SemTK: An Ontology-first, Open Source Semantic Toolkit for Managing and Querying Knowledge Graphs

PAUL CUDDIHY, GE Global Research, cuddihy@ge.com
JUSTIN MCHUGH, GE Global Research, mchugh@ge.com
JENNY WEISENBERG WILLIAMS, GE Global Research, weisenje@ge.com
VARISH MULWAD, GE Global Research, varish.mulwad@ge.com
KAREEM S. AGGOUR, GE Global Research, aggour@ge.com

ABSTRACT: The relatively recent adoption of Knowledge Graphs as an enabling technology in multiple high-profile artificial intelligence and cognitive applications has led to growing interest in the Semantic Web technology stack. Many semantics-related tools, however, are focused on serving experts with a deep understanding of semantic technologies. For example, triplification of relational data is available but there is no open source tool that allows a user unfamiliar with OWL/RDF to import data into a semantic triple store in an intuitive manner. Further, many tools require users to have a working understanding of SPARQL to query data. Casual users interested in benefiting from the power of Knowledge Graphs have few tools available for exploring, querying, and managing semantic data. We present SemTK, the Semantics Toolkit, a user-friendly suite of tools that allow both expert and non-expert semantics users convenient ingestion of relational data, simplified query generation, and more. The exploration of ontologies and instance data is performed through SPARQLgraph, an intuitive web-based user interface in SemTK understandable and navigable by a lay user. The open source version of SemTK is available at http://semtk.research.ge.com.

KEYWORDS
Visual SPARQL query generation, data triplification, data ingestion, semantic data management

1 INTRODUCTION

With the success of several commercial artificial intelligence and cognitive applications such as Siri, Cortana, Google Now and Watson, knowledge graphs have been rapidly gaining traction. However, the Semantic Web technology stack, which provides a foundation to maintain and query knowledge graphs, poses a significant barrier to their adoption by non-semantic subject matter experts in scientific and industrial communities. Tools such as Protégé[1] and SADL[1][2] have made rapid strides in reducing barriers for ontology design and creation. However, there exist very few tools with the same level of maturity to explore, query and manage semantic data in knowledge graphs, and to the best of our knowledge, none provide a seamless integrated experience to perform all three tasks within a single tool.

In this paper, we present the Semantics Toolkit (SemTK), an open source project designed to make semantics accessible to both expert and non-expert users in a user-friendly manner. SemTK allows users to explore, query and manage semantic data through its SPARQLgraph user interface. While existing approaches for natural language querying over RDF data and techniques for data triplification tend to abstract and hide the semantic model from users, SemTK leverages the domain ontology coupled with the domain expertise of subject matter experts to simplify these tasks. Chiefly through SPARQLgraph, it allows users to explore and search complex ontologies and construct a subgraph of interest both for query generation and data ingestion. SemTK has been developed in the context of the needs of a large industrial business, General Electric (GE), but with applicability

[1]http://sadl.sourceforge.net/

to a much wider audience well outside of the industrial domains in which GE operates. Given that the toolkit was initially developed in a relatively controlled industrial environment rich with subject matter experts with immense knowledge of their respective domains, we chose to develop SemTK with an "ontology first" mentality.

This paper introduces SemTK's novel capabilities to save semantic queries and execute them with runtime constraints in a manner similar to SQL stored procedures. We focus on the power and specificity of the SPARQL query language as opposed to higher level abstractions, and have sought to harness its power by making SPARQL easy to author for both data ingestion and retrieval by developing a toolkit designed to work with SPARQL 1.1 compliant data stores such as OpenLink's Virtuoso[2]. In this paper, we lay out the basic architecture of SemTK and use its SPARQLgraph user interface to describe its underlying concepts of connections and nodegroups (a subgraph of interest to the user) and how we leverage them for query generation, data triplification and ingestion. We explore local pathfinding and the use of automatically-generated SPARQL queries to suggest contents of VALUES clauses, and how these innovations greatly enhance a user's ability to quickly generate SPARQL. We discuss issues and optimization strategies for different query types, and lay out a novel approach for generating INSERT queries that ingest tabular data. The use of query storage with runtime constraints constitutes a stored procedure-like capability that facilitates the use of semantics within higher-level applications. An External Data Connectivity (EDC) service provides Ontology-Based Data Access to data residing outside of the semantic store. Overall, we show how these research contributions combine to create a powerful open source toolkit that accelerates the process of ingesting and exploring data, and provides a service layer on which other knowledge-driven applications can be built.

2 SEMANTICS TOOLKIT

SemTK is comprised of a suite of Java REST services that work with SPARQL 1.1 compliant triple stores. A web-based tool called SPARQLgraph has been built on top of those microservices. SPARQLgraph has been designed to both highlight and make the features of SemTK easy to use. The REST services layer is designed to be deployed in standard cloud environments such as Amazon Web Services, and to be used directly by a wide range of semantics-enabled applications. For the purposes of this paper, however, most functionality will be described in terms of the SPARQLgraph interface. We begin by describing fundamental concepts associated with SemTK and SPARQLgraph.

2.1 Ontology Connection and Ontology Info

A SPARQLgraph session first requires the specification of triple store endpoint connections, one to ontology information and another to instance data. An ontology connection contains the "model domain" which defines the base URI of classes and attributes contained in a model. The ontology may reference imported entities such as XMLSchema number, string, and date primitives.

Ontology information is loaded via SPARQL queries into a local cache for fast access. This cache is called "Ontology Info", and includes classes, subclass relationships, class properties and types, and permitted values for enumeration classes. This information is displayed to the user as a hierarchical list in the ontology panel (the left-hand pane of SPARQLgraph) as shown in Figure 1. This panel allows users to explore the loaded domain ontologies in preparation for querying or data ingestion. The hierarchical list conveys the subclass relationship between classes. On the expansion of each class, it also displays the associated datatype and object properties along with their respective ranges. To deal with the challenge of ontologies with 100s to 1000s of classes and

[2] https://virtuoso.openlinksw.com

properties, this panel provides a search control enabling users to quickly drill down and highlight any part of the ontology matching the search string. Users begin constructing a subgraph by selecting classes from this panel.

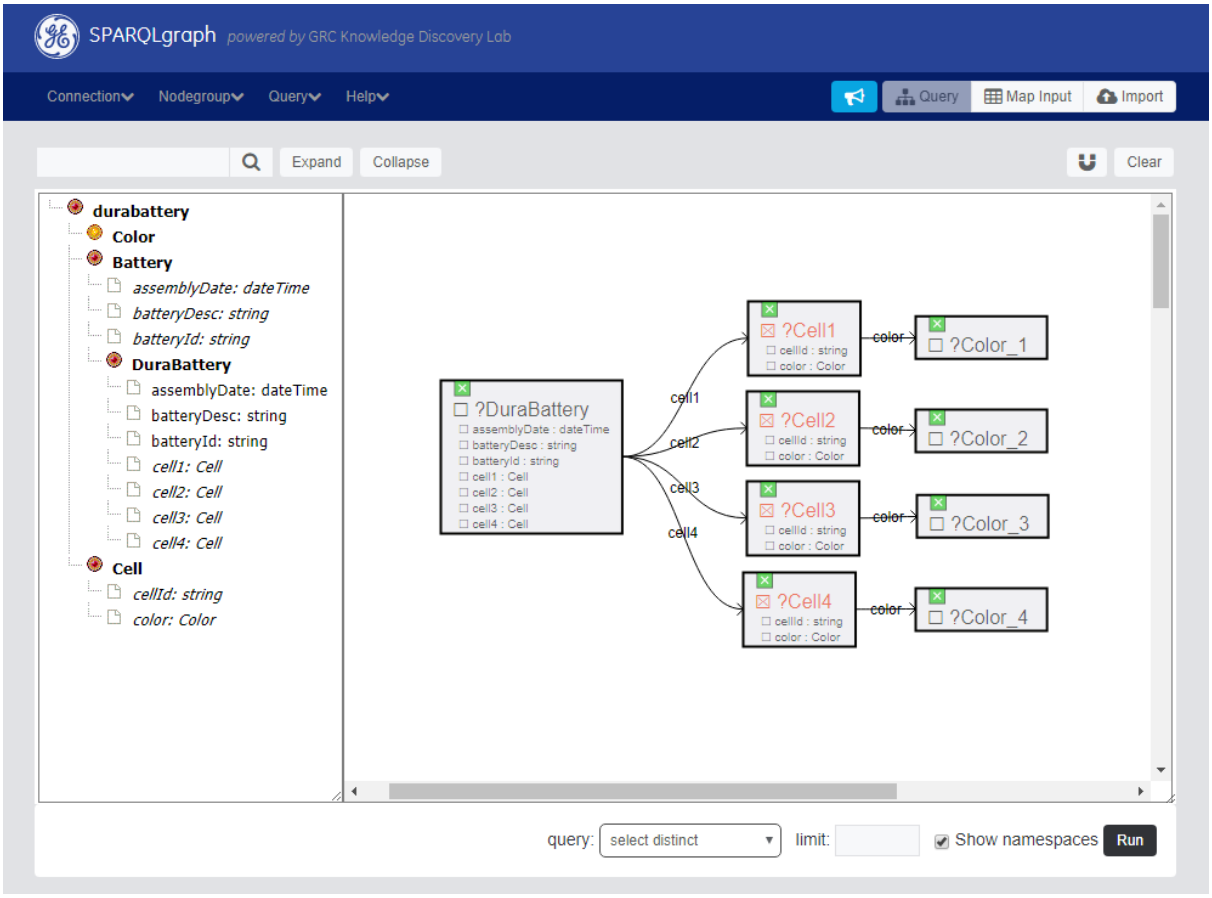

Figure 1: The SPARQLgraph Interface displays the ontology in the left panel and the subgraph of interest on the right

2.2 Query Construction and Generation

With the ontology information loaded and searchable, the user can begin drag-and-drop query generation and query execution against the second type of connection: the data endpoints. SemTK supports the generation of the following subset of SPARQL query types available in SPARQL 1.1 and SPARQL Update: select, construct, ask, delete, and insert. Query generation with SemTK allows for several advanced features, including regex, "values" clauses, counting, limits and optional values.

Query generation begins with the construction of a nodegroup. A nodegroup is a SemTK-specific representation of a subgraph of interest. It is built by dragging and dropping classes from the ontology panel to the visualization window. A simple nodegroup is shown in Figure 2.

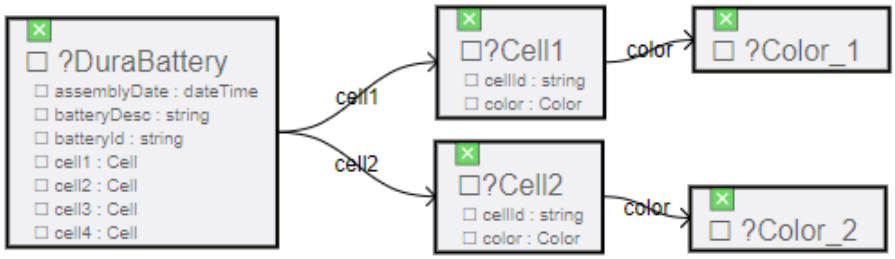

Figure 2: Visual representation of the nodegroup showing multiple instances of the same class (Cell) connecting to different object properties (cell1, cell2)

Each node in the nodegroup represents a class variable in a query, and has a unique name and a list of properties. The properties are split such that those outside the ontology domain are listed first. These properties are often primitive OWL datatype properties, but may also be properties linked to objects outside the domain (e.g. objects in DBpedia). When nodes are linked together into a subgraph as shown, the nodegroup is a powerful representation crucial to almost every SemTK function.

2.2.1 Pathfinding. It is often the case that a user may want to create a query incorporating entities separated by many links in a large ontology. This would require a complex SPARQL query consisting of many clauses representing the chain of relationship between the two entities. Further, there may be more than one possible path to connect the two entities. SPARQLgraph and SemTK simplifies the construction of SPARQL queries over such ontologies with a feature known as pathfinding. Pathfinding allows users to drop two classes on the canvas and select the desired connecting path. Pathfinding greatly enhances the ease-of-use in building nodegroup; as a new class is dragged from the ontology panel onto the nodegroup canvas, SemTK uses a slight variation of the well-known A* algorithm[3] to suggest various paths by which the new class might connect to the nodes already in the nodegroup. SemTK uses object property ranges to identify possible classes in a path.

To accomplish this, the A* algorithm has been modified with stopping logic. The search is limited to a maximum path length (typically 10 links) and/or search time (typically 30 seconds). Further, search path lengths can be limited such that once a path of length n is found, searching ends when all paths of length (n+m) have been explored. This search, restricted to local paths instead of a theoretically complete set, provides an indispensable feature for the efficient assembly of nodegroups. The combination of pathfinding with the performance enabled by the ontology cache allows users to quickly build complex nodegroups, and subsequently auto-generate SPARQL.

2.2.2 Property Selections and Filters. Once the nodegroup is constructed, the user may click to select the properties of interest to return from each of the nodes. Further, SemTK provides a user-friendly way to add SPARQL FILTER and VALUES clauses to constrain any property in the query. Users can easily apply regular expression filters by hand-editing a simple FILTER statement template pre-populated by SemTK for the given property. For example, such a statement for a battery id property might look like this: FILTER regex(?batteryId, "AABB"), with the user typing in the text between the quotes. Alternatively, SemTK can suggest valid values for a property based on the contents of the triple store, which the user may then select for inclusion in a VALUES clause. In this case, SemTK queries under the hood for all possible values, and then presents them in list form to the user. The queries used to populate this list are SELECT DISTINCT queries with two modifications: only the target variable is returned, and all filters or values clauses are removed from that variable. Only those values that satisfy the constraints and relations specified in the nodegroup are returned. A

[3] https://en.wikipedia.org/wiki/A*_search_algorithm

search control helps users narrow down values to select in cases where there are a large number of possible values.

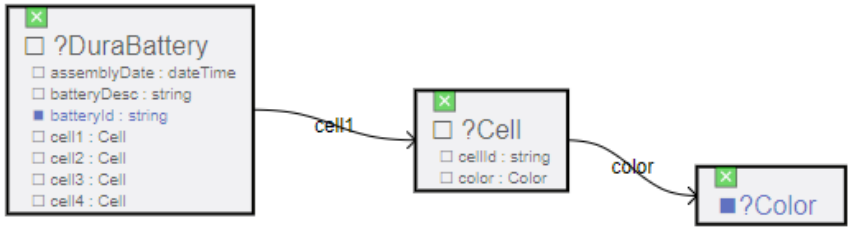

Figure 3: Nodegroup for values clauses on ?BatteryID and ?Color

For example, the nodegroup in Figure 3 represents all DuraBattery instances that have a Cell object property on cell1 where the Cell has a Color. When the user opens the dialog to build a values clause on ?Color, the system uses the algorithm above to execute the query shown in Figure 4 in the background.

```
prefix generateSparqlInsert:<belmont/generateSparqlInsert#>
prefix XMLSchema:<http://www.w3.org/2001/XMLSchema#>
prefix durabattery:<http://kdl.ge.com/durabattery#>

select distinct ?Color where {
  ?DuraBattery a durabattery:DuraBattery .
  ?DuraBattery durabattery:cell1 ?Cell .
    ?Cell durabattery:color ?Color .
}
```

Figure 4: Example of automatically generated query

The results of this query are a list of cell colors in the position cell1 in any DuraBattery. When the user selects one or more colors from the list, the system automatically builds a values clause similar to Figure 5.

```
VALUES ?Color { <http://kdl.ge.com/durabattery#blue>
                <http://kdl.ge.com/durabattery#red>}
```

Figure 5: Generated values clause

This functionality becomes more powerful when it is chained together. For example, if the user were now to ask for possible values for ?DuraBattery's batteryId, the same process would result in a list of only batteryId values that are attached to ?DuraBattery instances that have cell1 objects with colors of blue or red. Using this iterative method of building values clauses, a user can quickly down-select and navigate through complex data.

2.2.3 Query Generation. Once a nodegroup is constructed, SemTK can automatically generate SPARQL to perform various operations (e.g. select, delete) using the nodegroup. To generate a SPARQL select statement, SemTK walks the nodegroup and generates clauses for each node, including clauses for class type, relationships to other nodes, and relationships to properties. Filter and values clauses are generated where the user has specified them, and items marked for return are listed at the top of the SELECT DISTINCT statement. Consider

the nodegroup in Figure 2. If ?DuraBattery's batteryId was selected for return, then SemTK would generate SPARQL shown in Figure 6.

```
prefix generateSparqlInsert:<belmont/generateSparqlInsert#>
prefix XMLSchema:<http://www.w3.org/2001/XMLSchema#>
prefix durabattery:<http://kdl.ge.com/durabattery#>

select distinct ?batteryId where {
   ?DuraBattery a durabattery:DuraBattery .
   ?DuraBattery durabattery:batteryId ? batteryId .

   ?DuraBattery durabattery:cell1 ?Cell_1 .
      ?Cell_1 durabattery:color ?Color_1 .
   ?DuraBattery durabattery:cell1 ?Cell_2 .
      ?Cell_1 durabattery:color ?Color_2 .
}
```

Figure 6: Generated SPARQL query based on nodegroup shown in Figure 3

2.2.4 Optional. When constructing a query, a user may mark a class or property as optional, indicating that they wish to retrieve the requested data whether the marked class or property is present or not. In this case, SemTK query generation encloses all nodes downstream (or upstream) of the connecting arrow inside a SPARQL OPTIONAL clause. Clauses are nested as the SPARQL is generated.

2.2.5 Additional Clauses. The user interface enables specification of a maximum number of results to return, which SemTK implements by adding a LIMIT clause to the query. Further, users can select the "count" query type to return the number of results rather than the result contents. In this case, SemTK transforms the original select query into a count query by wrapping it using "SELECT (COUNT(*) as ?count)".

2.2.6 Delete Queries. A user may construct a nodegroup representing data to be deleted from the triple store. SemTK generates DELETE queries from such nodegroups. Deletion is straightforward for class attributes (including links between nodes), and consists of simply removing any instance of the given attribute from instances of the given class, taking account any surrounding constraints. In contrast, in the case of deleting instances of classes, the user may choose from several modes to control the scope of the deletion. The simplest mode removes triples of the specified class. Other modes provide the option to delete all triples with the node variable in the subject or object, or to limit the deletion to predicates in the loaded model, or predicates in the nodegroup. Together, these delete modes provide the user with a wide range of options to support practical applications.

2.3 SPARQL Query Optimization

SemTK's SPARQL query generation required proactive optimization to maintain acceptable performance. In early testing, the benchmark triple stores (OpenLink Virtuoso[7], Jena TDB[4] and Fuseki[5]) showed poor performance when attempting to execute generated queries. These early queries were generated by naively stepping through the contents of the nodegroup without regard for the impact of clause ordering on SPARQL query performance.

[4] https://jena.apache.org/documentation/tdb/

[5] https://jena.apache.org/documentation/serving_data/

To optimize query performance, the SemTK strategy is to order clauses from those expected to be most specific to least specific. The system assumes that the relationships described in the ontology are directional. Any relationship not stated explicitly as being bi-directional is assumed to be outgoing from the subject to the object. It is also assumed that requested instances of classes with no incoming relationships are, in many cases, more tightly specified than other instances. This is because in the sort of use cases for which the nodegroup is most useful, these instances often have outgoing connections. These instances are placed first in the generated queries, followed immediately by their outgoing connections. Applying these outgoing connections act as constraints. This ordering has the effect of decreasing the search space the SPARQL engine must examine to fulfill the later clauses. By the time the least specified instances are bound, the potential space has been decreased because the outgoing connections from the more specified entries have limited their scope.

The use of the above technique produced significant improvements over the original naïve implementation. In the case of Virtuoso, this led to greatly improved query execution times, making SPARQLgraph usable as a practical, interactive web-based tool. In the case of Fuseki and Jena TDB, it resulted in rapid responses, instead of queries which ran for multiple minutes before failing due to timeout errors. Further improvements to the query optimization are in progress and will be discussed in the future direction section.

2.4 Data Triplification and Ingestion

SemTK also provides capabilities to convert CSV data into RDF triples and ingest them into a triple store. Using the same ontology-based interaction model as the rest of the system, data triplification ingestion is intuitive and provides advanced features for checking the incoming data. The triplification and ingestion of data using SemTK takes place in three steps. The first step is the constructing a nodegroup in SPARQLgraph that will be used to define the structure of the triples to be generated. The second step is the drag-and-drop mapping of columns from a CSV (or table) to the constructed nodegroup. The third step is running the ingestion process itself, applying the mapping to the input data, generating triples, and inserting them into the triple store.

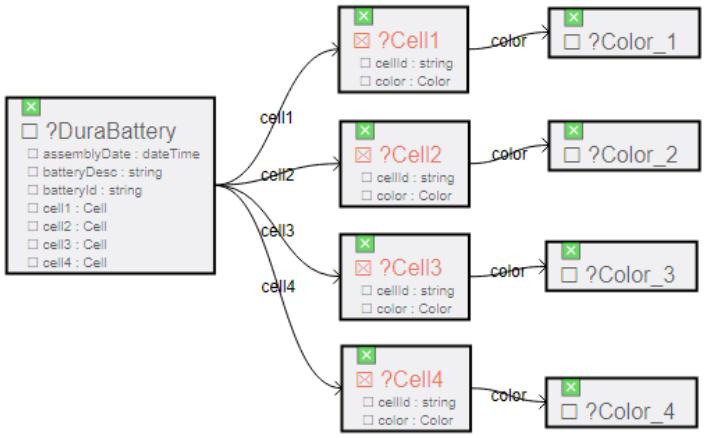

Figure 7: Example nodegroup used for ingestion

2.4.1 Building a Nodegroup for Ingestion. Continuing with the DuraBattery example, consider the case of ingesting a table containing ids for batteries and their four cells, cell colors, and battery assembly date and description. The first step of ingestion is building a nodegroup to represent the subgraph describing the instance data, as shown in Figure 7.

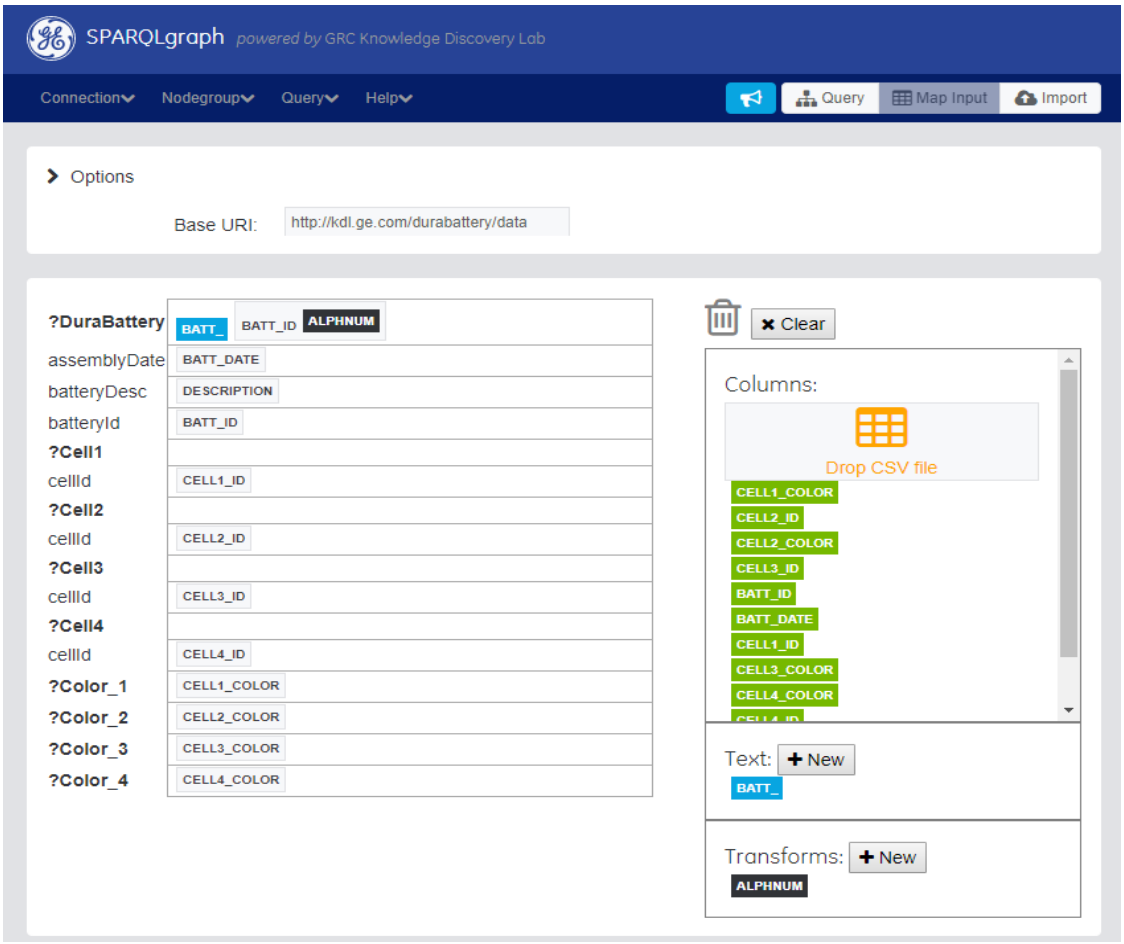

Figure 8: Mapping of properties in nodegroup from Figure 7 (left) to columns from a csv file (right)

2.4.2 Creating the Ingestion Mapping. Next, SPARQLgraph provides drag-and-drop generation of ingestion mappings. The mappings associate one or more columns in the CSV data (shown in green at right) to one or more attributes in the nodegroup (shown on the left). Additionally, transformations are available, allowing the column values to be altered before being assigned. Free text values can be added to the mapping as well. Figure 8 shows the mapping, which in this case is composed primarily of one-to-one mappings of table columns to properties. The URI values are slightly more complex. The value of the DuraBattery URI will be the text "BATT_" prepended to the value from the table's BATTERY_ID column. The BATTERY_ID value will also be processed by a user-defined text transformation named "ALPHANUM" which removes non-alphanumeric characters in order to form a legal URI. This type of user-specified URI is particularly useful for linking data when instance data is ingested from multiple sources and mappings. The user is currently responsible for managing uniqueness and legality of URIs and matching them across multiple ingestion mappings. On the other hand, Cell URIs have been left blank, so they will be automatically generated by the ingestion process.

2.4.3 Ontology-based Data Checking. SemTK's ingestion uses the ontology to validate ingestion attempts before proceeding. This validation occurs via two mechanisms. The first is a check of the nodegroup-based mapping. The second is a per-record check of the incoming data to guarantee conformance. This primary check

compares the mapping and its nodegroup against the current version of the ontology. The entire ontology need not match, but all classes used in the mapping must still exist. Further, the domains and ranges of all properties used in the mapping must still match those in the ontology. If this check passes, the mapping is valid and ingestion proceeds.

The secondary check is more exhaustive and relies on the results of the first check. For a consistent ingestion that respects the type and structural constraints imposed by the ontology, each incoming record from the input data is checked and each class instance relationship must be valid.

Using the nodegroup to give structure to the ingested records enforces the correctness and validity of the class instance relationships. This allows SemTK to avoid having to perform this check for each incoming record. Consolidating these actions into a single check improves performance dramatically over sequential solutions. The checking of the actual data is straightforward, and consists of confirming that each incoming record conforms to the type specified by the ontology. If each value in a record is valid, the record is considered conformant. In the case of URIs, the outgoing value must be a proper URI.

Finally, the ingestion tool handles blank values in incoming records by pruning any path in the nodegroup that has no non-blank values. For example, if a record has neither an id nor a color for cell3, then no triples will be generated for ?Cell3 or ?Color_3.

2.4.4 Preflight data checking mode. Ingestion using SemTK offers two modes. Preflight mode examines all data in the input set prior to ingestion, and checks for per-record failures which would be encountered during ingestion. If any failures are found, the insertion to the triple store does not occur and an error report is generated. This report lists each failing record, its original record number in the set and the first reason encountered that would have caused a failure for that record. If no problems are encountered, the ingestion is run normally and data is inserted into the triple store. This is the default mode.

An alternative mode ingests records without checking the entire set first. In this mode, failures are treated as independent and acceptable. This mode is treated as an advanced user feature and is less commonly used than preflight mode, as records in an input set are often not independent.

2.4.5 Ingestion Optimizations. SemTK employs optimizations that accelerate the ingestion process, reducing the amount of work required to validate the inserted triples. This is achieved by taking advantage of the nodegroup structure and the ontology itself. Using the nodegroup as its structural underpinning allows the ingestion process to check the single structure once, complete with the expected mappings and data types, regardless of the number of records that are to be ingested. A second optimization treats class instance relations as implicit and allows these associations to also be validated exactly once. Optimizations to reduce technical overhead are also employed, but are highly implementation-specific and thus not described here.

2.4.6 Graph Management Operations. SPARQLgraph provides a collection of operations used to manage data in the graphs storing both the ontology and instance data. These features are intended to facilitate operations that cannot be performed using the nodegroup-based generated queries. These include:

- Upload OWL: uploads an OWL file to the target graph
- Clear prefix: removes all triples which have URIs containing the given prefix
- Clear graph: removes all the triples from the target graph

2.5 Stored Procedure Support

Central to SemTK is the mission of making a subset of Semantic Web technologies readily accessible to non-experts. A critical step is making ingestion and query support available to existing workloads in a way easily understood by users. To this end, SemTK includes support for nodegroup-derived stored procedures. In relational database systems, a stored procedure is a subroutine available to simplify the execution of a complex query or operation. Typically, these are accessed by name and allow the caller to pass a collection of parameters. SemTK adopts this concept by allowing a user to store a nodegroup for subsequent execution. The stored nodegroup acts as the basis of a stored procedure but, unlike in SQL, the stored nodegroups can perform multiple tasks based on the selected query type. SemTK's stored procedure support allows the definition of parameters to be specified at runtime. These parameters are defined when the nodegroup is created. They are specifically intended to be overridden at runtime, as opposed to filter constraints which are permanently fixed in stored nodegroups. This dichotomy allows the creator of the nodegroup to divide the constraints into those that are meaningful to the nature of the workload from those that can be used to alter results. Runtime parameters are intended to be convenient to use by users unfamiliar with SPARQL. To this end, inserting arbitrary SPARQL is not permitted. Rather, the user must provide the parameter name, operation type, and collection of values. For each data type supported (Date/Time, String, URI, numeric), the number of operations supported is tightly controlled. SemTK provides features to store and execute stored nodegroups as well as to set, examine and apply runtime parameters. Together, these features form basic stored procedure support ready to be integrated into traditional workloads.

2.6 External Data Connection

While the triple store is an effective storage mechanism for many datasets, it is not suitable for many types of data, particularly those binary in nature (e.g., image data, files) or those requiring high overhead to store in a triple store (e.g. time series data). To address this, SemTK includes extensive capabilities for Ontology-Based Data Access (OBDA) [3], enabling data stored outside of the triple store to be linked, browsed, and queried in domain-relevant terms as if it were part of the triple store. This functionality is achieved by storing metadata in the triple store that describes data stored elsewhere. SemTK's OBDA features are referred to as External Data Connection, and are closed-source at present. While this feature is not a focus of this paper, further discussion of this capability is described in [4].

3 CONCLUSION AND FUTURE WORK

We presented SemTK, a Semantics Toolkit that enables expert and non-expert users alike to triplify, query and manage semantic data in an easy, user-friendly and intuitive manner. Using SPARQLgraph, we detailed how users can connect to existing triple stores to retrieve their domain ontologies and explore them. We also described how users can easily generate SPARQL queries via a drag and drop web-based user interface. The process of constructing a query is highly simplified with features such as pathfinding, which helps users connect two arbitrary classes without deep knowledge of the semantic model. We also described how SemTK can be used to not only generate different types of queries but also for generating mappings to translate CSV data into triples and ingesting them into a triple store. Finally, we detailed how users can save and re-use SPARQL queries via a SQL stored procedure-like capability with runtime constraints. Demos are available at http://semtk.research.ge.com, and the source code is provided under the MIT license at https://github.com/ge-semtk/semtk.

There are several exciting future directions for SemTK. We plan to focus on further optimizing the generated SPARQL queries by intelligently reordering the clauses. For instance, by keeping a count of the number of instances of each class present in the triple store, SemTK could automatically rearrange SPARQL queries, placing the clauses involving the fewest number of instances toward the top of the query. SemTK's ontology-first approach can be expanded by inferring an approximate semantic model from instance data, thereby

allowing users to query against arbitrary SPARQL endpoints in Linked Open Data. Finally, SemTK's triplification and ingestion process can be improved by reconciling and linking data cells in a CSV to existing instances and objects in triple store.

ACKNOWLEDGMENTS

The authors acknowledge the technical contributions of Ravi Palla and Christina Leber, and program support from Steven Gustafson, Matthew C. Nielsen, Arvind Menon, Tim Healy, David Hack, Eric Pool, Parag Goradia and Ryan Oattes.

REFERENCES

[1] Mark A. Musen. 2015. The Protégé Project: A Look Back and A Look forward. AI matters 1 4 (2015), 4–12

[2] Andrew Crapo and Abha Moitra. 2013. Toward a Unified English-Like Representation of Semantic Models, Data, and Graph Patterns for Subject Matter Experts. International Journal of Semantic Computing 7, 03 (2013), 215–236.

[3] Holger Wache, Thomas Voegele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. 2001. Ontology-based integration of information-a survey of existing approaches. In IJCAI-01 workshop: ontologies and information sharing, Vol. 2001. Seattle, USA, 108–117.

[4] Jenny Weisenberg Williams, Paul Cuddihy, Justin McHugh, Kareem S. Aggour, Arvind Menon, Steven M. Gustafson, and Timothy Healy. 2015. Semantics for Big Data access & integration: Improving industrial equipment design through increased data usability. In Proceedings of the IEEE International Conference on Big Data. 1103–1112. https://doi.org/10.1109/BigData.2015.7363864