NeuroQL: A Domain-Specific Query Language for Neuroscience Data

Hao Tian[1, 3], Rajshekhar Sunderraman[1], Robert Calin-Jageman[2], Hong Yang[1], Ying Zhu[1], and Paul S. Katz[2,3]

[1] Department of Computer Science, Georgia State University, Atlanta, GA 30303, USA
{haotian, raj, hyang9, yzhu}@gsu.edu
[2] Department of Biology, Georgia State University, Atlanta, GA 30302, USA
{rcalinjageman, pkatz}@gsu.edu

Abstract. In this paper, we propose a domain-specific query language called NeuroQL for the neuroscience domain. NeuroQL is designed primarily for neuroinformatics database users and aims to enable users to directly interact with neuroscience databases in their professional concepts and terms with the help of a conceptual data model. NeuroQL is DBMS independent and can be translated into traditional query language such as SQL, OQL and XQuery. It integrates some object-oriented features, and supports neuron domain-specific data types and query operators, which can dynamically evolve when the underlying database schema evolves.

1 Introduction

In recent years, neuroscientists have been accumulating data at an exponential rate. To accommodate this expansion of data, a variety of neuroscience databases have been developed to let neuroscientists store, retrieve, manipulate, and analyze their data. A list of neuroscience databases can be found at the society for Neuroscience website [2]. Usually users are provided a graphical query interface, through which users can create their queries. However the major problem of the query GUI solution is that it is application-dependent and supports limited types of queries. To provide a more flexible and powerful tool for neuroscientists to interface with neuroscience databases, we are proposing a neuron domain-specific query language (NeuroQL).

The concept of domain-specific language (DSL) is not new. There are many domain-specific languages for different application domains. Best known are classic examples like PIC, SCATTER, CHEM, LEX, YACC, and Make, which are described in [6]. For a variety of domain-specific problems, domain-specific languages are more attractive than general-purpose languages (GPL) because of the features of easier programming, systematic re-use, and easier verification in DSLs [4, 5]. However, designing and developing a domain-specific language is not an easy task, sometimes it is even harder than developing the application itself. SQL can be considered as a

[3] This work is funded in part by a Georgia State University Brains & Behavior Program, and by grants to Paul S. Katz from NIH and NSF.

domain-specific language focusing on the database domain. But it is still too general and not application-user-friendly for life science database applications as mentioned above. So far, only a few domain-specific query languages (DS-QL) have been proposed [7, 8, 9] for life-science data. Hammer and Schneider proposed a high-level data model and object-oriented query algebra for genomic information [10]. However, none of them can be adapted or extended to neuroscience due to the significant difference on the underlying data structure and semantics. To our knowledge, no domain-specific query language exists for neuroscience.

NeuroQL is motivated by the development of the NeuronBank system [1], which is an effort to develop an online reference source and informatics tool for exploring the vast knowledge of identified neurons and the circuits that they form. During the design of the NeuronBank system, neuroscientists expressed a desire of having a simple query language in their familiar concepts and terms so that they can query the database directly in a way close to their research language. This new query language will provide an advanced query function for experienced users as a supplement to the naïve query GUI, and should work on most types of DBMSs. Obviously, the current common query languages such as SQL, OQL [12], and XQuery [13] do not meet the requirements because they are all DBMS-dependent, and do not have built-in support to the neuroscience domain-specific data types and functions. Therefore, NeuroQL is designed for use by neuroinformatics database users, rather than the database administrators or developers. And it is mostly based on the neuron domain knowledge, unlike SQL and OQL that are based on the underlying relational database schema (see Fig. 1.).

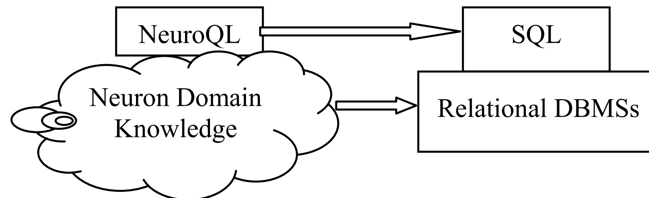

Fig. 1. The bases of NeuroQL and SQL

NeuroQL is a high-level canonical query language, whose basic idea was firstly introduced in [3]. The initial format and syntax of NeuroQL was proposed by the neuroscientists in our NeuronBank team. It integrates object-oriented concepts and supports most of neuron domain-specific data types, for example, neuron, soma, axon, and so on. These data types are systematically derived from the conceptual neuron data model - NeuroDM [3]. NeuroDM defines a small core data structure that holds on the neuron data in all species. It should not be changed in any neuron data model extension for an application. The underlying database schema is generated from NeuroDM as well. Thus, it is easy to set up the mapping between the database schema and the NeuroQL data types, which can dynamically update the data types in NeuroQL whenever the database schema evolves. NeuroQL also defines many advanced neuron domain-specific query operators corresponding to the functions among some common objects in neuroscience. For example, the operator "bilateral" represents the bilateral function between two neurons, and the operator "project"

represents the projection function from a neuron's axon to a nerve. The conceptual and implementation architecture of NeuroQL is shown in Fig. 2. The domain-specific data types and query operators are the major constructs of NeuroQL so that the NeuroQL query statements are very close to the questions or queries that the neuroscientists ask in their research.

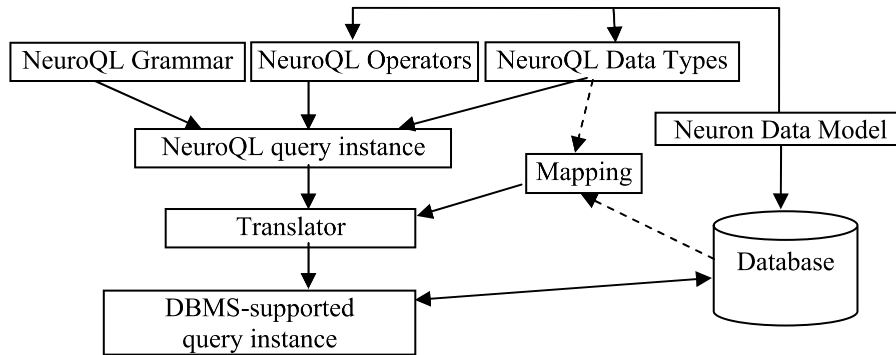

Fig. 2. The conceptual and implementation architecture of NeuroQL

In NeuronBank system, NeuroQL is used in the Meta Search Engine (Fig. 3) to provide the advanced query functionality on integrated heterogeneous species databases. Although each species database represents the neuron data in different ways, they have the same core structure defined in NeuroDM. They can be built on different types of DBMS such as relational DBMS, object-oriented DBMS, and XML native database. The NeuroQL queries will be translated into corresponding SQL, OQL or XQuery queries according to the type of the underlying DBMS.

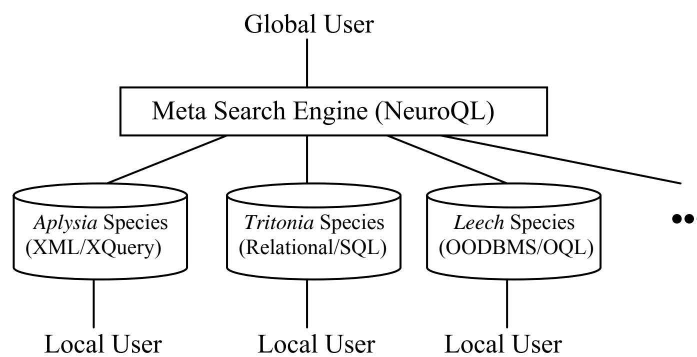

Fig. 3. NeuroQL in NeuronBank

Overall, NeuroQL supports neuroscientists to query the databases using their professional concepts and operators after they understand the abstract conceptual data structures in the data models. We believe that understanding a conceptual data

diagram (close to the users' domain knowledge) is much easier than figuring out and remembering the details of the relational database schema, especially the referential constraints.

2. Data types and query operators in NeuroQL

2.1 Data types

In addition to the common primitive data types, such as Integer, Float, Character, and Boolean, NeuroQL supports most of neuron domain-specific data types as well, such as neuron, soma, axon, nerve, ganglion, firing_pattern, connection, molecule, electrical synapse, chemical synapse, neuromodulation, and component. These domain-specific data types are systematically derived from the entities in the conceptual neuron data model. Fig. 4 shows an extension of NeuroDM used in the *Tritonia* species database of NeuronBank system.

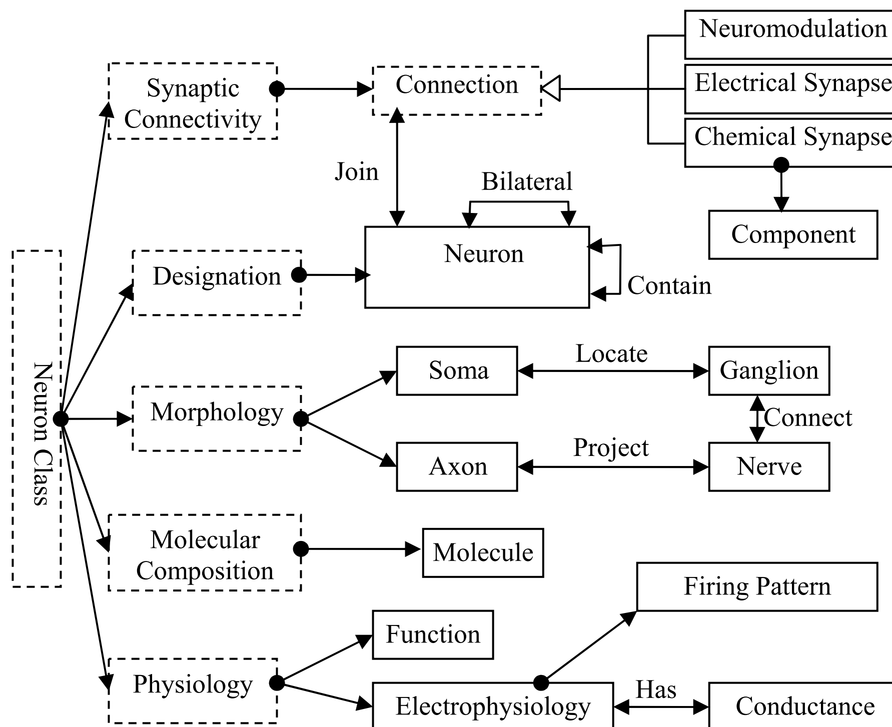

Fig. 4. An extension of NeuroDM [3] used in NeuronBank *Tritonia* species database

The algorithm translating a neuron data model into the data types of NeuroQL should be straightforward because it is similar to the process of converting an ER data model to the object-oriented classes. Following is an algorithm that generates the NeuroQL data types in the NeuronBank system from the NeuroDM, an extended ER data model with some OO features.

NeuroQL_Data_Type_Generation_Algorithm:

```
Input: NeuroDM
Output: NeuroQL data types

Begin:
For each entity in NeuroDM
   Create a new NeuroQL data type
   For each attribute of the entity
      If it is an atomic attribute
         Create an atomic property
      Else
         Create a container property for that attribute
   End of For
   For each inheritance relationship
      Inherit all properties and functions from the
                super data type to the new data type
   End of For
   For each aggregation relationship
      If the maximum number of members == 1
         Create a new property for the aggregated data
      Else
         Create a container for the aggregated data
   End of For
   For each association relationship
      Create a function returning the associated data
   End of For
End of For
For each special table in NeuroDM
   Create a new NeuroQL data type
   For each column in the table
      If it has an atomic data
         Create an atomic property
      Else
         Create a container property for that attribute
   End of For
End of For
End of Algorithm;
```

Next is a data type example ‘neuron’ that is derived from the entity ‘neuron’ in NeuroDM. We adopt object-oriented concept in our data type definition as well.

```
Data Type: neuron

   Properties:
   //Properties derived from the entity ‘Neuron’
      neuronID              String
      name                  String
```

```
    synonym              Set[String]
    type                 String
    genus                String
    species              String
    minSize              Integer
    maxSize              Integer
    functionCategory     String
    remark               String

  //Properties derived from the aggregation
  //relationships
    itsSoma                 Soma
    itsAxon                 Axon
    itsFunctions            Set[Function]
    itsElectrophysiology    Electrophysiology
    itsMolecules            Set[Molecule]
    itsFlaggedFeatures      Set[String]

  Functions:
  //Functions derived from the association
  //relationships
    Neuron  bilateral();
    Set[Neuron]  homologue();
    Set[Neuron]  contains();
    Set[Connection]  connect_neuron(Neuron);
```

Since the mapping between NeuroQL data types and database schema is set up systematically, NeuroQL can be updated dynamically when the database schema evolves. This mapping will be used by NeuroQL Translator to translate a NeuroQL query instance to a corresponding DBMS-supported query instance like SQL query. By adding, deleting and changing the entities in the neuron data model, NeuroQL can be easily extended and customized. This feature is especially useful when NeuroQL is integrated into the database applications focusing on the neuron data in different species.

2.2 Query operators in NeuroQL

The query operators (see Table 1) in NeuroQL are divided into 2 categories: primitive query operators and advanced query operators. Primitive query operators includes the common operators used in traditional query languages, and advanced query operators are corresponding to the functions among the objects in neuroscience The introduction of advanced query operators makes the NeuroQL query statements much closer to the questions and queries asked by the neuroscientists in their research. The semantics of the query operators are discussed in the next section.

Table 1. Some query operators in NeuroQL

Category	Operators
Primitive query operators	+ - * / % > >= = <= < != . LIKE IN
Advanced query operators	connect, parent, children, neighbors, project, electricalSynapse, chemicalSynapse, location neuromodulation, hasComponent, pass, circuit, itsFiringPatterns, presynapticCell, postsynapticCell.

3. Syntax and Semantics

3.1 Syntax

Following is the BNF grammar of NeuroQL and the interpretation of terminals is given in Table 2:

```
<NeuroQL_query> ::= ( <result_list> ) [ <condition_list> ]
<result_list> ::= <result_term> | <result_term> , <result_list>
<result_term> ::= <single_class_term> | <single_class_term>.SET_PROPERTY
<condition_list> ::= <condition_term> | <condition_term> , <condition_list>
<condition_term> ::= /*EMPTY*/ |<declaration_term> | <comparison_term>
<declaration_term> ::= <single_class_term> IDENTIFIER
                | <single_class_term>.SET_PROPERTY IDENTIFIER
                | <non_bool_op_term> IDENTIFIER
<comparison_term> ::= <bool_op_term>
                | <operand> COMPARISON_OP <operand>
<single_class_term> ::= DATATYPE | IDENTIFIER
                | <single_class_term>.SINGLE_PROPERTY
<bool_op_term> ::= ADV_BOOL_OP(<parameter_list>)
                | <single_class_term>.ADV_BOOL_OP(<parameter_list>)
<non_bool_op_term> ::= ADV_NON_BOOL_OP (<parameter_list>)
                | <single_class_term>.ADV_NON_BOOL_OP (<parameter_list>)
<parameter_list> ::= <parameter> | <parameter>,<parameter_list>
<parameter> ::= /*empty*/ | <operand>
<operand> ::= CONSTANTS | <single_class_term> | <non_bool_op_term>
                | <arithmetic_operand> ARITHMETIC_OP <arithmetic_operand>
<arithmetic_operand> ::= CONSTANTS
                | <single_class_term>.SINGLE_PROPERTY
```

Table 2. The interpretation of terminals in NeuroQL grammar

Terminals	Interpretation
DATATYPE	{dt | dt is the name of a neuron-domain data type in NeuroQL}
SINGLE_PROPERTY	{p | p is a property of a neuron-domain data type in NeuroQL, which references to a single object}
SET_PROPERTY	{sp | sp is a property of a neuron-domain data type in NeuroQL, which references to a set of objects}
IDENTIFIER	{id | id ::= [_a-zA-Z][_a-zA-Z0-9]*}
ADV_BOOL_OP	{bop | bop is an advanced query operators in NeuroQL, which returns a Boolean value}
ADV_NON_BOOL_OP	{nbop | nbop is an advanced query operators in NeuroQL, which returns a non_boolean value}
CONSTANTS	{c | c is a constant value of a primitive data type}
COMPARISON_OP	{'>', '>=', '=', '<=', '<', '!=', 'LIKE', 'IN'}
ARITHMETIC_OP	{'+', '-', '*', '/', '%'}

We adopt the dot notation in NeuroQL, which simplifies the reference to a class or its properties. For example, the path expression of neuronA.itsSoma.colorPattern references the property "colorPattern" of a soma of a neuron instance, neuronA. And in SQL, this is usually done by joining multiple tables based on some referential constraints.

3.2 Semantics

A NeuroQL query has a very simple structure like *(<result_list>)[<condition_list>]*. The <result_list> is similar to the SELECT clause in SQL, which consists of a list of objects (i.e. <result_term> in grammar). Each object in the <result_list> can be an instance of a NeuroQL data type or a property of it. The <condition_list> lists all query criteria (<comparison_term>), that is, the conditions that the query result must satisfy.

Let's use RT_1, RT_2, …, RT_n to represent the domain of each <result_term> respectively, then the domain of <result_list>, RL, is the set of n-tuples, whose first element is from RT_1, the second element is from RT_2, and so on. The query result includes the tuples in RL, which satisfy all conditions in the <condition_list>.

The advanced query operators (Table 3) can be used in <condition_list> in order to simplify the expression of a query, and make it closer to a professional expression in the neuron research domain.

Table 3. The signatures and semantics of some advanced query operators in NeuroQL

Operators	Signatures	Semantics
Parent	parent()	parent: {t | t is a neuron instance } → {n | n is the neuron instance in cluster type, and t is a member cell of n.}
Children	children()	children: {t | t is a neuron instance in cluster type } → { n | n is a neuron instance that is a member cell of t.}
Neighbors	neighbors(double r)	neighbors: {(t, r) | t is a neuron instance, r ∈ R} → {n | n is a neuron instance, and the distance between t and n is less than or equal to r}
Project	project(nerve v)	project: {(t, v) | t is a neuron instance, v is a nerve instance} → {bool | bool = true if the axon of t projects to v, otherwise bool = false}
Connect	connect(String name, String type)	connect:{(t, name, type) | t is a neuron instances, $name is a name of another neuron instance s, and $type ∈ {'electrical', 'chemical', 'neuromodulation', 'any'}} → {bool | bool = true if there is a connection from t to s or from s to t , whose type matches the value of input $type, otherwise, bool=false.}
electricalSynapse	electricalSynapse(neuron s)	electricalSynapse: {(t, s) | t and s are two neuron instances} → {bool | bool=true if there is an electrical_synapse instance between t and s; otherwise, bool=false}
chemicalSynapse	chemicalSynapse(neuron s, String pos)	chemicalSynapse: {(t, s, pos) | t and s are two neuron instances, and pos ∈ {'pre', 'post', 'any'}} → {bool | bool=true if there is a chemical_synapse instance from t to s if pos= 'pre', or from s to t if pos= 'post', or between t and s if pos= 'any'; otherwise, bool=false}
itsFiringPatterns	itsFiringPatterns()	itsFiringPattern: {t | t is a neuron instance } → {fp | fp is a firingpattern instance that t has}
Location	location()	location: {t | t is a neuron instance } → {g | g is a ganglion instance, at which t is located}
presynaptic_cell	presynaptic_cell()	presynaptic_cell: {t | t is a neuron instance } → {n | n is a neuron instance, and there is a chemical synapse from n to t}
postsynaptic_cell	postsynaptic_cell()	postsynaptic_cell: {t | t is a neuron instance } → {n | n is a neuron instance, and there is a chemical synapse from t to n}
circuit	circuit(Vector[neuron] v)	circuit: {n | n ∈ v, and v is a set of neuron instances} → {c | c is a circuit instance whose nodes include all neuron instances in v}
hasComponent	hasComponent(component p)	hasComponent: {(t, p) | t is a neuron instance and p is a component instance} → {bool | bool = true if p is a component of a chemical_synapse instance, whose presynaptic cell is t, otherwise bool = false}

*: In order to simplify the syntax of NeuroQL, there is an implicit input of a neuron instance for all advanced query operators.

4. Some NeuroQL Query Examples

In this section, we illustrate some NeuroQL query statements and their corresponding SQL query statements. Comparing the two kinds of query statements, it is obvious that it is easier for neuroscientists to write a NeuroQL query than an SQL query.

Example 1:
Query statement: all neurons that project to Nerve ‘Pd N 1’.
(Note: ‘Pd N 1’ is the name of a nerve)
NeuroQL query:

```
(neuron)[project(v), v.name = 'Pd N 1']
```

SQL query generated by SQL-Generator:

```
SELECT  n.neuronid, n.name, n.synonym
FROM    nb_neuron n, nb_nerve v, nb_project p
WHERE   n.neuronid = p.neuronid AND
      v.nerveid = p.nerveid AND v.name = 'Pd N 1'
```

Example 2:
Query statement: all neurons that have an electrical synapse connection with the neuron “R3-13”.
NeuroQL query:

```
(neuron)[connect('R3-13','electrical')]
```

SQL query generated by SQL-Generator:

```
SELECT   n.neuronid, n.name, n.synonym
FROM     nb_neuron n, nb_neuron s, nb_esynapse e_syn
WHERE   n.neuronid =e_syn.presyncell AND
        s = e_syn.postsyncell AND s.name = 'R3-13'
UNION
SELECT   n.neuronid, n.name, n.synonym
FROM     nb_neuron n, nb_neuron s, nb_esynapse e_syn
WHERE   n.neuronid =e_syn.postsyncell AND
        s = e_syn.presyncell AND s.name = 'R3-13'
```

Example 3: This is a complicated query example.
Query statement: all neurons satisfying following conditions: 1) they have molecule ‘5HT’; 2) they are the post-synaptic cell of some chemical synapse; 3) the pre-synaptic cell of the chemical synapse in condition 2) has a firing pattern ‘Irregular with Burst’.
NeuroQL query:

```
(neuron)[hasMolecules('5HT'),chemicalSynapse(s,'post'),
    s.itsFiringPatterns sfp, sfp.name= 'Irregular with
    Burst']
```

SQL query generated by SQL-Generator:

```
SELECT  n.neuronid, n.name, n.synonym
FROM   nb_neuron n, nb_molecule m, nb_hasmolecule hm
```

```
WHERE    n.neuronid=hm.neuronid AND
     m.moleculeid=hm.moleculeid AND m.name='5HT' AND
     n.neuronid IN
         (SELECT  c_syn.postsyncell
          FROM    nb_neuron s, nb_csynapse c_syn
          WHERE s.neuronid = c_syn.presyncell AND
             s.neuronid IN
                (SELECT fp.neuroid
                 FROM  nb_firingpattern fp
                 WHERE fp.name='Irregular with Burst'))
```

The SQL-Generator is being implemented using Java compiling tools JFlex and JCup. The application is a terminal based interpreter in which the user can enter NeuroQL queries and see responses. The system checks for syntax, generates and executes SQL code and displays the answers to the query.

5. Conclusions and future work

In this paper, we propose a domain-specific query language (NeuroQL) specifically for the neuroscience domain. The motivation is from the neuroscientists' desire of having a neuron query language supporting neuroscience concepts and functions so that they can directly interact with their data. Therefore, NeuroQL is designed primarily for neuroinformatics databases users, rather than the database administrators or developers. Based on the initial format and syntax proposed by the neuroscientists, we design NeuroQL as a high-level canonical query language, and implement it as an advanced query tool that can be used on any NeuroDM compatible neuroinformatics database applications. NeuroQL can be translated into most common types of query language such as SQL, OQL, and XQuery. Therefore, it is DBMS independent. NeuroQL integrates some object-oriented features, such as the dot notation, from OQL, and supports most of neuron domain-specific data types, for example, neuron, soma, axon, nerve, ganglion, synapse, molecule, firing-pattern, and so on. These data types are systematically derived from the conceptual neuron data model and mapped to the underlying database schema. This mapping will dynamically update NeuroQL data types whenever the database schema evolves. NeuroQL also defines many advanced neuron domain-specific query operators corresponding to the functions among the objects in neuroscience. The domain-specific data types and query operators are the major constructs of NeuroQL so that NeuroQL query statements are very close to the way, in which the neuroscientists ask questions or make a query in their research.

Writing an common query statement like SQL requires the users to know exactly what the database schema and constraints are, for example, table names, column names, foreign keys, etc. The common solution to this problem is that most of database applications provide users some graphical query interface with some predefined query functions. But these graphical query interfaces are usually application-dependent, and provide only limited and predefined query functions to users. NeuroQL solves this problem by providing users an abstract and succinct conceptual data model diagram. This diagram models the neuron data structure in the

database from the neuroscience point of view. Therefore, it is very close to neuron domain knowledge and easy to understand. With the conceptual data model diagram, neuroscientists can write a NeuroQL query in their familiar concepts and operators. Besides, the object-oriented concept, built-in domain-specific data types, and advanced query operators all help neuroscientists master NeuroQL quickly.

Because NeuroQL is designed for the database application users, it does not have the data definition and data update operations as in SQL, such as CREATE, ALTER, UPDATE, and INSERT operations. In the future, we will add some data update functions with restricted access to support neuroscientists to manipulate their data. Another future project is to develop more advanced query operators to facilitate interaction between neuroscientists and the database.

References

1 NeuronBank: Knowledgebase of Identified Neurons & Synaptic Connections. Website: http://www.neuronbank.org
2. The SfN Neuroscience Database Gateway. Website: http://big.sfn.org/NDG/site/
3. Tian, H., Wang, Y., Yang, H., Sunderraman, R., Katz, P.S., Zhu, Y.: A Novel Neuron Data Model with Domain-Specific Query Language. 27th Annual International Conference of the IEEE Engineering In Medicine and Biology Society (2005).
4. Deurse, A.V., Klint, P., Visser, J.: Domain-Specific Languages: an Annotated Bibliography. SIGPLAN Notices, ACM, Vol. 35 (2000) 26-36.
5. Domain-Specific Languages – An Overview. Website: http://compose.labri.fr/documentation/dsl/dsl_overview.php3
6. Bentley, J. L.: Programming Pearls: Little Languages. Communications of the ACM, Vol. 29, (1986) 711-721.
7. Amarall, V., Helmer, S., Meorkottel, G.: A Visual Query Language for HEP Analysis. Conference Proceedings of Nuclear Science Symposium, IEEE (2003).
8. Collberg, C.S.: A Fuzzy Visual Query Language for a Domain-Specific Web Search Engine. Proceedings of the Second International Conference on Diagrammatic Representation and Inference (2002) 176-190.
9. Goodman, N., Rozen, S., Stein, L.: Requeirements for a Deductive Query Language in the Mapbase Genome-Mapping Database. Workshop on Programming with Logic Databases, ILPS (1993) 18-32.
10. Hammer, J., Schneider, M.: The GenAlg Project: Developing a New Integrating Data Model, Language, and Tool for Managing and Querying Genomic Information. SIGMOD Record, ACM, Vol. 33 (2004).
11. Chen, P.P.: The Entity-Relationship Model: Toward a Unified View of Data", ACM Transactions on Database System. Vol. 1 (1976) 9-36.
12. Object Data Management Group. Website: http://www.odmg.org
13. XQuery 1.0. Website: http://www.w3.org/TR/xquery