OntoQuest: Exploring Ontological Data Made Easy

Li Chen[1] Maryann Martone[2] Amarnath Gupta[1] Lisa Fong[2]
[1]San Diego Supercomputer Center [2]Center for Research in Biological System
University of California, San Diego, 9500 Gilman Drive, CA 92093

ABSTRACT

Recently, there is a large demand by many scientific applications for managing, querying and reasoning ontology concepts and instances. We demonstrate OntoQuest, a system that provides powerful yet easy-to-use query and reasoning utilities by which the ontological data exploration experience is made easy. Even without any knowledge of ontology query languages, one can easily get a hands-on ontological data exploration experience using OntoQuest. The method is to categorize commonly asked queries based on their usage contexts so to prompt the user with context-aware guidance throughout the exploration process. OntoQuest is also designed to offer extended mapping schemes for storing OWL ontologies into back-end databases. Most existing ontology storage systems support mappings only for RDF data. Lastly, OntoQuest supports bulk insertion and updating of instances.

In this demonstration, we show how OntoQuest guides a user through the inquiry (querying and reasoning) process and also have a peek at the underlying handling of storing, querying, reasoning, and interacting with the user.

1. INTRODUCTION

The OntoQuest system is motivated by an OWL[5]-based neural anatomy ontology development project initiated by neuroscientists at UCSD. An ontology is a way of describing a shared common understanding about domain knowledge (concepts and their relationships) in formal logic such that the data is comprehensible to both human beings and machines.

To represent the knowledge base of a scientific domain, e.g., the subcellular anatomy of the nervous system in our case, a large-scale ontology needs to be developed to capture several thousands of concepts including cell types, subcellular components and molecules, and multicellular domains. Furthermore, instances that represent facts described in voluminous neuroscience literature need to be extracted and classified under concept definitions of the ontology. Developing such an ontology is extremely important for allowing neuroscientists to share the data, to validate hypotheses, and to make scientific discoveries through querying and reasoning.

1.1 Related Work

There is a growing sense among researchers and practitioners that ontologies will play a significant role in forthcoming information management. Numerous commercial and open-source software tools are available for building and deploying ontologies, and for integrating inference with database infrastructures. Existing ontology tools and systems[1] can be categorized into the following. (1) Ontology editors. Protege[6] is an ontology editor that supports frames, RDF(S) and OWL. OilEd is an ontology editor supporting DAML+OIL. (2) Program API libraries. Jena[4] is a programming toolkit that supports parsing, creating and searching primarily RDF models. In Jena2, a preliminary OWL API is implemented for accessing OWL ontologies, but the supported OWL reasoning is limited and has performance issues. (3) Ontology reasoners. The well-known DL (description logic) reasoners are FaCT (now FaCT++) [3] and Racer [2]. Instance reasoning is not supported by FaCT++, and nominals are not available neither in FaCT++ nor Racer. The new comer Pellet promises reasoning with instances (including nominal support). (4) Ontology stores. RDFSuite and Sesame are two representative RDF schema-based repositories and querying facilities. Jena also provides APIs via JBDC to talk to back-end RDBMSs. DLDB is another relational store for RDF/OWL data. It represents the inferred concept class hierarchy using relational views but does not capture the expressive OWL restrictions.

Limitations. These existing ontology tools and systems are less satisfactory for our purpose because either these tools are browsing and editing oriented, or their storing, querying, and reasoning capabilities are limited in the following ways. *First*, all existing ontology query systems demand users to master certain ontology query languages such as RDQL and RQL in order to ask any questions. With this requirement, the user interfaces of these systems are not friendly to most domain scientists. *Second*, the process of a scientific exploration often involves a series of question asking. Sometimes the results of a previous query may inspire the user to ask the next one using part(s) of the results of interest. Currently, the typical query interfaces are simply a

[1]This review and the citations are not meant to be complete due to the space.

set of unrelated query text fields which facilitate no formation of such interrelated inquiries. *Third*, existing systems support limited querying capabilities concerning graph properties, aggregate and grouping functions, etc. Furthermore, the available reasoning interfaces are so far restricted and are handled separately from queries. *Fourth*, current ontology stores are mainly relational ones compatible with RDF ontologies. They commonly ignore the expressive OWL restrictions. *Fifth*, there is little support for bulk insertion and updating of instances.

1.2 OntoQuest Feature Highlights

We demonstrate four main features of the OntoQuest system. (1) An intuitive and easy-to-use interface for domain scientists. A majority of ontology users are biologists who appreciate tools that do not require them going through a steep learning curve of ontology query languages. OntoQuest allows biologists to easily query ontologies without knowledge of a particular query language. Commonly asked queries are categorized into groups and shown via menus and parameterized forms. Example queries include finding the most general/specific classes for a class/instance, retrieving the instances of a class which satisfy some condition (e.g., value of property p is y), etc. Users simply click and check the provided options to issue queries. (2) "Context-aware" capabilities during an exploration session. Sometimes a user's query only makes senses within a context, e.g., she first asks for all properties of an instance o, and then picks one output property and wants to get its value. In this case, two queries are interrelated; the second one can only be answered by knowing that the asked property value is for that instance o. Context-aware is also reflected through the automatic choice of the group of queries to be prompted in a menu to the user. Queries that are not in the menu are not likely to be asked next. (3) The ability to process interesting aggregate operations that may involve reasoning. We will elaborate on this one through examples in Section 2.4. (4) Extended mapping from OWL constructs to database schema and constraints. OntoQuest not only supports storage for OWL ontologies, but also exploits indexing and encoding techniques [1] for optimizing graph transitivity computations. Frequently asked results can be materialized to trade off for query efficiency. (5) Interface for bulk editing and updating of instances. This may be very useful in situations where a large number of instances share the same property values, or the value distribution is describable by some function, or a selected set of instances are subject to the same value change. OntoQuest also supports automatic incremental validation realized by triggers in databases.

To the best of our knowledge, OntoQuest is the first ontology system targeted at domain scientists that facilitates their scientific explorations through an integrated environment for storing, interrelated querying, reasoning, etc.

2. DEMONSTRATION OVERVIEW

We use the following example ontology (Figure 1 shows part of it) for our demonstration.

2.1 Our Example Ontology

This ontology describes the basic structure of subcellular anatomy for nerve cell. Two main classes of cells are *neurons* and *glia*, with subclasses categorized under each. Four main categories of entities are *compartment, component, property* and *molecule* which are related to the cell classes by the "*has_a*" relationships. These entities may be related to a cell class at any level. Take compartment and component for example. Compartment captures a functional subdomain of the nerve cell. For neurons, the four main compartments are *cell body, dendrite, axon* and *spine*. Component refers to cellular structures common to all cells and they are taken from and cross-reference to the cell component hierarchy of the Gene Ontology. The instances of these classes may come from the literature or locally curated data repositories.

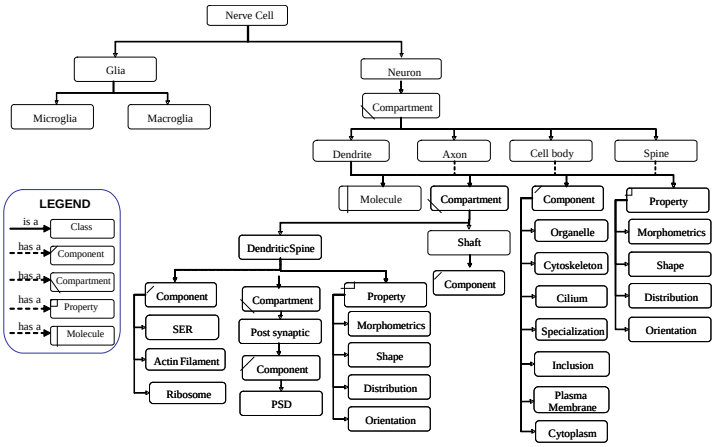

Figure 1: Ontology for Neuron Subcellular Anatomy

2.2 User Interface of OntoQuest

We first demonstrate the user interface of OntoQuest.

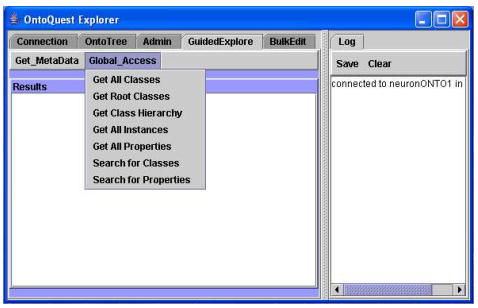

Figure 2: User Interface of OntoQuest

As shown in Figure 2, the interface is mainly a split pane – the right side consists of a scrollable pane displaying a log of past activities (e.g., connections, querying and reasoning tasks) while the left side allow users to switch among a set of tabbed panels each assigned a specific functionality. The first panel is for configuring the connection to the backing store for storing or fetching the ontology model. The second one is for browsing the expandable ontology hierarchy. The third is for users who are familiar with ontology query languages (e.g., RDQL) to directly post queries for administrating and debugging purposes. The fourth is for general

users to explore ontologies and issue interrelated queries. The last one is for bulk insertion and updating of instances.

2.3 Queries Categorized by Usage Context

We now demonstrate the feature of categorized queries that are designed for easy of use for general users.

Query Category	*Query APIs (shown in top and popup menus)*
Ontology Metadata	getVersion(), getCreator(), getOntologyURI(), getImports()
Global Access	getAllClasses(), getRootClasses(), getAllInstances(), getAllProperies(), getClassesMatching(String s_pattern), getPropertiesMatching(String s_pattern)
Inquiry for Class	getInstances(), getInstancesWithCond(String p, String p_value), groupInstancesByPValue(String p), getProperties(), getSuperClasses(boolean direct), getSubClasses(boolean direct), isSubsumedBy(String c), isSubsuming(String c)
Inquiry for Instance	getClasses(boolean direct), getPropValues(), getNeighborInstances(int direction, int dist), isClassifiedUnder(String c), isSatisfyingCond(String p, String p_value), degree()
Inquiry for Property	getDomain(), getRange(), getSuperProperties(boolean direct), getSubProperties(boolean direct), isApplicableTo(String c)
Aggregate or Graph Property	count(), sumOfPValues(String p), aveOfPValues(String p), steinerNet(), diameter(),

Table 1: Categorized Query Groups

Instead of providing a text field for users to type in a query, the interface provides or pops up menus with the "right" query options for the context, i.e., those that are likely to be asked next. Queries in the first two categories listed in Table 1 are shown as top menu items where a user may start her exploration. The first category assembles queries about the ontology metadata, e.g., version, creator. Queries in the second category allow a user who may have no knowledge about the content of the ontology to get all the classes, properties, or instances with their URIs matching a given string pattern. From here onward the exploration may continue with the user being prompted with appropriate popup menus presenting query options of the other four categories.

For example, assume a user first clicks on "*getAllClasses()*" on a top menu to initiate the exploration. From the returned classed in the result panel, she then picks one of interest and right-clicks to pop up a menu presenting queries of the third category – *inquiry for class*. Suppose that '*getInstances()*" was clicked next and all instances of that class are returned. In this case she may issue further inquiries for a particular instance, and so on.

Query Context and Its Transition. Generally speaking, it is unlikely that a domain scientist can plan ahead exactly what a single complicated question to ask to reach a finding. Instead, she tends to ask a series of small questions, one leading to another. The answers of a previous query

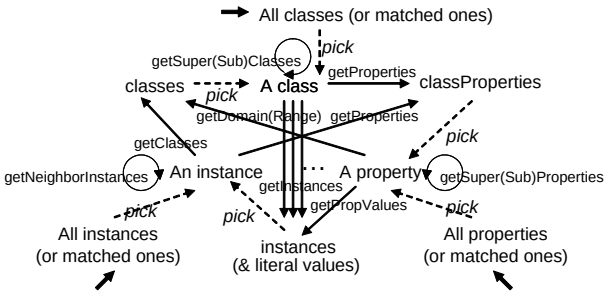

Figure 3: Transition of Query Context

constitutes the "context" from where the next query starts. Possible transitions among query contexts are shown in Figure 3. A starting point is usually the query options provided in the top menus, as indicated by those bold arrows. The context then transits along with the "picked" previous result and the clicked query option. At any point, a user may restart the exploration process again by clicking a top menu item. OntoQuest also offers a "back" button for users to retrovert to any previous context.

Queries in the "aggregation or graph property" category result in numbers or a new result panel showing the neighborhood graph (e.g., for steinerNet). They require multiple, rather than a single instance, to be "picked". Neither these queries nor those boolean queries (i.e., the ones prefixed with "is" which are actually access points of reasoning functionalities) cause any context transition.

2.4 Implementation of Query APIs

Among the provided queries listed in Table 1, some are implemented by calling Jena APIs or OWL APIs, while others have complicated logics that may also incorporate reasoning and post processing. Below we illustrate how to implement the query API $groupInstancesByPValue$.

The semantics of $groupInstancesByPValue(String\ p)$ is to group all the instances (direct and indirect) of a class c (the context) by their values of property p. The complexity of this query arises from the OWL-DL constructs allowed in our ontologies, such as class and property hierarchies, inverse roles, nominals, number restrictions, etc. Suppose c is Dendrite and p is is_Spiny whose domain is a boolean value. In the class hierarchy, Spiny_Dendrite is a subclass of Dendrite, which is defined with a $hasValue$ restriction $r1$: is_Spiny$\ni true$. For the 17 instances defined under this class, although none are explicitly given a value for is_Spiny, the value of $true$ is implied to all 17 instances.

Existing DL reasoners (i.e., FACT++, Racer) do not handle reasoning for nominals ($\ni$ is a nominal construct). Therefore, to process this query correctly, we need to gather the instances for each subclass c_j of c, check their p values as well as whether there is a $hasValue$ restriction on c_j which asserts that all the instances have some default value. Furthermore, each superclass c_i of c also needs to be checked to see whether a $hasValue$ restriction exists to affect the p values of all the instances of c.

Besides the method we use to traverse the class hierarchy and check for special restrictions for reasoning the p values of c's instances, we also need to sort and group these instances by their p values. Such *groupby* operations are not yet readily available in any well-recognized ontology query language. Hence we implement them in a manner similar to the counterpart relational operators. This example query and its results are illustrated in Figure 4.

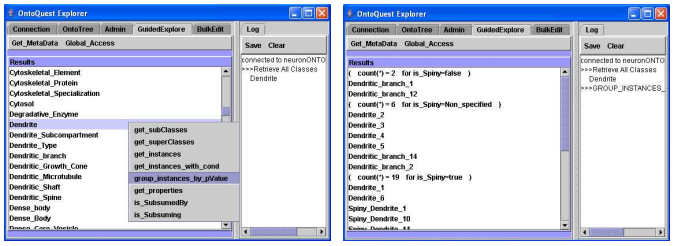

Figure 4: Groupby Query and Its Results

2.5 Mapping Ontologies to Store Schema

Due to the size of the application ontologies, it is necessary to use database technology in order to provide persistence to the knowledge described by the ontologies, and scalability to the queries and reasoning on this knowledge.

In OntoQuest we use Jena's libraries for processing and parsing OWL ontologies. However, we implement our own algorithms for generating a mapping between ontologies and the backend store schema. By our mapping, ontologies are stored by creating a relational table for each class or property definition. The DAG-structure of the ontology class hierarchy is stored in the system using an advanced encoding and indexing technique [1], by which transitive computations such as class subsumption can be efficiently performed. For example, to retrieve the instances of a class, a dynamic view of this class is generated which unions all instances of the subclasses.

Alternatively, we provide a more flexible mapping scheme that maps OWL ontologies into an object-relational model. In this scheme, classes are mapped to tables, and properties are mapped to set-valued attributes or pointer/references (there are exceptions, e.g., functional properties are mapped to a single-valued attribute). Furthermore, OWL restrictions such as $allValuesFrom$ ($\forall p.c$) and $someValuesFrom$ ($\exists p.c$) are mapped to corresponding table assertions that require exclusiveness or not null of p. For $hasValue$ ($p \ni v$), the initial value set of p contains a default value v.

2.6 Bulk Editing of Instances

OntoQuest provides the interface for users to edit property values that will apply to a large number of instances. When "bulk insertion" is clicked for a class, a customized property form (with some fields partially filled and some given restricting instructions) is dynamically generated based on the class definition. In addition, a number of string or number automatic generation functions are available as choices for producing instance uris. For bulk updating, the user needs to decide the set of instances to apply, which can be the result of a previous query. Regarding automatic maintenance and evolution of ontologies, we implement an incremental validation mechanism using triggers in the database which are automatically invoked for each bulk editing operation.

3. SYSTEM ARCHITECTURE

The architecture of OntoQuest is shown in Figure 5. We made use of some Jena APIs for parsing ontologies and then manipulating them using Java objects. Application ontologies can be put in the persistent store, while data in the store can also be serialized into RDFS/OWL ontologies. The query APIs and context manager are explained in Section 2.3. The query processor takes as input a query API call and the context, then retrieves the plan and evaluates it. The plan may simply be a Jena API, or a complicated process which may involve reasoning and result processing. An example is described in Section 2.4. Some boolean query APIs are translated into calls to a third-party reasoner. The store mapping schemes and instance editor are described in Section 2.5 and 2.6 respectively.

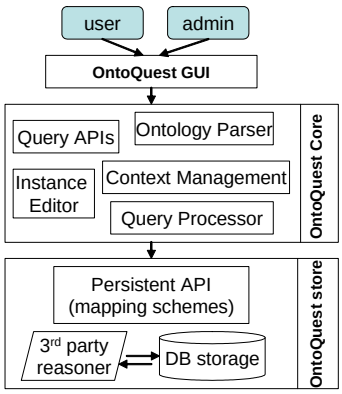

Figure 5: Architecture of OntoQuest

The prototype of OntoQuest is implemented with Java 1.4.2 on top of Oracle 9i. The module for query optimization and efficient reasoning is under development. Other work considered for the future includes the development of a web-interfaced system that allows for multiple instance editing and alignment for multiple ontologies.

4. REFERENCES

[1] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *International Conference on Very Large Data Bases (VLDB), Trondheim, Norway*, pages 493–504, 2005.

[2] V. Haarslev, R. Mller, and M. Wessel. Racer systems. http://www.sts.tu-harburg.de/ r.f.moeller/racer.

[3] I. Horrocks. The fact system. http://www.cs.man.ac.uk/ horrocks/FaCT.

[4] HP Labs Semantic Web Research. Jena a semantic web framework for java. http://jena.sourceforge.net.

[5] D. L. McGuinness and F. V. Harmelen. Owl web ontology language overview. http://www.w3.org/TR/2004/REC-owl-features-20040210.

[6] Stanford University. Protege: An open source ontology editor tool. http://protege.stanford.edu.