Scientific Analysis by Queries in Extended SPARQL over a Scalable e-Science Data Store

Andrej Andrejev[#1], Salman Toor[#2], Andreas Hellander[*3], Sverker Holmgren[#4], Tore Risch[#5]

[#] *Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden.*

[1]andrej.andrejev@it.uu.se, [2]salman.toor@it.uu.se,

[4]sverker.holmgren@it.uu.se, [5]tore.risch@it.uu.se

** Department of Computer Science, University of California Santa Barbara, 93106, USA.*

[3]andreash@cs.ucsb.edu

***Abstract*— Data-intensive applications in e-Science require scalable solutions for storage as well as interactive tools for analysis of scientific data. It is important to be able to query the data in a storage-independent way, and to be able to obtain the results of the data-analysis incrementally (in contrast to traditional batch solutions). We use the RDF data model extended with multidimensional numeric arrays to represent the results, parameters, and other metadata describing scientific experiments, and SciSPARQL, an extension of the SPARQL language, to combine massive numeric array data and metadata in queries. To address the scalability problem we present an architecture that enables the same SciSPARQL queries to be executed on the RDF dataset whether it is stored in a relational DBMS or mapped over a specialized geographically distributed e-Science data store. In order to minimize access and communication costs, we represent the arrays with proxy objects, and retrieve their content lazily. We formulate typical analysis tasks from a computational biology application in terms of SciSPARQL queries, and compare the query processing performance with manually written scripts in MATLAB.**

I. Introduction

With data-intensive science realized as a fourth paradigm [17], a number of efforts have been made to address data management and analysis needs in science. Applications in astronomy, geosciences, high-energy physics, computational biology, and many other fields of natural sciences produce large amounts of observation and simulation data, typically in form of multidimensional numeric arrays accompanied by metadata describing the complex structure and parameters of the experiments. While the source of the generated data is different in each case, they all face the same challenge: how can massively generated data, ramping towards exascale and beyond, be managed and analysed in a collaborative environment?

In early solutions, the complete workable datasets had to be moved from the data store to a computing node, before the analysis could be initiated. As the datasets kept growing, the need for finer-grained workflows became more evident, so that application-side processing could be started as soon as data begins to arrive. More interactive data analysis tools that also provide simple ways to express complex search conditions and combine existing data processing methods are certainly wanted by scientists in application areas.

The use of relational databases in scientific computing environments has been limited so far. One of the hindrances is the need to design the database schema; an efficient use of a database requires a certain level of expertise in data engineering. Another key issue is the nature of the scientific data itself: it frequently consists of complex data types such as matrices and tensors, which are normally not handled by relational databases. Instead, scientific data is often stored in ad-hoc text or binary file formats or specialized databases, like ROOT [10] or NetCDF [4].

The RDF data model was initially proposed as a schema-free alternative to relational and hierarchical databases, and is so general that most other data models can be easily mapped to RDF. Mapping from tree-like (hierarchical, XML) data to an RDF graph is straightforward, and mappings from relational data model are examined e.g. in [26]. The fact that no schema has to be specified upfront, but can optionally accompany the data, gives greater freedom for the applications, especially in experimental sciences where the organization of data constantly evolves from one experiment to the next. SPARQL [7] is the W3C standard query language for RDF.

However, as we have shown in [8], to make the RDF model useful and accepted in scientific applications one needs to extend the RDF data model and the SPARQL query language with capabilities to represent, search and process large numeric arrays. . For this we have defined our extension, SciSPARQL, and implemented it in our prototype system, Scientific SPARQL Database Manager, SSDM [8]. In this work we show how our approach scales to handle large datasets efficiently. To be able to query massive scientific data in terms of RDF, two usage scenarios are possible:

1) Loading data into a storage back-end system once and querying it afterwards. This is primarily useful if the original data is represented by a set of binary or text files. Converters to an RDF format (e.g. Turtle) need to be written. In section IV we define a general relational storage schema to represent scientific RDF data on top of any RDBMS supporting standard SQL. SciSPARQL queries are then translated to SQL calls to retrieve metadata and binary chunks representing the desired fragments of arrays.

2) Accessing data in a native e-Science data repository by mapping it to the RDF model. Since RDF is extremely flexible, it is usually easy and straightforward to map any structured data to RDF. The SciSPARQL queries can then be translated to native queries or API calls of the storage system. In this paper we define an RDF view of scientific experiment data stored in a distributed data store, Chelonia [27] , extended for scientific applications in order to store simple and complex (e.g. vectors and matrices) values of variables describing the parameters and results of experiments.

Our main contributions in this paper are:

- We extend the prototype system SSDM implementing SciSPARQL queries with the ability to query massive scientific data residing in any relational DBMS or in external distributed storage systems, such as Chelonia.
- We define a generic relational database schema for RDF with arrays, and use it with SSDM to implement a back-end storage in any RDBMS supporting SQL.
- We show how to map a flat spreadsheet-like application metadata model in Chelonia to the RDF data model by defining an RDF view over it. In this way, we are able to expose exactly the same relational schema as in Chelonia, allowing SciSPARQL queries to be formulated independently of the storage system.
- We evaluate our approach within the context of a highly data-intensive kinetic Monte Carlo simulation application from molecular systems biology called BISTAB. We demonstrate the expressivity of the SciSPARQL language on a for the application representative set of queries, present the run times with different storage back-ends, and show that this approach results in comparable performance to the use of hand-written MATLAB scripts reading files directly from disk; the previously employed post-processing workflow by the users of the application.

The rest of the paper is organized as follows. Section II gives an overview of related work. Section III describes the SciSPARQL language and its implementing system, SSDM. Section IV describes storage and retrieval of RDF data with a relational DBMS back-end. Section V describes the Chelonia data store and its integration into our architecture. Section VI introduces the systems biology application. In section VII we define the queries from section VI in SciSPARQL and evaluate the performance in different settings. Finally, section VIII discusses future directions and concluding remarks.

II. Related Work

In this section we review related work in which database technology is employed to facilitate data analysis in scientific applications.

The SciDB project [9] addresses similar issues of management of complex data but with a different approach. SciDB provides a native parallel data management system for large scale array processing. The whole database is organized as a collection of n-dimensional arrays. For extensibility SciDB provides user-defined-types (UDT) and user-defined-functions (UDF) similarly to Postgres [6]. The architecture of SciDB is based on a share-nothing approach in which each node runs a semiautonomous instance of the SciDB engine and communicates with a centralized catalog system, which makes it a scalable e-Science data store. In contrast to our approach, the SciDB data model requires that all data is contained in named arrays, that can be updated and queried for content or shape, but no other named properties are supported. In the context of e-Science data management this would mean a proliferation of single-element arrays containing strings, dates and other metadata. Managing scientific experiments require both metadata that describe properties of experiments and numeric data to represent the results. The example SciSPARQL queries in this paper illustrate the importance of this synergy. Furthermore, with our framework, array-structured data residing in SciDB can be made available using SciSPARQL queries, once an RDF view is defined in terms of the AQL interface language of SciDB.

There have been several projects extending relational databases with array semantics. Lerner and Shasha [19] are generalizing the idea of arrays as 'ordered data', and discuss the optimization opportunities of AQuery, an order-aware query language. Kersten et. al. [18] 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.

SciHadoop [11], as an extension of Hadoop's Distributed File System (HDFS) [23], is another effort to facilitate scientific applications by managing array data efficiently. In Hadoop data is managed by utilizing the MapReduce strategy for writing fault-tolerant 'embarrassingly parallel' batch programs. Since the actual distribution of data is formed regardless of its contents, it can be very inefficient for handling array data. To alleviate this SciHadoop implements a plugin for Hadoop which employs NetCDF to manage array data and also to help choosing the data distribution strategies. To achieve these results SciHadoop modifies the standard task scheduler. A customized array based query language is used to explicitly specify operations on the available data in each parallel Hadoop node (as the map function).

ArrayStore [24] is another similar effort for enabling complex parallel array processing. It is a specialized storage manager and not built on top of any relational DBMS. The work includes an array partitioning strategy to gain better performance and also demonstrates how to define operators for efficient array access in neighboring array partitions during parallel processing. The whole system is written in standard C++. The article also presents a detailed performance evaluation based on different partitioning strategies together with different operators.

Compared to the above-mentioned projects [11] and [24], we follow a different approach: storage distribution is handled entirely by a back-end system, and can be configured independently of the application. For example, nodes in Chelonia are autonomous and can be geographically distributed. We also provide a complete solution both for

simple data types and massive numeric arrays, so it is fairly easy to analyze complex numeric data by writing declarative queries using SciSPARQL. The extensibility of SciSPARQL allows applications to run custom modules for advanced data analysis. Currently the technique we are using to store multidimensional numeric arrays in the relational database is based on an array chunk partitioning scheme. To speed up the interaction, on SSDM side we use array descriptor objects in memory and array proxy objects pointing to a back-end database to effectively accumulate array projection and transformation operations without physical access to array contents.

III. SciSPARQL – Scientific SPARQL

A. Syntax and semantics

SciSPARQL is an extended version of the W3C query language SPARQL 1.1 [7]. SciSPARQL extends SPARQL with syntax and semantics for accessing numeric arrays of arbitrary dimensionality, array slicing, projection and transposition, and the ability to connect and invoke foreign functions and define functional views.

For example, using array slicing for selecting a matrix containing every second column of the original matrix bound to variable *?a* is expressed as *?a[:,::2]*, and accessing every row of the same matrix is expressed as *?a[?i,:]*. The latter expression also generates all corresponding bindings for the variable *?i* used as row subscript.

The slicing syntax is borrowed from Python, where *lo:hi:stride* values are specified for every dimension, any of the specifiers may be omitted, and *':'* denotes a complete range in a dimension. Elements are enumerated starting from 0, and elements corresponding to *hi* (the upper bound) are never included into the result. When projecting or selecting an element, a single subscript value is supplied instead of range, e.g. *?a[?i,?j]*.

One of the important features of SciSPARQL is the array aggregate functions, like array *array_sum(?a)* that returns the sum of all elements of a given array argument. Array aggregate functions operate over arrays, unlike SPARQL 1.1. where aggregate functions such as *SUM()* operate over bags. In SciSPARQL bag-oriented aggregate functions are overloaded to operate on arrays of uniform dimensionality, so using *SUM(?a)* in the *SELECT* clause will compute an array whose elements are sums of respective elements of all arrays bound to variable *?a* within a query or grouped query values.

SciSPARQL is extensible with algorithms expressed in conventional algorithmic languages, like Python, Java, or C. It is possible to define both regular and aggregate functions.

For example, the aggregate function

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

can be implemented in Python as

```
def mysum(b): return sum([i[0] for i in b])
```

We believe that this kind of extensibility is vital for a query language to be adopted in the scientific community, where historically the information processing logic has been evolving in the context of algorithmic programming languages, and libraries of quite sophisticated solutions exist and play the role of de-facto standards.

Another distinctive feature of SciSPARQL is *functional views*, which are user-defined functions expressed in terms of *SELECT* queries. This allows for building up libraries of SciSPARQL subqueries defining standard computation formulas or commonly used data retrieval tasks for further use. We define two such functions in Chapter VII, both for purpose of re-using the code, and encapsulating the argument to the second-order system function *ARGMAX()*.

During query processing the functional views are expanded similarly to SQL views, in order to give the query optimizer greater freedom for finding the optimal order of execution. As a query language, SciSPARQL is declarative, optimizable, and terse. This means that most kinds of conditions and constraints on the data retrieved from a database can be expressed directly as algebraic equations. It is the responsibility of the DBMS to come up with an optimal execution plan, taking into account storage statistics, distribution, communication and computation time. Even the foreign functions can be associated with cost models used for query optimization.

B. Scientific SPARQL Database Manager (SSDM)

SSDM is our prototype system built on top of the object-relational DBMS Amos II [22] providing a cost-based query optimizer, extensible type system, and foreign functions.

We have extended Amos II with a new type representing numeric multidimensional arrays, NMAs. In SSDM arrays stored in main memory are represented by storage and descriptor parts. Most array operations, like slicing, projection, and transposition are implemented in terms of producing new descriptors for the same storage type, without copying or otherwise accessing array elements. A number of derived arrays, e.g. the ones representing the columns of an original matrix, share the same storage object in memory as explained in [8]. In section IV we also define array proxy objects that operate exactly as array descriptors but point to arrays stored in the back-end database instead.

SSDM supports a stateless query-only interface for SciSPARQL, like most other SPARQL endpoints working as a basis for the Semantic Web. In addition a convenient interpreter-style session interface is provided, where SciSPARQL directives like *DEFINE* are available. There are also directives, like *SOURCE()* for batch execution of SciSPARQL scripts (typically consisting of function definitions), *LOAD()* for loading RDF data from files, e.g. in Turtle format, *DUMP()* for saving RDF data to Turtle files, and *QUIT* for ending the session. Any SciSPARQL function can be called as a directive, with results returned the same way as results of *SELECT* queries.

IV. External Storage of RDF with Arrays

We have extended SciSPARQL Database Manager with the following features:

- utilizing any relational DBMS with SQL support as back-end storage for RDF data including *multidimensional numeric arrays*;
- representing stored arrays, their subsets and elements with *array proxy* objects;
- retrieving array data in binary *chunks*, and effectively cache and reuse these chunks;
- downloading and storing in indexed main-memory all metadata as RDF triples. In our e-Science applications the amount of memory required to store the metadata information represented by triples is several orders of magnitude smaller than the size of data in numeric arrays and therefore fits in main memory.

A. SQL-based RDF storage schema

To allow the stored amounts of numeric array data to scale beyond the capacity of main memory, we have defined a simple SQL-based storage schema, as shown on Fig. 1. The table representing the set of RDF triples is horizontally partitioned based on object type: *URITriples*, *LiteralTriples* and *ArrayTriples*. Sice URIs are commonly repeated in RDF datasets, we introduce a *URI* dictionary table, and store integer identifiers of URIs in all other places.

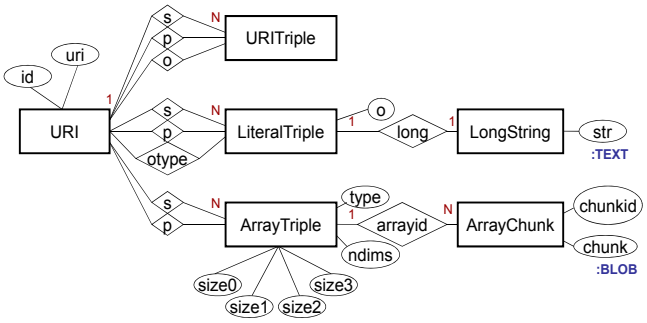

Fig 1. The SQL-based storage schema for RDF with arrays.

We also introduce the *LongString* table with an arbitrary-length TEXT field to store literal values beyond the limit of 32 characters, and the *ArrayChunk* table with a BLOB field to store binary chunks of numeric arrays. Arrays are identified by auto-incremented *arrayid* values, and chunks with *chunkid* values, unique within the scope of an array. The element type, number of dimensions *ndims,* and the sizes of the first four dimensions of an array are explicitly stored in the *ArrayTriple* table. For arrays of higher dimensionality the shape information is stored as a sequence of *(ndims-4)* integer values in the beginning of the first *chunk*.

B. Array proxy objects

A special kind of descriptor is used when accessing large arrays stored in relational back-ends or external repositories such as Chelonia. For each reference to an external array an array proxy is created that stores all information necessary to retrieve the array. When using an SQL back-end, this information includes a unique array identifier, dimensions, and range. For Chelonia this is database name, task identifier, variable name, shape, and range information. As with the in-memory descriptors mentioned above, an array proxy representing a derived array is computed as the result of applying array operations over original array proxies, without accessing the external repository. Typically each application of such array operations leads to a smaller derived array. A derived array is retrieved 'lazily', i.e. as late as possible, when returning it as a result or passing it to an external algorithm for numeric processing.

The chosen approach enables the streaming of array data into memory, and applying the computations at reasonably low granularity. A really large array cannot be put into memory in its entirety, however, the operations of interest, like summing up or averaging, can be performed row-by-row or column-by-column. A SciSPARQL query includes the expressions specifying what parts of the array need to be retrieved per operation, as shown in section VII A. These queries are executed in such a way that they retrieve one array piece at a time, and completely avoid accumulating massive intermediate results.

V. CHELONIA STORAGE SYSTEM

The Chelonia storage system [20] is a joint effort of the NorduGirid collaboration [5]. It has been developed with the next generation components of Advanced Resource Connector (ARC) middleware [1] under the knowARC project [3]. Chelonia is a service-oriented storage system originally designed to manage data on geographically distributed storage nodes. The design goal was meeting the requirements ranging from simple data sharing to the support for scientific analysis. Being part of the ARC middleware, Chelonia provides flexible data accessibility for the grid jobs running under ARC.

A. Architecture

The architecture of Chelonia is based on four core services, Bartender; front-end of the system, Librarian; catalog service, A-Hash; metadata store and Shepherd; service running at the storage node. The system architecture ensures scalability, efficiency, and ease to deploy and maintain. [20] describes its features and architectural details whereas [2] documents technical aspects of the system.

The extended version of Chelonia operates on top of MySQL DBMS, and defines a generic schema shown on Fig. 2, typically used by the scientists to store experimental data. It features Experiment and Task entities, and Variables, that might have different types and values corresponding to different tasks.

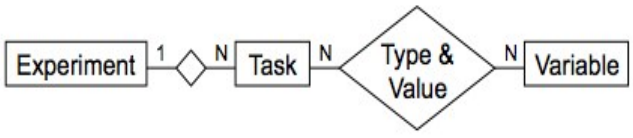

Fig. 2. ER-diagram of Chelonia storage schema

The storage types supported are character strings, integer and real numbers, and the multidimensional arrays of numbers. Chelonia is highly optimized for storing and querying large

arrays, and provides transparent array chunking: the SQL queries passed to Chelonia Bartender service are translated to address physical data on the storage nodes, and if array range is provided in a query, only the corresponding array chunks will be accessed.

B. Integration

In the current architecture Chelonia works as a distributed storage manager for BISTAB data and the access from SSDM is provided through a wrapper interface. The aim is to provide a flexible and scalable solution for data analysis. Chelonia provides the scalable and distributed storage of scientific data and SciSPARQL enables data analysis through queries.

Chelonia exposes interfaces accessible by SOAP over HTTP(S) requests and provides access to arrays, subarrays, and simple values. The stored data is presented as RDF triples inside SSDM and becomes queryable with SciSPARQL and accessible to the foreign functions.

In SSDM an RDF view of Chelonia is generated where tasks and experiments are mapped to RDF subjects, names of variables to RDF predicates, and values (including arrays) to RDF objects. This is achieved with a call to the SciSPARQL directive

```
CheloniaExperiment(namespace, db, experiment_name);
```

where *db* is the name of the Chelonia database being mapped, and *experiment_name* denotes the experiment that will be available as RDF triples enhanced with arrays. The prefix *namespace* is used in all queries during the session. This exposes the application-level schema for BISTAB to SciSPARQL queries.

When a dataset is connected, the metadata about all tasks and their respective variables gets downloaded and cached inside SSDM. In RDF terminology, this becomes a cache of all subjects and all subject-property pairs. Query joins of triple patterns are not executed by Chelonia, so most of the pattern-matching job is done in the SSDM in-memory database containing the cache. Only object values are retrieved from Chelonia.

It should be noted that the current interface to Chelonia is read-only. In the BISTAB application discussed in this article, the experimental data is loaded into Chelonia using a separate shell script.

V. URDME – An Application from Systems Biology

a) Stochastic reaction-diffusion kinetics: Simulation of reaction kinetics is frequently used to study the dynamics of regulatory biochemical pathways inside the cell. Key regulatory macromolecules, such as transcription factors or messenger RNAs, are often present in few copies, rendering the dynamics stochastic in its nature. This so called *intrinsic noise* is a factor that needs to be accounted for when studying cellular regulatory pathways since it can be expected to have a profound impact on the regulatory mechanisms. For example, some network motifs make the pathways more robust to molecular fluctuations [25].

In a discrete stochastic setting, the most common modeling framework is continuous-time discrete-space Markov processes. Statistically correct realizations of the process can be generated using kinetic Monte Carlo (kMC) methodology, such as the Stochastic Simulation Algorithm (SSA) [16]. To introduce spatial heterogeneity in the models, the computational domain is discretized into non-overlapping mesh cells, and diffusion is modeled as discrete jump events along the edges of the mesh. Recent computational studies have highlighted scenarios where both spatial and stochastic effects are essential to explain the behavior of the system [15], [13].

Analysis of the behavior of a spatial stochastic model for different input parameters would benefit from a systematic, observationally driven approach in which statistical approaches from e.g. machine learning and bioinformatics would be applied to the simulated data in order to discover input combinations where the model displays interesting behavior. In its simplest form, such an analysis could consist of aggregation of the full time series data to a set of biologically significant scalar or vector quantities, followed by the application of clustering algorithms to find groups of input cases displaying similar behavior. Such an approach is currently limited by the existing infrastructure, and would benefit greatly from integration with database solutions that simultaneously support knowledge discovery in databases and online selection and post-processing through queries, in our case SciSPARQL queries.

b) The URDME framework: BISTAB is implemented using the URDME framework for stochastic simulation of reaction-diffusion processes on unstructured meshes [12], [14]. It relies on the scientific computing environment MATLAB as a front-end, while the core simulation routines are implemented as stand-alone C programs. Another third party software, Comsol Multiphysics, is used to provide a modeling environment for the geometry and to provide unstructured mesh generation. If used interactively, URDME behaves much as a MATLAB toolbox. It is designed to provide flexibility for the applied users in terms of (biochemical) model design, execution and post-processing via e.g. customized MATLAB scripts. Given a description of the chemical reactions (in the form of C code) and of the geometry (in the form of a Comsol .mph file), the URDME MATLAB layer creates all necessary data structures and serializes the model to an input file in .mat format. URDME then compiles an executable specific to the model under consideration, launches the simulation, and then imports the output data back into the MATLAB interface.

The raw output from a simulation with URDME is a time series, or trajectory, with the number of molecules of each species recorded in every cell in the mesh for each output time point. It thus resembles the output of most partial differential equation (PDE) solvers, such as those based on the finite element method. An important difference from most standard PDE applications is that, since each run provides only one out of many possible realizations of the stochastic process, it is typically necessary to gather many independent trajectories into ensembles to form a basis for statistical analysis. Frequently, some model parameters such as the kinetic rate constants or diffusion constants are undetermined by

biological experiments or known with low precision. It is therefore necessary to conduct "parameter sweeps" in order to tune model parameters to an experimentally observed behavior, or to study the robustness of the model to changes in the input. A computational experiment may thus require the generation of tens or hundreds of thousands trajectories. The computational cost to generate the ensembles is large, but each realization can be simulated independently of the others.

c) Post-processing: The large amount of output data generated by URDME for a typical computational experiment poses a big challenge, both in terms of storage requirements and in terms of infrastructure for post-processing. While output data could be aggregated to e.g. mean values at the time of simulation, a computational experiment will likely require many different post-processing queries, and many of them will not be known in detail at the time of generation of the data. It is hence desirable that raw simulation output be persistent at least for the duration of a modeling project. The earlier solutions were based on either storing simulation output files locally on the user workstation, or transferring them to a central URDME server when they need to be accessed in the computation [21]. In the first case, hardware will likely limit high-throughput analysis of the model, and in the second case, the performance of the system will be limited by the data transfer cost. A more general approach with lazy access to a data repository through queries, as one described in the previous sections, is desired.

d) Model problem: The BISTAB dataset is a model of a bistable system [13], and was one of the first models used to demonstrate the use of spatial stochastic simulation in computational systems biology. For some parameter combinations, the system will be globally bistable, and for other combinations the proteins will self-organize in local areas of higher concentrations, leading to loss of global bistability. The BISTAB dataset consists a parameter sweep of 1900 realizations, where each realization is a file containing the result of a simulation with randomly chosen parameters. Processing this dataset and analyzing the biochemical model's behavior for the different parameter combinations requires both compute intensive post-processing of the time series data and the ability to manage and filter the post-processing results based on metadata such as parameter values.

e) Example queries: To demonstrate the utility of the proposed system we have applied it to run a number of different queries that are representative of the kind of array slicing and aggregate functions that are frequently needed as primitives in more complex post-processing routines. These queries often constitute the data-intensive part of the post-processing workflow, where the complete dataset is mapped to derived quantities of biological interest in a lower dimension. The queries we consider in this paper are:

• Q1: Compute the number of molecules over the whole spatial domain of a certain species as a function of time.

• Q2: Compute the number of certain species at a certain time point for all the realizations that have kinetic rate constants in a certain range.

• Q3: Retrieve the identifier of the trajectory that resulted in the maximal result for Q2.

The operation in Q1 is typical for visualization of the realizations and is for example needed to produce the time series plots in [13], [15]. While simple array slicing and aggregate functions like that done on a single matrix in Q1 can be expressed easily and efficiently in a scripting language such as MATLAB, already simple queries such as Q2 and Q3 will place the responsibility for managing all the many different files and their properties on the user of the URDME application, while SSDM uniformly manages all data and metadata. With the traditional approach the management and analysis of e.g. large parameter sweeps quickly becomes tedious when metadata is stored separately and may be a bottleneck that limits the productivity of the user. We show how the system efficiently combines the utility of a database to select subsets of the data based on the metadata describing the experiments in terms of a high-level declarative language capable of expressing array operations.

VII. Results and Discussion

A. SciSPARQL queries for the BISTAB experiment

We have developed a database schema for the BISTAB dataset, as described by the ER-diagram shown on Fig. 3. It is used both for generating an RDF dataset to be stored in SSDM with a relational database backend, and for defining an RDF view over the scientific database stored in Chelonia.

Once mapped dataset is accessible as the default RDF graph for SciSPARQL queries during the session, i.e. the queries do not need a FROM clause when addressing this dataset. In the BISTAB schema, all values are contained in properties of *Experiment* and *Task* instances. There are 1900 task instances currently stored in Chelonia. To test SQL-based storage, we use a sample of 100 instances.

The performance evaluation uses a relational database backend. We furthermore show the possibility to run the same SciSPARQL queries against a Chelonia native repository for scientific data mapped to RDF.

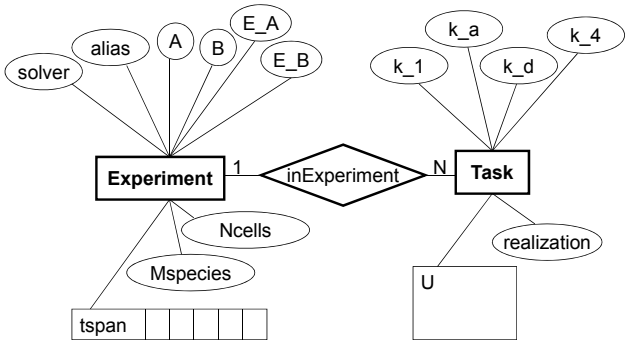

Fig 3. Entity-relationship diagram of BISTAB dataset

The time series being the result of an URDME simulation are stored in the matrix *U*, containing a row per mesh cell per species type, and a column per time point (Fig. 4). The number of cells (*Ncells*), species (*Mspecies*), and the time values for every time point (*tspan* vector) are part of

Experiment metadata. The values of the species names *A*, *B*, *E_A,* and *E_B* are used to access rows corresponding to these species types.

Together with each *U* matrix, a set of simulation parameters *k_1, k_a, k_d,* and *k_4* are stored. Since the simulation is stochastic, several different results per parameter set can be generated, and the *realization* number is used to distinguish between them.

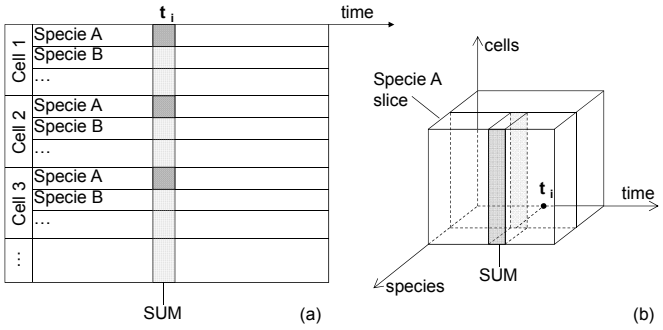

Fig 4. Simulation results stored in U matrix

Query Q1: Compute the sum of all species *A* over all mesh cells in the experiment as a function of time for the trajectory matrix *U* of task *Task1*. Q1 always selects one matrix, associated with *Task1*, and aggregates the information on species of type *A*, effectively accessing 12.5% of the matrix elements in the database. This query is representative of a frequently occurring use case; to reduce the data of an individual sample point to e.g. plot the 3D spatial data as a 1D aggregated time series.

First, we define a function *total_species(U, species, mspecies)* that sums up the given *species* type in *U*, applying a vector sum to the corresponding rows. Every simulation cell occupies *mspecies* rows in *U*:

```
DEFINE FUNCTION total_species(?U ?species ?mspecies)
AS SELECT (SUM(?U[?i]) AS ?res)
    WHERE { FILTER (mod(?i, ?mspecies)
                  = ?species-1) };
```

The variable *?i* will be bound to all possible subscripts in *?U* constrained by the filter expression. Every *mspecies* row is retrieved, and a (scalar) sum is computed over the elements of respective columns, as shown on Fig. 4a. If we think about the same data as a 3D array, with the 'species' and the 'cells' dimensions unnested, each summed-up subset can be represented as a slice, shown on Fig. 4b.

Query Q1 can now be formulated as

```
SELECT (?tspan[?j] AS ?t)
       (total_species(?U,?a,?mspecies)[?j] AS ?sum_A)
 WHERE { :Task1 :U ?U ;
                :inExperiment ?experiment .
         ?experiment :A ?a ;
                     :MSpecies ?mspecies ;
                     :tspan ?tspan };
```

The variable *?j* joins the possible subscript values for elements of the *?tspan* vector with those of a vector returned by *total_species()*. The WHERE clause specifies triple patterns used to extract (1) the *U* matrix associated with the experiment task instance named *Task1,* (2) the corresponding experiment instance (property *inExperiment*), and (3) other metadata associated with the experiment. The query returns pairs of *(timepoint, sum)* that can be directly used for plotting the wanted function of time.

Query Q2: Select the sum of all species *A* for time point 10s for all trajectory matrices *U* with parameters *k_a* and *k_d* in given ranges. Q2 selects just one column of a matrix, and the column index *?j* by looking up the *tspan* vector for the time point of interest. There can be many tasks falling into the specified parameter range.

```
SELECT (array_sum(?U[?a-1::?mspecies,?j]) AS ?res)
 WHERE { ?task :U ?U ;
               :k_a ?k_a ;
               :k_d ?k_d ;
               :inExperiment ?experiment .
         ?experiment :A ?a ;
                     :MSpecies ?mspecies ;
                     :tspan ?tspan .
         FILTER (?tspan[?j] = 10 &&
                 1.0E8 <= ?k_d && ?k_d <= 1.0E9 &&
                 50 <= ?k_a && ?k_a <= 90 ) };
```

This query sums up only one column with index *?j* expressed by the constraint *tspan[j]=10*. The *SELECT* expression sums up elements in one column and every *mspecies* row (grey in Fig 4a). Since we are interested in only one time point of the trajectory, here we do not use a vector sum as we do in function *total_species()*.

Query Q3: Find the task that has the maximal total population of species *A* or *B* for any time point. Q3 is an example of typical batch processing job. It makes a complete (unselective) sweep across all *U* matrices in the dataset, computes aggregated statistics for each matrix, and identifies the task that has received the maximum score.

We will need a helper function *max_AB_sum(task)*, aggregating the vectors returned by different calls to *total_species()*:

```
DEFINE FUNCTION max_AB_sum(?task) AS
SELECT
   (max(array_max(total_species(?U,?a,?mspecies)),
        array_max(total_species(?U,?b,?mspecies)))
   AS ?res)
 WHERE { ?task :U ?U ;
               :inExperiment ?experiment .
         ?experiment :A ?a ;
                     :B ?b ;
                     :MSpecies ?mspecies };
```

For each matrix *U* two vectors are computed: summing up the populations of species *A* and *B* and returning their maximum. The function is similar to Q1 and Q2 in the triple patterns involved, but the *?task* instance is now the function's argument. This allows us to apply the second-order function *ARGMAX()* to express query Q3:

```
SELECT (ARGMAX( max_AB_sum ) AS ?maxtask);
```

This query applies *max_AB_sum()* to all possible bindings of variable *?task* to task instances – computed by the triple patterns inside *max_AB_sum()*. The argument corresponding to the maximal function result is returned, and can be further queried for properties, e.g. parameters of the trajectory.

During the query execution *A* and *B* rows are accessed and summed up separately, and referenced by separate array proxies. Since the chunks contain two rows each, the chunk-level caching prevents double retrieval of the same chunks. A very small cache, capable of storing 25% chunks of a matrix, completely eliminates the problem. The matrices are accessed one at a time, so that the chunk cache is automatically refreshed (according to LRU replacement strategy) when function *max_AB_sum()* proceeds to the next task.

B. Setup and data loading

We have deployed both SSDM and the backend database on a single HP Compaq 8100 workstation with Intel Core i5 CPU @ 2.80 GHz, with 8Gb RAM and running Windows Server 2008 R2 Standard SP1.

The parameters (metadata) of the BISTAB experiment and each simulation were collected into a Turtle file, enhanced with links to binary data files (in standard MATLAB .mat files) containing the experimental results (*U* matrices). We used a dataset consisting of 100 realizations of *U* matrices, each about 71.5 MB, containing an integer element for each of (11107 cells × 8 specie types × 201 time points).

As SQL back-end we experimented with two different DBMSs accessed via JDBC: MySQL 5.6.10 and Microsoft SQL Server 2008 R2. The back-ends are configured to use small chunks of 1608 bytes each, so that a chunk contains two successive rows of a *U* matrix. This amounts to 44 428 chunks per matrix, and 4 442 801 array chunks in total (one chunk stores *tspan*).

To evaluate different data loading methods, we compared the performance of naive one-by-one insertion of each chunk with loading the complete dataset at once using the bulk-loading facility of the DBMS. The results are shown in Table 1. In case of bulk loading, the system first has to prepare a set of bulk-load input files to be sent to the bulk-loader. Here the data to be loaded into each table in our general relational storage schema for RDF (Fig 1) needs one or several prepared input files. If the data to be bulk-loaded into a table is larger than allowed by the OS (8 GB in our setting), the system splits the bulk-load input into several files.

TABLE 1. DATA LOADING TIMES FOR 100 MATRICES

task	MySQL	MS SQL Server
Preparing files for bulk-loader	980 s	82 s
Bulk loading	1 543 s	1 275 s
Total	**2 523 s**	**1 357 s**
Naïve one-by-one insertion	7 577 s	7 827 s

The bulk-loading into MySQL is slower since its bulk-loader requires text-based input. Here the array chunks are represented in hexadecimal form and the preparation work includes converting the binary data into hexadecimal representation. The gain is still a factor three compared with inserting the chunks one a time, mainly because incremental updates of internal DBMS structures in the latter case.

MS SQL Server allows bulk-loading binary files. Preparing these files becomes simply moving binary data from memory. The bulk-loader does not have to do any parsing. This is therefore the fastest option.

C. Query execution

Once the data was loaded into SSDM with its connected relational backend RDF storage, we ran the queries Q1 - Q3 and measured the execution time and the number of resulting tuples emitted. Our experiments on smaller datasets showed that all queries take practically constant processing time per matrix. As a comparison, we also made MATLAB scripts to perform the equivalent computations on a set of (uncompressed) binary .MAT files. Since MATLAB stores the matrix elements as floating-point numbers, the size of data it is reading from disk is in the best case twice bigger than the data we retrieve from a relational database, which explains why MATLAB is sometimes slower.

Table 2 shows "cold cache" run times where all data reside on disk before the query is executed:

TABLE 2. QUERY EXECUTION TIMES (IN SECONDS) WITH COLD CACHE

task	U matrices	results	MySQL	MS SQL Server	MATLAB
Q1	1	201	1.748	2.15	1.826
Q2	36	36	80.703	44.512	30.042
Q3	100	1	187.073	192.365	133.279

We can see that on smaller amounts of data our system slightly outperforms MATLAB with .MAT files. All results fall within same order of magnitude, which proves that the benefits provided by our solution combine with quite competitive performance.

Table 3 shows "warm cache" results, obtained by repeated runs of the same query. There are three cache levels involved: OS-level file cache, DBMS-level query cache, and SSDM-level array chunks cache. Due to massive amounts of data processed, Q3 does not benefit from any of these, and the results are same as in Table 2. In contrast, for Q1 there is an interesting case possible when all the data processed fits entirely into the SSDM cache, so the DBMS is not accessed at all; it only runs in the background, consuming some system resources. This particular case is shown as Q1*.

TABLE 3. QUERY EXECUTION TIMES (IN SECONDS) WITH WARM OS/DBMS LEVEL CACHE

task	U matrices	results	MySQL	MS SQL Server	MATLAB
Q1	1	201	0.434	0.526	0.157
Q1*	1	201	0.138	0.152	*N/A*
Q2	36	36	63.542	13.378	1.203

Here we can see that the SSDM cache is faster than the OS-level cache utilized by MATLAB. However, Q2 reads just a single column from every matrix, but it has to retrieve the same amount of chunks from DBMS. This makes it significantly slower than a system without chunking per row,

which is currently used in the current SSDM. Single-column access can be regarded as particular worst case for row-based array storage, as the useful data load is relatively small for the array retrieval operations.

VIII. Conclusion

In this article we have presented the architecture of SSDM providing the SciSPARQL language front-end to access scientific data stored in a relational database back-end. Two widely-used SQL DBMSs were used as alternative choices for the back-ends. Using the system's extensibility we also used the Chelonia system as an example of interfacing a back-end e-Science data repository. We demonstrated the strengths of our system by applying it to datasets from a highly data-intensive application from systems biology. The SSDM system was shown to provide a scalable storage mechanism with efficient data analysis capabilities to the scientific application. By formulating as SciSPARQL queries typical real world post-processing tasks arising in the use of the URDME framework, we showed that the system is capable of addressing non-trivial online data analysis tasks with competitive performance. The queries presented for the URDME framework illustrate how other applications that need interactive data analysis can access the data in terms of declarative SciSPARQL queries combining metadata and massive numerical data in order to accelerate the application workflow. Our solution is thus adequate to support upcoming data-intensive scientific applications and to build comprehensive systems for managing scientific information.

Future work includes development of distribution and replication strategies for data-intensive scientific and engineering applications. Different kinds of query processing and optimization techniques for such an architecture should be developed. Architectures providing more direct file access, completely transparent to SciSPARQL queries, are another important direction of our research. The discussion of different partitioning strategies in [11] [24] opens up another direction for possible future work.

Acknowledgment

This project is supported by eSSENCE and the Swedish Foundation for Strategic Research under grant RIT08-0041, (U.S.) Department of Energy (DOE) Award No. DE-SC0008975 and NIBIB of the NIH under Award No. R01-EB014877-01. The content of this paper is solely the responsibility of the authors and does not necessarily represent the official views of these agencies.

We acknowledge valuable discussion with Brian Drawert.

References

[1] Advanced Resources Connector. http://www.nordugrid.org/arc/
[2] Chelonia Web page. http://www.nordugrid.org/chelonia/
[3] EU KnowARC project. http://www.knowarc.eu/
[4] NetCDF. http://www.unidata.ucar.edu/software/netcdf/
[5] NorduGrid Collaboration. http://www.nordugrid.org/
[6] PostgreSQL: . http://www.postgresql.org/
[7] SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/
[8] A. Andrejev and T. Risch. Scientific sparql: Semantic web queries over scientific data. The 3rd International Workshop on Data Engineering Meets the Semantic Web (DESWEB), 2012.
[9] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In Proceedings of the 2010 international conference on Management of data, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM.
[10] R. Brun and F. Rademakers. ROOT – An object oriented data analysis framework. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 389(1-2):81–86, April 1997.
[11] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. A. Brandt. SciHadoop: array-based query processing in hadoop. In Scott Lathrop, Jim Costa, and William Kramer, editors, SC, page 66. ACM, 2011.
[12] Brian Drawert, Stefan Engblom, and Andreas Hellander. URDME 1.1: User's manual. Technical Report 003, Department of Information Technology, Division of Scientific Computing, Uppsala University, 2010.
[13] Johan Elf and Mans˚ Ehrenberg. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. Syst. Biol., 1(2):230–236, 2004.
[14] S. Engblom, L. Ferm, A. Hellander, and P. Lotstedt.¨ Simulation of stochastic reaction-diffusion processes on unstructured meshes. SIAM J. Sci. Comput., 31:1774–1797, 2009.
[15] D. Fange and J. Elf. Noise induced Min phenotypes in E. coli. PLoS Comput. Biol., 2(6):e80, 2006.
[16] Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reacting systems. J. Comput. Phys., 22:403–434, 1976.
[17] T. Hey, S. Tansley, and K. Tolle. The fourth paradigm: data-intensive scientific discovery. Microsoft Research, Redmond, WA, 2009.
[18] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. Sciql, a query language for science applications. In Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases, AD '11, pages 1–12, New York, NY, USA, 2011. ACM.
[19] Alberto Lerner and Dennis Shasha. Aquery: query language for ordered data, optimization techniques, and experiments. In Proceedings of the 29th international conference on Very large data bases - Volume 29, VLDB '03, pages 345–356. VLDB Endowment, 2003.
[20] J. K. Nilsen, S. Toor, Zs. Nagy, and A. Read. Chelonia: A self-healing, replicated storage system. Journal of Physics: Conference Series, 331(6):062019, 2011.
[21] P-O Östberg, A. Hellander, B. Drawert, E. Elmroth, S. Holmgren, L. Petzold, Reducing Complexity in Management of eScience Computations, Proceedings of CCGrid 2012 - The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 845-852, 2012
[22] T.Risch, V.Josifovski, and T.Katchaounov. Functional Data Integration in a Distributed Mediator System, Functional Approach to Data Management. Modeling, Analyzing and Integrating Heterogeneous Data, ed. by in P.Gray et.al. Springer, ISBN 3-540-00375-4, 2004.
[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
[24] E. Soroush, M. Balazinska, and D. L. Wang. Arraystore: a storage manager for complex parallel array processing. In SIGMOD Conference, pages 253–264, 2011.
[25] P. S. Swain, M. B. Elowitz, and E. D. Siggia. Intrinsic and extrinsic contributions to stochasticity in gene expression. Proc. Natl. Acad. Sci. USA, 99(20):12795–12800, 2002.
[26] S.Stefanova and T.Risch. Optimizing unbound-property queries to RDF views of Relational databases. The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS), 2011.
[27] S. Toor, M. Sabesan, S. Holmgren, and T. Risch. A scalable architecture for e-Science data management. e-Science, IEEE International Conference on, 0:210–217, 2011.