Scientific SPARQL:
Semantic Web Queries over Scientific Data
Andrej Andrejev*1, Tore Risch*2
* Department of Information Technology, Uppsala University
Box 337, SE-75105 Uppsala, Sweden
1andrej.andrejev@it.uu.se
2tore.risch@it.uu.se

Abstract— We define an extended version of the Semantic Web query language SPARQL called Scientific SPARQL, SciSPARQL. It is targeted mainly at scientific computing and laboratory data management. SciSPARQL includes expressions, numeric multi-dimensional array operations, user-defined functions, aggregate functions, and function views. A prototype system translates SciSPARQL to a Datalog dialect which is extensible by external functions implemented in a regular programming language. The system automatically recognizes collections in RDF Turtle statements that represent numerical multi-dimensional arrays in order to represent them with a special native data type. A back-end relational database provides persistent storage.

I. INTRODUCTION

Historically, results from scientific experiments were stored in ASCII or binary files. The experimental results stored in the files are typically large multi-dimensional arrays with spatial and/or temporal semantics. Often meta-data describing classes, relationships, properties, domain constraints, keys is implicit and built into the software developed to process the results. With these traditional solutions meta-data becomes separated from the experimental results proper.

As the original goal of the Semantic Web [2] was to support semantic interoperability between applications exchanging data on the web, its technical standards, most prominently, RDF [22], RDF Schema [23], and the SPARQL query language [24] are now widely used by researchers worldwide for schema-independent storage and processing of experimental data. The RDF Data Cube vocabulary [29] provides an ontology for mapping statistical data to RDF.

The Semantic Web is based on the RDF data representation where all kinds of data and meta-data are represented as 'triples of knowledge'. The SPARQL query language provides a general way to search such triples. By using RDF to represent both data and meta-data uniformly it is possible to make semantics-aware queries that combine all kinds of data.

Scientific data processing often involves complex numerical data, such as multi-dimensional arrays and functions over such numerical data. However, SPARQL does not provide a general way of representing multi-dimensional arrays. They have to be broken down into triples, which is both unnatural and slow. Furthermore, scientific applications also require a rich set of functions operating over numerical data, which are not part of SPARQL.

In this work we propose extensions to SPARQL to support scientific applications. The extended language is called Scientific SPARQL, SciSPARQL. It is enriched with array semantics, numeric expressions, aggregate functions, function views, and external functions.

The Scientific SPARQL Database Manager, SSDM, implements SciSPARQL and provides an efficient representation of numerical arrays and functions. In SSDM arrays are represented by a dedicated data type. Basic array operators are implemented as functions over this data type. These functions can be used in SciSPARQL queries. Functions provide array transformations, like slicing, transposition, and projection avoid copying the data, which saves memory and opens to array-specific query optimization.

As the notation for numerical arrays we transparently use the Turtle notation for collections [25] by identifying rectangular (cubical etc.) collections of numbers as multi-dimensional arrays. This makes the array notation fully compatible with SPARQL’s notation for collections.

Other highly wanted functionality for scientific applications includes aggregate functions integrated into queries, and extensibility of the query language with external functions written in some programming language to implement numerical algorithms and operators. We provide both of these. We implement aggregate functions following the latest W3C SPARQL 1.1 recommendations [26]. We also make SciSPARQL extensible by providing a general way to define external functions in Python, Java, or C.

SSDM adds numerical array representations and SPARQL processing to an extensible DBMS Amos II [13]. Amos II can access other data managers, such as relational databases, RDF stores, and files though its wrapper interfaces. In particular, existing relational databases can be queried [5] in terms of RDF resources and triples [15][18].

This paper is organized as follows. Section II describes the query language SciSPARQL, explaining its major extensions with regard W3C SPARQL 1.1 recommendations. Section III provides an overview of the SSDM architecture and Section IV describes the array storage techniques and implementation of array operations. Section V presents a real-life usage scenario, Section VI gives an overview of related work, and Section VII summarizes our contributions and future work.
II. SCIENTIFIC SPARQL

Scientific SPARQL extends SPARQL with the following functionality to support scientific applications:

- Certain Turtle collections are recognized as multi-dimensional numerical arrays and internally represented as a dedicated data structure called numeric multi-dimensional arrays (NMAs) to enable efficient storage of such collections and queries involving array operations.
- SciSPARQL provides extended syntax in the SELECT class for expressions, in particular numerical expressions.
- User defined external functions enable arbitrary computations to be implemented in some external programming language, e.g., Python. This provides access to Python’s numerical packages.
- User defined aggregate functions enable implementation of summary values over collections.
- User defined functions implemented as SPARQL queries enable to define common expressions and queries as function views.

A. Multi-dimensional Arrays and Numerical Expressions

SciSPARQL allows the use of arbitrary expressions in the SELECT clause, including numerical expressions, named by variables, e.g.:

```
SELECT (f(2*?x+3) AS ?y) ...
```

Numerical expressions in SciSPARQL are infix expressions where basic terms are variables, numeric constants, function calls, or array dereferences.

Slices are specified as in NymPy [20] using the notation lo:hi or lo:hi:stride for each dimension where lo denotes the lower bound of a slice and hi is the upper one. Subscripts are 0-based and therefore the element hi is never part of the array defined by the slice. lo defaults to 0 and hi to the dimension, while stride defaults to 1, for example:

```
SELECT (?a[::1,0:5] AS ?firstFiveColumns) (?b[1::2] AS ?everySecondRow)
```

Array elements and slices can be passed to functions or returned like any other values.

The SciSPARQL syntax for projections is similar to array slicing with a colon (:) indicating a projected dimension, for example A[1:,1], B[1:,1:2:3:5].

Regular array dereferencing is specified by supplying all subscripts, for example A[1,2].

B. External functions

SciSPARQL functionality can be extended by defining external functions implemented in, e.g., Python. For example, a plus function can be defined in SciSPARQL as:

```
DEFINE FUNCTION plus(?a) AS PYTHON 'myplus';
```

The implementation in Python of `plus` would be:

```
def plus(a, b): return a+b;
```

Notice that the arrays themselves are stored and managed by SSDM, not by Python; only an external function’s logic is implemented in Python.

C. Aggregate Functions

Aggregate functions differ from normal functions in that they produce one result value for a bag of input values. In SciSPARQL, they are defined as external functions:

```
DEFINE AGGREGATE sum(?a) AS PYTHON 'mysum';
```

The corresponding Python implementation code is:

```
def mysum(b): return sum(b);
```

Actually, in this case user function `mysum` is not required - the built-in Python `sum` function can be used directly.

D. Function Views

A SciSPARQL function can also be implemented entirely in terms of SciSPARQL, thus defining a function view by a subquery. Function views cannot be recursive. For example, the following function will calculate the sum of the \(x\) and \(y\) properties of a given subject \(s\):

```
DEFINE FUNCTION sumxy(?s) AS SELECT (?x+?y AS ?res)
WHERE { ?s :x ?x ; :y ?y }
```

III. THE SCI SPARQL DATABASE MANAGER

We have developed a prototype system to process SciSPARQL queries, called the SciSPARQL Database Manager (SSDM). SSDM utilizes the extensible DBMS Amos II [16], for its in-memory data storage and interfaces to relational databases. Fig. 1 shows the overall architecture.

The RDF loader is used to load both instances of experiments (parameter values, measurements), and meta-data describing classes, relationships, and attributes in experiment data. The latter can be designed by the user with help of ontology design tools like Protégé [7]. Both kinds of data are loaded from RDF files, and stored in the relational database back-end RDB as RDF(S) triples.

A main memory local database inside SSDM represents temporary data required for processing SciSPARQL queries over RDF, such as cached triples and numeric arrays.

Numeric arrays are automatically recognized when Turtle collections are imported or in SPARQL queries. They are represented as numeric multi-dimensional arrays (NMAs) in the local database and are converted to a corresponding BLOB representation when stored in the RDBMS back-end.
SciSPARQL is an extensible language where functions can be implemented in a foreign language and used in queries. The SSDM is integrated with the Python engine, which is used as run-time system for SciSPARQL external functions.

A storage object represents a one-dimensional array of either integer, double, or complex numbers. It contains a small header and is therefore self-descriptive. A descriptor object stores a pointer to a storage object, the number of dimensions `dims` of the array, the index `offset` of the first element in the storage object referenced in a derived array, and a sequence of dimension access descriptors (DADs), each describing one dimension of a derived array enumerated from 0 and up.

A given storage object can have many descriptors corresponding to different derived arrays. When a new array is created, both the descriptor and storage objects are allocated in main memory. When a derived array is produced, a new descriptor object is created linking to the storage object of the original array. Descriptor objects are automatically freed by a garbage collector whenever no variable or object refers to it. When the last descriptor object is freed, the garbage collector frees the storage object as well.

### IV. ARRAY FUNCTIONALITY

All RDF datatypes [22], including URIs and Unicode Strings, are natively supported in SSDM, its query processor, and its storage system. Distinctive SciSPARQL features such as NMAs are integrated into the core of our system. Detailed explanation of in-memory array storage and operations to produce derived arrays is given below. As a relational back-end storage system we interface Chelonia [19], which has a capacity to handle large multi-dimensional arrays.

#### A. In-memory Representation

In our native main-memory data storage, NMAs are represented as descriptor objects referring to storage objects, as shown on Fig. 3a. The storage objects compactly stores the elements of an array in continuous memory, while the descriptor objects provide very space efficient representations of derived arrays. This allows us to compute derived arrays without copying data.

For each array, its dimension sizes are stored in corresponding `dim` fields of its DADs. The storage order (so) values enumerate the dimensions from outmost to inmost dimension. The lower bounds (lo) are initialized to 0, and the iteration strides (stride) are initialized to 1. In this simple case, the access function \(a(i_0, \ldots, i_{n-1})\) that maps the array subscripts to the storage index takes the form:

\[
a(i_0, \ldots, i_{n-1}) = \sum_k i_k \cdot \prod_{m > k} \dim_m
\]

This expression is simplified by pre-computing the access multipliers (am), representing invariant parts of the formula per array dimension:

\[
am_k = \prod_{m > k} \dim_m
\]

#### B. Array Operations

In the most general case, an array access involves a physical offset, some iteration strides, and some lower bounds of each logical index. The complete form of an access function \(a_k\) for accessing one element \(i_0, \ldots, i_{n-1}\) of the array \(A\) is:

\[
a_k(i_0, \ldots, i_{n-1}) = \sum_k i_k \cdot \prod_{m > k} \dim_m
\]

This expression is simplified by pre-computing the access multipliers (am), representing invariant parts of the formula per array dimension:

\[
am_k = \prod_{m > k} \dim_m
\]
$$a_A(i_0, ..., i_{n-1}) = \text{offset} + \sum_k p_k^A(i_k) \cdot am_k$$

where

$$p_k^A(i_k) = l o_k + i_k \cdot \text{stride}_k$$

The function $p_k^A(i_k)$ projects a logical subscript $i_k$ of the derived array $A$ to the logical subscript of the basic array.

1) **Permutation of dimensions** is a multi-dimensional generalization of the matrix transposition operation. Given an $n$-dimensional numeric array $A$, the order of logical subscripts used to access its elements can be changed without affecting the physical order. This involves swapping the DADs, while retaining their access multipliers (am) intact, Fig. 3b.

The operation $\text{Permute}(A, h_0, ..., h_{n-1})$ takes an array $A$ and a vector of distinct permutation indices $h_0, ..., h_{n-1}, 0 \leq h_i < n$ and returns a derived array $B$ such that the access functions map to the same elements as pointed to by the permuted subscripts:

$$a_A(i_0, ..., i_{n-1}) = a_B(h_0, ..., h_{n-1})$$

$\text{Permute}$ is a SciSPARQL function that can be used in the SELECT clause or FILTER expressions. Matrix transposition is defined by the function view:

**DEFINE FUNCTION Transpose(2matrix)**

**AS SELECT Permute(2matrix, 1, 0);**

2) **Slicing** is an operation that can be applied to each array dimension independently, resulting in a subset of array slices specified by subscripts and stride, as shown on Fig. 3c. Given an $n$-dimensional numeric array $A$, the operation $\text{Sub}(A, l o_k, h_i, \text{stride}_k)$ results in an derived array $B$ of the same dimensionality, where the first element is defined by $l o_k$:

$$a_A(0, ..., l o_k, ..., 0) = a_B(0, ..., 0)$$

so that, effectively, the $l o$ value in the $k$-th DAD of the resulting array $B$ is:

$$l o_k = p_k^A(l o_k)$$

Analogously, the iteration stride of $B$ is multiplied by the $\text{stride}_k$ argument:

$$\text{stride}_B = \text{stride}_A \cdot \text{stride}_k$$

The dimensions of $B$ are defined as:

$$\text{dim}_B = \left[ \frac{p_k^B(h_i - 1) - l o_k}{\text{stride}_k} \right] + 1$$

When applied to different dimensions, slicing operations are completely orthogonal and commutative.

3) **Projection** involves reducing the dimensionality of an array by selecting one subscript value in a specified dimension - either row (Figure 3d) or column (Figure 3e) of a matrix, slice of a cube, etc. Projection removes one of the DADs, while retaining the access multipliers for the other dimensions untouched. The operation $\text{Pr}(A, l o_k)$ results in a derived array $B$ where the offset references the first element of $B$:

$$\text{offset}_B = a_A(0, ..., i_k, ..., 0)$$

C. Data input

When a Turtle file is imported by the RDF loader, collections that can be converted to numeric arrays are identified and represented as NMAs. For example, the RDF object of the Turtle statement

```turtle
:x :a (1 2 3) (4 5 6).
```

is in SSDM represented as a single NMA literal with dimensions two and three. The conventional naïve representation as RDF triples would require 16 triples and eight blank nodes. In contrast, the object of Turtle statement

```turtle
:y :a (1 (2 3) 4).
```

cannot be represented as an NMA, since it is not rectangular. Therefore it is represented as a regular RDF sequence where the 2nd element is represented as a one-dimensional NMA. This saves the storage of four triples and two blank nodes.

In practice, the numeric sequences are expected to be very large, so NMA vastly outperforms the standard RDF triple representation.

V. EXAMPLE

For evaluation of our system we use the following real-life scenario from the field of computational biology. In the scenario experimental data about yeast polarization experiments are represented in RDF. Fig. 4 shows the schema.

![Fig 4 Yeast Polarization Experiment: EER diagram](image-url)

0prefix : <http://udbl.uu.se/YeastPolarization#> .

```turtle
:Experiment001 a :YeastPolarizationExperiment ;
:ModelName "ALL_Alt" ;
:ModelVersion 1 ;
:SimulationAlgorithm "ISSA" ;
:InputType "GradientWithSwitching_Input" ;
:Diffusion 0.01 ;
:TimeStep 30 .
```

Fig 5 RDF data file in Turtle format

The class YeastPolarizationExperiment represents properties of experiments about yeast polarizations. Each experiment is recorded as a number of trajectories represented by the class Trajectory. Each trajectory has a property Width that contains the measured values of the trajectory as a time series array of floating point numbers (class FloatNMA). The trajectories are computed using some SimulationAlgorithm and InputType. Each trajectory has an associated trajectory ID.
TrajNo and the simulation parameters Km and kon. A
collection of such time series is associated with an instance
of a simulation experiment through the property inExperiment.
The attribute TimeStep of an experiment represents the time
step of the trajectory in seconds.

Fig. 5 contains an example of an RDF data file in Turtle
format compliant with the above schema. Empty brackets
denote blank nodes – the RDF equivalent of surrogate keys.

Different experiment instances might have different
attributes, and multi-parameter search can be implemented
across different parameter sets. If we would store the data in a
relational database, this could effectively mean a separate
schema for each instance of an experiment. The RDF data
model is more flexible, as we do not require all instances of a
class to have the same attributes and relationships, which, in
fact, we cannot foresee in a rapidly evolving laboratory
information context.

Examples of queries to the above schema:
1) What is the mean and variance of the values of each
trajectory, having kon parameter below 0.05? This is
expressed in SciSPARQL as:

```
PREFIX : <http://udbl.uu.se/YeastPolarization#>
SELECT ?km ?kon
WHERE {
  ?trData a :TrajectoryData ;
  :inExperiment :Experiment001 ;
  :Km ?km ;
  :kon ?kon ;
  :TrajNo ?trajNo ;
  :Width ?Width .
  FILTER (?kon < 0.05) };
```

The functions mean and variance take a single array as
argument and return a scalar. In the result we get a set of
tuples for each trajectory containing bindings for the variables
Km, kon, TrajNo, WidthMean, and WidthVariance.

2) For each combination of the parameters km and kon,
where kon is below 0.05, compute the mean trajectory where
each value is the average of the stored trajectory values.

```
PREFIX : <http://udbl.uu.se/YeastPolarization#>
SELECT ?km ?kon
WHERE {
  ?trData a :TrajectoryData ;
  :inExperiment :Experiment001 ;
  :Km ?km ;
  :kon ?kon ;
  :TrajNo ?trajNo ;
  :Width ?Width .
  FILTER (?kon < 0.05) }
```

The aggregate function meanAgg takes a bag of arrays as
argument and computes a new array of the mean values element-wise of the input arrays. The query looks similar to
the previous query, except that here the aggregate function
meanAgg is applied on a bag of ?Width arrays. Here we
effectively do grouping by Km and kon values as specified in
[26], and aggregate across different trajectories.

3) What is the mean of the last five trajectory values in
trajectories with time step of 2 minutes?

```
PREFIX : <http://udbl.uu.se/YeastPolarization#>
SELECT ?km ?kon ?TrajNo
WHERE {
  ?trData a :TrajectoryData ;
  :inExperiment :Experiment001 ;
  :Km ?km ;
  :kon ?kon ;
  :TrajNo ?trajNo ;
  :Width ?Width .
  :TimeStep ?timestep
  .
  FILTER (?timestep > 0) }
```

Here we define a function view in SciSPARQL - the
function final_time that takes a resource of class Trajectory
as argument and returns a numeric value. The function uses a
SciSPARQL query to find the corresponding experiment
entity and its TimeStep value. It also accesses the first
dimension size of the Width property of the trajectory, which
is a one-dimensional array.

```
WHERE { ?trajectory :inExperiment ?experiment ;
  ?experiment :TimeStep ?timestep }
```

Several other extensions to SPARQL were proposed to
make the query language more useful for certain tasks. Lausen
et. al. [10] suggest specifying RDBMS-style data constraints
(primary and foreign keys, domain ranges, subclasses and
superproperties) when mapping to RDF, and making use of this
information during SPARQL query optimization. Kochut and
Janik [9] extend SPARQL with functionality of discovering
semantic paths of possibly unknown length, specified as
regular expressions over subjects and properties. Barbieri et.
al. [1] propose syntax extensions to define windows over
streams of RDF triples. These proposals are orthogonal to
ours. We are not aware of any extension of SPARQL with
array functionality insofar.

VI. RELATED WORK

Several other extensions to SPARQL were proposed to
make the query language more useful for certain tasks. Lausen
et. al. [10] suggest specifying RDBMS-style data constraints
(primary and foreign keys, domain ranges, subclasses and
superproperties) when mapping to RDF, and making use of this
information during SPARQL query optimization. Kochut and
Janik [9] extend SPARQL with functionality of discovering
semantic paths of possibly unknown length, specified as
regular expressions over subjects and properties. Barbieri et.
al. [1] propose syntax extensions to define windows over
streams of RDF triples. These proposals are orthogonal to
ours. We are not aware of any extension of SPARQL with
array functionality insofar.

As SSDM Sesame [27] and CORESE [28] also allow
extensibility of SPARQL with simple external functions in
Java. In addition SSDM provides user defined aggregate
functions and function views. Since our extensions are
targeted at array processing, we choose Python with the
NumPy library [20] as the primary imperative-language to
implement external functions in SciSPARQL operating over
array data stored in SSDM.

There have been several projects extending relational
databases with array semantics. Lerner and Shasha [11] are
generalizing the idea of arrays as 'ordered data', and discuss
the optimization opportunities of AQuery, an order-aware
query language they define. Kersten et. al. [8] view arrays as
relational tables, where dimension values comprise the
primary key. They suggest an extension to SQL, called SciQL
where basic array operations are defined. These systems
introduce arrays on the schema level, while in SciSPARQL
arrays are data instances.
Baumann et. al. [6], takes a different approach in the RasDaMan system, which internally stores sections of arrays called tiles in an RDBMS and represents arrays as abstract data types in queries and applications. Recently, Dobos et. al. [4] have studied the requirements of scientific applications for storing arrays and implemented their own relational database extensions on top of Microsoft SQL Server, storing the array data in BLOBs and accessing it via T-SQL UDFs. Recently, the SciDB [3][21] project was started, targeted at storing large amounts of chunked array data in distributed environments. Their latest work [17] studies the strategies for array tiling and amounts of chunked array data in distributed environments.

By contrast, we define arrays as an abstract data type for SPARQL. Furthermore, we use array descriptor objects to efficiently derive new arrays from large base arrays, while in the other systems raw base array sections are stored in a database. We use the Chelonia system [19] as a distributed relational back-end for scalable storage of very large arrays also based on chunking/tiling.

Dedicated array databases and query languages have been developed actively in past decades. In 1996 Libkin et. al [12] defined the query language AQL, including calculus and algebra for accessing multi-dimensional arrays, e.g. concatenation, slicing, and aggregates. In AML [14] bit patterns are used to optimize array access and transformations exploiting operational associativity and externally defined locality. Rather than having arrays as the only data type, SciSPARQL handles other kinds of data than arrays as well.

VII. CONCLUSIONS AND FUTURE WORK

Extensions to SPARQL to handle scientific data were defined. The extended language SciSPARQL includes array operations, function views, expressions, and definition of external functions implemented in e.g. Python. SciSPARQL is implemented in a prototype system called SSDM (SciSPARQL Database Manager). SSDM provides efficient representation for arrays identified in RDF collections and implements functionality to efficiently produce derived arrays without copying data. The system architecture of SSDM was overviewed along with descriptions and examples of how arrays are represented and processed in SSDM.

SSDM is being integrated with the distributed relational back-end system Chelonia [19] designed for scalable array storage and access. This includes strategies for distributed processing of multi-dimensional array queries.

Evaluation work is in progress based on a real-life usage scenarios. The scenarios provide motivations for introducing new features required for scientific computing. Among these foreseeable features are second-order functions. Another direction is efficient processing of RDF streams, by adding stream and window semantics to SciSPARQL.

ACKNOWLEDGMENT

This work was supported by eSSENCE and the Swedish Foundation for Strategic Research, grant RIT08-0041.

REFERENCES