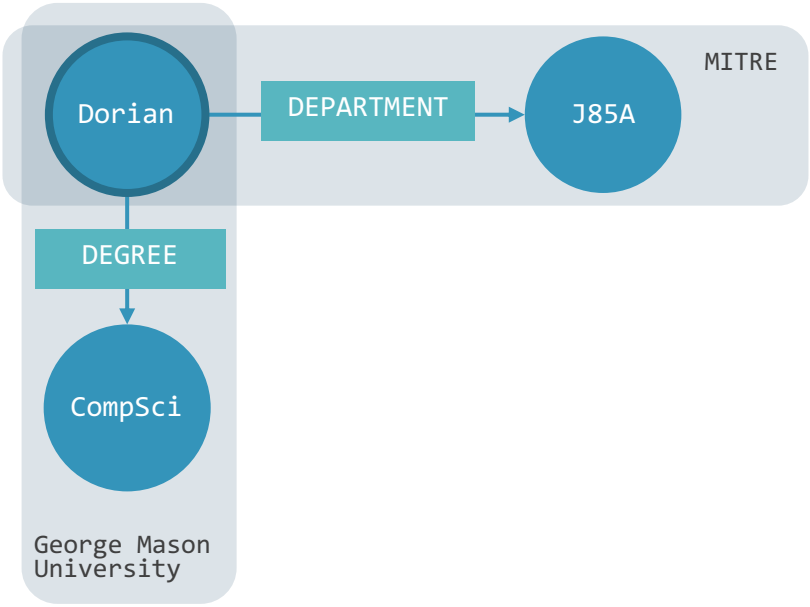


Figure 5 – Aligned data across different databases

Item 24: Consider adding or preserving equivalent relationships

It is common for data to have multiple aliases for a data point. For example, this will happen when a person has several nicknames, or if the same person has different roles in different contexts.

One technique of dealing with this duplicate data is to use a relationship that states that the data items are the same, such as a `SAME_AS` relation. Doing so makes the data input robust and aids alignment across data sets. Other databases may need to use different names in their data set, and the `SAME_AS` relation allows a database to adhere loosely to some other standard. An automated process for producing `SAME_AS` relations will make for a more efficient transformation process; however, adding such relations is a manual process.

A good place to store a `SAME_AS` relationship in a property graph is in the node properties. In Figure 6, the node `Dorian` has several nicknames stored as properties, which show the `SAME_AS` relationship.

Figure 6 – Equivalent relationship in a property graph

Item 25: Predicates and object types help decompose compound literal objects

Objects can have values that are a literal, which is actually just a string. Such a literal is commonly a *terminal node*, which means that they can no longer be decomposed. An example of this is a person's name. There are no constraints as to what is in a name, so it is best to treat it as a literal.

However, there are situations where the user can decompose a literal—for example, a timestamp. A timestamp is a specially formatted literal. The format follows some convention[10] that informs the user about how to parse the timestamp.

The user should make sure to capture that convention information when he or she is exporting. This information can come from the label on either the object or the predicate of the triple.

Remember [11]

In AllegroGraph, the functions `toPointXY` and `toPointLonLat` need the predicate argument to decide which geospatial subtype to use to construct the point.

[10] Usually, this is the name of the convention, such as the ISO 8601 standard.

Chapter 10

Translate

Item 26: Be careful when translating a SPOGI index

Most RDF databases have something like the *SPOGI index*, which is an index of each element of the quintuple. SPO stands for subject, predicate, object, and GI stands for graph and a unique identifier.

Unless the RDF data is very basic, translating the SPOGI index may be complicated, usually due to unconventional patterns associated with the quadruple and quintuple. This particularly happens with certain types of reification, especially those that use the fourth element to hold properties instead of the graph context.

Item 27: Translate to an object representation

A property graph works very well with the concept of an object model. An *object model* is an entity that has properties. Nodes and edges in a property graph can have properties, so, in this regard, a property graph is similar to an object model. JSON is an object model, which makes it very easy to translate for a property graph.

Translating an RDF triple to an object model representation may make the import process easier. The user can look for patterns where it seems like there are objects with properties. Quite often, a predicate and an object can become a key-value property on a node or edge. In addition, the user should look out for the pattern of storing properties in the fourth element of a quadruple.

Chapter 11

Encode

Item 28: Be careful with the types on predicate-object properties

Most of the time, RDF types will translate to labels in property graphs, or they can serve as labels for indexing. However, there are some cases where this does not work.

This does not work with a predicate-object property. A predicate-object property is a triple and describes a property of a subject. For example, in Figure 7, with the triple `[Dorian, WEIGHT, 150]`, the predicate and the object form a property of the subject. However, triples usually have a type, as in `[Dorian: intern, WEIGHT, 150: lbs]`. When transforming such a triple into a property graph, it creates a situation where properties have properties, which is usually not directly possible in a property graph.

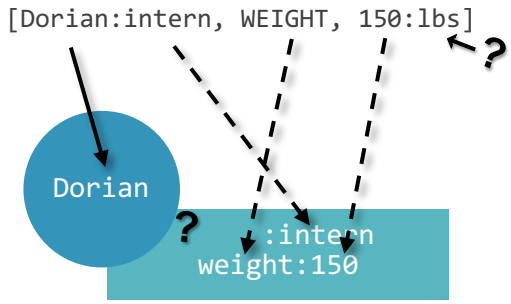


Figure 7 – A property group cannot have properties of properties

Item 29: Preserve the classification capabilities of RDFS

If triples use the class-describing ability of an RDFS, it is possible to translate the class structures into the property graph. In addition, through the class structuring of RDFS, it is possible to use the class as an index in the property graph. Finally, the user should try to translate as much information about the RDFS classes in his or her triples, as the classes may be invaluable while creating graph contexts and relationships.

Tip[4]

RDFS does not offer application-specific classes and properties. Instead, RDFS provides the framework to describe application-specific classes and properties. Classes in RDFS are much like classes in object-oriented programming languages. This allows a user to define resources as instances of classes, and subclasses of classes. The following RDF / XML format describes an intern as a subclass of the employee:

```
<rdf:RDF
   xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
   xmlns:rdfs=http://www.w3.org/2000/01/rdf-schema#
   xml:base="http://www.mitre.fake/employees#">

   <rdfs:Class rdf:id="employee" />

   <rdfs:Class rdf:id="intern">
      <rdfs:subClassOf rdf:resource="#employee"/>
   </rdfs:Class>
/>
```

Chapter 12

Transformation

Item 30: Consider when to use either a direct or indirect transformation

A direct transformation is a triple that transforms into a node-relation-node. Such is the case when a subject-predicate-object is describing a relation between two entities. Figure 8 illustrates a direct transformation between two entities.

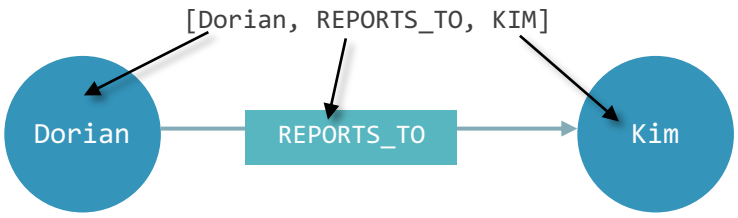


Figure 8 – Direct transformation from RDF to property graph

However, there may be many cases where a direct transformation is not possible. The most predominate example of this is a predicate-object property in RDF.[11] The predicate and object paring form a key-value property, which is not a one-to-one transformation. Figure 9 illustrates an indirect transformation.

[11] See Item 28: Be careful with types on predicate-object properties

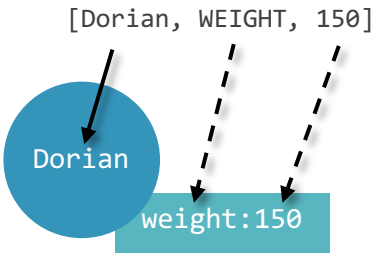


Figure 9 – Indirect transform of predicate-object property from RDF to property graph

Item 31: Property graph properties cannot contain nodes

Due to the differences between RDF and property graphs, there are cases where a transformation will cause significant differences in the data model. This is usually common with reification.

For example, there is a reification pattern to convert a predicate into a subject, which adds an extra node to the model so that the predicate can have properties.[12] If a user tried to transform this model into a property graph, he or she would end up with a situation where it is convenient to make a node represent a value. Unfortunately, this is not possible in a property graph.

[12] See Item 14: Every reification pattern has its compromises

Chapter 13

Linkage

Item 32: Be careful to preserve links to other databases

A way to enhance a data set is to link it to another data set or, alternatively, to import a data set. For several reasons, it may not always be possible to import another data set. Instead, the user may need to create links to another data set. This most commonly occurs when the data set is very large. Usually, queries to these external links return read-only data.

How this linking occurs will complicate the ETL process. The user should decide whether a property graph supports linking or if an external solution is needed, such as querying an HTTP interface of the database. In addition, to avoid inadvertently altering the data set, the user should take caution when accessing an HTTP interface.

Item 33: Be on the lookout to preserve links to documents

An RDF database can have links to documents through which the user can search using either a free-text ability or a free-text search engine. The free-text search is usually an external program integrated into the RDF database.

Triples can also represent connections between databases and documents that contain free-flowing, unstructured text. These connections are also extremely valuable. They can link entities from the database to the documents that mention them. Some RDF databases have specific support for such connections, which allow data and text to form a single,

interconnected information space. That information space allows access through hybrid queries, combining the richness of the full-text search with the selectivity and precision of a database [13].

However, the property graph database may not natively support this. One solution is to create a server layer application that sits between the two. This could entail creating an HTTP interface to query both databases. Doing so is typically a significant effort with little chance to create a generalized, re-usable solution—at least, not easily [16].

Chapter 14

Literal: Transform

Item 34: Beware of a literal in the place of either the subject or predicate

The RDF specification does not allow a literal in the place of subjects or predicates.[13] Breaking this rule causes many problems, such as losing inference abilities and reference ambiguity.

Usually, in a property graph, a good place to put a literal is in the properties of either a node or relationship. Transforming a literal into either a node or edge is bad data modeling. It is right to either turn the literal into a legitimate node or edge or to place it into the properties of a node or edge.

Item 35: Watch out for either missing or dropping a literal's type

A literal can also be a compound literal, and, with such a literal, decomposing is necessary. The compound literal is serialization of some sort that requires a method to decompose it. The decomposing method is commonly a standardization. For instance, a

[13] According to Section 3.4 of the W3 definition of RDF 1.1 Concepts and Abstract Syntax, "A literal may be the object of an RDF statement, but not the subject or the predicate."

timestamp is a compound literal with a standard method to decompose it. Commonly, a user stores the reference to the standardization as the type.[14]

The literal's type will point the user to a specification that has the correct method of decomposing the compound literal. Without the literal's type, it may be impossible to decompose the literal, which can, in turn, cause the user to instead misinterpret the compound literal. In addition, even after decomposing a literal, the user should try to not drop the type if they expect to re-aggregate it later.

[14] A real example of a standardized timestamp is the ISO 8601 standardization, which can be the timestamp's type (e.g., `type:ISO_8601)`.

Chapter 15

Reification: Transform

Item 36: Triple unique identifier in the subject or object is difficult to transform

Using a triple ID for the subject or object is a very elegant pattern of reification in RDF [15], but this pattern of reification is difficult to portray in a property graph.

One method is to make the predicate-object of the reified triple a key-value property on the original statement, as in Figure 10. However, in this pattern, it is not clear that `WITNESSED_BY->Bob` refers to the node-relation-node `Dorian-TALKED_TO->Kim`, instead of to the relation `TALKED_TO`, which is what it seems in Figure 10.

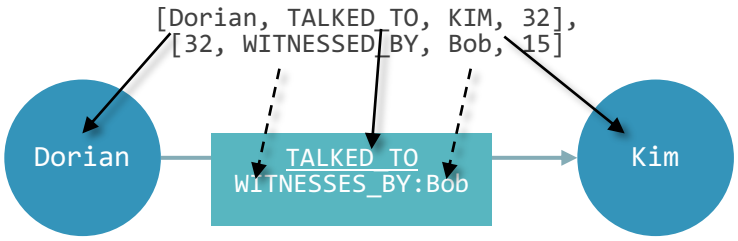


Figure 10 – Reification by using a unique identifier in the subject or object

[15] See Item 15: Triple ID in the subject or object

Item 37: Beware of reification that uses blank nodes

Adding four extra triples that refer to a triple[16] will not only complicate the property graph model during the transform process, but sometimes a direct transformation is not possible.

Direct transformation is not always possible because this pattern of reification can rely on attaching the four triples to a blank node; the problem is that a property graph does not allow blank nodes. Therefore, in the case of this type of reification, careful consideration must go into developing an automated process that can handle this case.

Item 38: Using the fourth element for properties of a predicate

The fourth element of an RDF sometimes holds properties on a predicate.[17] This type of reification is the easiest to transform, but only if the user is aware of the reification type, as illustrated in Figure 11.

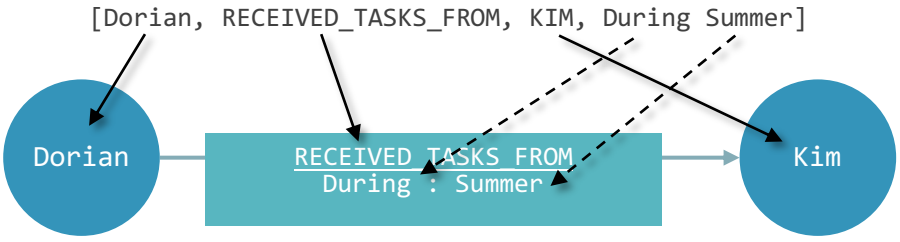


Figure 11 – Transforming the properties on a predicate with the fourth element

[16] See Item 16: Four extra triples that refer to a single triple
[17] See Item 17: Fourth element is properties on the predicate

Unit Three

LOAD

Chapter 16

HTTP Interface: Load

Item 39: Differences between built-in import tools and HTTP interfaces

It is advisable that the user check the trade-offs for the different approaches to loading data into a property graph. There are usually at least two methods for loading data into a property graph: a built-in import tool and an HTTP interface.

The main difference between these approaches is their *ACID* compliance. ACID stands for atomicity, consistency, isolation, and durability. A database needs these properties to guarantee successful processing of a query. The built-in import tool is faster because it loads data directly into the data set, thus circumventing ACID compliance. In the case of an HTTP interface, because loading data occurs through queries, the import will be ACID compliant. However, if a database import tool wraps the HTTP interface, then neither approach will offer an advantage over the other.

Chapter 17

Import

Item 40: Memory cache affects import efficiency

It is noteworthy that importing a data set will need a certain amount of memory for the database cache and heap sizes. The database needs this memory as a staging area for the import transactions.

Usually, there is a formula to control the ideal sizes for cache and heap sizes, and the user can look for it by checking the property graph's documentation. If the documentation does not contain a formula, it is simple to derive an approximate formula by the number of nodes, relationships, and properties that the user is trying to import. The example below is a good example of how to create a formula for approximating the memory size.

Warning – Proprietary Solution [17]

In Neo4j, make sure to increase the heap size generously, especially if importing large data sets. In addition, the user needs to make sure that the file buffer caches fit the entire data set.

For example, if the user is going to import 100K nodes, 1M relationships, and a fixed-size property per node/relationship (i.e., an integer number), then these are the smallest values that the user should use in the file buffer cache configuration.

Nodes:	100,000	*	15B	=	1.5 MB
Relationships:	1,000,000	*	34B	=	34MB
Properties:	1,100,000	*	41B	=	45.1 MB

Item 41: Gremlin is an emerging property graph query language

If the user has maximum portability in mind for data sets, then the user will want to use standards or at least open-source material that the community considers the de-facto standard. One such language is Gremlin, which is under the care of the Apache Foundation. Gremlin is declarative, and the user should consider that when determining if it is a good choice versus the native language of the user's property graph database.

The problem with Gremlin is that it must go through a few layers before reaching the property graph. Gremlin communicates to the property graph through the Blueprints API, with the implication being that the property graph will need to implement the Blueprints API. In other words, the Gremlin language merely wraps the database's native language.

Item 42: Try a periodic commit for large data sets

The main risk with committing large data sets into the database in one pass is that the process could encounter an error and thus fail. This could mean hours of time waiting for the commit to finish, only to have it fail at the end. Depending on the database, this could be a total rejection of the data, or it could mean a corrupted commit. There are a few ways around this, either manually or automatically.

The user can decide if the property graph has a feature like a *periodic commit*, which guarantees a commit at some determined interval. If the property graph database does not have a periodic commit, then the user can consider making a custom application for a periodic commit.

Warning – Proprietary Solution [6]

In Neo4j, when importing a large amount of data (more than 10,000 rows), the user should prefix the `LOAD CSV` clause with a `PERIODIC COMMIT` hint. This allows Neo4j to regularly commit the import transactions to avoid memory churn for large transaction-states.

Item 43: Setup index to speed up importing

With a property graph, an index allows faster updates to the data set; therefore, the user needs to set up the indexes and constraints on the data. This is faster because the database references what part of the graph needs to be updated.

Setting up the indexes and constraints on the data also makes importing data more efficient because look-up times of where to place data will be faster. This is the difference between committing new data in a matter of minutes versus a matter of hours—or worse, not committing at all. Setting up indexes and constraints is especially important if the user wants to make this a permanent data set.

Warning – Proprietary Solution [17]

In Neo4j, indexes will do lookups faster during and after the load process. The user should make sure to include an index for every property that is used to find nodes in `MERGE` queries.

A user can create an index with the `CREATE INDEX` clause. Example:

```
CREATE INDEX ON :User(name);
```

If a property is unique, adding a constraint will also implicitly create an index. For example, if the user wants to make sure not to persist any duplicated user nodes in the database, then the user needs to use a constraint for the email property.

```
CREATE CONSTRAINT ON (u:User) ASSERT u.email IS UNIQUE;
```

Item 44: Backup data before attempting a batch import

Usually, a batch importer is used for placing a data set into a newly created database. With this in mind, it usually will not offer an ACID guarantee. However, the user may want to use the tool to import into an already populated data set. If this is the case, then the user should make sure to back up the data set before doing so.

An issue that comes with this approach is the difficulty in verifying if a particular transaction is correct.

Warning – Proprietary Solution [6]

Pertaining to Neo4j, the batch insert feature will import data into an existing database, although it is a less common use case. However, the user will need to shut down the existing database before he or she writes to it.

The batch importer bypasses transactions, so it is possible to experience data inconsistency if the import process crashes midway. The user must back up his or her existing database before using the batch insert feature.

Chapter 18

Idempotence

Item 45: Use a repeatable (idempotent) update operation

It is common for multiple teams to load data into a data set. This may mean an increase in attempting to load duplicate data, which is like denormalized data. The user should try to put safeguards into place to prevent a user from updating the database with duplicate data, as all that this will do is lock up database resources.

An easy solution to this problem is for the user to check if the property graph database has some sort of idempotent commit feature. When the data is already in the database, the *idempotent updates* silently fail because the database can tell that the data is already present. Additionally, this usually prevents a log entry, which makes troubleshooting less difficult.

Warning – Proprietary Solution [6]

In Neo4j, when the user gets data from external systems or is not sure if certain information already exists in the graph, he or she needs access to a repeatable (idempotent) update operation. In Cypher, Neo4j's query language, `MERGE`, has this function. `MERGE` combines `MATCH` and `CREATE`, which allows it to check if the data exists first before creating the data. With `MERGE`, the user defines a pattern to find or create. Usually, as with `MATCH`, the user only wants to include the key property to look for in his or her core pattern. `MERGE` allows the user to set more properties when using `ON CREATE`.

Chapter 19

CSV: Load

Item 46: The structure of the CSV files determines the query statements

During the import phase, the structure of the CSV determines the query statement for loading the data. Because of this, the user may want to explore this loading phase first.

Additionally, the user can try to find a common pattern in the CSV files to create queries for nodes and relationships. This will help during both the export and the transform phases.

Warning – Proprietary Solution [6]

For example, with the following in Neo4j's Cypher language,

```
Nodes.csv                          Relations.csv
Node,Name,Label                From,Name,Relationship Type,To,Name
1,Dorian,Intern                1,Kim,TASK_LEAD_OF,1,Dorian
```

For the nodes:

```
CREATE (n:#{Label} {id:{Node}, name:{Name}}) RETURN *
```

For relations:

```
MATCH (from: {id:{From}}), (to {id:{To}}) create from-[:#{Relationship Type}]->to RETURN *
```

Notice that the two name fields are missing.

Item 47: Load a line of the CSV file before proceeding

Errors are bound to happen when loading data, so it is a good idea to catch those errors as early as possible. The user should try to not load thousands of nodes or relationships before learning this lesson, as it is difficult to expect how the database will parse the data. For example, a CSV file that is opened in a spreadsheet program may automatically format each year as a two-digit number, instead of a four-digit number. If the database expects a four-digit number for the year, then there will be a loading error.

The user should load a single line of the CSV before committing to loading the rest of the file. However, if this one line is error-free, that does not mean that the rest of the lines are error-free, though it is indicative that the rest of the file will not cause errors if it is similar to the first line. The user should come up with a format that will work, and then alter his or her queries to anticipate errors with the current data set.

Chapter 20

JSON: Label

Item 48: Use a function to traverse the JSON structure

The user should find out if the property graph is able to import JSON files, as the JSON structure can translate well into property graphs.

Usually, if the property graph accepts JSON, then it is also a feature to parse, or walk through, the JSON structure. The ability to walk through a JSON structure will make loading queries more human-readable, and thus create fewer errors.

Warning – Proprietary Solution [6]

In Neo4j, the Cypher statement `UNWIND` expands a collection into a sequence of rows. This works for JSON, which is essentially a container for collections. With `UNWIND`, the user can transform any collection back into individual rows.

One common usage of `UNWIND` is to create distinct collections. Another usage is to create data from parameter collections that the user provides to the query. `UNWIND` requires the user to specify a new name for the inner values.

Chapter 21

Label

Item 49: Uniqueness constraints for nodes with a specific label

In many cases, the user will want a certain level of uniqueness within the data set. Uniqueness constraints are important because they can help to cut duplicates in the data set. This particular item focuses on labels as uniqueness constraints. The user should consider if the property graph is capable of enforcing constraints upon labels. Uniqueness constraints do not mean that all nodes have to have a unique value for the properties [6].

If a label has a uniqueness constraint, then properties of the node or edge are unique (e.g., a user and his or her email address may only exist once in a system). If multiple, concurrent threads try to create the user, then this strategy will reduce the duplicates [6].

Item 50: Create an index from a label

The user can create an index on just about every element of a property graph, not just nodes and relationships; labels will allow the user to create indexes on properties.

In order to create an index for properties, the properties need to have some other constraint, such as a specific label.

Warning – Proprietary Solution [6]

In Neo4j, to create an index for a property, for all nodes that have a label, use `CREATE INDEX ON`. Note that the index is not immediately available because Neo4j has to create it in the background.

Chapter 22

Relationship

Item 51: Use properties for information about relationships

What makes a property graph very useful is that the user can put properties on nearly everything. Properties are not just for the nodes (objects and subjects in RDF) of the property graphs, but also for the relationships (predicates in RDF).

In RDF, the various methods of reification are the only techniques for placing properties on relationships. There are several methods of reification for placing properties on a predicate, so the user will have to recognize and transform these predicate properties. This becomes particularly cumbersome if the data set has multiple methods of reification.

Tip [18]

Unlike the RDF data model and SPARQL query standard, a property graph database enables the user to add properties to relationships, instead of only to nodes. In addition, a property graph database can contain metadata about relationships. A good example is in-car route planning, where a user will store each town as a node. The distance between each town is a property of the relationship.

Item 52: A relationship does not always imply a bi-directional traversal

Typically, the user will need to create an explicit direction for a relationship between nodes. However, in some property graphs, this direction is not explicit, and traversal happens in both directions. Note that just because traversal can happen in either direction, this does not imply the relationship is symmetrical, such as the case with an employee-employer relationship.

Item 53: Undirected relationship for arbitrary relationships

In property graphs, a relationship's direction is sometimes symmetrical. This can occur with something like a `SAME_AS` relationship. For example, this is the case when a person has several aliases.

In this situation, the user can check if there is a special syntax to denote this relationship. Otherwise, it is necessary to add two relationships between two nodes (one relationship going in each direction), and then add properties to each relationship to signify the symmetrical relationship. This can add a lot of noise to the data model.

Warning – Proprietary Solution [6]

In Neo4j, when it is the case that a relationship's direction is arbitrary, the user can leave off the arrowhead. `MERGE` will then check for the relationship in either direction and will create a new directed relationship if no matching relationship exists.

Chapter 23

Transform: Load

Item 54: Order-free RDF triples and node-dependent structure of a property graph

The most important thing to notice between RDF triples and a property graph is their structures. RDF triples do not have an order, while a property graph is a node-dependent structure. These differences will cause problems throughout the extract, transform, and load processes.

This structural difference can mean that the user will come across triples that do not fit into the structure of the property graph. The best tip to avoid this is to carefully model the RDF triples to avoid later difficulties in deriving a model from a set of order-free RDF triples.

Appendix A

Modeling

Tips about modeling in the ETL process

Here are a few tips on how to start the ETL process [19]:

- Remember that modeling is important.
- Create a graph model first.
- Clarify what are entities, relationships, and properties.
- Focus on the use cases.
- Be conscious about a data model.
- Do not reconstruct data structures from other databases.
- Import into a graph model.
- Add graph indexes later in the process.

The data model is different from data representation

The model, also known as a whiteboard graph, is different from the representation. A *representation* is the implementation in the database. A *model* is the concept to which the representation refers; for example, the diagram on the whiteboard is the model, and the RDF triple data set is the representation. In addition, a model transformation is more complex than a representation transformation.

Appendix B

Inference

There are several standard profiles for inference

There are several standard profiles, or languages, for inference. These include RDFS and the three profiles of OWL 2 – RL, DL, and QL. Only RDFS OWL 2 RL and OWL 2 QL are right for applications that need to deal with large volumes of data. The user should investigate which profile best fits the needs of a domain, data, and application. Then, the user can check whether or not the vendor provides full support for the standard profiles, particularly the one that best fits the needs of an application. Finally, the user should check if the inference engine passes standard compliance tests and if it supports independent verification and evaluation [13].

Appendix C

RDF with Property Graphs

The industry's leading products are rapidly changing and are becoming more like each other, in that they are trying to offer both RDF and property graph capabilities.

Open-source software for using RDF and SPARQL with property graphs

TinkerPop[18] is an open-source project that provides an entire stack of technologies within the graph database space. The Blueprints API is at the core of this stack of technology. The Blueprints API is like the JDBC (Java Database Connectivity)[19] of graph databases. By providing a set of generic interfaces, it allows a user to develop graph-based applications without introducing explicit dependencies on concrete graph-based database implementations. Additionally, the Blueprints API provides concrete bindings for the Neo4J, OrientDB, and Dex Graph Databases. In addition to the Blueprints API, the TinkerPop team developed an entire range of graph technologies, including Gremlin, a powerful, domain-specific language designed for traversing graphs. Hence, once a Blueprints API binding is available for a particular graph database, an entire range of technologies is available [9].

[18] http://tinkerpop.incubator.apache.org/

[19] This technology is a generic method of connecting to databases through Java.

There is a difference between SPARQL and SPARQL-like

Warning – Proprietary Explanation [20]

All of the input and output of the Neo4j SPARQL Plugin[20] is JSON. The user sends data and queries to Neo4j that are embedded in JSON, and the results are JSON, but not the W3C SPARQL Query Results JSON Format. This use of JSON is the default for the Neo4j RESTful API, which provides the context for all SPARQL-oriented communication with a Neo4j server. While the plugin's documentation refers to an endpoint, it is not a SPARQL endpoint in the sense that it supports the SPARQL Protocol, but rather, that it has its own interface for accepting SPARQL queries and delivering results.

[20] https://github.com/neo4j-contrib/sparql-plugin/

Appendix D

Error Messages

Determine the approach that provides the most-robust error messages

Different error messages will occur in different contexts. The user needs to find the approach that has the most-robust error messages. Even if the user chooses to not use the approach with the best error messages, it will allow the user to have good intuition of what is happening with environments that have less-robust error messages. The shell that comes with a database usually offers the best error messages.

List of Tables

List of Figures

Glossary

A

Alignment—This occurs when separate data sets have common data points.

Apache Software Foundation—This is a non-profit corporation that supports open-source Apache software projects.

Atomicity, Consistency, Isolation, and Durability (ACID)—These are properties that a database needs so that it can guarantee the successful processing of a query.

B

Bison Query Language (BiQL)—This graph query language is for representing, manipulating, and transforming information networks.

Blueprints API—This framework provides a collection of generic interfaces for property graphs capabilities.

C

Comma-Separated Values (CSV)—This simple format stores tabular data in plain text.

Compound Object Literal—This string can undergo further decomposition, such as a timestamp.

Cypher—This is the declarative graph query language for Neo4j that allows for expressive and efficient querying and updating of the graph database.

D

Data Point—This is a discrete unit of information.

Data Set—This is a collection of related data points.

Database Model—This is a type of data model that determines the logical structure of a database and fundamentally determines in which manner data can be stored, organized, and manipulated. For example, the diagram on a whiteboard is the model.

Database Representation—This is an implementation of a model within the database. For example, the RDF triple data set is the representation.

E

Extract, transform, and load (ETL)—This process describes the procedure of transferring a data set from one database into another.

Extract—This step reads data from a specified source database and extracts a desired subset of data.

F

Free-Text—This is unstructured text, such as a paragraph of text.

G

Graph Context—This is like the concept of a subgraph.

GraphLog—This query language uses graphs for expressing data, queries, and generic rules.

Gremlin—This declarative query language has traversal operators that chain together to form a path-like expression.

G—This high-level, dataflow graphical programming language is for developing interactive applications that are executed in parallel by using multicore processors.

H

HTTP Interface—This interface is a communication and data-transfer process using HTTP verbs (e.g., GET and POST). The advantage of this interface is an independence from a particular programming language, software, or operating system.

I

Idempotent Update—This update operation will silently fail, thus sparing the use of database resources, or creating extraneous log entries.

Inference—The process of having a database use RDFS or OWL vocabulary to prove new RDF triples that are based off the current data set of triples.

J

Java Database Connectivity (JDBC)—This technology is a generic method of connecting to databases through Java.

JavaScript Object Notation (JSON)—This format is a lightweight data-interchange format that is easy to read and write for both humans and machines.

JavaScript Object Notation for Linked Data (JSON-LD)—This format is for transporting Linked Data by using the JSON format.

K

Knowledge Organization Systems (KOS)—These are systems such as thesauri, classification schemes, subject heading systems, and taxonomies.

L

Linked Data—This format is for publishing structured data so that the data can interlink to become more useful through semantic queries.

Linter—This is a program that determines syntax validity.

Literal—This is a data point in a string format.

Load—This step writes either all of the subset or just the changes of the data to a target database that may or may not have previously existed.

N

Notation 3 Logic (N3)—This non-standard serialization is similar to Turtle, but has some additional features, such as the ability to define inference rules.

N-Quads—This superset of N-Triples is for serializing multiple RDF graphs.

N-Triples—This simple, easy-to-parse, line-based format is similar to Turtle, though not as compact.

O

Object Model—This is an entity with properties.

P

Periodic Commit—This is a process guaranteeing a commit at some determined interval.

Property Graph—This is a graph that supports labels, edges, and nodes, all of which can hold any number of attributes in the form of key-value-pairs.

Provenance—This information is about entities, activities, and people that are involved in producing a piece of data or thing; this is useful in forming assessments about its quality, reliability, or trustworthiness.

PROV—This is a W3C family of documents that defines a model, corresponding serializations, and other supporting definitions to enable the interoperable interchange of provenance information.

R

RDF Data Query Language (RDQL)—This language is for extracting information from RDF graphs and is a precursor to SPARQL.

RDF/XML—This syntax is to serialize an RDF graph as an XML document.

Reification—This is the process of making statements about an RDF triple.

Representational State Transfer (REST)—This is a system of guidelines to make an HTTP interface more scalable and maintainable.

Resource Description Framework (RDF)—This W3C standard defines data items and relationships in a triple-format that is similar to a graph representation.

Resource Description Framework Schema (RDFS)—This W3C standard is for defining basic aspects of data models for data that is in RDF format, and it provides the ability to organize data items as sets and define relationships between those sets.

Resource Query Language (RQL)—This query language is for use in URIs with object-style data structures.

S

Semantic Web—This W3C extension of the web that promotes common data formats and exchange protocols, especially regarding RDF.

Serialization—This is the process of formatting data, for either storage or transmission, that is reconstructed at a later time.

Sesame RDF Query Language (SeRQL)—This language for RDF/RDFS combines features from other languages, mainly RQL, RDQL, N-Triples, and N3, and is part of Sesame.

Sesame—This system/framework for storing and querying RDF data includes various storage back ends, query languages, inference engines, and client-server protocols.

Simple Knowledge Organization System (SKOS)—This specification and standard supports the use of KOS, such as thesauri, classification schemes, subject heading systems, and taxonomies, within the framework of the Semantic Web.

Social Network Query and Transformation Language (SNQL)—This query language does social network matching and social network construction.

Social Query Language (SoQL)—This SQL-like query language is for social networks that are focused on identifying groups and paths over a classical network.

SPARQL Protocol and RDF Query Language (SPARQL)—This query language by the W3C specifies queries against data graphs that use the RDF format.

Subject, Predicate, Object, Graph, Identifier (SPOGI) Index—This is an index of each element of the quintuple, where SPO stands for subject, predicate, object, and GI stands for graph and unique identifiers.

T

TinkerPop—This open-source project provides an entire stack of technologies within the graph database space.

Transform—This step works with the acquired data to convert it to a desired state by using rules or lookup tables or by creating combinations with other data.

Triple Set—This data model breaks down knowledge into subject-predicate-object assertions.

Turtle—This serialization format allows RDF graphs to be represented in a compact and natural text form with abbreviations for common usage patterns and data types.

U

Uniform Resource Identifier (URI)—This string of characters provides a unique identity to an internet resource. This may be either a uniform resource locator or unique resource name.

Uniform Resource Locator (URL)—This reference to a resource specifies the location of the resource on a computer network and a mechanism for retrieving it.

Unique Resource Name (URN)—This string of characters identifies a name of a web resource.

Uniqueness Constrains—These are constraints to making a data point unique so as to prevent duplicate data points.

V

Versa—This compact query syntax is in 4Suite with Python and is not based on SQL.

W

Web Ontology Language (OWL)—This language can represent rich and complex knowledge about things, groups of things, and relations between things.

Web Ontology Language (OWL)—This W3C standard is a superset of RDFS and provides extensive techniques for defining and relating sets of data items, such as set operations, cardinality, relationships (symmetrical, inverse, transitive), equivalence, and difference relationships.

World Wide Web Consortium (W3C)—This international community develops open standards to ensure the long-term growth of the web.

X

XML User Interface Language (XUL)—This markup language has a template element to declare rules for matching data in RDF and uses RDF extensively for data modeling.

References

[1] M. Rouse, "The TechTarget Network," [Online]. Available: http://searchdatamanagement.techtarget.com/definition/extract-transform-load. [Accessed 12 June 2015].

[2] RDF Working Group, "Resource Description Framework (RDF)," 25 February 2014. [Online]. Available: http://www.w3.org/RDF/. [Accessed 13 July 2015].

[3] OWL Working Group, "Web Ontology Language (OWL)," 11 December 2012. [Online]. Available: http://www.w3.org/2001/sw/wiki/OWL. [Accessed 22 June 2015].

[4] W3Schools, "Introduction to RDF," [Online]. Available: http://www.w3schools.com/webservices/ws_rdf_intro.asp. [Accessed 25 June 2015].

[5] DATAVERSITY Education, LLC., "DataVersity Answers," [Online]. Available: http://answers.semanticweb.com/. [Accessed 16 June 2015].

[6] Neo Technology, Inc., "The Neo4j Manual," [Online]. Available: http://neo4j.com/docs/2.2.2/. [Accessed 8 June 2015].

[7] "Neo4J, RDF and Kevin Bacon," [Online]. Available: https://tommorris.org/posts/2462. [Accessed 17 June 2015].

[8] Apache TinkerPop, "TinkerPop3 Documentation," [Online]. Available: http://tinkerpop.incubator.apache.org/docs/3.0.0.M9-incubating/. [Accessed 17 June 2015].

[9] Apache TinkerPop, "Blueprints Homepage," [Online]. Available: https://github.com/tinkerpop/blueprints/wiki. [Accessed 17 June 2015].

[10] A. Harth, K. Hose and R. Schenkel, "5.3.4 Physical Operators: Joins, Path Traversals," in *Linked Data Management*, Boca Raton, CRC Press, 2014, pp. 134,135.

[11] Franz, Inc., "AllegroGraph 5.1 Documentation Index," [Online]. Available: http://franz.com/agraph/support/documentation/current/. [Accessed 8 June 2015].

[12] Wikimedia Foundation, Inc., "Serialization," [Online]. Available: https://en.wikipedia.org/wiki/Serialization. [Accessed 27 August 2015].

[13] Ontotext, *The Truth About Triple Stores,* Ontotext, 2014.

[14] W. Suny, A. Fokouez, K. Srinivasz, A. Kementsietsidisx, G. Huy and G. Xiey, *SQLGraph: An Efficient Relational-Based Property Graph,* Melbourne: SIGMOD, 2015.

[15] Office of Naval Research, Code 31 and U.S. Navy TENCAP, "Achieving a Naval Data Strategy: Achieving a Naval Data Strategy: Leveraging UCD the UCD Ecosystem as a Pathfinder for a Naval Data Ecosystem," 2014.

[16] M. Hunger, "Document Oriented Access to Graphs," 27 May 2014. [Online]. Available: http://www.slideshare.net/neo4j/document-oriented-access-to-graphs. [Accessed 22 July 2015].

[17] GrapheneDB, "Importing Data Into Neo4j via CSV," [Online]. Available: http://blog.graphenedb.com/blog/2015/01/13/importing-data-into-neo4j-via-csv/. [Accessed 6 June 2015].

[18] A. Fowler, NoSQL for Dummies, Hoboken: John Wiley & Sons, Inc., 2015.

[19] M. D. Marzi, "Importing Data into Neo4j," 27 March 2014. [Online]. Available: http://watch.neo4j.org/video/90358900. [Accessed 18 June 2015].

[20] B. DuCharme, "Storing and querying RDF in Neo4j," [Online]. Available: http://www.snee.com/bobdc.blog/2014/01/storing-and-querying-rdf-in-ne.html. [Accessed 17 June 2015].

Index

W