

Andrej Andrejev

Semantic Web Queries
over Scientific Data



UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in Lecture hall 2446, Polacksbacken, Uppsala, Wednesday, 23 March 2016 at 14:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Gerhard Weikum (Max Planck Institute for Informatics).

Abstract

Andrejev, A. 2016. Semantic Web Queries over Scientific Data. *Uppsala Dissertations from the Faculty of Science and Technology* 121. 214 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-9465-0.

Semantic Web and Linked Open Data provide a potential platform for interoperability of scientific data, offering a flexible model for providing machine-readable and queryable metadata. However, RDF and SPARQL gained limited adoption within the scientific community, mainly due to the lack of support for managing massive numeric data, along with certain other important features – such as extensibility with user-defined functions, query modularity, and integration with existing environments and workflows.

We present the design, implementation and evaluation of *Scientific SPARQL* – a language for querying data and metadata combined, represented using the RDF graph model extended with numeric multidimensional arrays as node values – *RDF with Arrays*. The techniques used to store *RDF with Arrays* in a scalable way and process Scientific SPARQL queries and updates are implemented in our prototype software – *Scientific SPARQL Database Manager, SSDM*, and its integrations with data storage systems and computational frameworks. This includes scalable storage solutions for numeric multidimensional arrays and an efficient implementation of array operations. The arrays can be physically stored in a variety of external storage systems, including files, relational databases, and specialized array data stores, using our *Array Storage Extensibility Interface*. Whenever possible SSDM accumulates array operations and accesses array contents in a lazy fashion.

In scientific applications numeric computations are often used for filtering or post-processing the retrieved data, which can be expressed in a functional way. Scientific SPARQL allows expressing common query sub-tasks with functions defined as parameterized queries. This becomes especially useful along with functional language abstractions such as *lexical closures* and *second-order functions*, e.g. array mappers.

Existing computational libraries can be interfaced and invoked from Scientific SPARQL queries as *foreign functions*. Cost estimates and alternative evaluation directions may be specified, aiding the construction of better execution plans. Costly array processing, e.g. filtering and aggregation, is thus preformed on the server, saving the amount of communication. Furthermore, common supported operations are delegated to the array storage back-ends, according to their capabilities. Both expressivity and performance of Scientific SPARQL are evaluated on a real-world example, and further performance tests are run using our mini-benchmark for array queries.

Keywords: RDF, SPARQL, Arrays, Query optimization, Second-order functions, Scientific workflows

Andrej Andrejev, Department of Information Technology, Division of Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Andrej Andrejev 2016

ISSN 1104-2516

ISBN 978-91-554-9465-0

urn:nbn:se:uu:diva-274856 (<http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-274856>)

Contents

1	Introduction	9
2	Background and Related Work.....	14
2.1	Semantic Web	14
2.2	RDF Repositories	15
2.2.1	SPARQL endpoints and Linked Data.....	16
2.2.2	SPARQL extensions	16
2.2.3	Storing RDF graphs	17
2.3	Exposing Non-RDF Data as RDF	18
2.3.1	Relational data to RDF	18
2.3.2	Objects to RDF	19
2.3.3	XML to RDF	20
2.3.4	Spreadsheets to RDF.....	21
2.3.5	Multidimensional data in RDF	22
2.4	Array Models.....	24
2.5	Array Databases	25
2.6	The Amos II System.....	27
3	SPARQL Language Overview	29
3.1	Example Dataset.....	29
3.1.1	Turtle Syntax	30
3.2	Graph Patterns	31
3.3	Combining the Graph Patterns	32
3.3.1	Optional Graph Patterns	33
3.3.2	Matching Alternatives	33
3.3.3	Existence Quantifiers and Other Filters.....	35
3.3.4	Addressing Multiple Graphs.....	35
3.4	Property Path Expressions.....	36
3.4.1	Precedence of Path Operators	37
3.4.2	Algebraic Properties of Path Operators	37
3.5	Aggregation and Grouping.....	38
3.6	Error Handling.....	39
3.7	Ordering and Segmentation.....	40
3.8	Constructing New RDF Graphs	41
3.9	Updating the Datasets.....	42

4	Scientific SPARQL.....	43
4.1	Array Queries.....	44
4.1.1	Array Dereference Syntax.....	45
4.1.2	Variables Bound to Array Subscripts.....	47
4.1.3	Built-in Array Functions.....	48
4.1.4	Array Arithmetic.....	48
4.1.5	Intra-array Computations.....	50
4.1.6	Array Equality.....	50
4.2	Parameterized Queries - Functional Views.....	51
4.3	Lexical Closures and Second-Order Functions.....	53
4.3.1	Array Algebra Second-order Functions.....	54
4.4	Foreign Functions.....	55
4.5	Calling SciSPARQL from Algorithmic Languages.....	57
5	Scientific SPARQL Database Manager.....	59
5.1	Architecture overview.....	60
5.1.1	Example Dataset.....	61
5.1.2	Example Query.....	63
5.2	Numeric Multidimensional Arrays.....	69
5.2.1	Storage of Resident Arrays.....	69
5.2.2	Array Transformations.....	71
5.3	Data Loaders.....	74
5.3.1	File Links.....	74
5.3.2	RDF Collections.....	75
5.3.3	Data Cube Vocabulary.....	76
5.4	Scientific SPARQL Query Processor.....	79
5.4.1	SciSPARQL Query Structure.....	80
5.4.2	Compositional vs. Operational SPARQL Semantics.....	86
5.4.3	AmosQL Query Structure.....	95
5.4.4	Extensions to ObjectLog and Physical Algebra.....	101
5.4.5	The Translation Algorithm.....	105
5.5	Polymorphic Properties Problem.....	127
5.5.1	Directionality Problem.....	127
5.5.2	Normalization Problem.....	128
6	External Storage of RDF with Arrays.....	130
6.1	Array Storage Extensibility Interface.....	131
6.1.1	Placing APR Calls into the Translation.....	133
6.1.2	APR Implementations.....	135
6.1.3	Problems and Solutions.....	136
6.2	Relational Back-end.....	138
6.2.1	Storage Schema.....	139
6.2.2	The Problem of Retrieving Array Content.....	142

6.2.3	Strategies for Formulating SQL Queries during APR	144
6.2.4	Resolving Bags of Array Proxies	145
6.2.5	Sequence Pattern Detector (SPD) Algorithm	154
6.3	Comparing the Storage and Retrieval Strategies.....	156
6.3.1	Query Generator	158
6.3.2	Experiment 1: Comparing the Retrieval Strategies	158
6.3.3	Experiment 2: Varying the Buffer Size	171
6.3.4	Experiment 3: Varying the Chunk Size	172
6.3.5	Summary of the Comparison Experiments.....	175
6.4	Real-Life Query Performance Evaluations	176
6.4.1	BISTAB: an Application from Computational Biology	177
6.4.2	BISTAB Data Model as <i>RDF with Arrays</i>	180
6.4.3	Experiment Setup and Data Loading	182
6.4.4	BISTAB Application Queries.....	183
6.4.5	Query Performance	186
7	Integration of SciSPARQL into Matlab	188
7.1	Usage Scenario	188
7.2	A Workflow Example	190
7.3	Matlab Interface to SSDM	192
7.4	Discussion	194
8	Summary and Future Work	195
	Summary in Swedish	198
	Acknowledgement	200
	References.....	201
	Glossary	212

Abbreviations

AAPR	aggregate array-proxy-resolve function
APR	array-proxy-resolve function
ASEI	Array Storage Extensibility Interface
API	Application Programming Interface
DBMS	Database Management System
DNF	Disjunctive Normal Form
ER-diagram	Entity-Relationship diagram
HDF	Hierarchical Data Format
JDBC	Java Database Connectivity
MCR	Matlab Common Runtime
RDBMS	Relational DBMS
RDB-to-RDF	Relational Database to RDF
RDF	Resource Description Framework
SciSPARQL	Scientific SPARQL
SIMD	Single Instruction, Multiple Data
SLR(1) parser	Simple Left-to-right reversed Rightmost derivation parser with single look-ahead
SPARQL	SPARQL Protocol And RDF Query Language (recursive acronym)
SSDM	Scientific SPARQL Database Manager
TCP	Transmission Control Protocol
TLA function	Top-Level Aggregate function
UDF	User-Defined Function
URI	Universal Resource Identifier
W3C	World Wide Web Consortium

1 Introduction

The amount of scientific and engineering data has grown exponentially in recent decades [163], and this growth includes a rapid increase in the amount of data sources publicly available on the web [76, 165]. Complexity and diversity (structural, terminological, etc.) of this data is also expected to rise steadily in the coming decades, as novel data models emerge along with new and unforeseen applications. The efforts directed towards data integration and interoperability are becoming of vital importance [22, 67, 112].

One promising direction of these efforts is the search for a *lingua franca* - a model general and flexible enough, so that the other, more specific data models can be mapped into it in a lossless way; and yet being meaningful and easy to understand and query. Semantic Web [23] and Linked Open Data [29] are conceived as a potential solution [79]: all kinds of data and metadata can be represented as a graph with nodes and (classes of) edges identified by globally unique URIs. The original aim of this data model was to describe the resources available on the web - hence the name: Resource Description Framework (RDF) [129].

For querying RDF datasets the graph-based pattern-matching query language SPARQL [155] was proposed and recommended by W3C. In its current state, SPARQL 1.1 allows queries that retrieve data from an RDF graph, filter the potential query solutions, and postprocess them before emitting the results. SPARQL bridges the gap between the traditionally separated *data* and *metadata*, the latter being the semantic, structural, statistical, and other kinds of descriptions of the former. A potential to fully combine *data* and *metadata* search and conditions in one query, thus simplifying the process and eliminating extra round-trips to the remote data sources, is contained within the Semantic Web paradigm but is not fully realized.

The main problem is that although most kinds of other data models can be mapped to RDF (as shown in Section 2.3), the efficiency and usefulness of such mappings might become unsatisfactory. For example, *numeric multidimensional arrays*, a data abstraction that is central in all natural sciences and constitutes the main bulk of accumulated data, when mapped to RDF have to be transformed into graphs, thus making even the simplest

array operations (e.g. *element access*) unfeasible to perform or even express in a general case.

So far RDF and SPARQL gained limited adoption within the scientific community, due to the lack of array support [102] and other important features – such as extensibility with user-defined functions, query modularity, integration with existing environments and workflows. Some users turn towards the 'more mature' relational database technology (e.g. [164], eventually extending it with missing array functionality [41, 49, 119, 125], while others find the idea of relational schema design too restrictive, resorting to specialized file formats (e.g. NetCDF [111]) or hierarchical databases (e.g. ROOT [36]). In either case, array data is separated from metadata and the latter sometimes ends up encoded into eventually very complex file names, so that data retrieval and processing become a nontrivial task for a programmer. While many complications arise from the need of manual data/metadata re-integration, another challenging task is the adequate estimation of data quantities and distributions, in order to come up with an optimal order of data retrieval operations.

Automating the task of programming the data retrieval and processing is the essence of *query optimization*. Relational database management systems (RDBMSs) were taking care of data statistics and evaluation cost models, in order to produce optimal execution plans since 1970s [148, 39]. The modern RDF stores [50, 65, 98, 112, 113, 168, 183] employ similar techniques based on indexing, query rewriting and materialized views in order to address the challenges of web-scale query processing [1, 66, 73, 88, 94, 126, 134, 144].

Addressing different data and metadata sources in a single query is possible within a data integration framework where machine-readable descriptions of the structure and semantics of the available data are present. RDF is specifically designed for publishing such descriptions by creating and referring to vocabularies of globally-scoped terms, and by defining the logical relationships within and across such vocabularies, using the RDF Schema [33] and OWL [19] formalisms.

The main research questions addressed in this Thesis are:

1. How can RDF and SPARQL be extended to be suitable for scientific and engineering numeric data representation and analysis tasks, in particular, those which combine data and metadata?
2. How can extended SPARQL query processing be implemented on the basis of a database management system? In particular:
 - a. What extensions to the underlying query processing and algebra operators are needed for efficient processing of SPARQL queries?

- b. How can existing state-of-art data persistence approaches (RDBMSs, specialized file formats, array databases) be utilized for scalable storage and querying of RDF data with arrays?
- c. How can query functionality of extended SPARQL be integrated into existing environments and workflows for scientific and engineering data analysis?
- d. How do we measure the impact of data storage decisions and retrieval strategies on the overall query performance?

In few words, the aim of this work is providing a viable solution (both conceptual and technical) opening the benefits of the Semantic Web approach to scientific data management, and making scientific data available and interoperable on the Semantic Web.

To answer Research Question 1, the RDF data model has been extended, so that numeric multidimensional arrays of arbitrary shape and dimensionality (including those exceeding the main memory limit) can be attached as *values* in *subject-property-value* RDF triples. We call this model *RDF with Arrays*, and it is backwards-compatible with the basic RDF model: arrays that are recognized within the imported RDF graphs are *consolidated*, i.e. their elements are co-located and the array shape is determined. Internal array storage facilities are used in that case, and such structured data becomes available to the queries using array-oriented features. In order to query *RDF with Arrays* collections, the W3C SPARQL language has been extended with array syntax and semantics, as well as other useful features, including user defined functions (UDFs), parameterized views, second-order functions, and lexical closures. We will refer to a SciSPARQL query containing array operations as an *array query*. Chapter 4 introduces the Scientific SPARQL (SciSPARQL) language and provides usage examples.

To answer Research Questions 2 we developed the publicly available and ready-to-use *Scientific SPARQL Database Manager, SSDM* [6]. It is an extensible main-memory DBMS built to process the SciSPARQL queries. SSDM loads and stores *RDF with Arrays* datasets and processes SciSPARQL queries over the stored data. It utilizes object-relational query optimization techniques, extensibility, and inter-process communication of the underlying main-memory DBMS Amos II [136], and, being a major system extension, introduces some novel features at all levels, including:

- physical representations of arrays and other RDF terms, together with their serializations,
- new execution algebra operators, to reflect distinctive SPARQL semantics,
- lazy data retrieval based on *array proxy* objects,

- a library of array-specific operations, and extensions to existing (scalar) arithmetic, designed to support array computations.

Chapter 5 presents the SSDM architecture. Regardless of the architectural choices, SSDM can be utilized as a stand-alone system, a client-server system, or a cluster of processes based on peer-to-peer communication.

To answer Research Question 2a, Chapter 5 describes the process of answering SciSPARQL queries including a complete definition of the translation of SciSPARQL queries into the domain calculus based query language of Amos, specialized query normalization and rewriting techniques, cost-based optimization, and extensions to the execution algebra with a library of array operators for executing SciSPARQL queries.

To answer Research Question 2b, Chapter 6 presents two approaches for how SSDM can be extended to store and query metadata and massive numeric array data by utilizing external data managers:

- utilizing back-end systems for the storage of array data loaded (e.g. binary file formats or SQL-compliant RDBMSs), by deploying an SSDM-managed relational storage schema or other external storage management - the *back-end scenario*, or
- linking arrays that are already stored in external storage systems into user-specified RDF graphs managed by SSDM – the *mediator scenario*.

To answer Research Question 2c, Chapter 7 presents a client-server integration of a SciSPARQL client into the scientific computing environment Matlab, thus providing tight integration of SciSPARQL queries into scientific workflows [7]. It is shown how handy SciSPARQL queries can be for Matlab users, especially in a collaborative environment. Furthermore, Semantic Web styled metadata can be used for annotation and, eventually, search for the numeric computation results, while essentially preserving the traditional workflows.

To answer Research Question 2d Section 6.3 presents a mini-benchmark featuring some typical array access patterns, including the best and worst cases for each storage choice. An extensive experimental evaluation of the array query performance of SSDM was performed, both benchmark-based and application-driven, [6]. The evaluation furthermore sets the context for our ongoing integration [8] with the Rasdaman array database [16].

The following papers were published in the course of this work:

- ***Scientific SPARQL: Semantic Web Queries over Scientific Data*** [5] introduces the query language, array data model, and in-memory implementation of array operations.
- ***Scientific Analysis by Queries in Extended SPARQL over a Scalable e-Science Data Store*** [6] puts SciSPARQL in the context of a real-

world scientific computing application. In order to accommodate for massive numeric data involved, storage extensibility mechanisms and lazy array data retrieval are introduced.

- ***Scientific Data as RDF with Arrays: Tight Integration of SciSPARQL Queries into Matlab*** [7] presents the integration of SciSPARQL queries and updates, facilitating the Semantic Web way of handling metadata about scientific experiments into Matlab and typical computational workflows, demonstrating the benefits and the low cost of adoption of our approach.
- ***Spatio-Temporal Gridded Data Processing on the Semantic Web*** [8] positions Scientific SPARQL as a next unification step in handling geographic and other kinds of gridded coverage data on the web. As an example of a hybrid data store approach suggested, it features SSDM as a SciSPARQL front-end, and the Rasdaman [16] system for scalable storage of massive gridded datasets.

The author of this Thesis is the main contributing author in all research papers listed above.

The outline of this Thesis is as follows: Chapter 2 gives an extensive overview of the background and related work, including Semantic Web, data integration approaches, other SPARQL extensions, and array databases. Chapter 3 introduces the SPARQL query language in detail, encompassing most of its features and can thus be regarded as an extended background, crucial for understanding Scientific SPARQL features and usage, which are described in Chapter 4. Chapter 5 describes the architecture and SciSPARQL query processing in general, and Chapter 6 focuses on providing the storage for array data, and presents performance evaluations. The integration of SciSPARQL queries into the Matlab environment is presented in Chapter 7. Finally, Chapter 8 summarizes the contributions of this work, and points out directions for further development.

2 Background and Related Work

2.1 Semantic Web

The Semantic Web initiative, first proposed in 2001 [23], promotes utilizing a graph data model (Resource Description Framework - RDF) for describing all kinds of resources on the web. Graph-oriented query languages (e.g. SPARQL 1.1 [155]) were designed for querying RDF graphs. The main intention is to provide a structured, yet easily extensible way of expressing the complex metadata in the evolving application contexts.

Universal Resource Identifiers (URI, or IRI if Unicode is used) are employed to identify classes, instances, and relationships in the RDF data model. The term 'universal' means that every publishing party is able to define their own manageable identifiers within their own namespace, which thus become globally unique. Generalizing the *Universal Resource Locators* (URL), which may look similar, URIs may or may not be dereferenceable on the web. Dereferenceable URIs point to RDF documents containing additional information about the identified resource.

Higher-order specifications of object-oriented data models, including class hierarchies - *ontologies* [31, 81, 117] are typically expressed with RDF Schema [33] vocabulary, featuring standard terms for inheritance, domain, and range specifications. Interactive visual tools (e.g. Protégé [67]) help in the development and presentation of such models, with the resulting metadata becoming an extension of the RDF graph it describes.

Further modeling, including disjointness, cardinality, and symmetry can be expressed with *Web Ontology Language* - OWL [19]. Knowledge inference and reasoning rules can be codified with RIF [92] / SWRL [78] on top of such data and metadata, opening way to classical symbolic AI approaches: making the human-oriented knowledge structured and available to computers for further processing.

All this information, including resource description data, schemata, and inference rules is normally merged into an RDF graph. The graph query language (and communication protocol) SPARQL is designed to query RDF graphs by formulating *graph patterns* and additional constraints as queries.

The result of a query is a set of bindings of query variables that reference values from the RDF graph in case of a SELECT query, or a new RDF graph in case of a CONSTRUCT query. Chapter 3 below provides an extensive introduction to SPARQL queries and updates.

Semantic Web has gained a lot of traction in recent years, as efficient RDF Stores and SPARQL query processors became available [4, 34, 37, 55, 112, 113, 115, 158, 183]. According to [68], already by 2013 more than four million Web domains contained RDF markup. Wide adoption of common vocabularies like DublinCore [51], FOAF [32], schema.org brings hope for automating data integration tasks (also reasoning, decision support, etc) at a new level.

Within the Scientific SPARQL project, we follow the Semantic Web approach for storing and querying metadata as a most promising solution, already earning attention by different communities in science e.g. [10, 87, 140, 150, 170] and engineering e.g. [30, 103], as well as in more interdisciplinary contexts e.g. [69]. We promote using the Semantic Web descriptions of experiments, parameter cases, data provenance etc. in order for the experimental data to become interoperable across different sources.

2.2 RDF Repositories

An RDF Repository is a DBMS capable of storing and querying RDF graphs. Querying is typically done with a graph query language. SPARQL is the most common option, though its predecessors (e.g. RQL [90], TRIPLE [152], Versa [174]) and alternatives native to a particular RDF Repository, e.g. SeRQL for Sesame [34] are supported by some systems. The diversity of RDF query languages in pre-SPARQL era led to emergence of layered mediation frameworks, e.g. Datalog-based EDUTELLA [110]. Certain graph databases are not officially RDF repositories, but allow SPARQL mappings along with a native graph language, e.g. Cypher [77] for Neo4J [173]. There is also an ongoing project to integrate the essential SPARQL-like syntax and semantics into a superset of SQL [157].

A number of file formats, or *serializations* are defined to facilitate easy interchange and storage of RDF data outside the repositories. RDF/XML [130], Turtle [21] / Ntriples [20], and Notation3 [24] are the most widely used ones, along with embeddings of RDF information into the HTML documents, e.g. with RDFa [131]. Throughout this work we will use Turtle notation for our RDF examples.

2.2.1 SPARQL endpoints and Linked Data

Most RDF Repositories offer a *SPARQL Endpoint* - a web service answering SPARQL queries using a SPARQL communication protocol to encode the queries and results being transmitted. Thus, SPARQL became lingua franca in the decentralized Linked Data [29] environment, where, basically, everyone is free to publish their part of the global RDF graph, and RDF terms represented by URIs are dereferenced to obtain additional information. Figure 1 shows a fragment of the Linked Data cloud diagram, listing some representative RDF datasets publicly available. One of the major connectivity hubs is DBpedia [11], the RDF-encoded fact tables from Wikipedia articles.

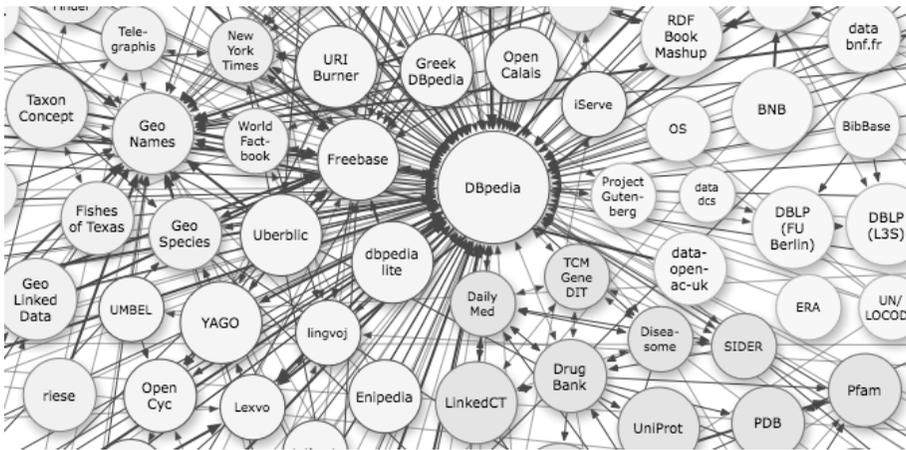


Figure 1. Linked Data Cloud Diagram (fragment)¹

2.2.2 SPARQL extensions

Application-specific extensions of SPARQL also exist, e.g. GeoSPARQL [15] for GIS applications was standardized by W3C. More general extensions include SPARQL Update [156], previously known as SPARUL, stream-processing C-SPARQL for continuous queries [14], A-SPARQL for archival [160], and many others. Presented in this Thesis Scientific SPARQL can be seen as another big extension, being a strict superset of W3C SPARQL 1.1 and adding substantial amount of new functionality, effectively extending the conceptual power of SPARQL beyond the traditional metadata queries.

We will be referring to our RDF Repository implementing SciSPARQL queries as *Scientific SPARQL Database Manager*, or *SSDM* for short.

¹ original at en.wikipedia.org/wiki/File:LOD_Cloud_Diagram_as_of_September_2011.png

Besides SciSPARQL, it is able to process the underlying systems native functional query language AmosQL [136]. A number of APIs, including C, Java, Python, and Lisp are available, making the system easy to extend or embed. Chapter 7 presents such an embedding of SciSPARQL into Matlab.

2.2.3 Storing RDF graphs

Storage-wise, RDF Repositories use one or more of the following approaches: in-memory, native RDF store / graph store, or built on top of either relational or NoSQL DBMSs.

In-memory storage is perhaps the most viable solution for most RDF applications up to the present day, since RDF is typically used to represent metadata and/or formalized knowledge, and the sizes of RDF graphs are still small enough to fit in main memory, especially when normalized properly. Other main-memory databases, like Starcounter [157] and SAP HANA [141] offer graph models. A memory snapshot can typically be dumped to disk and loaded back to memory in order to survive the server restarts. SSDM uses this approach, when not connected to a back-end storage for *RDF with Arays*.

Native RDF stores provide persistence mechanisms to store larger amounts of RDF triples on disk, including purposely-built indexing infrastructures. There is a wide spectrum of approaches presented: some systems (like RDF-3X [112]) store heavily-indexed normalized RDF triples, some (like Neo4J [173], though not officially an RDF store, but providing the RDF/SPARQL layer on top) store large graph structures with pointers. Many closed-source projects, including NitrosBase [115], AllegroGraph [4], and Stardog [158] also fall into this category.

RDBMS-based storage of RDF, for example Jena [84], Virtuoso [54], Ultrawrap [154], Ontop [137] rely on an underlying Relational DBMS to locate the data being queried, and to perform all the joins. They utilize the indexing and execution plan optimization capabilities of the underlying RDBMS. The relational schema used to store RDF is subject to further classification [139]: (a) single table, (b) partitioning by value type (c) partitioning by predicate, (d) partitioning by correlating predicates, or (e) wrapping from any arbitrary relational schema (typically read-only). SSDM supports options (b) and (e), as described in Chapter 6, with the RDB-to-RDF view definitions based on the SWARD [124] framework.

A correct SPARQL-to-SQL translation plays a central role for RDBMS-based RDF Repositories. There is an ongoing discussion [46, 121, 122, 40] within the Semantic Web community about the potential semantic mismatches between different approaches to translation in general. We revisit this problem in Section 5.4.2, even though we translate SciSPARQL

queries to our functional AmosQL language, where they can be further translated [182] to SQL queries or other API calls to different storage back-end.

NoSQL DBMS-based storage, utilizing the emerging 'not-only-SQL' databases (e.g. HBase [74] column store, Couchbase [43] document store), utilize data model flexibility of the underlying DBMS, while usually having to perform joins and other database operations externally. Cudré-Mauroux et. al. [45] offer a comprehensive overview of the current approaches, along with performance comparisons of RDF/SPARQL layers over these (generally, distributed) database systems. The conclusion is that column-store based RDF stores may outperform native RDF stores on simple SPARQL queries, the functional minimalism of the underlying DBMS results in lesser freedom for SPARQL query optimization, thus loosing the race on more complex queries. Still, we expect that NoSQL database APIs will become richer in the future, and are looking forward to interfacing such NoSQL databases as storage back-ends for SSDM. Some preliminary integration and performance tests are already presented in [101].

2.3 Exposing Non-RDF Data as RDF

2.3.1 Relational data to RDF

Creating RDF views reflecting relational data (and schemata) was a research issue from the early days of RDF adoption [124, 159], since the relational databases are by far the most prevalent source of structured data. Relaxing this structure, and mapping application-scoped relational table semantics to globally-unique RDF terms (typically defined by standard vocabularies/ontologies) is obviously a step towards greater data integration and query interoperability across disparate data sources.

Another reason why RDF models on top of relational storage have emerged so early was the substantial overhead in processing arbitrary RDF data in form of triples (before the native RDF Stores matured, and the computational power grew sufficient) due to the following reasons:

- a typical SPARQL query, when viewed as referring to a single subject-property-value table, contains a lot more join operations than a similar query to an equivalent relational model;
- cardinality of such a table of triples is also substantially bigger than the total cardinalities of tables in the corresponding relational schema, making the physical access paths longer;
- statistical information about distributions of different properties and values needs to be maintained in a novel way (e.g. RDF-3X indexes

also act as histograms [112, 113]), making old relational-style query optimization approaches blind and inefficient.

The Relational-to-RDF mapping approach offered a solution, since it is practically always possible to translate a SPARQL query back to SQL queries against the underlying relational databases. This way, the conceptual flexibility of RDF and SPARQL was combined with efficiency of the relational storage and query processing solutions, as long as the data originated from the relational databases anyway. This solution, however, is not simple [122], and there have been recent advances [182] on further optimizing the SQL query generation when translating SPARQL.

Practically, there have been a number of mappings defined. The current W3C standard recommendations include Direct Mapping of Relational Data to RDF [9], which automatically generates URIs to define tables (as node classes) and rows (as instances), but does not allow specifying custom URIs and does not map schema information. The first shortcoming is addressed by RDB to RDF Mapping Language recommendation [127]. Schema mapping is proposed in the Semantic Archival of Relational Data project [160, 161], and constraint mapping, which is potentially helpful to native SPARQL query optimization, is proposed in [97]

As a minimum, any Relational-to-RDF mapping is going to have the following components, for a given relational schema:

- a mapping of table names to RDF classes
- a mapping of attributes to RDF properties
- a mapping of primary key values in each table to RDF node instances
- for tables with no primary keys defined, a mapping of their rows to RDF *blank nodes*
- a mapping of foreign keys to RDF properties

Additional schema and constraints information can also be provided in the mapping. The software solutions implementing Relational-to-RDF mappings include D2RQ [47], SWARD [124], SARD [160], Virtuoso [55], Ultrawrap [149], Ontop [137] and others. SSDM is built on the same platform as SWARD / SARD, and thus can access mediated relational databases. However, this benefit concerns basic RDF models, and thus is orthogonal to the extensions introduced by SciSPARQL.

2.3.2 Objects to RDF

As a graph data model RDF supports object-oriented data modeling: relationships like class/instance, inheritance, declared properties, domain and range specifications, are available within RDFS and OWL frameworks. When viewed in terms of object-oriented programming, the model is multiple-inheritance, with static and dynamic properties, and extensible on-

the-fly - this allows stricter models to easily fit in. Additionally and alternatively, *RDF Literal* values, being comprised of type URI and string-serialized value, can also be seen as 'stringified' representations of arbitrary objects whose class is known.

There are object-oriented DBMS around, designed to provide persistence to objects exactly as they are defined in the programming languages, including ObjectStore [95], and many others. Some DBMS provide object-oriented APIs for the developers, along with other data models - e.g. Starcounter [157] and SAP HANA Open ODS Views [141].

An Object-to-RDF mapping may also be provided for classes of objects in a programming language, like C++ or Java. In fact, it is so straightforward that with the RDFBeans framework [132] it takes just a simple annotation to the classes and properties, for example

```
@RDFBean("http://xmlns.com/foaf/0.1/Person")
public class Person
{ ...

    @RDF("http://xmlns.com/foaf/0.1/name")
    public String getName()
    { ...
}}
```

Results in all instances of *Person* class to be accessible as RDF via the provided RDF Store API.

Another approach is when an Object (or Object-Relational) RDBMS exposes a SPARQL query interface for its objects, like Starcounter [157] does, effectively making it an RDF Store at the same time. In this case, details like RDF namespaces for classes and properties need to be provided to the DBMS.

As SSDM is built on top of the Amos II mediator architecture [136], that supports objects natively and implements interfaces to object databases, including ROOT [89], it is relatively easy to expose these mediated object models as RDF - one just needs to provide RDF namespaces for classes and properties.

2.3.3 XML to RDF

Mapping semi-structured data (like XML documents) to RDF requires certain conventions, but is nonetheless important, given that XML is a widely adopted information interchange format across a wide spectrum of disparate applications. XML Schema plays an important role in the process of formulating the mapping rules. The overview [25] presents the state of art in the field, and suggests the SPARQL2XQuery framework, further

elaborated in [26, 27]. There is, however, no publicly available software implementation of the mapping technique.

Another project, named XSPARQL [28, 176] extended by Ali et.al. [3] simply combines the essential parts SPARQL and XQuery syntax in one language, making it possible to natively query both RDF and XML. Both works are centered around translating SPARQL to XQuery expressions, including update functionality. Creation of metadata-rich, well-annotated XML documents available for semantic querying is certainly an important research direction for the Semantic Web adaptation, especially in business and industrial application.

2.3.4 Spreadsheets to RDF

While the general 'spreadsheet' paradigm assumes a 2D space of enumerated rows and columns (as traditionally seen in Lotus 1-2-3 and MS Excel), where each cell is an interactive model-view-controller element, it can also be treated as data alone, making no difference between the stored and derived values. Some specialized data stores can be easily adapted to this spreadsheet view, and some are built with this model in mind - for example the Chelonia [114, 166] data store developed for e-Science applications within the NorduGrid [116] project.

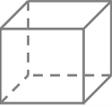
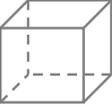
var task id	k_1	k_a	k_d	k_4	realization	result
1	32.159	79.279	782750669.857	53.286	1	
2	19.151	39.044	300035857.676	73.445	1	

Figure 2. An example dataset (BISTAB experiment (see Section 6.4.4) stored in Chelonia, with cubes denoting numeric array data stored as values

Chelonia organizes the dataset orthogonally into enumerated *tasks* and named *variables*, and stores *instances* of named variables, at most one per task (which might be regarded a row in an MS Excel workbook). An instance can hold a numeric value, a string, or a numeric array of arbitrary size, independently of other instances. Figure 2 shows an example of dataset stored in Chelonia. When expressed with an Entity-Relationship diagram (Figure 3) it turns out to be quite simple: an experiment can be seen as a

group of tasks, while tasks and variables comprise the 2D space of a (possibly sparse) spreadsheet.

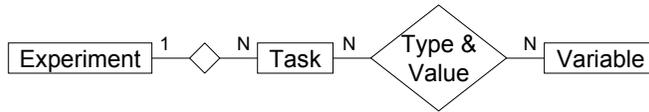


Figure 3. Chelonia storage schema

Within the scope of the SSDM project we have experimented with integrations of e-Science tools into the SciSPARQL environment. Reflecting Chelonia data, including experiments, tasks, variables, types and values of their instances with an RDF view proved to be conceptually straightforward, as explained in [6]. In short, every instance was represented by a single RDF triple, with *subject* derived from task number, and *property* derived from variable name. Since both Chelonia and SciSPARQL support numeric arrays as values, this array data was mapped without changes.

In general any spreadsheet data, for example MS Excel workbooks can be (with certain manual guidance) mapped to RDF in a similar way, with e.g. rows becoming *subjects* and columns becoming *properties* in RDF triples. More complex mappings, with a certain degree of programmability, are available in the RDF123 [71] and XLWrap [96] projects. This opens yet another horizon to the generality of the Semantic Web approach in querying disparate data in diverse models and formats. Additionally, spreadsheets are often used to contain numeric arrays, thus providing an extra motivation for using *RDF with Arrays* model, queryable with SciSPARQL.

2.3.5 Multidimensional data in RDF

There are several approaches to treating multidimensional data as RDF that have been adopted by the Semantic Web community. The simplest one is nested RDF collections. A more elaborate framework, designed for representing statistical data (e.g. OLAP Data Cubes [66]) is called RDF Data Cube [133].

2.3.5.1 Collections

Ordered collections of RDF terms are normally incorporated into an RDF graph as linked lists using `rdf:first` and `rdf:next` as relationships and `rdf:nil` as a terminating node - similarly to linked lists in e.g. Lisp. Such ordered connections can be nested and used to represent, among other things, multidimensional arrays of numbers.

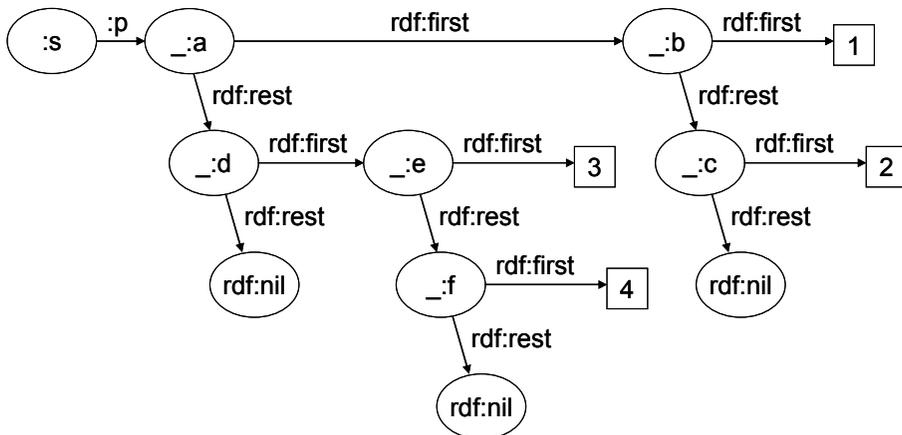


Figure 4. A graph with RDF collection representing a 2x2 matrix

Since any array should be integrated into the RDF graph (otherwise there is no way to navigate to it), it will be stored as a value of at least one other RDF triple (`:s :p _:a` in our example). Some RDF serialization formats provide a condensed syntax for expressing RDF collections. For example, the dataset from Figure 4 can be expressed by a single Turtle statement:

```
:s :p ((1 2) (3 4)) .
```

This, however, does not decrease the complexity of the RDF graph - the same 13 triples would need to be generated and made available to SPARQL queries. In order to navigate to an array element, a SPARQL query needs to use chains of `rdf:first` and `rdf:next` properties. A query addressing element `[2,1]` in the above example (value 3), can be expressed in SPARQL as

```
SELECT ?element21
WHERE {
  :s :p ?array .
  ?array rdf:rest ?x .
  ?x rdf:first ?slice2 .
  ?slice2 rdf:first ?element21 }
```

In general, a query addressing an element `[x,y]` in a 2D array will contain a property path of `(x+y)` triple patterns, and `(x+y-1)` additional variables.

Apart from inefficiency arising from this 'too general' graph-based storage and processing of arrays, this representation also fails to give important guarantees about the data structure. For example, different leaf elements in the collections might be of different types, including numeric, string, and user-typed literals, URIs and blank nodes. The nested array slices might not match in their shape, and referring to array slices by the intermediate blank nodes (like `_:b` or `_:e`) between the queries is not officially allowed, since such blank nodes might change whenever two RDF datasets are combined.

As SciSPARQL extends the RDF data model with arrays, the graph representation of nested RDF collections becomes much more compact. While importing RDF into SSDM, such collections are recognized and stored internally as numeric arrays, as described in Section 5.3.2

2.3.5.2 RDF Data Cube Vocabulary

RDF Data Cube [133] was developed as a Semantic Web adaptation of SDMX (Statistical Data and Metadata eXchange) [147], the ISO standard for exchanging and sharing statistical data and metadata among organizations. RDF Data Cube builds upon a set of other vocabularies, including SKOS [154] for statistical concepts, VoiD [175] for data access specifications, and Dublin Core [51] for publication-related information.

SSDM interprets the RDF Data Cube semantics, consolidating the numeric multidimensional array data and thus drastically reducing the graph size of a Data Cube dataset, while preserving all information therein, as described in Section 5.3.3. Another important benefit is speeding up pattern-matching queries, as they have to deal with much smaller RDF graph.

2.4 Array Models

Since the emergence of APL [82], we have seen a wide spectrum of array data models, along with the algebras of array operators. Baumann & Holsten [18] give a comprehensive theoretical comparison of four representative models: including AQL [99], AML [104], Array Algebra [17], and RAM [12, 13, 42].

The array model used in SciSPARQL is similar Array Algebra used in Rasdaman [16], though it is a bit more narrow by design. In Rasdaman each array dimension is defined with lo_k and hi_k integer bounds, and the range is defined as a record of named and typed fields. SciSPARQL presents a simple particular case of Rasdaman arrays, however, the numeric Rasdaman arrays can be mapped losslessly to the SciSPARQL array model by providing an additional vector of lo_k values. Arrays of records of numeric types can be represented by collections of aligned arrays in SciSPARQL.

As for the more general array data models, i.e. ones with non-integer dimensions, or with non-numeric ranges, those can be modeled by creating dictionaries (one-dimensional vectors of arbitrary values) for each dimension/range. This is exactly the approach used to represent Data Cube datasets with numeric multidimensional arrays in SSSDM, as described in Section 5.3.3.

Regarding the array operators, recent developments of SciSPARQL [8] introduce the second-order functions, central to Array Algebra [17], directly as SciSPARQL language primitives.

2.5 Array Databases

Historically, there have been three kinds of approaches to handle arrays in the database context.

(1) Databases, normalizing arrays in terms of their main data model, representing each array element as one or several records. SciQL [91], along with its predecessor RAM [12, 13, 42] treat each array as a relational table, where columns are divided into dimension and non-dimension attributes, and SQL is extended to provide array operations in addition to the native relational operations, e.g. selection and join over arrays. Similar normalization technique is used under-the-hood in certain UDF-based array integrations into the relational DBMSs, including [119] and [41]. Data Cube Vocabulary [133] suggests a way to represent multidimensional statistical data in terms of an RDF graph, which can be handled by any RDF store.

While allowing to keep the original set of semantic primitives in queries and updates, and making all existing DBMS features (query optimizer, access paths, consistency control, etc.) work for arrays as well, this approach has important downsides, both in storage and access overheads, and sometimes in flexibility: every array in SciQL needs to have a name (as a relational table), and a numbered set of arrays can only be modeled as an extra dimension. Otherwise, insertion of an array instance effectively involves schema modification, as noted by Misev & Baumann [107]. Furthermore, iteration across a set of arrays becomes obviously problematic.

(2) Databases, incorporating arrays as a value type. This includes PostgreSQL [125], recent development of ASQL [108] on top of Rasdaman [16] system, and the extensions to MS SQL Server based on BLOBs and UDFs [49]. In the context of relational databases, this is regarded as the 'array-as-attribute' approach following the classification in [107].

There are also semi-declarative high-level dataflow programming languages centered around array processing, e.g. DSL [118], and Array-QL [64], both finding their origins in Single Assignment C [143] - a functional programming language supporting array operations. A similar functional approach was implemented earlier in Amos II system, specifying matrix expressions at a high level, while the implementations are automatically matched to the matrix subclasses [120].

SciSPARQL follows the 'array-as-attribute' paradigm beyond the relational world, bringing numeric multidimensional arrays as values into the RDF data model. It integrates the Semantic Web [23] flexibility in metadata management (including ontologies, knowledge inference, adding new properties 'on-the-fly', and querying based on these 'optional' properties) with efficient array storage and processing, so that array data and metadata search can be combined in the same query.

(3) Dedicated array-only databases, offering only specialized array query languages, (e.g. SciDB [35, 44] and the core Rasdaman system [16]). A number of earlier developments, including AQL [99], AML [104], RIOT [179, 180], and ArrayStore [154] also fall into this category. This would also include lightweight queryable database layers on top of popular array file formats, with SAGA [172] being the most recent example, inspired by NoDB approach [2] that does not require a data loading step.

The main problem with this approach is inherited from the underlying concept of array data formats: everything is arrays. For example, scientific users miss an infrastructure for storing and querying the descriptions of experiments, including parameters, terminology mappings, provenance records and other kinds of metadata. At best, this information is stored in a set of variables in the same files that contain large numeric arrays of experimental data, and thus is prone to duplication and is hard to update. Query (or dataflow programming) languages are designed as another abstraction layer on top of array file APIs, and thus are array-centered. In contrast, SciSPARQL is a superset of the standard W3C SPARQL 1.1 query language and its array semantics does not limit the underlying graph-based query semantics.

Storing the arrays in files has its benefits for performance and eliminating the need for data ingestion, as shown by comparison of SAGA to SciDB [35, 44]. SciSPARQL incorporates this option, as presented in the context of its tight integration into Matlab [7]. In that case, SSDM maintains a main-memory RDF database, and the massive array data is stored in native .mat files. Both data and metadata are queryable, array proxies refer to files but otherwise work exactly as main-memory array descriptors described in Section 5.2. Chunking and caching, however, is done entirely by the OS / file system. Still, in the present technological context we believe that utilizing state-of-the-art relational DBMS to store massive array data promises better scalability, thanks to cluster and cloud deployment of these solutions, and mature partitioning and query parallelization techniques.

In summary, SciSPARQL extends RDF with arrays as values, allows users to query and update the arrays together with RDF metadata (as shown on a real-world application in [6]), and stores the arrays either in specialized file formats, similarly to SAGA [172], or in BLOBs stored by RDBMS,

similarly to [49], but not relying on DBMS-side UDFs. SSDM is implemented based Amos II DBMS [136], making use of its flexible extensibility mechanisms.

One important difference from e.g. Rasdaman [16] is that we use a simpler partitioning approach for arrays. Instead of specifying dimension-aligned 'tiles', whose shape and overlap should be tuned for particular array processing tasks [60, 105], we split the arrays into one-dimensional chunks, so that the chunk size is the only parameter and its auto-tuning heuristics are simple. Instead of designing tiles to increase the chances of array access patterns becoming predictably regular, we instead discover that regularity at query runtime.

As SAGA system evaluation [172] has shown, even in the absence of SQL-based back-end integration, the sequential access to chunks provides a substantial performance boost over random access.

2.6 The Amos II System

Amos II [136] is an functional main-memory DBMS, employing its own functional and declarative domain calculus query language, AmosQL. Stored functions in AmosQL correspond to tables in the relational data model, and derived functions serve as parameterized views, effectively making the query structure modular. The system is easily extensible with foreign functions, implemented in algorithmic programming languages (currently supported C, Java, Python, and Lisp), and such foreign functions can be invertible and specify a cost and cardinality estimates for the optimizer.

Furthermore, AmosQL has aggregate functions, nested subqueries, disjunctive queries, quantifiers and second-order functions, and is relationally complete. The queries operate on atomic values, vectors, tuples, records, and bags (i.e. multisets), implementing the DAPLEX [151] semantics, which governs the evaluation of bag-valued functions. Inner (and other kinds of) joins, Cartesian products, and compositions of bag-valued functions are defined.

Internally, Amos II uses an extension of Datalog [169], called ObjectLog [100], to represent the structure of a query as a logical expression of stored and foreign predicates. Predicate flattening, normalization, and rewrite rules are applied. The ObjectLog representation of a query is translated into object algebra [86] by the cost-based optimizer. The cost-based optimizer reorders the predicates in each conjunction, minimizing the total cost of execution, according to the cost model provided. This process is shown by example in 5.1.2, where a SciSPARQL query is translated to AmosQL in the first step.

There are many features in Amos II making it an advanced object-oriented DMBS and a research vehicle, including late binding [57], active rules [145], distributed data stream processing [178, 177], extensible indexing [167], complex query optimization [59], and more. One characteristic trait relevant to SciSPARQL usage is the mediator architecture [136] of Amos II.

Federated queries are split into parts which can be delegated to the underlying data sources, taken account for their generic capabilities like joins, arithmetic operations, aggregates etc. The process is quite flexible, and any remaining predicates can always be executed by the mediator. This includes the process of query translation, and has allowed addressing e.g. both complete-functionality SQL [72], and limited-functionality SQL, offered by Google BigTable [181]. Also, the mediator architecture has enabled Amos II to wrap High Energy Physics datasets in the hierarchical ROOT [36] database format, and successfully optimize scientific queries searching for certain kinds of collision events [59] - the task which was traditionally solved by making ad-hoc algorithmic implementation of each query.

The last example has demonstrated how beneficial it is to use declarative queries to specify the database search criteria in a form of mathematical expressions: equations and inequalities. The DBMS is generally well-equipped to come up with a fairly optimal execution plan, making use of the available cost model and statistics. With SciSPARQL we make a step further, offering a superset of the standard and well-accepted query language SPARQL, already well-suited for data integration, and designed to operate in the context of Linked Open Data [29] - an internet-scale federation of RDF data sources. Another step further w.r.t. both AmosQL and SPARQL is the array functionality, addressing the needs of scientific and engineering data processing.

As a matter of related work, Datalog-based predicate calculus has been widely used for decades, and still maintains a good reputation. As pointed by J.Hellerstein [75], the Datalog extensions have the potential and elegance in addressing such challenging tasks as parallelization and asynchronous communication, apart from being well-suited for expressing recursion (as we show in 5.4.5.3) and implementing query decomposition. Besides, Datalog has been the basis for AI approaches to knowledge inference in database - so-called *deductive databases* [128] - a concept similar to OWL entailment and RIF/SWRL reasoning in the Semantic Web.

3 SPARQL Language Overview

Scientific SPARQL query language [5] is a superset of W3C SPARQL 1.1 standard [155], and is designed to query *RDF with Arrays* datasets. The semantics of SciSPARQL is thus focused both on graph pattern matching, defined by the SPARQL standard, and on array processing introduced in our extension

The purpose of this section is to introduce the essential features of SPARQL, as specified by the W3C Standard [155], including different kinds of graph patterns (basic, optional, alternative), property path expressions, filters, grouping and aggregation. This part should be regarded as an extended background, crucial for understanding the contributions of this work.

The next chapter continues this overview by discussing the extensions introduced in SciSPARQL, including array expressions, parameterized views, lexical closures, and second-order functions [8], together make an noticeable shift towards a functional query language, albeit retaining the property of declarativeness.

Neither part can be regarded as a substitute for the complete documentation on the query language. SciSPARQL User Manual is available on the project homepage [146], and W3C SPARQL 1.1 Specification [155] can also be recommended as a tutorial for the standard language.

3.1 Example Dataset

An RDF graph consists of nodes and edges. Edges are always identified by URIs, while nodes can be either URIs (globally unique), blank nodes (unique within a graph or union of graphs to be queried), or literals: numbers, text strings, temporal or logical values.

Figure 5 shows an example of an RDF graph using the FOAF [32] vocabulary. There is one class node for `foaf:Person`, four instance nodes for that class identified by blank nodes, and a `foaf:name` property for each of them. Additionally they participate in the `foaf:knows` relationships, which

happen to be symmetric - double-sided arrows indicate pairs of symmetric properties.

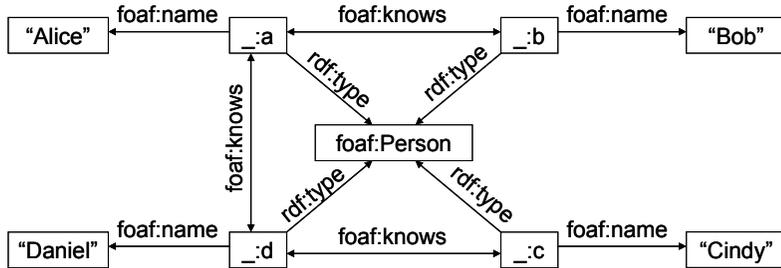


Figure 5. Example of RDF graph using FOAF vocabulary

At the same time, an RDF graph is also a set of $(subject, property, value)^2$ triples. Subject and value of each triple correspond to nodes in the graph, while properties correspond to edges.

3.1.1 Turtle Syntax

There is a number of ways to serialize RDF graphs to text. The RDF graph in Figure 5 can be expressed as a set of triples, e.g.

```

_:a a foaf:Person ;
    foaf:name "Alice" ;
    foaf:knows _:b , _:d .
_:b foaf:knows _:a .
...
  
```

Throughout this Thesis we will use *Turtle* [21] - Terse RDF Triple Language to present the RDF datasets. The fully specified triples are separated by dot '.', while triples sharing the same *subject* are separated by semicolon ';', and triples sharing both *subject* and *property* are separated by comma ',', and we usually place them in the same line. So the above fragment contains five triples, with two unique *subjects* and four unique *subject-property* pairs. The same syntax is used for specifying *triple patterns* in SPARQL, as shown in Section 3.2.

Generally, the dot sign separating the triples in RDF and SPARQL has the semantics of a conjunction (along with comma and semicolon). So what technically appears to be a set of triples, from the epistemological perspective is a conjunction of facts.

Both Turtle and SPARQL use prefixes in order to abbreviate URIs. The Turtle file with the dataset on Figure 5 would contain a prefix definition

² Another common way to refer to triple components is $(subject, predicate, object)$. We prefer to avoid the confusion with ObjectLog *predicates*.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

It specifies that e.g. `foaf:name` property is a shorthand for the URI `<http://xmlns.com/foaf/0.1/name>`. The reserved property `a` stands for `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`, otherwise commonly abbreviated as `rdf:type`. It indicates the relationship between instances and classes when both are represented by RDF nodes.

Blank nodes, e.g. `_:a` are used whenever no URI is provided to identify the node, and different blank node labels specify different nodes. Blank nodes are typically used to represent instances identified by the values of their key properties (as `foaf:Person` instances are identified by `foaf:name` values in our example). Another common use case are linked lists, formed with `rdf:first` and `rdf:rest` properties. Turtle has a compact syntax to represent such lists, e.g the following Turtle construct:

```
:s :p ((1 2) (3 4)) .
```

It encodes the graph shown on Figure 4 in Section 2.3.5.1, with six new blank nodes generated by the Turtle reader, along with 12 additional triples.

3.2 Graph Patterns

At the core of all non-trivial SPARQL queries there is at least one *graph pattern*, for example

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person
WHERE { ?person foaf:name "Alice" }
```

contains a graph pattern



This graph pattern consists of a single triple pattern, with the variable `?person` used as a wildcard to match a graph node. The result of such a query would be the set of bindings for the projected variable `?person`. If applied to the dataset on Figure 5, this would result in a single blank node `_:a`.

A graph pattern may be more complex and include a conjunction of several triple patterns, connected with the `'.'` operator. Whenever the triple patterns have the same *subject*, `'.'` is substituted with `','` for a more compact syntax³:

³ ... and whenever the triple patterns have the same *subject* and *property*, comma sign `','` is used to connect them - similarly to the Turtle syntax explained in Section 3.1.1.

```

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?friend_name
  WHERE { ?person foaf:name "Alice" ;
           foaf:knows ?friend .
           ?friend foaf:name ?friend_name }

```

Here we need to distinguish between the *query results*, which contain the binding only for the projected variable `?friend_name`, and the *solutions*, which contain the bindings for all variables in the `WHERE` block. Given the dataset on Figure 5, the solutions would consist of:

<code>?person</code>	<code>?friend</code>	<code>?friend_name</code>
<code>_:a</code>	<code>_:b</code>	"Bob"
<code>_:a</code>	<code>_:d</code>	"Daniel"

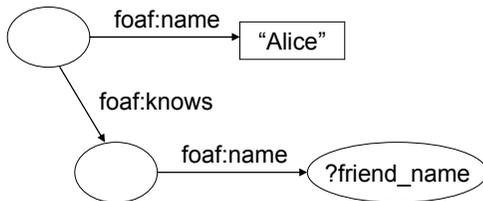
In cases when variables are used only once to connect the triple patterns, the common practice with SPARQL is to use the unlabelled blank nodes `[]` as a substitute. When a variable (like `?friend`) is used to connect a *value* of one triple pattern to a *subject* of another triple pattern, the property and value of the latter can be put inside these square brackets. With both of these reductions applied, the last query would be written as:

```

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?friend_name
  WHERE { [] foaf:name "Alice" ;
           foaf:knows [ foaf:name ?friend_name ] }

```

Here blank nodes are substituting some of the variables in the graph pattern:



3.3 Combining the Graph Patterns

SPARQL is designed to produce deterministic results in the cases of incomplete, redundant, and even conflicting data, which might be published by the independent parties, with little or no common guidelines besides the use of the RDF data model per se. In order to address these challenges, a SPARQL query may include optional or alternative graph patterns, existence and non-existence quantifiers, and explicitly match different graph patterns to the particular sources.

3.3.1 Optional Graph Patterns

Consider that the RDF graph in Figure 5 would feature additional `foaf:mbox` properties for some of the `foaf:Person` instances. The following query will return the emails of Alice friends, if they are available, and return their names in any case:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?friend_name ?friend_email
  WHERE { ?person foaf:name "Alice" ;
           foaf:knows ?friend .
           ?friend foaf:name ?friend_name .
           OPTIONAL { ?friend foaf:mbox ?friend_email } }
```

The nested `OPTIONAL` graph pattern is thus a source of unbound values in both query solutions and the results of the query:

<code>?friend_name</code>	<code>?friend_email</code>
"Bob"	mailto:bob@example.org
"Daniel"	

Being largely similar to the relational algebra *left outer join* \bowtie operator applied to the sets of solutions, the `OPTIONAL` keyword in SPARQL introduces certain issues with declarativeness, as discussed in Section 5.4.2. In short, there are cases where moving around two `OPTIONAL` graph patterns may result in a non-equivalent query.

3.3.2 Matching Alternatives

Assume some of the emails in the graph are listed using the FOAF standard `foaf:mbox` property, while others use a domain-specific property `<http://example.org/email>`. There are two ways to address this inconsistency. The general Semantic Web approach would use an OWL [19] equivalence statement `owl:sameAs`, so that all SPARQL queries, with *OWL entailment* enabled, would treat these two properties as equivalent. While establishing equivalence between the terms used in different datasets is one of the main tools for the data integration in the context of Semantic Web, the objectivity of the identity relation itself might be limited to some but not all possible contexts, leading to the so-called *Identity Crisis* [70].

One might instead prefer to treat a set of properties as equivalent just for the purpose of a specific SPARQL query, without manipulating the datasets and affecting the results of other queries. This would be one of the use cases for the alternative graph patterns, combined with `UNION`, as in the query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/>
SELECT ?friend_name ?friend_email
  WHERE { ?person foaf:name "Alice" ;
```

```

        foaf:knows ?friend .
    ?friend foaf:name ?friend_name .
    { ?friend foaf:mbox ?friend_email }
    UNION
    { ?friend ex:email ?friend_email } }

```

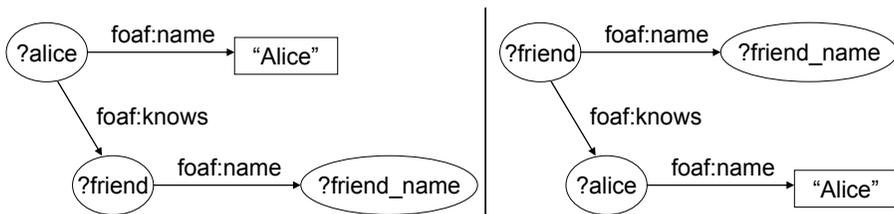
Arbitrary graph patterns can be used as alternatives. For the purpose of another example, consider that the `foaf:knows` relationship is not restricted to be symmetric in the dataset, so we would like to trace it in either direction. The following query returns the names of all people who know Alice and all people whom Alice knows:

```

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?friend_name
WHERE { ?friend foaf:name ?friend_name .
        ?alice foaf:name "Alice" .
        { ?alice foaf:knows ?friend }
        UNION
        { ?friend foaf:knows ?alice } }

```

This query will effectively express two alternative graph patterns:



However, if the `foaf:knows` relationship happens to be mutual in some case, the same bindings will be generated twice for `?friend` and `?friend_name`. To avoid this, and return every person at most once, one would use `DISTINCT` option on the `?friend` variable in the `SELECT` clause:

```

SELECT DISTINCT ?friend ?friend_name

```

Different branches of the same union might provide bindings for the different variables. For example, the following query might return a more informative result, while generating some unbound values as well:

```

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name_Alice_knows ?name_knows_Alice
WHERE { ?alice foaf:name "Alice" .
        { ?alice foaf:knows [ foaf:name ?friend_name] }
        UNION
        { [] foaf:knows ?alice ;
          foaf:name ?friendOf_name } }

```

3.3.3 Existence Quantifiers and Other Filters

The presence of at least a single solution to a graph pattern, or the absence of such, can be turned into a Boolean value using the existence quantifiers. For example, the following query checks for the persons who have foaf:homepage property but no foaf:mbox property:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name_Alice_knows ?name_knows_Alice
WHERE { ?p rdf:type foaf:Person .
        FILTER ( EXISTS { ?p foaf:homepage [] } &&
                NOT EXISTS { ?p foaf:mbox [] } ) }
```

The FILTER conditions in SPARQL queries may appear in a conjunction with graph patterns. They may contain any kind of logical expression, using the logical '&&' (conjunction), '||' (disjunction), and '!' (negation) operators. Besides the quantifiers used in these examples, a large variety of arithmetic and string expressions [155] can be used as terms in the filter conditions. If a filter expression evaluates to anything else than a Boolean value, the *Effective Boolean Value* of the expression is used. The values equivalent to true are non-zero numbers, non-empty strings and typed RDF literals, all possible date/time values and URIs.

The general expression syntax of SPARQL is fairly standard, and hence is omitted in this introduction. However, the exhaustive list of all possible expression constructs in SciSPARQL is presented in Section 5.4.5.4, for the purpose of defining their translation to AmosQL and ObjectLog.

3.3.4 Addressing Multiple Graphs

The queries presented so far did not explicitly identify the dataset they address - in this case, they were accessing the *default graph* of the SPARQL endpoint they are sent to. In the Semantic Web context, a multitude of graphs is typically combined for the purpose of querying. An explicit set of graphs to be combined can thus be specified in the FROM clause of a SPARQL query. Another option is to treat these graphs separately, addressing the specific graph patterns to each of them.

W3C Specifications [155] suggest the following example (presented here with minor simplifications):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?who ?g ?mbox
FROM NAMED <http://example.org/alice>
FROM NAMED <http://example.org/bob>
WHERE { ?who foaf:made ?g
        GRAPH ?g { ?x foaf:mbox ?mbox } }
```

This query retrieves the `foaf:mbox` information from either of the named source graphs, and returns it along with the source graph identifier and the publisher. Here, the graph pattern querying for the `foaf:mbox` property is matched against every available graph, which is listed in the default graph as a value in a `foaf:made` triple.

3.4 Property Path Expressions

A powerful feature introduced in the W3C SPARQL 1.1 standard are regular path expressions as another kind of graph patterns, making it easy to specify chains of properties, alternative and reversed properties. For example, the first two queries in Section 3.3.2 can be reformulated using patterns like

```
?friend foaf:mbox|ex:email ?friend_email
```

and

```
?alice foaf:knows|^foaf:knows ?friend
```

respectively, where the `|` operator denotes the *alternatives* and `^` specifies the *reversed* property.

Still, the main power of the regular path expressions is the ability to query for graph nodes connected by chains of properties of arbitrary length but with certain repeating structure. For example, the following query would list the names of people who are listed as Alice's friend, friend-of-a-friend (that's what FOAF vocabulary name actually stands for), and so on:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?friend_name
WHERE { [] foaf:name "Alice" ;
          foaf:knows+/foaf:name ?friend_name }
```

Here the `+` operator denotes the *transitive closure* of the `foaf:knows` property, and `/` denotes the *chaining* of property paths. If the `*` operator were used instead of `+`, the reflexive-transitive closure would include "Alice" among the results.

The transitive and reflexive-transitive closures are implemented as graph traversal algorithms, which internally check for equivalence of the nodes, and terminate at the point where no new nodes can be reached.

3.4.1 Precedence of Path Operators

Path operators can be freely combined in a path expression. According to W3C SPARQL 1.1 Specifications [155] the precedence order of the path operators⁴ is the following:

- *transitive '+'*, *reflexive '?'*, and *reflexive-transitive closure '*'*
- *reversal '^'*
- *chaining '/'*
- *alternative paths '|'*

Whenever a different precedence is desired, parentheses can be used to control associativity. For example, a graph pattern

```
?x (ex:motherOf|ex:fatherOf)+/foaf:name "Alice"
```

would bind ?x to all ancestors of a person named Alice.

3.4.2 Algebraic Properties of Path Operators

Even though the W3C Standard [155] does not list the properties of path operators explicitly, they are trivial to deduce, and are invaluable if one would like to transform the regular path expressions within their class of equivalence, for the purpose of simplification or normalization. The SPARQL users, formulating queries with path expressions, might also benefit from the structured summary presented in this section.

In the following triangular table (Table 1) we summarize the equivalent expressions that arise when one or two path operators are combined. Given A, B, and C are path fragments, the identities listed in the table cells always hold.

Table 1. Algebraic properties of path operators

	+	*	?	^	/	
+	$A++ = A+$	$(A+)* = A*$ $(A*)+ = A*$	$(A?) + = A*$	$^A+ = (^A)+$	-	-
*		$A** = A*$	$(A?)* = A*$ $(A*)? = A*$	$^A* = (^A)*$	-	-
?			$A?? = A?$	$^A? = (^A)*$	-	-
^				$^{^A} = A$	$^(A/B) = ^B/^A$	$^(A B) = ^A ^B$
/					$A/(B/C) = (A/B)/C$	$(A B)/C = A/C B/C$ $A/(B C) = A/B A/C$
						$A B = B A$ $A (B C) = (A B) C$

⁴ We do not include the *negated property set* operator in the current version of SciSPARQL, due to the problems with its standard definition, explored in [88]. Though not theoretically ambiguous, together with *reversal* it introduces certain counter-intuitive 'butterfly effect' in the set of query solutions.

In mathematical terms, Table 1 lists the following properties:

- **idempotence** of *closure* operators '+', '*', and '?',
- **subsumption** of *transitive* '+' and *reflexive* '?' *closures* into the *reflexive-transitive closure* '*' - the latter can also be constructed by applying *transitive closure* '+' on top of the *reflexive closure* '?' (but not the other way around),
- **involution** property of the *reversal* operator '^',
- **commutative** property of the *alternative* operator '|',
- **self-distributiveness** and **mutual distributiveness** of *chaining* '/' and *alternative* '|' operators,
- **distributiveness** of the *reversal* operator '^' with respect to *closures* and the *alternative* '|' operator, and
- **reversal** of the *chains* of path fragments with the *reversal* operator '^'.

The more formal definition of the regular path expressions, together with their translation to AmosQL and eventually ObjectLog, are given in Section 5.4.5.3.

3.5 Aggregation and Grouping

The `SELECT` part of a SPARQL query may contain a list of projected variables (as seen in all the queries presented so far), or named expressions. A variety of functions, including arithmetic and string manipulation, are available [155], and, in the case of SciSPARQL, easily extensible, as we show in Section 4.4. For example a query with the `SELECT` statement

```
SELECT (round(?x) AS ?result) ...
```

would return the rounded value for each `?x` binding among the query solutions, i.e. the `round()` function will be applied independently every time the query is about to emit.

There are, however, certain SPARQL functions which operate on bags (multisets) of bindings - the *aggregate functions*. Most of them, like `SUM()`, `AVG()` etc. operate only on numerical values, whereas `COUNT()` operates on all kinds of values. For example, the following query would return minimum and maximum age of persons listed in the graph:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT (MIN(?age) AS ?min_age) (MAX(?age) AS ?max_age)
WHERE { ?p rdf:type foaf:Person ;
         foaf:age ?age }
```

emitting a single result (or none if no persons or their age information is found).

If one would need to compute e.g. the average age of each persons friends, this would require grouping the query solutions by person, and applying the aggregate `AVG()` function within each group. This is achieved with the `GROUP BY` clause:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name (AVG(?friend_age) AS ?avg_friend_age)
WHERE { ?p rdf:type foaf:Person ;
         foaf:name ?name ;
         foaf:knows ?friend .
         ?friend foaf:age ?friend_age }
GROUP BY ?p ?name
```

Note that we need to group by the variable `?p`, bound to `foaf:Person` instance nodes, not by the person name (which might not be unique). Listing `?name` as an additional grouping variable might seem redundant, as `?name` is *fully functionally dependent* on `?p`, i.e. we do not expect different `?name` values for the same person. Unfortunately, SPARQL requires that every variable projected out from the aggregate query (or used for post-filtering or ordering) should be also listed in `GROUP BY` clause. In SciSPARQL we lift this restriction, implicitly adding such variables to the effective `GROUP BY` clause.

Additional post-filter conditions can make use of the aggregate values computed. For example, adding

```
HAVING (?avg_friend_age <= 30 && COUNT(?friend) > 3)
```

to the end of the last query would restrict the resulting groups of solutions by size and average age. Note that this adds another aggregate value to be computed for each group.

3.6 Error Handling

It is worth noting that in SPARQL every valid query is always evaluated without raising any exceptions. This is achieved by two separate mechanisms:

- I.** The validity of the query can be determined at *compile time* - a process separate from actually executing the query on a given dataset. A SPARQL query processor emits a wide range of error conditions at different phases of validating the query. The *lexical* and *syntactic* errors, corresponding e.g. to an unmatched quotation mark or an unexpected keyword, indicate that the query cannot be reconstructed from a given textual representation. Next, a range of semantic checks is performed - a *semantic* error can be raised e.g. if aggregate function calls are nested. Finally, the query is transformed to an execution plan (Section 5.1.2 illustrates how this is done in our system),

making sure that every variable gets a finite multiset of potential bindings. If this is found impossible, the query will be reported as *non-executable*.

II. A valid query may still produce errors, when applied to a certain dataset. Division by zero, or a non-numeric operand passed to an arithmetic operator (since SPARQL is *dynamically typed*) produce a special *error* value, which is passed further through the expressions. Query solutions containing an *error* value for a variable never produce a result. Hence, evaluating a `FILTER` expression to *error* is equivalent to evaluating it to *false*. A `SELECT` expression evaluating to *error* effectively discards the solution. This includes aggregate functions evaluating once per group.

For example, if a group of solutions contains a non-numeric binding for a variable under `SUM()`, the aggregate function would return *error*, and the group will not be part of query result. In our system, returning *error* value from a function is in all ways equivalent to returning no values at all. Saying that a function *does not return* in a certain case should be understood as returning *error* value in the standard SPARQL terms.

3.7 Ordering and Segmentation

By default, the result of a SPARQL query is a multiset of bindings for the query output variables. It is, however, possible to return these bindings in a certain order, by using the `ORDER BY` clause.

The following query would list the persons in the dataset sorted by age (in descending order) and, in the case of coevals, by name (alphabetically):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?age
WHERE { ?p rdf:type foaf:Person ;
          foaf:name ?name
          foaf:age ?age }
ORDER BY DESC(?age) ?name
```

Once the order of the results is defined, it becomes possible to retrieve certain portions of results. For example, adding

```
LIMIT 3
```

to the end of the query would make it return the information about the three oldest people (thus probably saving considerably on communication), and adding instead

```
OFFSET 500 LIMIT 100
```

would be typical for a query retrieving the portions of results on demand.

Since the SPARQL standard specifies that the comparison '<' and '>' operators are defined only on the values of the same type, the order of results where an ordering variable is bound to values of the different (incomparable) types is not defined, and hence the segmentation cannot be used in the reliable way. SciSPARQL addresses this problem by defining a certain order among the values of all possible types in *RDF with Arrays*, including URIs, blank nodes, all kinds of literals and arrays.

3.8 Constructing New RDF Graphs

As mentioned before, the result of a `SELECT` query in SPARQL is a list of mappings of its output variables to values (which might include *unbound* values). Sometimes, it is instead desirable to produce a set of triples, which can be regarded as a derived RDF graph. For this purpose, `CONSTRUCT` queries are available in the language⁵.

The following query would construct a derived graph, listing `ex:mutualFriend` properties for all pairs of persons connected with `foaf:knows` relationship both ways:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/>
CONSTRUCT { ?x ex:mutualFriend ?y }
  WHERE { ?x rdf:type foaf:Person ;
           foaf:knows ?y ;
           ?y rdf:type foaf:Person ;
           foaf:knows ?x }
```

The `CONSTRUCT` clause contains a graph construction pattern. For every solution of the `WHERE` block, the corresponding triples will be constructed and emitted. Note that since the graph pattern in the `WHERE` clause is symmetric there will be two solutions for each matching pair of persons.

The solutions with *unbound* variables will not produce triples in those *construction patterns* where these variables are used. We show how the `CONSTRUCT` statements are handled by the SciSPARQL query processor, by defining their translation in Section 5.4.5.11.

⁵ The W3C SPARQL standard [155] also specifies `ASK` queries, which are the shorthand of using `EXISTS` quantifier, and mentions `DESCRIBE` queries, not actually defined in the standard.

3.9 Updating the Datasets

The separate W3C Standard Recommendation [156] governs the SPARQL Update language.

The *Data Definition Language* is limited to creating (with the `CREATE` statement) and dropping (with the `DROP` statement) the named RDF graphs, since, in contrast to the relational data model, there are no schemas to be defined separately from the data.

The *Data Manipulation Language* is mainly represented by the `DELETE/INSERT` statement. For example, instead of deriving a new RDF graph (as in Section 3.8), one could insert the new triples into the same graph, by simply changing the `CONSTRUCT` keyword to `INSERT`.

Deleting triples is as simple - the following statement would delete all personal emails from the graph:

```
PREFIX ex: <http://example.org/>
DELETE { ?p foaf:mbox ?email }
WHERE { ?p rdf:type foaf:Person ;
         foaf:mbox ?email ;
```

For every solution of the `WHERE` block (i.e. for every combination of `?p` and corresponding `?email` values), this statement will delete all triples according to the *deletion triple pattern*. In principle, this would be possible to do with some of the pattern variables free, but SPARQL (and the current implementation of SciSPARQL) requires that all delete pattern variables should be bound. It is part of the future work on SciSPARQL to lift this unnecessary restriction.

The `DELETE` and `INSERT` clauses can be combined in a single statement, sharing the `WHERE` block (e.g. for replacing certain properties according to a pattern). Deletion and insertion patterns may include a named `GRAPH` specifier, similarly to the syntax shown in Section 3.3.4, or a named graph addressed by the whole statement can be specified using the `WITH` keyword. A different graph can be used in the `WHERE` block, introduced with the `USING` keyword instead. Section 5.4.5.12 details the translation of `DELETE/INSERT` statements to procedural AmosQL statements, containing declarative translation of the `WHERE` block.

A different mechanism is used for evaluating simple `INSERT DATA` and `DELETE DATA` statements: they do not contain a `WHERE` block, hence their patterns are free from variables and are purely constant. Their purpose is the massive insertion or deletion of RDF triples in a streamed fashion. They are evaluated at parse time, and thus can be arbitrarily long.

4 Scientific SPARQL

The main purpose of Scientific SPARQL is to enable data processing tasks common in science and engineering to be expressed as queries in extended SPARQL. These tasks are generally characterized by extensive computations, and also by large amounts of numeric data, typically ordered along a number of orthogonal axes [102]. Such data can be represented as *numeric multidimensional arrays*, which become a class of RDF terms in our extended *RDF with Arrays* data model.

Computations are used either for filtering or post-processing the retrieved data, and may typically be expressed in a functional way. Existing computational libraries (many of which became de-facto standards in scientific computing, and are often referred for reproducibility of results) can be interfaced and invoked from the query language as *foreign functions*. Cost estimates and alternative directions of evaluation can be additionally specified (see Section 4.4), in order to aid the construction of better execution plans - the process illustrated in the beginning of Chapter 5.

Though real-life scientific computing tasks, as we have shown in [6], find much more compact formulations in SciSPARQL than in high-level algorithmic languages like Matlab (mainly thanks to declarativeness and more natural metadata management), we expect complex tasks to be formulated as complex queries. Good query modularity becomes as important for scalability as good data design and annotation. In this respect, SciSPARQL allows expressing common query sub-tasks as *functional views*, i.e. SciSPARQL functions defined as parameterized queries.

Such flexibility in defining functions and using them in queries is further strengthened by functional language abstractions such as *lexical closures* and *second-order functions*. When it comes to the array processing tasks, besides a library of the most common functions, SciSPARQL offers *array constructors*, *mappers* and *condensers* as second-order functions. These constitute a highly flexible mechanism of expressing custom array operations, demonstrated on the example of Geo-Science applications in [8].

This chapter summarizes the contributions presented in Scientific SPARQL as a language extension in terms of syntax and semantics. Implementation details are reserved for the next chapter, however, certain

notes on potential scalability opportunities are given, in order to encourage the formulation of expressive and straightforward SciSPARQL queries that our system (SSDM) is well-optimized for.

4.1 Array Queries

We define an *array* as a mapping function A from a finite *domain* to an infinite *range*, which is stored explicitly:

$$A: D \rightarrow R$$

The *domain* of arrays in SciSPARQL is always a Cartesian product of the sets of integers ranging from 1 to dim_k :

$$D = \{1 \dots \text{dim}_1\} \times \dots \times \{1 \dots \text{dim}_n\}$$

Here n is the number of dimensions in the array, dim_k is the array size in the dimension k and the $\langle \text{dim}_1, \dots, \text{dim}_n \rangle$ vector is called the *array shape*. We call arrays of the same shape *aligned* arrays. The range of an array can either be a set of *Integer*, *Real* or *Complex* numbers, or *Boolean* values.

The *RDF with Arrays* data model, underlying SciSPARQL queries, incorporates arrays into RDF graphs as another kind of nodes, along with other literal types. Array values may only appear in the *value* position of RDF triples. However, due to compatibility concerns with pure RDF and SPARQL, the predicates `rdf:first` and `rdf:rest` commonly used with RDF collections are *polymorphic* in SciSPARQL and may be matched with arrays appearing on the *subject* position in queries (examples in Section 5.5) This makes arrays into a particular case of RDF collections - Section 5.3.2 explains the relationship between arrays and collections in greater detail.

A typical RDF with Arrays dataset contains numeric multidimensional data - in form of arrays, and the associated metadata - in form of an RDF graph. Figure 6 shows a simple example, which will be further extended in the next chapter (where also a serialization in extended Turtle format is presented). It features an *RDF with Arrays* description of an experiment (given in a generic way, it might be a stochastic simulation of a partial differential equations system, for example) denoted as an instance of `ex:OurExperiment`, and consisting of a number of realizations, connected using the `ex:inExperiment` property. Both *experiment* and *realization* instances have literal-valued properties representing associated data and metadata at the respective levels of detail. The properties `ex:initialState` and `ex:result` are array valued, and represent the numeric part of *RDF with arrays* dataset.

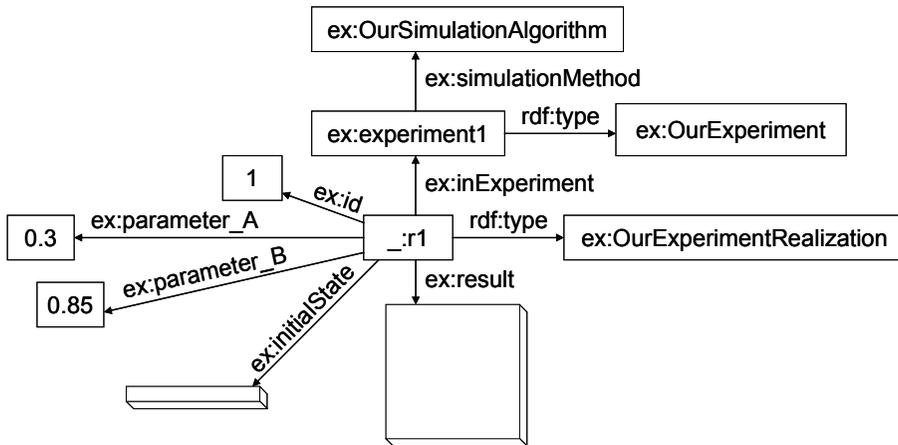


Figure 6. An example *RDF with Arrays* dataset (fragment)

We will refer to the queries aimed at retrieving arrays from *RDF with Arrays* datasets, and containing array-specific operations as *array queries*.

A trivial (but important) case is retrieving an array based on the associated metadata. For example, one might be interested in the `ex:result` arrays together with the corresponding realization ids, based on the experiment properties and realization parameters:

```

SELECT ?id, ?A
WHERE
  { ?e a ex:OurExperiment
    ex:simulationMethod ex:OurSimulationAlgorithm .
    ?r ex:inExperiment ?e ;
    ex:parameter_A 0.3 ;
    ex:parameter_B ?b ;
    ex:id ?id ;
    ex:result ?A .
    FILTER ( ?b > 0.8 ) }

```

The rest of this section introduces the key features of array queries. In the examples we deliberately omit the `PREFIX` part of the queries, since SciSPARQL allows the prefix declarations to be specified once per session - with a separate statement:

```

PREFIX ex: <http://udbl.uu.se/ex#>

```

4.1.1 Array Dereference Syntax

SciSPARQL allows array subscripts in square brackets, where subscripts for the respective dimensions are separated with commas.

For each dimension either *single subscripts* or *range selections* can be specified. By default, *range selections* are specified with a colon as *lo:hi*, and selections with a stride as *lo:stride:hi*, where both *lo* and *hi* address the elements that are included in the selection, and the elements are counted from 1. This design was chosen to make Matlab users feel at home⁶.

Either or both *lo* and *hi* values can be omitted, with default for *lo* being 1 and default for *hi* always being the array size in the respective dimension. Thus the expressions `?A[:]` and `?A[1:]` are always equivalent to `?a`.

If valid single subscripts for all array dimensions are specified, the array is dereferenced to a single element. Otherwise, complete ranges are assumed for the remaining dimensions. SciSPARQL thus makes a difference between three kinds of array dereferences:

- ***single element dereference***, for example `?A[2,1]` for a 2D array `?a`, where *single subscripts* are provided for all dimensions. The result is always a number, or *error* if a subscript falls out of range.
- ***projection dereference***, for example `?A[:,1]` or `?A[2]` or `?A[1:3,2]` or `?A[2,:5:]` for a 2D array `?A`, where *single subscripts* are provided for some dimensions, and *range selections* (explicit or implicit) for the others. The result is a smaller array with fewer number of dimensions (only those of the original dimensions for which ranges were provided), or *error* if a single subscript falls out of range or the *range selection* results in an empty selection.
- ***range selection dereference***, for example `?A[1:5,2:3]`, `?A[1:5]`, `?A[:5,:2:]`, where *range selections* (explicit or implicit) are provided for all array dimensions. The result is a smaller array with the same number of dimensions as the original one, or *error* if the *range selection* results in an empty selection.

The latter two are also collectively called *array slicing* operations. Each array slicing is resulting in an *array subset* Figure 7 shows the elements selected from a 2D array using *projection* on the first (rows) dimension, and *range selection* on the second (columns) dimension.

If a *range selection* effectively specifies a single element, it is still treated as a *range selection* with respect to the dimensionality reduction. Thus,

⁶ However, with the `_sq_python_ranges_` flag a user may opt for a different dialect of SciSPARQL, which supports Python notation for ranges. In this case, elements are counted from 0, *hi* element is never part of the selection, and optional strides are specified as *lo:hi:stride*. No other differences are introduced. This switch only takes effect at the stage when a SciSPARQL query, update, or function definition is passed to the interpreter. The definitions of SciSPARQL functions and parameterized updates are stored internally in a way that is invariant to these syntactic differences, so it is safe to switch back and forth between the two dialects in a session. In the rest of this work, the default (Matlab) notation is used.

(unlike Matlab) SciSPARQL makes a difference between arrays that have different number of "single-element" trailing dimensions, and between singleton arrays and numbers, so that `?A[2,3:3]` is not equal to `?A[2,3]`. For a 2D array `?A` where these subscripts are valid, the former expression would return a 1D-projection with a single element in it, whereas the latter expression would dereference directly to that element.

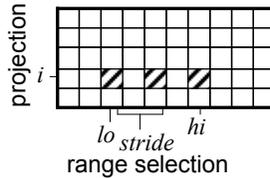


Figure 7. A projection and range selection `?A[4,3:2:7]`, applied to a 2D array

Since SciSPARQL is designed to handle very large arrays, any dereference operation that returns a derived array does not allocate any memory to store the new array's elements - internally, it just allocates a new *descriptor* object pointing to the same storage space. Thus, creating sets of projections and slices of arrays is very cheap (further explained in Section 5.2), and is encouraged as a simple way to formulate many data-reduction operations. This principle extends to arrays stored externally (and retrieved lazily), as we discuss in Chapter 6.

4.1.2 Variables Bound to Array Subscripts

One important feature of SciSPARQL as a declarative query language is the possibility to automatically bind a query variable to its valid range of values. Just as a triple pattern

```
?x foaf:name "Alice" .
```

binds variable `?x` to every node that has a property `foaf:name` with value "Alice", an array dereference expression

```
?A[?i]
```

with the otherwise unbound variable `?i` becomes an *array access pattern*: the variable `?i` will assume all valid subscript values, that is, integers from 1 and up to the size of array `?A` in its first dimension.

Unless otherwise restricted, such binding will form a Cartesian product with bindings for other variables in the query solution. So, for example,

```
SELECT ?i, ?j (?A[?i,?j] AS ?value)
WHERE { [] ex:id 1 ; ex:result ?A }
```

will return every element of the 2D array ?A (or respective projections if ?A is array of greater dimensionality, or nothing otherwise), together with subscript values. Similarly,

```
SELECT ?i, ?j (?A[?i,?j] AS ?value)
WHERE { [] ex:id 1 ; ex:result ?A .
        FILTER ( ?i >= ?j ) }
```

will return bottom-left triangle of ?A, and

```
SELECT ?i (?A[?i,?i] AS ?value)
WHERE { [] ex:id 1 ; ex:result ?A }
```

will return the diagonal elements. We will study the performance of such patterns in Chapter 6.

4.1.3 Built-in Array Functions

A number of basic functions are defined in SciSPARQL in order to access the array shape and element type, construct arrays and perform operations not covered by the array dereference syntax:

- `adims(?a)` - return the shape of an array as a 1D integer array containing sizes of `a` in each dimension. To obtain the number of dimensions, use `adims(adims(?a)) [1]`.
- `elttype(?a)` - return element type of array, with 0 for *Integer*, 1 for *Double*, 2 for *Complex*.
- `A(?e1, ?e2, ?e3, ...)` - construct a 1D array of the given numeric elements.
- `find(?a, ?e)` - return the indexes of elements in `?a` equal to `?e`, as 1D integer arrays.
- `permute(?a, ?d1, ?d2, ...)` - change the shape of array by rearranging its dimensions (generalized transposition). The integer values `?d1, ?d2, ...` denote the new order for the array dimensions. The effect is the same as with Matlab `permute()` function⁷.
- `transpose(?a)` - simple 2D matrix transposition, equivalent to `permute(?a, 2, 1)`.

Rearranging array dimensions, similarly to an array slicing operation, involves no copying of array elements, and thus produces a derived array.

4.1.4 Array Arithmetic

The standard binary operators operating on numbers in SPARQL are extended to operate element-wise on arrays in SciSPARQL. This includes addition '+', subtraction '-', multiplication '*', and division '/' operators. For

⁷ Officially documented in <http://mathworks.com/help/matlab/ref/permute.html>

example, an expression $?a + ?b$ will be evaluated in four cases, as shown in the Table 2.

Table 2. Polymorphism of an arithmetic operator in SciSPARQL (example)

$?a$ binding	$?b$ binding	value of $?a + ?b$
number	number	number
number	array	array, where $?a$ is added to each element of $?b$
array	number	array, where $?b$ is added to each element of $?a$
array	array	array of sums of corresponding $?a$ and $?b$ elements, if $?a$ and $?b$ have the same shape

However, in order to let the SciSPARQL query optimizer distinguish between scalar and array-valued operations (the latter are expected to be sufficiently more expensive, both in terms of computation and memory), SciSPARQL users are encouraged to use the special array-oriented dot-prefixed operators, for example $?.+$ in cases where array values are expected.

The expression $?a ?.+ ?b$ is semantically equivalent to $?a + ?b$ as described by Table 2, e.g. it produces a number if both operands are numbers. However, it hints the query optimizer that an array value is expected here, so it will try to schedule this operation at the point where fewer intermediate results (i.e. candidate bindings for $?a$ and $?b$) are anticipated.

This is different for the comparison operators $<$, $<=$, $>$, $>=$, which, when applied to an array (or two arrays of the same shape) will produce a deterministic albeit not a meaningful result, used only for ordering. Equality of arrays, however, is well defined below in Section 4.1.6. In the same cases, dot-prefixed comparison operators will produce a new array of type *Boolean*, containing the results of element-wise comparison.

Numeric aggregate functions, like $SUM()$, $MIN()$, $MAX()$, $AVG()$, etc. are also extended to handle bags of array values. They return only if all arrays in the bag have the same shape, and construct a new array value. No optimizer hints are available, since the evaluation of aggregate functions separates the *inner* and *outer* contexts of a query (more technicalities in Sections 5.4.1.2 and 5.4.3.4), and there is typically little freedom to move the predicates around it (see, e.g. [38]).

Another possibility is that due to the modular structure of SciSPARQL queries (as described in Section 5.2.2), there might be two parameterized aggregate subqueries invoked as functions from a third query on the same level - then the optimization might benefit from knowing which aggregation involves arrays and which one does not. We leave these optimization

opportunities, based on a more accurate cost estimate for the aggregate functions as a matter of the future work.

4.1.5 Intra-array Computations

Arrays, apart from bags, form another conceptual layer of collections in SciSPARQL. While it is possible to combine all elements of a bag of numbers (or arrays) with the aggregate function `SUM()`, it should also be possible to apply an aggregate function to all (or certain) elements of a given array. There are actually three ways to do this in SciSPARQL:

I. Shorthand functions as `array_sum()`, `array_avg()`, `array_min()`, and `array_max()` are available in SciSPARQL for the basic computation tasks, and should be preferred as the most efficient ones. They operate on all elements of a given array, and ignore the logical dimensionality.

II. It is always possible to "open" an array into a bag of its elements, as shown in Section 4.1.2, and then apply a traditional aggregate function. This allows arbitrary conditions on the element places and values to be expressed in a query. For example, the following query would sum up only positive elements on even positions in the main diagonal of `?A`:

```
SELECT (SUM(?A[?i,?i]) AS ?sum_diag_even_positive)
WHERE { [] ex:id 1 ; ex:result ?A .
         FILTER ( ?A[?i,?i] > 0 ) && mod(?i, 2) = 0 }
```

Here, the free variable `?i` binds to all valid values for the row and column subscripts of `?A`, and then is checked for an even value. Only in those cases, array elements are considered eligible to be summed up. As the example shows, this way is highly general, but might clutter the `FILTER` expression (which is typically used for metadata conditions) and also forces bag-based aggregation where it could have been avoided.

III. In order to alleviate for the said shortcomings, SciSPARQL borrows Array Algebra [17] primitives used in *Rasdaman* [16], as a matter of ongoing integration. The second-order functions `MAP()`, `CONDENSE()`, and `ARRAY()` are supported in our system, making use of the powerful *lexical closure* mechanism, explained in Section 4.3.

4.1.6 Array Equality

The only cases where dot-prefixed operators differ from the original ones is the comparison of arrays with `'='` and `'!='`, which results in a single Boolean value, and the comparison of array elements with `='.='` and `='.!='`, which results in *array of Boolean*. While the second case is trivial, the equality of arrays needs a definition.

Two arrays are equal **iff** all of the following conditions are satisfied:

- they have the same number of dimensions,
- they have the same size in each respective dimension,
- their respective elements are numerically equal.

Note that the same element type is not a requirement - an integer array might be equal to an array of real numbers. However, whenever the floating-point arithmetic is involved, it is always a good idea to round the array elements down to a certain precision before comparing, in order to avoid precision-induced artifacts. For this purpose the `round()` function is extended to handle arrays, taking the desired precision as a second argument.

SciSPARQL does not trim the trailing dimensions of size 1 as e.g. Matlab does, which might lead to the loss of structural metadata, important in our setting. Hence e.g. a 1-dimensional array of size 3 can never be equal to a 2-dimensional 3x1 array, even though they both might represent the same mathematical object - a *column vector*. Similarly, SciSPARQL does not treat simple numeric values as equivalent to singleton arrays: a number 5 is not equal to an array with a single element of 5.

4.2 Parameterized Queries - Functional Views

The good modularity of potentially complex SciSPARQL queries is achieved by isolating common parts as *parameterized queries*, also known as *functional views*. We use these two terms interchangeably, since by stressing different aspects of the same mechanism, together they convey the desired dualistic notion of the subject.

There is `DEFINE FUNCTION` statement in SciSPARQL. As shown below in Section 4.4, its use extends far beyond the *functional views* and SciSPARQL per se; however, for the purpose of this section its use is quite simple. The following example defines a function `resultById()` retrieving the value of `ex:result` property of a realization of the `ex:OurExperiment` experiment class, given the realization id:

```
DEFINE FUNCTION resultById(?id) AS  
SELECT ?A  
  WHERE { ?r ex:inExperiment [ a ex:OurExperiment ] ;  
           ex:id ?id ;  
           ex:result ?A }
```

Naturally, a call to this function can be used as a part of an expression. This has the potential of formulating short queries without a proper `WHERE` clause at all. For example, the following query returns the third row of the `ex:result` matrix of a realization with `id = 1`:

```
SELECT (resultById(1)[3] AS ?row3)
```

A function definition is parsed and validated (but not optimized) at the moment it is submitted as a SciSPARQL statement. This implies, in particular, that the prefixes used in a function definition (unless supplied directly before the `DEFINE FUNCTION` clause) should be already defined for a session. Similarly, any other functions called inside the definition should already be defined. This way SciSPARQL forbids *mutual-* and *self-recursion*, and imposes an acyclic dependency graph among the function definitions it maintains.

This principle does not extend to accessing the named RDF graphs. A graph specified in a `FROM`, `FROM NAMED`, or `GRAPH` clause inside a function definition does not need to be present among the available graphs at the time of function definition - thus the library of *functional views* can be loaded into a SciSPARQL session (using `SOURCE` directive) independently of loading or creating the named RDF graphs.

Apart from query modularity benefits, with *functional views* it is possible to express some otherwise inexpressible computations in a single query. In particular, it is possible to nest aggregate operations - for example computing the sum of positive diagonal elements of `ex:result` for each array, and then finding the average value across all realizations in the given experiment instance:

```
DEFINE FUNCTION sum_diag_positive(?r) AS
SELECT (SUM(?A[?i,?i]) AS ?res)
WHERE { ?r ex:result ?A .
        FILTER ( ?A[?i, ?i] > 0 ) }

SELECT (MAX(sum_diag_positive(?r)) AS ?max)
WHERE { ?r ex:inExperiment ex:experiment1 }
```

In the next section (4.3), we show how functions similar to `sum_diag_positive()`, returning numeric values, can be used with second-order functions like `ARGMIN()` and `ARGMAX()`.

Another important benefit of *functional views* is the ability to express *top-k selections* for a non-fixed parameter *k*. For example, the following function will find the given number of highest values on the `ex:result` diagonal:

```
DEFINE FUNCTION k_top_diag(?r ?k) AS
SELECT (?A[?i,?i] AS ?e)
WHERE { ?r ex:result ?A }
ORDER BY DESC(?e) LIMIT ?k
```

While the SPARQL Standard requires that `LIMIT` and `OFFSET` values should be constants, in SciSPARQL they can be expressions not depending on the variables inside the query. A parameter in a *parameterized query* thus may be used.

4.3 Lexical Closures and Second-Order Functions

SciSPARQL offers second-order functions that allow expressing common computational tasks easily, as demonstrated in [6, 8].

For example, optimizing a function over a finite domain is the in the general case done by evaluating it for every valid set of arguments and comparing the results. In order to express this declaratively, SciSPARQL features the `ARGMIN()` and `ARGMAX()` second-order functions. For example⁸, finding a realization having the greatest sum of positive diagonal elements in `ex:result` matrix is expressed as

```
SELECT (ARGMAX(sum_diag_positive(*)) AS ?r_max)
```

or, since SciSPARQL allows function calls as separate statements, simply:

```
ARGMAX(sum_diag_positive(*))
```

The free parameter denoted by the asterisk will sweep across all nodes in the RDF graph, matched as *subjects* by the triple pattern inside the function `sum_diag_positive()`, as it is defined in the previous section.

Another feature inspired by Array Algebra [17] are the generic *array constructor*, *mapper* and *condenser*, represented by the `ARRAY()`, `MAP()`, and `CONDENSE()` second-order functions in SciSPARQL, explained below in Section 4.3.1.

All of these take a functional argument - a *lexical closure*, consisting of a function name and values provided for some (or none) of its parameters, with other parameters marked by asterisk '*' placeholder. Inside a second-order function, a lexical closure is evaluated exactly like a normal function with a number of arguments equal to the number of asterisks. For example, `ARGMIN()` and `ARGMAX()` require unary functions - the lexical closures will always contain one asterisk. The rest of the arguments are bound to values provided at the point of closure formation.

For example, *Minkowski distance* is a function of three arguments - two vectors and one scalar exponent:

$$d_{Minkowski}(x, y, p) \stackrel{Def}{=} \left(\sum_i |x_i - y_i|^p \right)^{1/p}$$

In SciSPARQL, this example would look like

```
DEFINE FUNCTION Dminkowski(?X ?Y ?p) AS  
SELECT (power(SUM(power(abs(?X[?i] - ?Y[?i]), ?p)),  
1/?p) AS ?distance)
```

⁸ Section 5.4.5.10 has the translation of this example to AmosQL.

In many practical cases, however, the exponent p is provided upfront, whereas the two vectors are the "real" arguments that the function typically maps over. For example, *Euclidean distance* can be defined as a function of two arguments

$$d_{Euclid}(x, y) \stackrel{Def}{=} d_{Minkowski}(x, y, 2)$$

Lexical closures eliminate the need of defining and naming single-use functions. So, instead of separately defining, and then providing d_{Euclid} as a functional argument, one could directly use $Dminkowski(*, *, 2)$ as an equivalent binary function.

4.3.1 Array Algebra Second-order Functions

An **array constructor** returns an array of given type and shape. It expects a unary function (or closure) that takes a vector of *logical subscripts* as a single argument, and computes the array elements:

ARRAY(*type, shape, mapper*)

An **array mapper** maps over a collection of $n \geq 1$ *aligned arrays*. It returns a new array of given type aligned to that collection. It expects an n -ary function (or closure) that is mapped over the respective elements of the given arrays:

MAP(*type, mapper, v₁, ..., v_n*)

An **array condenser** computes an intra-array aggregate value applying a given aggregate operation to all array elements. No particular order is guaranteed; hence the aggregate operation (represented by a binary function or closure) is required to be commutative and have identical domain and range.

CONDENSE(*op, v*)

An additional unary *filter* function, if provided, will be applied first, in order to select elements based on their value:

CONDENSE(*op, v, filter*)

Intra-array aggregate functions like `array_sum()`, `array_avg()`, etc. are equivalent to particular condenser calls.

The usage examples of these second-order functions are given in [8], in the context of a geo-informatics application.

4.4 Foreign Functions

As mentioned above, a typical scientific or engineering data processing task involves both data retrieval and extensive computations. While the querying capabilities of SciSPARQL address the data retrieval task in a more general and expressive way than generally seen in manually written programs, calling various computational routines should stay similar to the way it is normally done in C, Python, or Matlab. At the same time, the query optimizer should retain the freedom to call the filtering and post-processing tasks in the optimal order, based on the *cost* and *cardinality* estimates, as explained below.

For this purpose, SciSPARQL offers a mechanism for extensibility with *foreign functions*. While being implemented in algorithmic languages (currently C/C++, Java, Lisp, Python, or Matlab), these functions are used directly in a query: the `SELECT` clause typically contains the post-processing expressions, and `FILTER/HAVING` clauses contain the expressions that filter the potential query solutions. In the same way as *functional views*, foreign functions can be used to form *lexical closures* and be passed to *second-order functions*, as explained in Section 4.3.

The process of introducing a foreign function to SciSPARQL typically involves three steps:

- providing a function implementation or a wrapper for a library function, with the signature (header) compatible to SciSPARQL,
- linking the implementation to SSDM (mechanisms for different languages vary), and
- defining the new SciSPARQL function using the `DEFINE FUNCTION` statement, optionally providing cost and cardinality estimates.

For example, the following function implemented in Java would return real square roots (if any) of its real or integer argument:

```
public class MyLib {
    public void sqroot(CallContext cxt, Tuple tpl) throws AmosException
    {
        double x;
        if (tpl.isDouble(0)) x = tpl.getDoubleElem(0);
        else if (tpl.isInteger(0)) x = tpl.getIntElem(0);
        else return;

        if (x >= 0.0) {
            double r = Math.sqrt(x);
            tpl.setElem(1, r);
            cxt.emit(tpl);
            if (x > 0.0) {
                tpl.setElem(1, -r);
                cxt.emit(tpl);
            }
        }
    }
}
```

Such a Java implementation of a SciSPARQL foreign function is effectively *static*, and returns the results by calling `cxt.emit()`. Each call to a foreign function may thus yield zero or more results. The arguments and results are passed using a single `Tuple` instance, where the first `tpl.getArity()-1` positions are filled with arguments, and the function has to fill the last one with its result before emitting. In all these respects, C/C++ and Lisp interfaces are similar and offer the same degree of flexibility, while Python and Matlab interfaces offer a direct mapping of SciSPARQL function arguments to those of the implementing function.

Since SciSPARQL is a dynamically typed language, in all cases a runtime type check is necessary. By convention, as explained in Section 3.6, a runtime error is not an exception, but instead the absence of any emitted result. An invalid value passed to a filter or postprocessing function is equivalent, e.g., to an unmatched triple pattern, simply resulting in a discarded solution. Hence, `AmosException` is reserved only for so-called *internal errors*, and cannot be thrown because of the wrong input.

Linking of such a Java implementation is achieved by including the bytecode for `MyLib` into Java's `CLASSPATH` when running SSDM under JVM. In case of Python, the source code needs to be placed in `PYTHONPATH`. In case of C/C++, linking involves compiling a separate dynamic-link library, and dynamically loading it into SSDM process, by issuing `LOAD_EXTENSION('mylib')`, referring to `mylib.dll` in Windows *path* or `libmylib.so` in Linux *library path*. Lisp source files are loaded in a similar way using `SOURCE_LISP()`. Matlab foreign functions require no additional linking, since they are available as callbacks from the SSDM process embedded into Matlab.

Finally, the SciSPARQL definition of `sqroot()` would look like:

```
DEFINE FUNCTION sqroot(?x)
AS JAVA 'MyLib/sqroot' COST 4 FANOUT 1
```

Here the optional `COST` and `FANOUT` parts specify the cost and cardinality estimates. Even very rough estimates would help the optimizer much better than the absence of any. By convention, the unit cost corresponds to a simple arithmetic operation like `+` or `*` over scalar operands. `FANOUT` specifies the average amount of results emitted per function call - in our case it averages to one (i.e. zero for negative arguments and two for positive).

Whenever possible, the users are encouraged to provide foreign functions as *multidirectional* [58] so that the optimizer might choose to compute the function arguments if the result happens to be bound earlier. Such definitions are made by specifying the alternative *binding patterns* as strings composed of 'b' for *bound* and 'f' for *free* (or, respectively, '-' and '+'), and providing an implementation for each. For example, if a similar

implementation `square()` is defined⁹ in `MyLib` Java class, the multidirectional definition would be:

```
DEFINE FUNCTION sqroot(?x) AS  
  FOR 'bf' JAVA 'MyLib/sqroot' COST 4 FANOUT 1  
  FOR 'fb' JAVA 'MyLib/square' COST 1 FANOUT 1
```

In this example we have shown a function dealing with simple types, like *Double* and *Integer*, which are mapped to Java's (or other languages') native type system. Since the *RDF with Arrays* data model introduces RDF-specific types, like language- and locale-annotated strings, typed literals, URIs, and most notably, Numeric Multidimensional Arrays; each language interface provides the additional classes for each of these. For example, a Java implementation would use `UString`, `TypedRDF`, `URI`, and `NMA` (array) wrapper classes defined in `ssdm` package. Each of them provides constructors and field accessors to facilitate the native data processing.

The complete extensibility interface documentation for each language is a part of the SciSPARQL User Manual [146].

4.5 Calling SciSPARQL from Algorithmic Languages

SciSPARQL queries can easily be incorporated into traditional algorithmic programs - this approach would be somewhat opposite to the one described in the previous section. However, both approaches are typically combined in sufficiently complex real-life applications. Declarative SciSPARQL queries may thus be embedded in traditional data processing routines, which might include data acquisition, logging, visualisation, user interactions, or feedback loops in a control system.

The process of calling SciSPARQL queries (or SciSPARQL functions as parameterized queries - see Section 4.2) relies on the concepts of *connection* and *scan* (result set), and involves the following steps:

- establishing a *connection* to SSDM server,
- passing a query string (or a function name and actual arguments) to the server, and retrieving a *scan*,
- iterating through the *scan*, effectively running the query execution plan just enough to retrieve yet another result,
- closing the *scan*,
- closing the *connection*.

⁹ The implementation of `square()` should be aware of the binding pattern it is called with, as it has to retrieve its de-facto argument from position 1 and write its result into position 0. For this reason, `sqroot_fb()` might be a better name for such implementation.

An important scalability feature is the lazy evaluation of SciSPARQL queries. A query does not have to be executed in its entirety in order to obtain a scan. Instead, it is the scan object that calls back SSDM in order to advance the query execution on demand. After retrieving each result, the application program is free to close the scan, thus terminating the query - a feature more powerful than the `LIMIT` clause inside a query, as any application logic can be involved. However, providing the `LIMIT` clause is still a good practice when the number of results to retrieve is fixed - this provides more freedom to the optimizer.

Chapter 7 describes the usage of embedded SciSPARQL queries in greater detail, in the context of Matlab integration. Java, C/C++, and Python programs may use the respective APIs, implementing the `Connection` and `Scan` classes.

Both this API and the *Foreign Functions* interface described in the previous section are the essential parts of the underlying Amos II [136] database management system. SSDM extends both of these in order to handle its type system - *RDF types* and *arrays*, and provides its own documentation and usage examples. The API documentation is part of the SciSPARQL User Manual [146], libraries, header files, and code examples are part of SSDM.

5 Scientific SPARQL Database Manager

Scientific SPARQL was developed along with its implementation - a software system called Scientific SPARQL Database Manager, or SSDM for short.

This chapter describes the internal architecture of SSDM's kernel, while the back-end and storage manager interfaces are explained in Chapter 6. Section 5.1 first presents an architectural overview including the query processing steps, illustrated by an example. Next, in Section 5.2 in-memory implementations of arrays and array operations in SSDM are explained in detail. Section 5.3 introduces data loaders from RDF serialization formats, including the *array apprehension* mechanisms, and external links that may be followed in a lazy fashion, resulting in *array proxies*.

Section 5.4. describes the process of translating SciSPARQL queries to AmosQL. It begins with the formal definition of the SciSPARQL statement structure in Section 5.4.1, followed by the necessary refinement of the standard SPARQL semantics in Section 5.4.2. Certain restrictions are lifted, and a clear *operational semantics* is defined for all valid SciSPARQL queries. Next, the target language query structure and semantics is described in detail in Section 5.4.3, along with the extensions made to it in order to implement SciSPARQL (Section 5.4.4). Finally, Section 5.4.5 gives a formal definition of the translation algorithm - we recommend this section mainly for those who are going to do their own implementation of SPARQL or a similar language, and are facing related challenges. Translations of aggregate functions, grouping, array access, functional views, second-order functions and path queries are explained in separate sub-sections.

Section 5.5 completes the discussion by raising interesting issues with the `rdf:first` and `rdf:rest` properties, which become polymorphic in SciSPARQL, due to the backwards-compatibility requirement. Solutions are presented, along with illustrative examples.

5.1 Architecture overview

Structurally, SSDM is comprised of a core (central box in Figure 8) capable of answering *SciSPARQL* queries, loading *RDF with Arrays* data for storage, executing external functions and implementing an open set of wrapper/mediator and storage back-end interfaces.

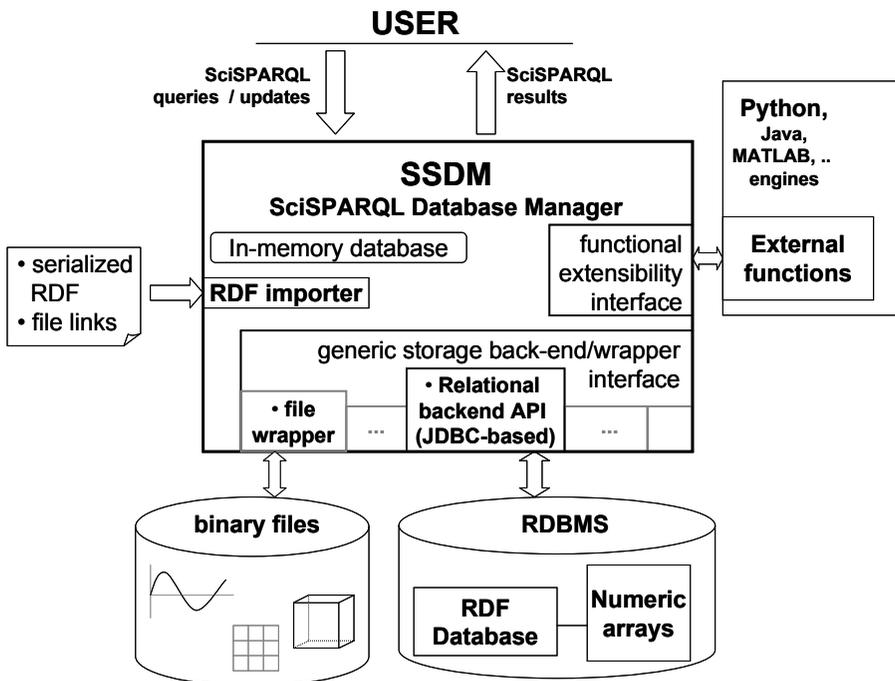


Figure 8. The SSDM Architecture, including interfaces and extensions

Technically, SSDM is a major extension to the Amos II main-memory DBMS, utilizing its query processing facilities including AmosQL and ObjectLog query representations, query optimizer, cost model, execution algebra, extensibility mechanisms, inter-process communications, and other facilities that proved quite useful both in research prototyping and production scenarios.

SSDM uses the *in-memory* database of Amos II to store/cache RDF graphs, so that the graph pattern matching is performed using its main-memory based indexing mechanisms. Array data can also be stored and processed in main memory - we will refer to this case as the *main-memory scenario* in Chapter 6. Since SSDM is built to accommodate large amounts of numeric data, the *generic storage back-end/wrapper mechanism* (described in Section 6.1) is used to retrieve data from (or store the data in) an open set of storage systems, and to delegate array processing to these

systems whenever possible. Sections 6.2 - 6.3 describe the *Relational backend API* and the specific optimization techniques. A setting where arrays are stored directly in binary files is described in Chapter 7.

RDF with Arrays graphs are defined either with a custom-built wrapper over a non-RDF data model, (as described in Section 2.3), produced internally using SPARQL Update syntax (Section 3.9), or imported from RDF files using the *RDF importer* (Section 5.3), where the numeric array data is either consolidated from an RDF-based notation (Sections 5.3.2 - 5.3.3) or loaded lazily from binary files using *file links* (Section 5.3.1) - a specific type of URIs.

SciSPARQL queries are extensible with foreign functions, which can be defined in one of the supported algorithmic languages. For this purpose, SSDM features a functional extensibility interface (Section 4.4). These foreign functions can be used in queries for filtering or post-processing the results, and the query optimizer can be provided the necessary information for optimizing such external calls (ibid).

An SSDM process can run either as a server, accepting connections from SSDM clients, as a client, or stand-alone. The communication is done low-level via TCP sockets, with all data objects being marshalled using serialization methods provided for their classes. The server instances can easily be clustered using centralized (star-shaped) or decentralized (peer-to-peer) network configurations.

For the query part, SSDM currently offers

- a text based-interpreter console for direct user interaction, in case of a stand-alone/client process,
- C and Java APIs allowing to send SciSPARQL queries and updates, and access the query results in terms of the host language data structures (explained in Section 4.5), and
- MATLAB front-end, allowing seamless integration of SciSPARQL queries into typical scientific and engineering workflows (described in Chapter 7).

The SSDM architecture is best illustrated by a scenario, which includes loading an *RDF with Arrays* dataset and answering a SciSPARQL query.

5.1.1 Example Dataset

As a running example, we are going to extend the RDF dataset from the previous chapter, illustrated in Figure 6. It features an *experiment* instance of a class `ex:OurExperiment`, with attached *realization* instances of another class `ex:OurExperimentRealization`. Both have a number of properties, including array-valued properties `ex:initialState` and `ex:result`.

Let's first consider the following Turtle file being loaded into SSDM:

```
@prefix ex: <http://udbl.uu.se/ex#> .

ex:experiment1 a ex:OurExperiment ;
               ex:simulationMethod ex:OurSimulationAlgorithm .

_:r1 a ex:OurExperimentRealization ;
     ex:inExperiment ex:experiment1 ;
     ex:id 1 ;
     ex:initialState (0 0.5 1 1 1 1 0.5 0) ;
     ex:iterations 1000 ;
     ex:parameter_A 0.3 ;
     ex:parameter_B 0.85 ;
     ex:result <file://realization_1.mat#Res> .
```

Essentially, this file combines data and metadata describing one instance of the `ex:OurExperiment` class, and one instance of the `ex:OurExperimentRealization` class. The schema of this RDF dataset is implicit, and can be illustrated by the ER-diagram in Figure 9, with array-valued properties shown as 3D rectangles.

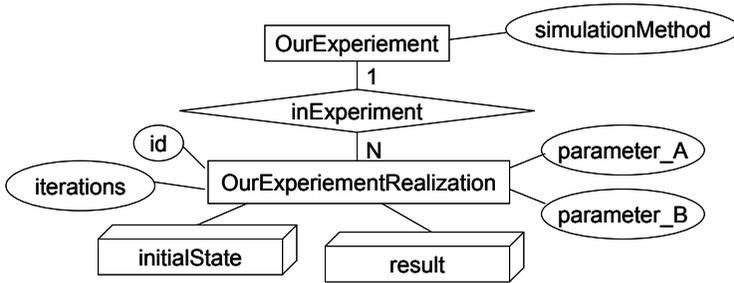


Figure 9. Implicit ER model, inferred from the first part of the *GI* example dataset

Two syntactic options are used to supply arrays in a Turtle file:

- *RDF Collections* syntax for the `ex:initialState` property, utilizing array apprehension syntax described in Section 5.3.2, and
- *file links* for the `ex:result` property, described in Section 5.3.1.

As this Turtle file is imported into SSDM, ten RDF triples become stored as a *default graph* in SSDM's in-memory database. Two triples are array-valued: one corresponding to `ex:initialState` stores a *memory-resident array* as its value, which is *consolidated* from an RDF collection of numbers. The value of the `ex:result` triple is read from a linked file, using the additional information (a variable name) provided after the '#' sign. As we show in Section 6.1, it is not necessary to load the array content into memory as SSDM allows for lazy array data retrieval. Still, the array shape and element type information need to be read from a linked file at the loading stage, so the linked files are required to be available on the server file system at this point.

One of the benefits of using RDF for metadata, compared with the relational data model, is that adding new properties is easy, and does not require redesign of the whole database. For example, at some later point one might decide to store an additional parameter C, and move the `ex:simulationMethod` property down to the `ex:OurExperimentRealization` class:

```
_:r314 a ex:OurExperimentRealization ;
      ex:inExperiment ex:experiment1 ;
      ex:id 314 ;
      ex:simulationMethod ex:OurSimulationAlgorithm_v2;
      ex:initialState (0 0 0 0.5 0.5 0 0 0) ;
      ex:iterations 2000 ;
      ex:parameter_A 0.3 ;
      ex:parameter_B 0.9 ;
      ex:parameter_C 3.14 ;
      ex:result <file://realization_314.mat#Res> .
```

Such a realization instance still belongs to the same class, and is connected to the same experiment instance as the realization above. As for the `ex:simulationMethod` property that can now be attached to instances of different classes, SPARQL makes it easy to query using UNION and OPTIONAL constructs, , as shown in Section 3.3.2.

We will refer to this example dataset, including all the triples attached to `_:r1` and `_:r314`, as graph *GI* throughout this chapter.

5.1.2 Example Query

A typical SciSPARQL query contains a graph pattern, filtering and post-processing expressions.

QI: Select the average of the simulation result values at the last iteration, together with realization id, for those realizations that have parameter $A \geq 0.25$ and the initial state values limited by 0.75 (assuming the `ex:result` arrays are 2-dimensional, and the second dimension is the iteration number):

```
PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id (array_avg(?R[:, ?iterations]) AS ?res)
WHERE { ?realization a ex:OurExperimentRealization ;
          ex:id ?id ;
          ex:result ?R ;
          ex:iterations ?iterations ;
          ex:parameter_A ?a ;
          ex:initialState ?initialState .
FILTER ( array_max(?initialState) < 0.75
          && ?a >= 0.25 ) }
```

This query has a single block of conditions, and, conceptually, the process of its evaluation consists of three different steps, as illustrated in Figure 10:

- applying a *graph pattern*,
- filtering the *solutions*, and
- postprocessing the results.

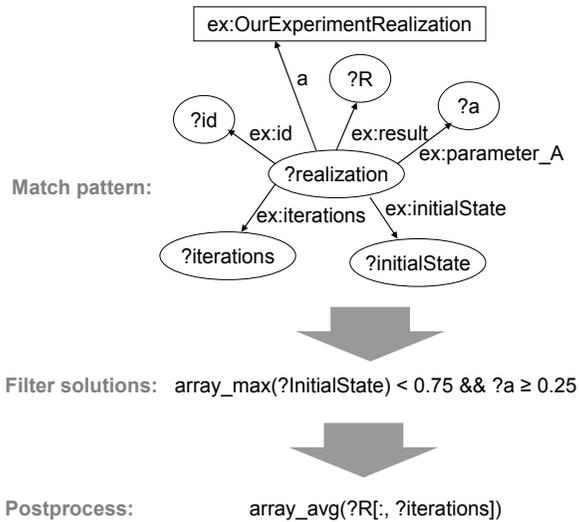


Figure 10. Conceptual stages of answering query *Q1*

By the *solution* of a *graph pattern* we will denote the set of bindings, one for each variable in a graph pattern, which belong to the underlying RDF graph and do not contradict that pattern. For example, the complete list of solutions for the single *graph pattern* in *Q1*, given the RDF graph *G1* in Section 5.1.1, can be represented as a table:

<code>?realization</code>	<code>?id</code>	<code>?R</code>	<code>?iterations</code>	<code>?a</code>	<code>?initialState</code>
<code>_:r1</code>	1	<code><array proxy></code>	1000	0.3	<code>(0 0.5 ...</code>
<code>_:r314</code>	314	<code><array proxy></code>	2000	0.3	<code>(0 0 ...</code>

At the filtering stage, the first solution is filtered out by the first conjunct, so only the second solution contributes to the result. However a more detailed look at the conditions in *Q1* might suggest that we do not really need to retrieve the complete solutions in order to evaluate the filters - by binding only the `?a` and `?initialState` variables first, we might skip retrieving the other data from the RDF graph in some cases.

In the next subsections we show how SSDM solves this typical query optimization task, and comes up with an efficient execution plan.

5.1.2.1 Query translation

The first step is translating SciSPARQL to AmosQL:

```
select id, rdf:array_avg(aref(R,1,rdf:minus(iterations,1)))
  from Literal realization, Literal a, Literal initialState,
       Literal iterations, Literal R, Literal id
 where (realization,
        URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
        URI('http://udbl.uu.se/ex#OurExperimentRealization'))
        in GRAPH(0)
 and (realization, URI('http://udbl.uu.se/ex#id'), id) in GRAPH(0)
 and (realization, URI('http://udbl.uu.se/ex#result'), R)
        in GRAPH(0)
 and (realization,
        URI('http://udbl.uu.se/ex#iterations'), iterations)
        in GRAPH(0)
 and (realization, URI('http://udbl.uu.se/ex#parameter_A'), a)
        in GRAPH(0)
 and (realization,
        URI('http://udbl.uu.se/ex#initialState'), initialState)
        in GRAPH(0)
 and rdf:array_max(initialState)<0.75 and a>=0.25;
```

Here we can see that all RDF triples are accessed via the `GRAPH()` function, whose argument denotes a specific graph, with 0 denoting the *default graph*. A triple pattern is matched using the '*s p o*' in' syntax for locating tuples in AmosQL. All SciSPARQL query variables are mapped to AmosQL query variables, and functions `array_avg()` and `array_max()` are translated to their corresponding AmosQL implementations. Prefixed URIs are expanded into string arguments to the `URI()` constructor function.

Since SciSPARQL is dynamically typed, while AmosQL is statically typed, a common supertype `Literal` for all RDF terms is used in variable declarations. Additionally while e.g. comparison operators are defined across any type of arguments, some other arithmetic operators like '-' have to be translated to calls to a generalized function like `rdf:minus()`, which does the dynamic type checking. That call is needed to translate 1-based array indexing in the current SciSPARQL dialect to the 0-based array index, as required by the AmosQL function `aref()`, introduced in SSDM. The `aref()` function extracts an array subset given the array value of variable `R`, along the second dimension (argument `I`) with supplied index *iterations-1*.

5.1.2.2 ObjectLog representation: predicates and binding patterns

The AmosQL query is then further translated into a logical expression of *ObjectLog* predicates. Extra variables are introduced to flatten out the nested functional-style expressions:

```
(*SELECT* ID+ V29+) <-
(AND (GRAPH 0 REALIZATION
      #[URI "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"]
      #[URI "http://udbl.uu.se/ex#OurExperimentRealization"])
 (GRAPH 0 REALIZATION #[URI "http://udbl.uu.se/ex#id"] ID)
 (GRAPH 0 REALIZATION #[URI "http://udbl.uu.se/ex#result"] R)
```

```

(GRAPH 0 REALIZATION #[URI "http://udbl.uu.se/ex#iterations"]
 ITERATIONS)
(GRAPH 0 REALIZATION #[URI "http://udbl.uu.se/ex#parameter_A"] A)
(GRAPH 0 REALIZATION #[URI "http://udbl.uu.se/ex#initialState"]
 INITIALSTATE)
(RDF:ARRAY_MAX INITIALSTATE _V26)
(< _V26 0.75)
(>= A 0.25)
(RDF:MINUS ITERATIONS 1 _V27)
(AREF R 1 _V27 _V28)
(RDF:ARRAY_AVG _V28 _V29)

```

Here, `URI()` constructor calls with constant arguments are evaluated at compile time, which is a practical way to reduce the size of the optimization problem. Any other AmosQL function (whether *stored* or *foreign*) with a signature $f(x_1, \dots, x_n) \rightarrow (y_1, \dots, y_m)$ is matched with an ObjectLog predicate $(F\ X_1 \dots X_n\ Y_1 \dots Y_m)$. Any *derived* AmosQL functions would have been expanded and flattened at this step, and any logical expressions would be normalized to *Disjunctive Normal Form*.

Every ObjectLog predicate has a number of allowed binding patterns, and a cost and fanout (i.e. cardinality multiplier) estimate associated with each of them. Predicates representing *stored functions* can be evaluated with all, any, or none of their arguments bound: the fanout estimates depend on the storage statistics and cardinality constraints provided, while the cost to return each result depends on the available access paths.

Foreign functions (whether built-in or user-defined) are different. For example, a comparison operator '`<`' can only be evaluated when both of its arguments are bound, whereas the `rdf:minus()` arithmetic function is represented by a ternary ObjectLog predicate $(RDF:MINUS\ A\ B\ X)$, which can be evaluated in either of three directions: each of the variables can be computed while the other two are bound. We will denote such binding patterns as `--+`, `+-`, `++-`, where '`-`' corresponds to the incoming bound variable and '`+`' corresponds to the variable that gets its binding as the result of predicate execution.

Another example is $(AREF\ A\ DIM\ IDX\ X)$, which can either compute a single slice `x` when the original array `A`, dimension `DIM` and slice index `IDX` are bound (pattern `---+`), or compute all possible slices of array `A` in the dimension `DIM`, together with the corresponding slice indexes `IDX` (pattern `--++`). The fanout estimates are largely different, while the cost of generating each result is the same for both binding patterns.

5.1.2.3 Execution plan

The task of the query optimizer is to find an optimal execution plan, by selecting the right (i.e. correct and most suitable) order of the predicates, and thus determining their binding patterns and the transition of information

through the variables. In this case, the *Nested Loop Join* operator is used to implement the conjunction.¹⁰

The stored GRAPH predicate can be regarded as a relational table with four columns that is repeatedly joined with itself, in order to retrieve the realization instances and the values of their properties. The first appearance of GRAPH in the execution plan will bind the `realization` variable, while the subsequent ones will use the discovered bindings for that variable.

The query optimizer (and SSDM in general) has no way of knowing the implicit RDF schema shown in Figure 9, so it is prepared to encounter, for example, multiple `ex:parameter_A` values for a single realization instance, and multiple `ex:result` values independently. Hence, the *Nested Loop Join* approach takes care of the possible multiplicity of properties, producing a result for each discovered combination.

```
(*SELECT* ID+ _V29+) <-
(NESTED-LOOP-JOIN
  (HASH-INDEX-SCAN GRAPH-+++
    0 REALIZATION+
    # [URI "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"]
    # [URI "http://udbl.uu.se/ex#OurExperimentRealization"])
  (HASH-INDEX-SCAN GRAPH-+++
    0 REALIZATION- # [URI "http://udbl.uu.se/ex#parameter_A"] A+)
  (CALL GE-- A- 0.25)
  (HASH-INDEX-SCAN GRAPH-+++
    0 REALIZATION- # [URI "http://udbl.uu.se/ex#initialState"]
    INITIALSTATE+)
  (CALL NMA-AGGREGATE--+ INITIALSTATE- 4 _V26+)
  (CALL LT-- _V26- 0.75)
  (HASH-INDEX-SCAN GRAPH-+++
    0 REALIZATION- # [URI "http://udbl.uu.se/ex#id"] ID+)
  (HASH-INDEX-SCAN GRAPH-+++
    0 REALIZATION- # [URI "http://udbl.uu.se/ex#result"] R+)
  (HASH-INDEX-SCAN GRAPH-+++
    0 REALIZATION- # [URI "http://udbl.uu.se/ex#iterations"]
    ITERATIONS+)
  (CALL PLUS-- 1 _V27+ ITERATIONS-)
  (CALL NMA-PROJECT-+++ R- 1 _V27- _V28+)
  (CALL NMA-AGGREGATE--+ _V28- 2 _V29+)))
```

The same execution plan is represented graphically in Figure 11. The vertical order of predicate boxes corresponds to their order in the nested loop, the 'output' or free variables are underlined, and the arrows show the data dependencies among the predicates. The stored predicate GRAPH appears with the *hash-index-scan* access method, and the foreign predicates are accessed by calling their implementations, corresponding to the binding patterns chosen.

¹⁰ In general, the Amos II optimizer may choose to implement the conjunction of predicates differently, e.g. with a *Merge Join* operator,

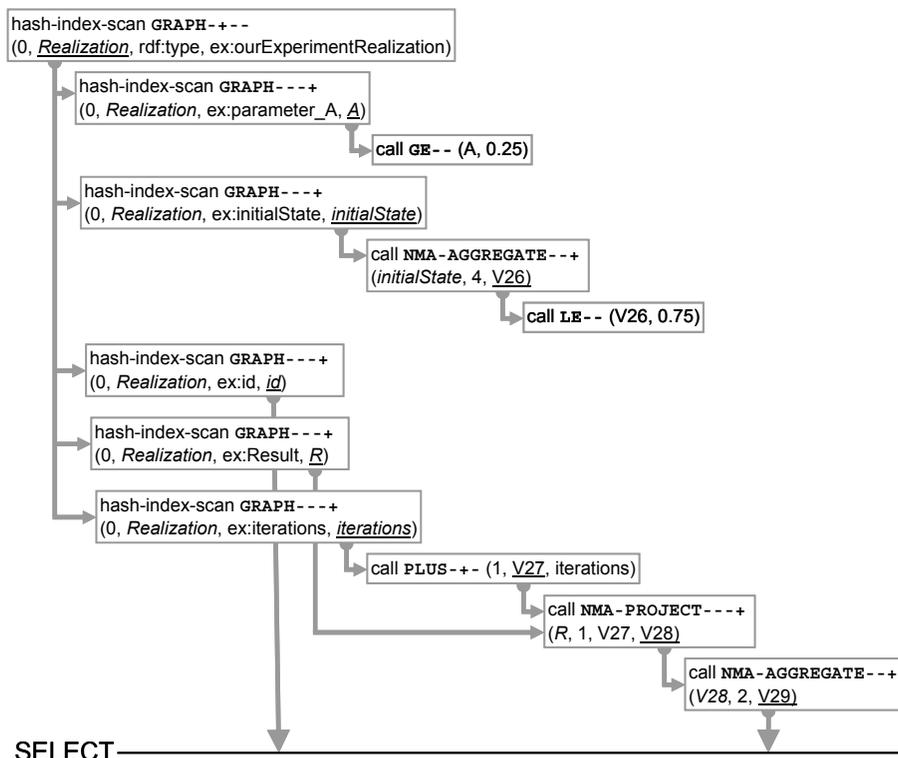


Figure 11. The graphical representation of *Q1* execution plan

The query optimizer wisely chooses to find the `ex:realization` instances using the predicate with a single unbound variable - a type predicate. Next, parameter *A* value is extracted, so that the simple inequality filter can be applied as early as possible. Next, the array value of the `ex:initialState` property is extracted from the RDF graph, and an intra-array aggregate function is evaluated - for another filter to take place. Given both filters are satisfied, the remaining properties are extracted and post-processing is performed.

5.1.2.4 Array operations

The execution plan above contains three array operations - two calls to the generic external function `NMA-AGGREGATE`, implementing built-in array condensers like `array_max()` and `array_avg()`, and one call to `NMA-PROJECT`. The principal difference between these two functions is that `NMA-AGGREGATE` actually needs the array contents in order to compute the result, while `NMA-PROJECT` does not.

As explained in Section 5.2, a memory-resident array consists of two objects - an *array descriptor* and an *array storage object*. A new *array descriptor object* is created whenever an operation like `NMA-PROJECT`, or

any other operation selecting a subset of an array, or altering its shape in some other way is applied. The new descriptor will be pointing to the same *array storage object*, so that the massive numeric data making up the array contents is not copied or even accessed.

Furthermore, when an array is stored externally, it is represented by an *array proxy* object in SSDM, which is very much similar to an *array descriptor*, except that it contains the information necessary to identify the external storage system (a back-end, or a wrapped database, or a linked file) and to locate a particular array instance within it. Whenever an operation like `NMA-PROJECT` is applied to an *array proxy*, a new *array proxy* object is created, and this storage-relevant information is copied.

Hence, the actual array data retrieval from a linked file may be done just before the call to `NMA-AGGREGATE` in case of externally stored `ex:result` arrays. This data retrieval is implemented by the *array-proxy-resolve*, `APR()` function, which materializes the specified subset of an external array, based on the information in an *array proxy*, into a *memory-resident array*. The general approach to implement `APR()` for arbitrary array storage systems is described in Section 6.1.

One benefit of this lazy data retrieval is that typically only a small subset of an array happens to be needed for the actual computation. In our example, we only retrieve the last iteration's result from each simulation. Another reason is that an execution plan might contain additional filter conditions between generating array proxies and materializing them. Section 6.2 discusses further optimizations of this process by aggregating these resolve operations into pipelined streams in case of SQL-based storage back-ends.

5.2 Numeric Multidimensional Arrays

The formal definition of the *Numeric Multidimensional Array* (or *array* for short) is given in the beginning of Section 4.1, followed by the definitions of array operations. This section explains implementation details of arrays and array operations in SSDM

5.2.1 Storage of Resident Arrays

In SSDM's native main-memory data storage, arrays are represented as *descriptor objects* referring to *storage objects*, as shown in Figure 12. A storage object compactly stores array elements in continuous memory, while descriptor objects provide very space efficient representations of derived arrays. This allows us to compute derived arrays without copying or otherwise accessing the array contents.

A storage object represents a one-dimensional array of *Integer*, *Double*, or *Complex* numbers. It contains a small header storing the element type. 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 directly pointing to the storage object of the original array. Descriptor objects are automatically freed by the garbage collector whenever no variable or object refers to it. The garbage collector frees the storage object when the last descriptor object referring to it is freed.

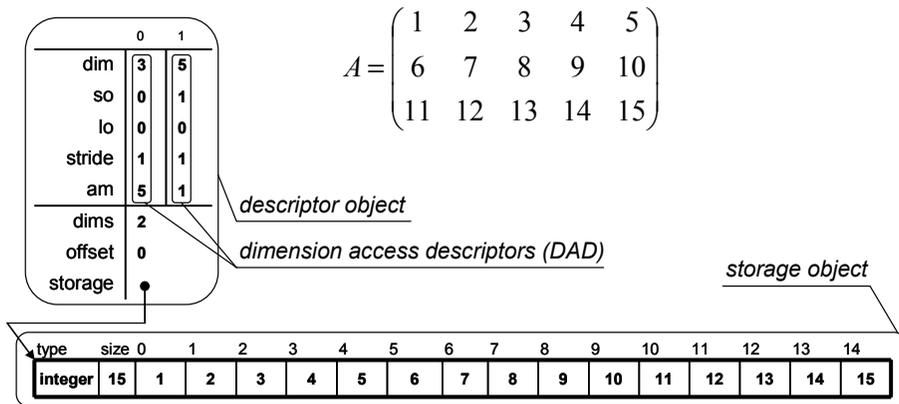


Figure 12 In-memory array representation

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 $a(i_1, \dots, i_n)$ that maps the array subscripts to the *storage index* takes the form:

$$a(i_0, \dots, i_{n-1}) = \sum_k (i_k - 1) \cdot \prod_{\substack{m \\ so_m > so_k}} \dim_m$$

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

$$am_k = \prod_{\substack{m \\ so_m > so_k}} \dim_m$$

In the example in Figure 12, $am_0=5$ and $am_1=1$, so that element $A[2,3]$ would be dereferenced to storage index $a(2,3) = 7$, containing the element 8.

In the most general case, 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_A for accessing one element (i_1, \dots, i_n) of the array A is:

$$a_A(i_1, \dots, i_n) = offset + \sum_k p_k^A(i_k) \cdot am_k \quad (1)$$

where

$$p_k^A(i_k) = lo_k + (i_k - 1) \cdot stride_k$$

The function $p_k^A(i_k)$ projects a logical subscript i_k of the derived array A to a (0-based) logical subscript of the basic array.

For example, the element $D[3]$ of the subarray e presented in Figure 13d below would be addressed as $a_D(3)=7$, which corresponds to the same element 8.

5.2.2 Array Transformations

Below we describe the three central array operations capable of producing new descriptor objects (or, similarly, new array proxies). These array transformations do not access the array content, whether it resides in a main-memory *storage object* or is externally stored. The fourth operation - array element access, is also *delayed* in case of external arrays.

1) *Permutation of dimensions* is a multidimensional 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, as illustrated in Figure 13b.

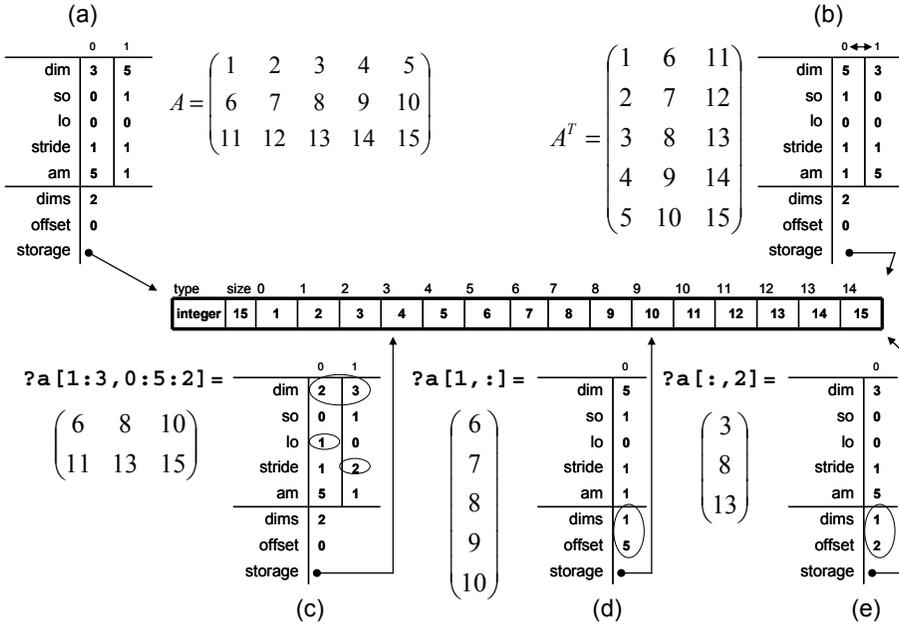


Figure 13. Array transformations in terms of descriptor objects

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

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

$Permute$ is a SciSPARQL function that can be used in a SELECT clause or FILTER expression. Matrix transposition is defined by the functional view:

```
DEFINE FUNCTION Transpose(?matrix)
  AS SELECT Permute(?matrix, 1, 0);
```

2) **Slicing** is an operation that can be applied to each array dimension independently, resulting in an array subset specified by subscripts and a stride, as shown in Figure 13c. Given an n -dimensional numeric array A , the operation $Sub_k(A, lo_k, hi_k, stride_k)$ results in an derived array B of the same dimensionality, where the first element is defined by lo_k :

$$a_A(0, \dots, lo_k, \dots, 0) = a_B(0, \dots, 0)$$

Effectively, the lo value in the k -th DAD of the resulting array B is:

$$lo_k^B = p_k^A(lo_k),$$

Analogously, the iteration stride of B is multiplied by the $stride_k$ argument:

$$stride_k^B = stride_k^A \cdot stride_k$$

The dimensions of B are defined as:

$$\dim_k^B = \left\lfloor \frac{p_k^B (hi_k - 1) - lo_k^B}{stride_k^B} \right\rfloor + 1$$

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

$$\begin{aligned} Sub_k(Sub_l(A, lo_l, hi_l, stride_l), lo_k, hi_k, stride_k) &= \\ = Sub_l(Sub_k(A, lo_k, hi_k, stride_k), lo_l, hi_l, stride_l) & \end{aligned}$$

for any dimension indices $k \neq l$ and the valid lo , hi and $stride$ values.

3) Projection involves reducing the dimensionality of an array by selecting one subscript value in a specified dimension - either row (Figure 13d) or column (Figure 13e) 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 $Pr_k(A, i_k)$ results in a derived array B where the offset references the first element of B :

$$offset_B = a_A(0, \dots, i_k, \dots, 0)$$

Similarly to slicing, the projection operations across different dimensions can also be arbitrarily superimposed. However, since the number of logical dimensions changes, the dimension enumeration has to be adjusted:

$$Pr_k(Pr_l(A, u), v) = \begin{cases} Pr_l(Pr_{k+1}(A, v), u), & k \geq l \\ Pr_{l-1}(Pr_k(A, v), u), & k < l \end{cases}$$

Projecting a 1-dimensional array results in an atomic value (in case of memory-resident arrays) or a 0-dimensional *array proxy* (see Section 6.1) pointing to such an element in an externally stored array.

4) Element access can be regarded as an ultimate superposition of array projections (and as such, a particular case of array projections). An element of array A identified by a vector of *logical subscripts* $\langle i_1, \dots, i_n \rangle$ can be defined as

$$A[i_1, \dots, i_n] = Pr_1(\dots Pr_n(A, i_n) \dots, i_1)$$

Technically, however, the multidimensional *storage function* $a_A(i_0, \dots, i_{n-1})$ defined in the previous section is used to compute the

particular storage address in a memory-resident array, or the *offset* in a new *0-dimensional array proxy* object used to represent single elements of externally stored arrays.

5.3 Data Loaders

There are two basic ways to load *RDF with Arrays* data into SSDM:

- using W3C SPARQL 1.1 `INSERT` and `INSERT DATA` update statements, or
- loading RDF data from *Turtle* or *NTriples* text-based formats.

In the latter case, array data is either

- *linked* using Turtle file links (introduced by SSDM), or
- *consolidated* from one of the standard RDF representations of the multidimensional numeric data, including RDF collections and RDF Data Cube datasets.

The Turtle file reader in SSDM also supports NTriples [20] as a subset of Turtle. Local files and files available on the Web via HTTP can be loaded into a default or named RDF graph in the database using the `LOAD()` directive. The parsing is performed in a streamed way, (which is also the case with `INSERT DATA` statements), so arbitrarily large Turtle files are supported.

5.3.1 File Links

According to the W3C RDF standard, a *value* in a *subject-property-value* RDF triple can either be a *URI*, a *blank node*, or an *RDF Literal*. A construct like

```
<file://realization_1.mat#Res>
```

is formally a URI, and hence would raise no error if loaded into another RDF Store. SSDM tries to interpret it as an array value in the respective triple.

First, the Turtle reader extracts the file extension and checks whether an array reader plug-in is registered to handle that kind of files. SSDM is extensible with *array readers* for different file formats, and any such array reader is free either to immediately load the array into memory or produce an *array proxy*, based e.g. on the array size or other properties. In the latter case, the corresponding *array-proxy-resolve* routine also needs to be registered for this particular kind of *array proxies* - this option is described in Section 6.1.

SSDM checks that the file is available (in this case, the file `realization_1.mat` in the SSDM server's file system), and calls the array reader, passing the part of the file link after '#' as a parameter. In the example scenario described in 4.1, the `.mat` file reader accesses the file to check whether the variable `Res` exists, extracts the type and shape of the array, and returns an array proxy containing the file and variable names, besides the usual *array descriptor* information. These proxies may be *resolved* later or immediately.

In case of no *array reader* registered, or the file being not available, or file links are disabled altogether when the `_sq_resolve_file_links_` flag is *false*, the URI from the Turtle file will be stored as a *value* of the RDF triple, treating it in the same way as any standard RDF store.

5.3.2 RDF Collections

RDF collections are described in Section 2.3.5.1 and might be used to explicitly represent multidimensional array data in a Turtle file. The main problem is that they are merely 'syntactic sugar' introduced by the Turtle format to compactly represent such collections with large numbers of the underlying RDF triples.

For example a Turtle triple

```
:s :p ((1 2) (3 4)) .
```

masks 12 additional RDF triples and 6 introduced blank nodes, as shown in Figure 4. All this underlying information can be consolidated into a 2x2 array of integers, and SSDM does that.

An RDF collection would be identified as an array if the following conditions about it hold:

- each element is either a number or another collection;
- all numbers appearing in collections are nested on the same level, and only numbers do appear at that level;
- a uniform number of elements in collections are nested on each level.

The widest numeric type among the values found in the collection will be used as the array element type.

For example, the *value* of the Turtle triple

```
:x :a ((1 2.25 3) (4 5 6)) .
```

will be represented with 2x3 array of real numbers in SSDM.

As another example, the value of the Turtle triple

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

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

This physical compression adds new array-based semantics to collections, while retaining the original linked-list semantics. The standard `rdf:first` and `rdf:next` predicates are redefined over arrays, virtually connecting them with their respective subsets. Such polymorphism leads to an interesting optimization problem, discussed in Section 5.5, together with current solutions.

5.3.3 Data Cube Vocabulary

As the Semantic Web mainly concentrates on providing a framework for publishing metadata, the RDF Data Cube [133], introduced in Section 2.3.5.2 provides a rich vocabulary for:

- defining the data structures - using the classes
`qb:DataStructureDefinition`, `qb:ComponentSpecification`;
- defining flat and hierarchical enumerations, a.k.a. code lists - using the classes `skos:ConceptScheme`, `skos:Concept`;
- identifying the instances of Data Cube datasets (with an open set of metadata attached) - using the class `qb:DataSet`;
- storing the observations (array elements) in terms of dimensions, measures and attributes - using the class `qb:Observation`;
- defining standard slices, e.g. time series, snapshots, or otherwise grouping the observations - using the classes `qb:SliceKey`, `qb:Slice`, `qb:ObservationGroup`.

The RDF Data Cube encodes the array data results in much bigger RDF graphs, compared to RDF collections. It suggests no particular nesting order, and might be better suited for sparse array data. For example, the 2x2 matrix shown on Figure 4 would take 5 RDF triples per cell:

```
ex:o4 a qb:Observation ;
      qb:dataSet ex:dataset1 ;
      ex:i 2 ; ex:j 2 ; ex:value 4 .
```

with the additional structural metadata in place:

```
ex:dataset1 a qb:DataSet ;
            rdfs:Label "My 2x2 matrix example" ;
            qb:structure ex:dsd1 .

ex:dsd1 a qb:DataStructureDefinition;
        qb:component
          [ qb:dimension ex:i ; order 1 ],
          [ qb:dimension ex:j ; order 2 ],
          [ qb:measure ex:value ] .
```

A `qb:DataStructureDefinition` instance defines the components of a data cube - i.e. its dimensions, measures, and attributes. Attributes may be attached to the dataset as a whole or to particular slices. For example, a unit of measure is semantically an attribute of each measurement, while it can be stored once for a given dataset, to avoid redundancy. Dimension values may also be attached to particular slices. Hence, an RDF Data Cube can be expressed as a graph in equivalent *normalized* and *abbreviated* forms. The concept of 'normalization' is defined as somewhat opposite to the normalization in relational databases: dimensions, whose values are only given once per slice (or attributes whose values are only defined once per dataset), are said to form an *abbreviated* Data Cube. The equivalent *normalized* cube will replicate these values for each individual measurement.

SSDM *consolidates* RDF Data Cube datasets, drastically reducing the graph size, while preserving all information therein. New array properties, containing the numeric data extracted from the observations, will be attached to the corresponding `qb:DataSet` instance. The property names would be the same as the component names used in the original observations (`ex:value` in our example). The distinct dimension values will be sorted and attached as collections to the same `qb:DataSet` instance. For numeric dimensions these lists will automatically be represented as 1D arrays. However, if for a particular dimension all values are positive integers, their set is contiguous and includes 1 (as for `ex:i` and `ex:j` dimensions in our example), no mapping is needed, and those values can be used directly as array subscripts.

The conversion is done in two phases: first, the distinct dimension values are collected, and then the allocated arrays are filled with observation data. No `qb:Observation` nodes need to be stored anymore, however, they may still be inferred (in other words, virtually reconstructed) when processing basic SPARQL queries.

A realistic example is given in [133], both in abbreviated (with slices) and normalized forms. The dataset represents 24 numeric observations (life expectancy per region, per time period, per gender), along with the structural and publication-related metadata, in accordance with the SDMX [147] practices for the statistical data modelling. The abbreviated form consists of 206 triples. Once read into SSDM, 150 of these triples are *consolidated* into a single array-valued triple, and three RDF collections for dimension values enumeration (total 9 ordered values)¹¹.

¹¹ Currently, SSDM uses RDF collections to store the ordered sets of non-numeric values, though a generalized array type might help further reducing the graph size in this example - the current approach adds 18 triples to store RDF collections as linked lists.

Figure 14 below shows this Data Cube example the way it is represented in SSDM (most `rdf:label`, `rdf:comment`, and `dct:description` properties are omitted). The central node is `eg:dataset-le3`, representing the dataset instance. The top-left part of the figure is occupied by the publication-related metadata, which is crucial to finding this dataset among the others on the web. The bottom half of the picture shows the definition of the dataset's structure - as an instance of `qb:DataStructureDefinition`, with its `qb:component` properties: dimensions, measures, and attributes. Some of these are borrowed directly from the SDMX vocabulary, while some others are mapped to the corresponding statistical concepts via the `qb:concept` properties.

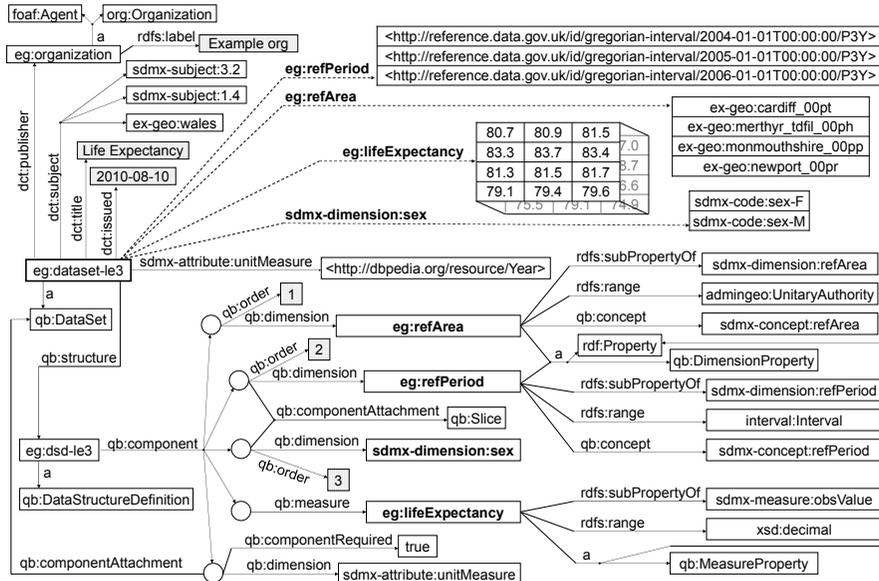


Figure 14. The *RDF with Arrays* consolidation of the RDF Data Cube example dataset from [133]

The top-right part of Figure 14 shows the parts of the graph (with dashed arrows) added by SSDM, consolidating the data from the 150 original triples. This includes the array-valued property `eg:lifeExpectancy`, storing the measure, and three collection-valued properties, storing the dimension values. Notice that RDF node URIs shown in bold font, associated with the `qb:DataStructureDefinition` instance via blank nodes, are also used as property URIs for the corresponding `qb:DataSet` instance - this is the standard RDF Data Cube way of defining and using custom properties.

Effectively, as a result of consolidation, all Data Cube components become attached to the `qb:DataSet` instance - whereas some remain single-valued (the `sdmx-attribute:unitMeasure` attribute), some become ordered

sets (the dimensions), and some become multidimensional numeric arrays (the `eg:lifeExpectancy` measure in this example). Slices are no longer used for abbreviation - instead, the array representation of the multidimensional data allows extracting slices on demand, e.g. for the purpose of aggregation.

Consolidating RDF Data Cube datasets helps to drastically reduce the graph size, providing physical-level separation of data and metadata. Under certain SSDM configurations, the resulting arrays might be stored in a specialized back-end, while metadata is retained in memory. Another important benefit is speeding up pattern-matching queries, by letting them deal with much smaller RDF graphs.

5.4 Scientific SPARQL Query Processor

As illustrated by the example in Section 5.1.2, processing a SciSPARQL query involves a number of steps:

1. Translation to AmosQL
2. Parsing to ObjectLog representation
3. Flattening and applying rewrites
4. Cost-based optimization
5. Execution runtime

This section mainly explains steps 1 and 5, which are largely interconnected, and most of the work on implementing a SciSPARQL query interpreter was concentrated there.

As for the intermediate steps 2 - 4, SSDM largely relies on the underlying Amos II functionality. In this section we show that it is possible to implement the complete SPARQL semantics (as defined by the W3C standards [155, 156]) using the general Datalog-based query processing architecture. Some interesting semantic mismatches were encountered, and some of the solutions involved extensions to AmosQL, the predicate calculus representation, and the query optimizer.

The translation process is based on the internal representation of the query structure. The process includes:

- parsing the string representation of a SciSPARQL query, using an SLR(1) ascending parser, resulting in a data structure Q ;
- performing a number of transformations on Q , collecting lists of variables, applying expression rewrites;
- generating the string representation $tr(Q)$ of the generated AmosQL query, using a recursive tree-to-text writer.

This section describes the SciSPARQL query structure in a formal way, discusses the algebraic interpretations of SciSPARQL along with the related concept of *well-designed queries*, and explains the structure of AmosQL queries used for the translation and the extensions made to AmosQL. It concludes with a formal definition of the translation function $tr(Q)$. More examples are introduced to illustrate important concepts as well as the translation cases.

5.4.1 SciSPARQL Query Structure

Following is a formalized description of the W3C SPARQL 1.1 query structure, which completely describes the SciSPARQL query structure as well. The extensions introduced by SciSPARQL are the new kinds of expressions (e.g. array expressions), and the new kinds statements (e.g. the `DEFINE FUNCTION` statement). Also, any SciSPARQL function call with arguments explicitly bound is also a valid SciSPARQL statement - e.g. directives like `LOAD()`, `SOURCE()`, etc. are, syntactically, top-level function calls.

According to the W3C SPARQL specifications, there are four basic types of queries, differing in the kind of output they produce:

- ASK queries, returning a *Boolean* value;
- CONSTRUCT and DESCRIBE queries, returning sets of triples (i.e. resulting RDF graphs);
- SELECT queries, returning sets of bindings for the output variables.

In SPARQL the result of a `SELECT` query always includes names of variables, hence the `SELECT` list may contain either query variables, or *named expressions*, where the result of an expression is bound to a new variable. Example *Q1* contains a named expression, bound to variable `?res`.

For the purpose of translating SciSPARQL queries, the query structure needs to be represented as a data structure, including named fields, sets, sequences etc. This section describes the structure of SciSPARQL queries on such conceptual level. The query outline, conditions and expressions have their corresponding logical representations inside SSDM query translator.

5.4.1.1 Basic query structure

An internal representation of a SciSPARQL query Q , on the most basic level, includes the following components:

- *Q.type*: either `SELECT`, `CONSTRUCT`, `DESCRIBE` or `ASK` query;
- *Q.what*: a list of returned variables or *named expressions* for a `SELECT` query, a list of result-generating triple patterns for a `CONSTRUCT` query, or a single RDF term for a `DESCRIBE` query.

In case of SELECT queries, we will further refer to this list of expressions as a *select* list.

- ***Q.from***: a list of RDF graph names identified by URIs. These graph names are translated to AmosQL function names and arguments as a part of the triple pattern translation. For example, `GRAPH(2)` for a certain named graph, `GRAPHS({2,4,5})` for a set of merged graphs, and `GRAPH(URI2GraphId(URI('http://udbl.uu.se/example')))` for a triple pattern inside a SciSPARQL user-defined function (the RDF graph does not have to be in the dictionary at the time of function definition).
- ***Q.where***: a list of conditions, typically represented by single *condition block*, or a list of *condition blocks*, as explained below.
- ***Q.distinct***: whether the `DISTINCT` option is specified for the query in general. It controls whether only distinct results will be emitted from the query.
- ***Q.orderby***: a list of variables on which to perform sorting of the query results, where each variable supplemented with a direction flag. The sorting is done by default in ascending order, and if multiple variables are specified, each following variable is only used for comparison if all previous variables have equal values across the two *result rows* being compared.
- ***Q.offset*** and ***Q.limit*** expressions, defining the desired partitioning of the query results. These must evaluate to *Integer* values, and can not depend on variables bound inside the query (i.e. they must be constant in SELECT queries, and may depend on function parameters in SciSPARQL function views).

5.4.1.2 Aggregate query structure

Built-in SciSPARQL functions are categorized into aggregates, e.g. `SUM()` or `MAX()`, which operate on bags of argument values, and non-aggregate functions, e.g. `round()`, `mod()`, which normally return results for separate argument bindings. Similarly, functional views and foreign functions in SciSPARQL (see Chapter 4) are defined using either `DEFINE FUNCTION` or `DEFINE AGGREGATE` syntax, hence SSDM is able to recognize the aggregate functions used in queries.

Based on this, there are two kinds of SELECT queries, differing in semantics and translation approach used: basic queries and *aggregate queries*. The latter, by definition, have at least one aggregate function call used in the `SELECT` or `HAVING` clause. ***Q1*** above is an example of a basic SELECT query, while ***Q2*** below is an example of an *aggregate query*:

Q2 (standard W3C SPARQL 1.1): Select the number of realizations and the total number of iterations for each distinct value of parameter *A* with multiple realizations stored:

```

PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?a (COUNT(?r) AS ?realizations)
          (SUM(?iterations) AS ?total_iterations)
WHERE { ?r a ex:OurExperimentRealization ;
          ex:iterations ?iterations ;
          ex:parameter_A ?a }
GROUP BY ?a
HAVING ?realizations >= 2
ORDER BY ?a DESC(?total_iterations)

```

Aggregate queries have *inner* and *outer* layers, each having their own lists of variables, conditions, *select* expression lists and *distinct* flag. Graph pattern matching is only performed in the *inner layer*, while result ordering and partitioning are only done in the *outer layer*.

Q2 contains two aggregate function calls in *named expressions* in the *select* list, and both variables defined by these *named expressions* are additionally used in the *outer layer* - one in the **HAVING** and one in the **ORDER BY** clause. The only variable used on both levels is the ?a variable, listed in the **GROUP BY** clause. Other *inner* variables cannot be used in **SELECT**, **HAVING** or **ORDER BY**, except in argument expressions to the aggregate functions.

A special case of an aggregate query is a so-called *total aggregate* query, with no **GROUP BY** or **HAVING** clauses provided, so that the results always get assembled into at most one group. If the underlying group of solutions is non-empty, such queries always return a single result. **Q3** below illustrates the concept:

Q3 (standard W3C SPARQL 1.1): Compute the number of distinct values for parameter *A*:

```

PREFIX ex: <http://udbl.uu.se/ex#>
SELECT (COUNT(DISTINCT ?a) AS ?result)
WHERE { ?r a ex:OurExperimentRealization ;
          ex:parameter_A ?a }

```

Another peculiar feature of **Q3** (defined in the SPARQL 1.1 standard) is the use of the keyword **DISTINCT** before the argument to the aggregate function **COUNT()**. This enforces the application of the **DISTINCT** option to the results of the inner query, so its **SELECT** clause might look like

```
SELECT DISTINCT ?a
```

while the outer query simply applies the **COUNT()** on the resulting bag of solutions.

Formally, aggregate queries have the following additional structural components:

- ***Q.groupby***: a list of *inner* variables to perform grouping on. The order of variables is not important, as a single *solution* for the *outer level* of the query will be assembled for each group of solutions of the *inner level*, where values of the listed ***groupby*** variables remain the same;
- ***Q.inner-distinct***: whether a `DISTINCT` option was used under any aggregate function, as illustrated by ***Q3*** above;
- ***Q.having***: a filter condition applied on the solutions of the outer query. Any kind of expression is allowed and the *Effective Boolean Value* is used (see Section 3.3.3).

There are more fields introduced while rewriting *Q*, as explained in the following sections. This includes ***Q.agg*** list of aggregate expressions, and ***Q.select-extra*** to accommodate for ordering on variables/expressions that do not appear in *Q.select*.

5.4.1.3 Condition block structure and sets of variables

Most non-trivial queries have a `WHERE` clause, effectively containing a conjunction (although the order sometimes matters, see Section 5.4.2) of:

- triple and path patterns,
- `FILTER` conditions,
- explicit bindings with `BIND` and `VALUES` constructs,
- `OPTIONAL` *blocks*,
- disjunctive *blocks* introduced with `UNION`,
- nested blocks with the `GRAPH` specifier (nested *blocks* without this specifier can be flattened and merged in the parent *block*),
- subqueries projecting variables into the basic query.

These conditions are syntactically grouped into non-empty ***blocks***, and each block may be annotated by a `GRAPH` specifier. According to the W3C SPARQL 1.1 standard, there is always one parent block in the `WHERE` clause, provided without any `GRAPH` specifier.

These blocks as concepts are represented directly in the SciSPARQL translator. The ***where*** part of a query, if non-empty, always contains a single top-level ***block*** structure. The block serves as a container for the sequence of conditions, together with the sets of variables important for the translation, including *bound*, *partially bound* and *referenced* variables, which are defined here. The definitions are recursive, as they rely on the same properties of the nested blocks, and a tree-traversal algorithm is used to build these sets.

A variable *v* is ***bound*** in block *B* if and only if at least one of the following conditions is satisfied:

- it participates in a triple (or path) pattern inside *B*,
- it is *bound* in a nested sub-block with a `GRAPH` specifier,
- it is returned to *B* from a subquery,

- it is *bound* in all of the branches of a UNION condition inside *B*,
- it is explicitly assigned a value (or a set of values) by VALUES condition inside *B*, or
- it is assigned a result of an expression depending only on *bound* variables with a BIND condition inside *B*.

An important superset of *bound* variables is the set *Partial(B)* of *partially bound* variables. A variable *v* is ***partially bound*** in block *B* if and only if at least one of the following conditions is satisfied:

- it is *bound* in block *B*,
- it is *partially bound* in an OPTIONAL sub-block of block *B*,
- it is *partially bound* in any of the branches of a UNION condition in block *B*, or
- it is assigned a result of an expression depending only on *partially bound* variables with a BIND condition inside *B*.

Note that despite its name (and due to the lack of a better word), in any block *b* the set *partially bound* variables includes the set of *bound* variables. We call the set difference as a set of ***semibound*** variables:

$$\text{Semibound}(B) = \text{Partial}(B) \setminus \text{Bound}(B)$$

Semibound variables are exactly those which might be bound or not bound in solutions for block *B*.

Finally, we define the set *Ref(B)* of ***referenced*** variables inside block *B*, as those variables participating in conditions of any kind inside the block, including FILTER expressions, OPTIONAL and UNION sub-blocks. This, however, does not include the 'internal' variables of subqueries - only the variables selected from a subquery inside *B* are *referenced* and *bound* in the *B*. The set of *referenced* variables for any block *B* subsumes the other three sets.

Besides the sets of *bound*, *partially bound*, and *referenced* variables, a block also lists *B.blanks* - the set of variables introduced to represent *blank nodes* in a SPARQL query.

All the above sets are defined for the *inner layer* of a query, since the *outer layer* of aggregate queries does not feature any blocks per se. In the above examples ***Q1 - Q3***, all variables *referenced* in the WHERE clause are also *bound* there, and there are no semibound or free variables. An example featuring *semibound* variables must include an OPTIONAL or UNION construct, like ***Q4*** and ***Q5*** below:

Q4 (Standard W3C SPARQL): Select all realization ids and parameters *A* and *C*, if the latter is applicable:

```
PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id ?a ?c
```

```

WHERE { ?r a ex:OurExperimentRealization ;
           ex:id ?id ;
           ex:parameter_A ?a .
OPTIONAL { ?r ex:parameter_C ?c } }

```

Two blocks constitute the WHERE clause of this query. In the nested OPTIONAL block variables ?r and ?c are *referenced* and *bound*, and in the parent block all query variables are *referenced*, while ?r, ?id, and ?a are *bound*, and ?c is *semibound*.

The result of this query may contain absent bindings for certain variables. For the example dataset **G1** in Section 5.1.1, this query will produce the following two partial mappings for its *select* variables:

?id	?a	?c
1	0.3	
314	0.3	0.9

Q5 (Standard W3C SPARQL): Select ids of all realizations with B or C parameters values applicable, together with their parameter values:

```

PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id ?b ?c
WHERE { ?r a ex:OurExperimentRealization ;
           ex:id ?id .
           { ?r ex:parameter_B ?b }
UNION
           { ?r ex:parameter_C ?c } }

```

Q5 contains three blocks, one basic block with ?r and ?id variables bound, and two UNION branches, additionally binding variables ?b and ?c, respectively. According to [122], this query is equivalent to a *union* of two queries, each containing one of the union branches. Their respective solutions would be:

?r	?id	?b
_:r1	1	0.85
_:r314	314	0.9

 \cup

?r	?id	?c
_:r314	314	3.14

A *union* of multisets of query variable *mappings* is different from the *relational union* operator: it does not require the same sets of variables being mapped in the operand multisets - i.e. it does not rely on the concept of *relational compatibility*. It simply appends the multisets, as if the queries were evaluated independently, hence the result of **Q5** being:

?id	?b	?c
1	0.85	
314	09	
314		3.14

Note that the sets of variables defined in this section are defined for the whole blocks, and do not depend on the particular place inside a block where the translation takes place. This assumes a declarative nature of SciSPARQL, which is not always the case, since SciSPARQL completely incorporates the semantics of the W3C SPARQL 1.1 standard. There are certain cases where the order of the conditions (in the standard language) does matter, as we discuss in the next section.

5.4.2 Compositional vs. Operational SPARQL Semantics

Let us introduce some notation first. We will denote query blocks by letters A, B, C, \dots . A **simple graph pattern** is a SciSPARQL query block which does not contain nested `OPTIONAL` or `UNION` sub-blocks. It consists of a conjunction of triple patterns.

Without loss of generality, we can include `GRAPH`-annotated sub-blocks as additional sets of triple patterns, if we define a **triple pattern** as a quad $\langle G, s, p, v \rangle$ where $s, p,$ and v are either RDF terms or variables, and G is either a finite non-empty set of (default and/or named) RDF graphs, or a variable whose potential solutions are the graphs listed as `FROM NAMED` in **from** part of the query.

There are operators defined on such blocks, reflecting the `OPTIONAL` and `UNION` relationships. For example, the **where** part of $Q4$ can be described as $A \text{ OPT } B$ where

$$\text{bound}(A) = \{?realization, ?id, ?a\},$$

$$\text{bound}(B) = \{?realization, ?c\}$$

Since the `OPT` operator implies nesting of blocks, it is right-associative, i.e.

$$A \text{ OPT } B \text{ OPT } C = A \text{ OPT } (B \text{ OPT } C)$$

is a double nesting of an `OPTIONAL` block, whereas

$$(A \text{ OPT } B) \text{ OPT } C$$

describes a basic block with conditions from A and two `OPTIONAL` sub-blocks B and C , not nested into each other.

By $A \text{ AND } B$ we denote a simple conjunction of conditions from A and B , i.e. effectively merged block. We will also use the notation

$$A \text{ AND } (B \text{ U } C)$$

for a basic block with conditions from A and a `UNION` with branches B and C . According to [122], a query can always be normalized to DNF, with all unions pushed to the top level, using the properties

$$A \text{ AND } (B \text{ U } C) = (A \text{ AND } B) \text{ U } (A \text{ AND } C)$$
$$(A \text{ U } B) \text{ OPT } C = (A \text{ OPT } C) \text{ U } (B \text{ OPT } C)$$
$$A \text{ OPT } (B \text{ U } C) = (A \text{ OPT } B) \text{ U } (A \text{ OPT } C)$$

Finally, there are two (in most cases identical) sets of solutions for a SPARQL query block A

- a **compositional solution**, defined by the W3C SPARQL standard [155], here denoted as $[[A]]$, and explained in Section 5.4.2.2
- an **operational solution**, here denoted as $eval(A)$ and explained in Section 5.4.2.3.

5.4.2.1 Example

The following example is borrowed from [40] and is a basic RDF graph with four isolated star-shaped components. We will refer to it as graph G_2 and, for uniformity, present it here in Turtle notation:

```
@prefix : <http://udbl.uu.se/ex2>

_:b1 :name "Paul" ;
     :phone "111 - 1111" .
_:b2 :name "John" ;
     :email "john@john.edu" .
_:b3 :name "George" ;
     :web <www.george.edu> .
_:b4 :name "Ringo" ;
     :email "ringo@ringo.edu" ;
     :web <www.starr.edu> ;
     :phone "444 - 4444" ;
     :cell "444 - 4444" .
```

G_2 is supposed to represent information about different persons; however, the information is structurally non-uniform, incomplete, and redundant. This would be a problem with a relational or object-oriented DBMS, but the Semantic Web / Linked Data solutions, by design, should handle these aspects in a graceful manner.

We begin by analyzing the following query, also borrowed from [40]. It might seem a bit contrived, but illustrates the potential problem. Its English formulation is not simple either, and is given later in Section 5.4.2.3.

Q6 (Standard W3C SPARQL):

```
PREFIX : <http://udbl.uu.se/ex2#>
SELECT ?x ?y ?z
WHERE { ?x :name "Paul" .
        OPTIONAL { ?y :name "George" .
                   OPTIONAL { ?x :email ?z } } }
```

Using the above-introduced block notation, the query structure can be reflected as $A \text{ OPT } (B \text{ OPT } C)$ where

$bound(A) = \{?x\}$
 $bound(B) = \{?y\}$
 $bound(C) = \{?x, ?z\}$

5.4.2.2 Compositional semantics

A SPARQL query $A \text{ OPT } (B \text{ OPT } C)$, where A , B , and C are simple graph patterns according to the W3C SPARQL 1.1 specifications should be evaluated as the following SPARQL Algebra expression

$$[[A \text{ OPT } (B \text{ OPT } C)]] = [[A]] \bowtie ([[B]] \bowtie [[C]])$$

where \bowtie is the relational *left outer join* operator defined on bags of solutions.

This evaluation process should consist of:

- 1) finding the sets $[[B]]$ and $[[C]]$ of mappings that satisfy B and C ,
- 2) *left-outer-joining* them, resulting in $[[B]] \bowtie [[C]]$,
- 3) finding the set $[[A]]$ of mappings that satisfy A , and
- 4) *left-outer-joining* it with the result of (2).

This process is illustrated in Figure 15a for query Q_6 and graph G_2 .

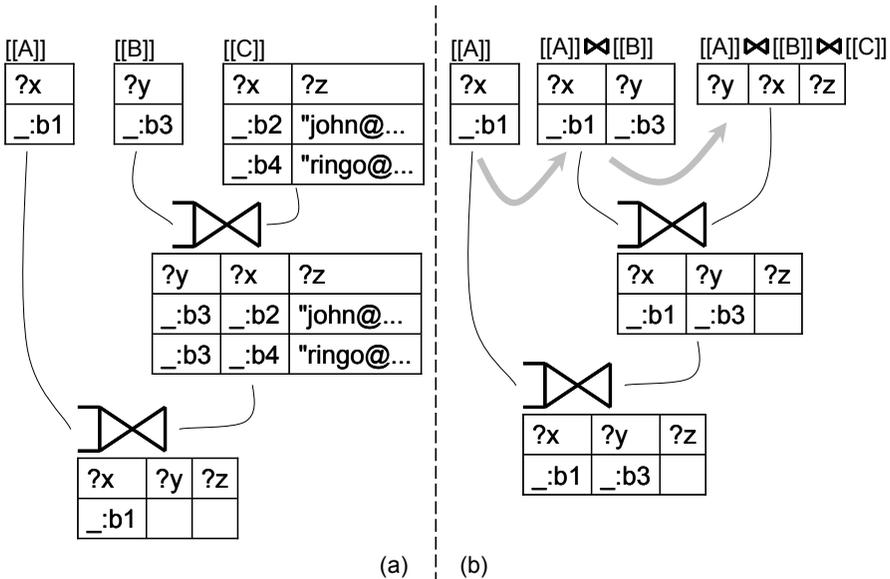


Figure 15. Compositional (a) and operational (b) evaluation order of query Q_6 over graph G_2

However, on large datasets this approach is generally inefficient. Even if we avoid storing intermediate results of $[[B]]$ and $[[C]]$, we will probably need storing $[[B \text{ OPT } C]]$ anyway (unless we are able to retrieve the results

in some known order for a merge-join algorithm to help). This is the known problem of *bushy* execution plan trees vs. *left-deep* trees [80].

5.4.2.3 Operational semantics

A traditional way to evaluate a query like **Q6**, adopted by present-day DBMSs, including SSDM, would include the steps to:

- 1) find the set of mappings that satisfy A , resulting in $eval(A) = [[A]]$;
- 2) for each result in $eval(A)$ try to extend it with mappings of variables $bound(B) \setminus bound(A)$, while the mappings of variables $bound(B) \cap bound(A)$ are retained, thus ensuring that the mappings are compatible and providing a more informative binding pattern for evaluating B . This results in $eval(A \text{ OPT } B)$;
- 3) for each result in $eval(A \text{ OPT } B)$ that was extended on the previous step¹², try to extend it with mappings of variables $bound(C) \setminus (bound(A) \cup bound(B))$, similarly to the previous step. The mappings of variables $bound(C) \cap (bound(A) \cup bound(B))$ are retained, ensuring compatibility of solutions and providing an even more informative binding pattern for evaluating C .

Effectively, by re-using the mappings of variables found on the left-hand \bowtie operand while evaluating the right-hand \bowtie operand, this approach results in an additional natural join on the right-hand side of \bowtie , as shown in [117]. Consequently,

$$\begin{aligned} eval(A \text{ OPT } B) &= [[A]] \bowtie ([[A]] \bowtie [[B]]) \\ eval(A \text{ OPT } (B \text{ OPT } C)) &= \\ &= [[A]] \bowtie ([[A]] \bowtie [[B]]) \bowtie ([[A]] \bowtie [[B]] \bowtie [[C]]) \end{aligned}$$

The addition of extra inner joins, compared to *compositional* semantics, might look more restrictive, however, there are important classes of graph patterns where the latter operational approach $exec(Q)$ actually results in false positives w.r.t. the standard *compositional semantics* of $[[Q]]$. **Q6** is one such example, as illustrated in Figure 15b. The mappings retrieved while resolving the outer patterns are used to restrict the search space while resolving the inner patterns, which helps to greatly reduce the amount of information retrieved, better utilize indices, and avoid storing intermediate results. The flow of this useful information is shown by the gray arrows.

We can see that the extra binding for $?y$ arises from one of these extra joins: $[[A]] \bowtie [[B]]$, which becomes a Cartesian product since the graph patterns A and B do not share any variables.

¹² By this extra condition we ensure that we are evaluating $A \text{ OPT } (B \text{ OPT } C)$, which is not equivalent to $(A \text{ OPT } B) \text{ OPT } C$. The optional pattern C is nested into another optional pattern, and mappings of C -specific variables may only appear together with non-NULL mappings of B -specific variables.

The query **Q6** looks a bit strange indeed, and its English formulation would be: 'Find Paul node, on success find George node and, if both are found, return Paul's email if available', which also sounds inherently 'operational', and is correctly answered by $exec(Q6)$, not by $[[Q6]]$. The reader is welcome to experiment with alternative, perhaps more 'compositional' English formulations of the same query.

5.4.2.4 Well-designed queries

Pérez et.al. [122] formulate the condition for *well-designed* queries, and prove that the two semantics are equivalent if the query is *well-designed*.

According to [122]:

"A graph pattern P is **well-designed** if for every occurrence of a sub-pattern $P' = (P_1 \text{ OPT } P_2)$ in P and for every variable $?x$ occurring in P , the following condition holds: if $?x$ occurs both inside P_2 and outside P' , then it also occurs in P_1 ."

In other words, it requires that no variable that can be either bound or unbound as result to *OPT* can be used outside P' . Formally, this condition serves to avoid the Cartesian products, introduced by the *operational semantics*, during the evaluation of any sub-pattern P' that has an optional part.

Sub-patterns that have *AND* instead of *OPT* are provably not problematic. In the same example in Figure 15, if the 'inner' left join \bowtie would be substituted with a simple natural join \Join : the *compositional* evaluation tree (Figure 15a) is not affected, and in the *operational* evaluation tree (Figure 15b) the result of the changed operation would be the empty set, eventually leading to a query result equivalent to the *compositional* one.

Researchers who have studied this problem [122, 40] agree that it is extremely hard to make a realistic example of a query where *compositional* and *operational* semantics do not coincide. However, the class of *not well-designed* queries is much wider than that, and in the next sections we will study classes of useful queries that are *not well-designed*.

5.4.2.5 An important class of not well-designed queries

One interesting case where *compositional* and *operational* semantics agree, but the fact that the query is *not well-designed* finds its implication in the loss of *declarativeness* of the query. Pérez et.al. state that for *well-designed* queries the following property holds:

$$((A \text{ OPT } B) \text{ OPT } C) = ((A \text{ OPT } C) \text{ OPT } B)$$

Indeed, the left-join operator \bowtie is generally sensitive to the order of its application. For the relations R_1, R_2 and R_3 , the property

$$R_1 \bowtie R_2 \bowtie R_3 = R_1 \bowtie R_3 \bowtie R_2$$

can only be guaranteed if R_2 and R_3 extend R_1 with non-overlapping sets of attributes. With a set of relation R attributes denoted as $att(R)$, this condition can be written as

$$att(R_2) \setminus att(R_1) \cap att(R_3) \setminus att(R_1) = \emptyset$$

In the case of both *compositional* and *operational* interpretations of the above queries, this condition means that

$$bound(B) \setminus bound(A) \cap bound(C) \setminus bound(A) = \emptyset$$

so that in *operational* semantics the evaluations of patterns B and C can be done independently, and no variable mapped in $[[B]]$ can influence evaluation of C and vice versa.

However, sometimes it might be useful to express several 'tries' to bind the same variable, so that the order of these tries becomes important. The expression of such queries in relational calculus will require a *coalesce* operator (suggested by Chebotko et. al. [40]). Here's an example:

Q7 (Standard W3C SPARQL): Select names of persons, together with landline or cellphone numbers if available:

```

PREFIX : <http://udbl.uu.se/ex2#>
SELECT ?name ?phone
WHERE { ?x :name ?name .
          OPTIONAL { ?x :phone ?phone } .
          OPTIONAL { ?x :cell ?phone } }

```

This query will prioritize the `:phone` property, and, if the two `OPTIONAL` sub-blocks were reordered, the `:cell` property would be prioritized instead. This example shows how the order of *where* conditions in the query influences the result, meaning that the query is not completely *declarative*.

5.4.2.6 Coalesced expressions

To make query **Q7** *well-designed*, and thus preserve the *declarativeness*, it is sufficient to bind the `:phone` and `:cell` properties to different variables, thus making the `OPTIONAL` sub-blocks independent of each other:

```

PREFIX : <http://udbl.uu.se/ex2#>
SELECT ?name ?phone ?cell
WHERE { ?x :name ?name .
          OPTIONAL { ?x :phone ?phone } .
          OPTIONAL { ?x :cell ?cell } }

```

However, this solution might not be suitable in certain cases, where different properties, or chains of properties, lead to semantically equivalent values. In our example graph **G1**, given in Section 5.1.1 the `ex:simulationMethod` property was originally attached to

ex:OurExperiment instances, but at some point (thanks to the flexibility of the RDF model) ex:OurExperimentRealization instances started to store the same properties, in order to refer to the newer versions of the simulation algorithm used. The following query retrieves these property values, prioritizing the *Realization*-bound ones:

Q8 (Standard W3C SPARQL): Select all *Realization* ids, together with simulation method information if available:

```

PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id ?method
WHERE { ?r a ex:OurExperimentRealization ;
          ex:id ?id .
          OPTIONAL { ?r ex:simulationMethod ?method } .
          OPTIONAL { ?r ex:inExperiment ?e .
                    ?e ex:simulationMethod ?method } }

```

In **Q8** it would be unnatural to project out different variables for the same simulation method information retrieved in two different ways. Fortunately, W3C SPARQL 1.1 provides the **COALESCE** macro that returns the result of the first listed expression which is neither *unbound* nor *error*. With **COALESCE**, the query **Q8** can be made *well-designed* while preserving exactly the same semantics and result width:

Q8a (W3C SPARQL 1.1):

```

PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id (COALESCE(?m1, ?m2) AS ?method)
WHERE { ?r a ex:OurExperimentRealization ;
          ex:id ?id .
          OPTIONAL { ?r ex:simulationMethod ?m1 } .
          OPTIONAL { ?r ex:inExperiment ?e .
                    ?e ex:simulationMethod ?m2 } }

```

The application of **COALESCE** then maps directly to the relational calculus *coalesce* operator, making both formulations of **Q8** structurally equivalent with regard to the *SPARQL to relational algebra* translation proposed in [40].

The possibility of query re-formulation, as shown with **Q8**, does not eliminate the need to handle important classes of *not well-designed* queries in a deterministic and uniform way. SciSPARQL query processor recognizes the cases where the order of *Q.where* conditions is important, and preserves that order under the operational semantics.

5.4.2.7 Binding by filters

Consider another query, which is *not well-designed* - **Q9** (Standard W3C SPARQL):

```
PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id ?a ?c
  WHERE { ?r a ex:OurExperimentRealization ;
           ex:id ?id ;
           ex:parameter_A ?a .
          OPTIONAL { ?r ex:parameter_C ?c } .
          FILTER ( 2 * ?c = ?a ) }
```

Since, according to the W3C SPARQL standard definition [155], any expression depending on an *unbound* value, except expressions under the `bound()` function, will evaluate to *unbound* (an *unbound* FILTER is equivalent to *false*) the English formulation of **Q9** should sound like: "Select those *Realization* ids where parameter *C* is stored and is equal to half of the value of parameter *A*".

Q9 is *not well-designed*, since the variable `?c` is *referenced* in the basic block, whereas it is not *bound* there. One way to make it well-designed would be to remove the `OPTIONAL` keyword altogether, merging the corresponding pattern into the basic block.

Still **Q9** is a valid query in SPARQL and SciSPARQL, but, due to its loss of declarativeness, the `OPTIONAL` and `FILTER` conditions cannot be reordered. Figure 16 shows the *compositional* execution tree for **Q9a**:

```
PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id ?a ?c
  WHERE { ?r a ex:OurExperimentRealization ;
           ex:id ?id ;
           ex:parameter_A ?a .
          FILTER ( 2 * ?c = ?a ) .
          OPTIONAL { ?r ex:parameter_C ?c } }
```

In Figure 16 the filter condition is denoted by f , and the solutions of the basic block with filter conditions applied - as $[[A \text{ AND } f]]$.

The typical way to apply *equality* filters in databases is finding the optimal *binding patterns*, as shown by example in Section 5.1.2.2. Even though the filter expression depends on both `?a` and `?c`, during the query evaluation we are only interested in cases where the filter evaluates to *true*. Consequently, the same filter expression can be used to compute the satisfying value of `?c` when `?a` is already known. Technically, for a predicate $(\text{TIMES } 2 \text{ C } A)$ the query optimizer will choose the binding pattern '-+-', instead of '---', as in the original **Q9**.

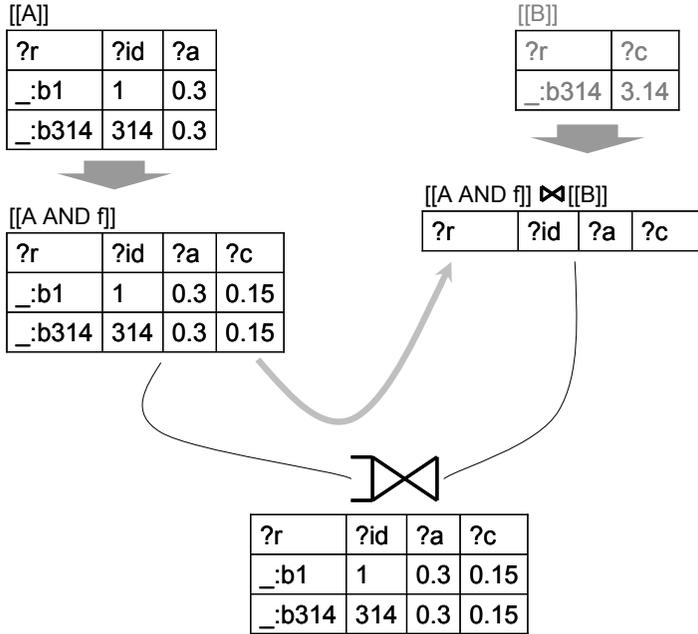


Figure 16. Compositional evaluation tree of Query $Q9a$ over graph $G1$

With this way of evaluating queries, a transition from a solution set $[[A]]$ to $[[A AND f]]$ involves adding the variables 'bound' by the filter, and their respective bindings, as shown in Figure 16. In the next step, extending this *solution set* with values from the database includes a *natural join* of two solution sets, resulting in an empty set. (Note that $[[B]]$ does not even need to be evaluated separately). Finally, due to the nature of the *left join* operator \bowtie , the left set of solutions is retained and projected to the query result.

5.4.2.8 Relaxing the procedural semantics of BIND

The W3C SPARQL 1.1 standard [155, section 10.1] dictates the following restriction on the use of the BIND condition:

"The variable introduced by the BIND clause must not have been used in the group graph pattern up to the point of use in BIND."

This implies that the order of *triple pattern* and BIND conditions is important in all kinds of queries, including *well-designed* ones, and those without any *semibound* variables. For example, while the following query is valid:

Q10 (W3C SPARQL 1.1): Select the *Realization* ids where parameter B is three times greater than parameter A :

```

PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id ?a ?c
  WHERE { ?r a ex:OurExperimentRealization ;
            ex:id ?id ;
            ex:parameter_A ?a .
            BIND ( 3 * ?a AS ?b)
            ?r ex:parameter_B ?b }

```

an alternative formulation *Q10a* is not allowed:

```

PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id ?a ?c
  WHERE { ?r a ex:OurExperimentRealization ;
            ex:id ?id ;
            ex:parameter_A ?a ;
            ex:parameter_B ?b .
            BIND ( 3 * ?a AS ?b) }

```

In both cases it is obvious that **BIND** conditions are used as filters, and an equivalent **FILTER** condition would have been allowed at any point in the **WHERE** clause. It is also beneficial to evaluate the **BIND**/**FILTER** before the remaining triple pattern (as in *Q10*), since it narrows down the graph search and helps to better utilize database indexes.

SciSPARQL removes the restrictions concerning the position of **BIND** conditions in the *where* block, and effectively makes no difference between **BIND** and **FILTER**, making **BIND** potentially *multi-directional*. This feature has two benefits: (i) it widens up the set of valid and correct queries, allowing users to express different intentions with **BIND** and **FILTER** interchangeably, and (ii) it further encourages the users to state the expected correspondences among the values retrieved from an RDF graph, thus opening more opportunities for the query optimizer.

Apart from this, SciSPARQL defines the clear *operational semantics* for the queries which are not *well-designed* according to [122]. This includes the ability to express coalesced expressions with a sequence of **OPTIONAL** patterns, and careful evaluation of **FILTER** and **BIND** expressions when the variables used in these expressions might not always be *bound*.

5.4.3 AmosQL Query Structure

As a first step of query processing, SciSPARQL queries are translated to AmosQL queries. This includes the translation from *graph* to *functional* data model, and from queries returning bags of *mappings* (a.k.a. *solutions*) to the queries returning *relations* (bags of tuples).

The result of an AmosQL query may include **NIL** values representing *unbound* values in SPARQL. Those are introduced by **OPTIONAL** and **UNION** conditions, as shown by examples *Q4* and *Q5* in Section 5.4.1.3.

This section describes the structure of an AmosQL query both on the conceptual level, and by example translations from SciSPARQL. Though AmosQL is certainly a background for the presented work, in this section we focus on features useful for the translation.

5.4.3.1 Basic AmosQL query structure

A simple AmosQL query includes the following components

- ***select*** - a list of projected variables or expressions. The result of a query does not include variable names (hence the distinction between *mappings* and *tuples*). The length of this list is the ***width*** of a query;
- ***distinct*** - whether the query results will be filtered to exclude the duplicate tuples. For tuples containing *NIL* values in the same positions, those values are treated as equal for this purpose.
- ***from*** - a set of all variables in the query, except the parameters for the derived function definitions. The latter are used to translate SciSPARQL *functional views*, as described in Section 5.4.5.8;
- ***where*** - a logical expression, combining the query conditions by the means of *and* and *or* operators.

For example, in the translation of ***Q1*** given in Section 5.1.2.1, the query ***width*** is 2, the ***select*** list includes the variable *id* and an expression involving array operations, the ***from*** list contains all AmosQL counterparts of SciSPARQL query variables, and ***where*** is a conjunction of eight simple conditions, six being *lookups* into the *default graph*, and the other two being inequalities.

5.4.3.2 Cross-referenced named expressions

One important issue the translator needs to take care of is *cross-referenced named expressions*. SciSPARQL has no restrictions on how a variable defined by a named expression can be used in a query. For example, if the query contains

```
SELECT (?length * ?width AS ?area) ...
```

the *?area* variable can be used in any other place in the query, including other **SELECT** expressions, triple patterns, or filters. The problem is that normally the translation does not contain the names of **SELECT** expressions. Additional **BIND** conditions need to be introduced for this purpose.

It is easy to identify such cases, by first computing the set *ref(B)* of variables referenced in the basic block *B* (as described in Section 5.4.1.3), and then checking whether the expression name *name(ne)* from the **SELECT** list appears in *ref(B)*. If so, a new condition *name(ne) = expr(ne)* is added to the basic block. ***Q11*** below provides a complete example:

Q11 (SciSPARQL): Select realization ids where 'factor Y' is less than 10^{-3} , together with the values of 'factor Y', as well as and 'factor X' used in its computations:

```
PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id (?a * ?b AS ?factorX)
      (?factorX / ?iterations AS ?factorY)
WHERE { ?r a ex:OurExperimentRealization ;
          ex:id ?id ;
          ex:iterations ?iterations ;
          ex:parameter_A ?a ;
          ex:parameter_B ?b .
        FILTER (?factorY < 0.001) }
```

The translation of **Q11** contains the translations of the `?factorX` and `?factorY` variables (both of them are detected as being cross-referenced), and the extra equality conditions that bind them:

```
select id, factorx, factory
  from Literal r, Literal factory, Literal iterations,
        Literal factorx, Literal b, Literal a, Literal id
 where (r, URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
        URI('http://udbl.uu.se/ex#OurExperimentRealization'))
        in GRAPH(0)
  and (r, URI('http://udbl.uu.se/ex#id'), id) in GRAPH(0)
  and (r, URI('http://udbl.uu.se/ex#iterations'), iterations)
        in GRAPH(0)
  and (r, URI('http://udbl.uu.se/ex#parameter_A'), a) in GRAPH(0)
  and (r, URI('http://udbl.uu.se/ex#parameter_B'), b) in GRAPH(0)
  and factorx = rdf:times(a, b)
  and factory = rdf:div(factorx, iterations)
  and factory < 0.001;
```

The task of identifying and rewriting cross-referenced expressions is orthogonal and independent of the tasks of identifying *Q.select-extra* expressions for ordering or collecting *aggregate function calls* in aggregate queries, as shown in the next two sections.

5.4.3.3 Ordering and segmentation

In general, a result of an AmosQL query is a bag (i.e. multiset) of tuples, which is, by definition, unordered. However, different *scan* interfaces (as discussed in Section 4.5) to Amos II and SSDM retrieve the results one by one or in sequential batches, and their application might benefit if a certain order would be enforced among the resulting tuples.

SciSPARQL, along with SPARQL 1.1, allow this by including an ORDER BY clause in SELECT queries. Translations of queries with ORDER BY involve a call to `sortbagby()` AmosQL function, and an outer query to transform the resulting sequence of vectors to a sequence of tuples. The translation is illustrated by **Q12** below. The function `sortbagby()` takes three arguments: (i) arbitrary bag of tuples, (ii) a vector containing positions inside tuples to perform the sorting on, and (iii) a vector of the same length

containing sorting direction flags - either 'inc' or 'dec' strings. The function returns a sequence (i.e. vector) of vectors, containing the argument tuples ordered in the specified way.

Some applications might also specify the maximum number of resulting solutions they would like a SciSPARQL query to retrieve - by introducing LIMIT clause. If used together with ORDER BY, this results in *top-k selections*, for example:

Q12 (Basic W3C SPARQL): Select top 5 realizations having the longest simulation (in the number of iterations), returning the realization ids:

```
PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id
WHERE { ?r a ex:OurExperimentRealization ;
         ex:id ?id ;
         ex:iterations ?iterations }
ORDER BY DESC(?iterations) LIMIT 5
```

Similarly to LIMIT, it is also possible to specify OFFSET for a SciSPARQL query in order to retrieve a specific section of the query results. This allows splitting a single query with a long result section to a sequence of queries with a limited number of results. Though *Scan* interfaces in SSDM handle this problem on an interface level, so the query execution does not proceed until the next result is requested, segmentation with LIMIT and OFFSET is part of SPARQL, and is supported in our translation.

Q12 is translated using `sortbagby()` and `bsection1()` Amos functions:

```
bsection1((select o:v[0] from Vector of Literal o:v
          where o:v in sortbagby((
select id, iterations
from Literal r, Literal iterations, Literal id
where (r, URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
        URI('http://udbl.uu.se/ex#OurExperimentRealization'))
in GRAPH(0)
and (r, URI('http://udbl.uu.se/ex#id'), id) in GRAPH(0)
and (r, URI('http://udbl.uu.se/ex#iterations'), iterations)
in GRAPH(0)), {2}, {'dec'})), 0, 5);
```

As shown, the {2} and {'dec'} arguments specify the sorting order, and then the outer query 'select o:v[0] ...' transforms vectors to tuples. The resulting ordered bag is passed to `bsection1()`, specifying zero (default) *offset* and *limit* of 5 results. The `bsection1(b, start, stop)` function iterates through the bag `b` of inputs, and begins emitting when an element counter is at value `start`, and stops when the counter reaches the `stop` value.

5.4.3.4 Grouping and aggregation

An *aggregate function* in Amos is a function that takes a bag argument and returns an atomic result. The useful class of aggregate functions for SSDM, which can be used together with grouping, are those which accept a *Bag of*

Literal as a single argument. This includes `count()`, which accepts all kinds of bags.

The Amos aggregate functions with numeric semantics, like `sum()`, `avg()`, `min()`, `max()` and some others have been extended to handle bags of scalar numbers and bags of aligned arrays. According to the SPARQL 1.1 standard, in case of incompatible values in the input bag, such aggregate functions return *error* value, which in SciSPARQL and AmosQL terms is equivalent to terminating silently without emitting a result. Such extended versions of aggregate Amos functions are named with `rdf:` prefix, e.g. `rdf:sum()`.

We have defined SciSPARQL *aggregate queries* in Section 5.4.1.2. An aggregate query having *width* of 1 simply applies an aggregate function to the bag of the *inner query* solutions, like the *Q3* example. The AmosQL translation of *Q3* is as follows:

```
count((select distinct a
      from Literal r, Literal a
      where (r,
             URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
             URI('http://udbl.uu.se/ex#OurExperimentRealization'))
            in GRAPH(0)
      and (r, URI('http://udbl.uu.se/ex#parameter_A'), a)
            in GRAPH(0));
```

This translation also shows that the *Q.inner-distinct* property of an aggregate SciSPARQL query is translated to `distinct` option in the inner query in AmosQL.

An aggregate query of greater *width* might include several aggregate operations across different variables in the inner query solutions, or expressions involving those variables. Additionally, it might include *grouping* on certain variables or expressions. *Q2* is one such example, being translated as follows:

```
select o:v[0], o:v[1], o:v[2] from Vector of Literal o:v
where o:v in sortbagby((
  select a, realizations, total_iterations
  from Literal total_iterations, Literal realizations, Literal a
  where (a, realizations, total_iterations) in groupby((
    select a, r, iterations
    from Literal r, Literal a, Literal iterations
    where (r, URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
            URI('http://udbl.uu.se/ex#OurExperimentRealization'))
           in GRAPH(0)
    and (r, URI('http://udbl.uu.se/ex#iterations'), iterations)
           in GRAPH(0)
    and (r, URI('http://udbl.uu.se/ex#parameter_A'), a) in GRAPH(0)
        ), {'count', #'rdf:sum'})
  and realizations>=2), {1,3}, {'inc', 'dec'});
```

The translation shows a straightforward mapping of conceptual *inner* and *outer* query structures in SciSPARQL to the pair of nested queries in

AmosQL (starting at lines 3 and 6). The inner query contains a graph pattern, and declares all variables used in that pattern, projecting out three of them.

The outer query declares variables corresponding to the *outer* context of **Q2**, including the *grouping* variable `?a` and the names of aggregate expressions. The main *where* condition is a call to the `groupby()` function, binding all these variables, and specifying the aggregate functions `count` and `rdf:sum` as functional arguments. The additional condition translates the one found in the `HAVING` clause of **Q2**.

We will refer to the part of the translation that appears to be the first argument to `sortbagby()` as the *core translation*, and the translation applying `sortbagby()` and/or `bsection1()` as the *finalized translation*. In the absence of `ORDER BY`, `LIMIT` and `OFFSET` clauses these translations are the same.

Notice that sorting with `sortbagby()`, as described in the previous subsection, makes use of the variables in the outer query - including the results of grouped aggregate operations. If, for example, some of the `ORDER BY` variables (or *named expressions*) were not included in the `SELECT` clause in SciSPARQL, their translations would still appear in the *select* list of the outer AmosQL query, but the respective vector elements would not be selected into the result after sorting. We will refer to such additional expressions as *Q.select-extra*.

This principle complements the process of discovering cross-referenced named expressions, explained in Section 5.4.3.2, and the process of collecting aggregate subexpressions, more formally described in Section 5.4.5.7 below. The `HAVING` and `ORDER BY` clauses in a SciSPARQL query might include additional unique aggregate expressions, not found in the `SELECT` clause. In the absence of names, surrogate names for such expressions will be created, and some of these might be added to *Q.select-extra*.

Consider the query **Q2a**, differing from **Q2** only in the `SELECT` clause, and the fact that the corresponding aggregate expressions are instead incorporated into *Q.orderby* and *Q.having* lists, which do not require expressions to be named:

```
PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?a
  WHERE { ?r a ex:OurExperimentRealization ;
            ex:iterations ?iterations ;
            ex:parameter_A ?a }
  GROUP BY ?a
  HAVING COUNT(?r) >= 2
  ORDER BY ?a DESC(SUM(?iterations))
```

The translation of *Q2a* would be the same as for *Q2* as shown above, except for the shorter vector-to-tuple projection on the first line, and the fact that the former `total_iterations` and `realizations` variables are now re-introduced by the translator with surrogate names `agg:1` and `agg:2`:

```
select o:v[0] from Vector of Literal o:v
where o:v in sortbagby((
  select a, agg:2
    from Literal agg:2, Literal agg:1, Literal a
  where (a, agg:1, agg:2) in groupby((
    select a, r, iterations
      from Literal r, Literal a, Literal iterations
    where (r, URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
            URI('http://udbl.uu.se/ex#OurExperimentRealization'))
          in GRAPH(0)
        and (r, URI('http://udbl.uu.se/ex#iterations'), iterations)
          in GRAPH(0)
        and (r, URI('http://udbl.uu.se/ex#parameter_A'), a) in GRAPH(0)
          ), {'count', #'rdf:sum'})
    and agg:1>=2), {1,3}, {'inc','dec'});
```

The *named expressions* corresponding to `agg:1` and `agg:2` comprise the *Q.agg* list, and `agg:2` is needed for ordering is in *Q.select-extra* list. By means of these rewrites, the SciSPARQL translator handles uniformly all the aggregate expressions found anywhere in the *outer level* of a query.

5.4.4 Extensions to ObjectLog and Physical Algebra

Some of the SPARQL standard behavior have proven to be quite challenging to implement using the original definitions of AmosQL and the underlying ObjectLog [100]. These challenges include:

- generating unique values inside the query: *blank nodes* for CONSTRUCT queries, random numbers and GUIDs
- lazy evaluation of IN lists
- IF and COALESCE operators
- OPTIONAL operator

While the first challenge conflicts with the idea of side-effect-free queries, the remaining three contradict the declarative nature of AmosQL and ObjectLog, where the query conditions constitute a logical expression that can be transformed e.g. to *Disjunctive Normal Form*. Predicates are the atomic terms in such expressions, each having a set binding patterns for the optimizer to choose from.

The predicate calculus is, in a certain sense, a more restrictive model than the relational algebra. Whereas the latter can easily be extended with e.g. *coalesce* operator, as shown by Chebotko et.al. [40], the predicate calculus can not - since *coalesce* is neither an atomic predicate, nor a logical expression of any atomic predicates.

This difficulty is akin to the conceptual distinction between functions and macros in Lisp: whereas a function is evaluated only after all its arguments are computed, and thus can be mapped to collections of its argument bindings, the macro decides the evaluation order of its arguments, and might even choose not to evaluate some of them. In Lisp, logical AND and OR operators are naturally defined as macros, thus implementing lazy evaluation, and *n*-ary OR, while not restricted to logical operands, being also the equivalent for *coalesce*.

In AmosQL a disjunction is by definition equivalent to the relational UNION ALL, always evaluating every branch. However, the conjunction of predicates assumes a kind of lazy evaluation: if a predicate does not return a value, the nested-loop execution (as shown by example in Section 5.1.2.3) skips the subsequent predicates, backtracking instead. However, the user normally has no control of how the predicates in a conjunction will be ordered in the nested loop so that, for example, a condition $f(x)$ and $g(x)$ is totally equivalent to $g(x)$ and $f(x)$.

The only feature of an AmosQL query that restricts the order of evaluation is the dependency relationships among the variables. One way to make sure that $g(x)$ is only evaluated if $f(x)$ returns, would be to make $f(x)$ return x (instead of a *Boolean* value) and make sure it has only "forward" binding pattern '-+'. Then the condition $y = f(x)$ and $g(y)$ would guarantee the desired order.

The AmosQL implementations of OPTIONAL, IF, COALESCE, and "lazy" IN operators all require macro-like behavior at the execution time, and the specialized representations at the intermediate steps, including ObjectLog. The rest of this section elaborates on the extension of both physical algebra and ObjectLog with OPTIONAL operator, as the most essential feature needed for SciSPARQL, with the rest remaining a near-future work.

5.4.4.1 OPTIONAL operator

As mentioned above, if a predicate placed in the nested-loop-join does not return, the current solution is discarded, and the execution backtracks to the previous predicates to generate a new solution. For example, let us consider a query *Q4a*, differing from *Q4* only so that the OPTIONAL graph pattern is merged into the basic one:

```
PREFIX ex: <http://udbl.uu.se/ex#>
SELECT ?id ?a ?c
WHERE { ?r a ex:OurExperimentRealization ;
           ex:id ?id ;
           ex:parameter_A ?a ;
           ex:parameter_C ?c }
```

The optimizer is potentially free to reorder the four stored predicate instances that this query translates to - based on selectivity statistics or any other considerations (Amos is extensible in its optimization strategies too!) The stored predicates have a nice property of offering the full combinatorial set of binding patterns. One of the possible execution plans would be:

```
(*SELECT* ID+ A+ C+) <-
(NESTED-LOOP-JOIN
  (HASH-INDEX-SCAN GRAPH-+++
    0 R+ #[URI "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"]
      #[URI "http://udbl.uu.se/ex#OurExperimentRealization"])
  (HASH-INDEX-SCAN GRAPH-+++
    0 R- #[URI "http://udbl.uu.se/ex#id"] ID+)
  (HASH-INDEX-SCAN GRAPH-+++
    0 R- #[URI "http://udbl.uu.se/ex#parameter_C"] C+)
  (HASH-INDEX-SCAN GRAPH-+++
    0 R- #[URI "http://udbl.uu.se/ex#parameter_A"] A+))
```

While being evaluated on the **GI** dataset, the first partial solution (with realization `_:r1`) would be discarded at the point when the predicate on `ex:parameter_C` fails to return, and the predicate on `ex:parameter_A`, placed further down the loop, is not executed. Since the predicate on `ex:id` returns only a single result given the realization `_:r1`, the execution backtracks to the first predicate, returning another realization `_:r314`, which, after applying the rest of the predicates, becomes an emitted solution.

If we would like to implement **Q4** instead, the main requirement is that the partial solution with realization `_:r1` is not discarded even if the `ex:parameter_C` tuple is not found in the graph. The inbound value for the variable `c` will need to be propagated through the remaining predicates, and emitted as part of the solution.

This is achieved by adding the `OPTIONAL` operator into the execution plan, which does exactly this: introduces *unbound* values into the current solution, instead of discarding it:

```
(*SELECT* ID+ A+ C+) <-
(NESTED-LOOP-JOIN
  (HASH-INDEX-SCAN GRAPH-+++
    0 R+ #[URI "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"]
      #[URI "http://udbl.uu.se/ex#OurExperimentRealization"])
  (HASH-INDEX-SCAN GRAPH-+++
    0 R- #[URI "http://udbl.uu.se/ex#id"] ID+)
  (OPTIONAL (HASH-INDEX-SCAN GRAPH-+++
    0 R- #[URI "http://udbl.uu.se/ex#parameter_C"] C+))
  (HASH-INDEX-SCAN GRAPH-+++
    0 R- #[URI "http://udbl.uu.se/ex#parameter_A"] A+))
```

The `OPTIONAL` operator is not limited to containing a single predicate - arbitrary conjunctions and disjunctions of predicates can be put under `OPTIONAL`, the same way as in SPARQL. The use of `OPTIONAL` extends all the way up to ObjectLog representation, and the AmosQL syntax has been

extended with the `optional()` construct (not really a function), so that the AmosQL translation of **Q4** is as follows:

```
select id, a, c
  from Literal r, Literal c, Literal a, Literal id
 where (r, URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
        URI('http://udbl.uu.se/ex#OurExperimentRealization'))
        in GRAPH(0)
 and (r, URI('http://udbl.uu.se/ex#id'), id) in GRAPH(0)
 and (r, URI('http://udbl.uu.se/ex#parameter_A'), a) in GRAPH(0)
 and optional((r, URI('http://udbl.uu.se/ex#parameter_C'), c)
              in GRAPH(0))
```

Extending AmosQL, ObjectLog and the physical algebra with a new operator provided the most simple and straightforward implementation of SPARQL semantics inside the functional DBMS framework.

5.4.4.2 Query optimization restrictions

The query optimizer was made to be aware of the new `OPTIONAL` operator in ObjectLog, treating it as a container and propagating it to the physical algebra. However, in some cases this handling is not sufficient. As has been discussed in Section 5.4.2, there are important classes of *not well-designed* queries, i.e. queries which are not completely declarative, where reordering of certain conditions might affect the result.

For example, in queries **Q7** and **Q8** reordering of the two `OPTIONAL` conditions entails a potentially different result. Also, as shown by **Q9** and **Q9a**, swapping an `OPTIONAL` condition with a `FILTER` condition depending on variables bound in `OPTIONAL` also affects the results.

A simple to implement solution currently used in Amos is the restriction for the query optimizer to move any conjunctive predicates across the `OPTIONAL` block. The addition of an `OPTIONAL` condition to a query might split the flattened list of predicates into three parts, optimized separately: conjunctive predicates listed before and after `OPTIONAL`, and the predicates inside the `OPTIONAL` block.

Even though simple and easy to understand, this restriction can be largely relaxed. It is enough to make sure that only the predicates that depend on variables bound inside an `OPTIONAL` block, and not bound outside it (i.e. on *semibound* variables), may not be moved across the optional block. This is never the case in *well-designed* queries, which remain purely declarative.

The relaxed restrictions have yet to be implemented in the future versions of SciSPARQL Database Manager. Since *semibound* variables are detected at the translation stage, it should be made possible to communicate their set to the optimizer. Another option would be to delegate the detection of *semibound* variables to the optimizer itself.

5.4.5 The Translation Algorithm

As described in the previous sections, an internal representation Q of a SciSPARQL query is a data structure with named fields, lists, and nested data structures. Of course, the translation is not limited only to queries. A more general term is a SciSPARQL statement, with the internal representation of S . Such a statement can either be

- a *query*,
- a *function definition*, containing $S.name$, $S.params$ and $S.body$ fields, with $S.body$ being a SELECT query,
- a *stand-alone expression*, considered to be a degenerate form of SELECT query. (Certain functions with side effects, like `LOAD()` and `SOURCE()` can only be called this way),
- an *update statement*, containing ***with***, ***insert***, ***delete***, ***using***, and ***where*** fields (this does not include `INSERT DATA` and `DELETE DATA`, as these statements are executed directly at parse time), or
- a session-scoped *prefix declaration*.

This section formally defines the translation function $tr(S)$, generating AmosQL textual translation of a SciSPARQL query parsed into S . The function $tr(S)$ is recursive, and is mainly applied depth-first in leaf-to-root order. For example, the translation of a simple SELECT query begins with translating the leaf elements of $Q.what$ expressions, and produces the translation of the whole query in the last step by combining the translations of its constituent parts.

There are, however, exceptions to this principle. For example, when a SciSPARQL query declares a set of named graphs addressed, or a set of prefixes used - these should be accessible when translating the triple patterns. Also, translation of certain expressions (like `ARGMAX()` explained below) involves adding new conditions at certain levels and introducing additional query variables. In order to accommodate for this flexibility, a translation context data structure is introduced.

5.4.5.1 Translation context

The translation context TC data structure is created at the beginning of translating a statement S . Below are the fields and their initial values:

- ***TC.prefixes*** - the set of statement-scoped prefix declarations. Together with the set session-prefixes of session-scoped prefix declarations, introduced with `PREFIX` statements, it is used to translate prefixed URIs to URI constructor calls in AmosQL.
- ***TC.source*** - the set of named graphs given in a `FROM` clauses of a query or a `USING` clauses of an update statement being translated; the particular content of this field is explained in the next subsection.

- ***TC.namedsources*** - the set of the alternative bindings for otherwise unbound graph variables, given in `FROM NAMED` or `USING NAMED` clauses.
- ***TC.newconds*** - the translated representations of the additional conjunctive conditions that need to be added to the current block, initialized to the empty set.
- ***TC.newvars*** - the list of additional variables introduced during the translation of the current query, initialized to the empty set.
- ***TC.bound*** - the list of variables *bound* outside the current query (e.g. in a host query when translating a subquery), initialized with *S.params* for function definitions and to the empty set for standalone queries.

All the listed fields serve for the root-to-leaf or horizontal propagation of the data important for the translation. However, not all this data is propagated all the way down to the leaf translations; there are particular thresholds where the translation context is cloned partially:

- When translating a subquery, *TC.newconds* and *TC.newvars* are left on the outer level; moreover if the subquery contains `FROM` and/or `FROM NAMED` clause, the *TC.source* or *TC.namedsources* fields will be overridden.
- Similarly for subqueries introduced by translations of `ARGMIN()` and `ARGMAX()`.
- When translating an aggregate query, outer and inner contexts are created, with *TC.newvars* and *TC.newconds* independently accumulated on these separate layers.

5.4.5.2 Source graphs and triple patterns

Following the W3C SPARQL standard, SciSPARQL allows specifying a set of source graphs in a query with a `FROM` clause and with a `USING` clause for updates. In the absence of such a specification, the default graph is used.

The translation of a triple pattern condition *tp*, containing RDF Terms or variables *tp.s*, *tp.p* and *tp.o*, corresponding to *subject*, *property* and *object* of the pattern, in general has the form

$$tr(tp) = (tr(tp.s), tr(tp.p), tr(tp.o)) \text{ in } tr_{Source}(g)$$

where *g* is the innermost `GRAPH` specification (either a URI or a variable), or, in the absence of such, using the pre-computed translation of the source graph or graphs:

$$tr(tp) = (tr(tp.s), tr(tp.p), tr(tp.o)) \text{ in } TC.source$$

Since a query might specify more than one graph URI in its `FROM` clauses, either `GRAPH()` or `GRAPHS()` function is used for the translation, the latter accepting a vector of *graph ids*, and implementing *RDF Merge* of these

graphs, as defined by the W3C Standard [155]. Such a merge operation is a simple *union* of the graphs, since all blank nodes stored in different graphs are unique in the scope of the given SSDM database.

SSDM maintains a dictionary `NGDict()` mapping URIs identifying named graphs to the internal *graph ids*, with the default graph always having *id* of 0. When translating queries, and the graph's URI is known, this dictionary is looked up at translation time, and *tr(Q.from)* might look like `GRAPH(4)` for a single graph or `GRAPHS({5,7})` for a set of graphs. When translating function definitions, the source graph is not required to exist at the definition time; hence delayed lookup is put into a translation, making it, for example:

```
GRAPH(NGDict(URI('http://udbl.uu.se/g2.ttl')))
```

Either of such translations is put into *TC.source* at the beginning of the translation process. When a variable is given after the `GRAPH` keyword, similar lookup is utilized:

```
GRAPH(NGDict(g))
```

for a SciSPARQL variable `?g`.

Alternative named graphs specified with `FROM NAMED` or `USING NAMED` clauses, as W3C Specifications suggest, are only useful to provide a finite set of bindings for a graph variable. Since SSDM operates within a *closed world assumption*, there is always a finite number of named graphs accessible to a query, i.e. a finite number of entries in the `NGDict()` dictionary. Hence, an otherwise unbound graph variable `?g`, used in a graph pattern

```
GRAPH ?g { ... }
```

will effectively match this pattern against all named graphs stored in SSDM's database. The only reason to use `FROM NAMED` and `USING NAMED` syntax is to restrict the set of possible source graphs, as if the

```
VALUES ?g { ... }
```

condition were added to the basic block. This idea is used for the translation of *TC.namesources*, effectively resulting in an extra condition put into *TC.newconds* for any such variable - such conditions are similar to translations of `VALUES` conditions.

5.4.5.3 Translating path expressions

SPARQL is a graph language, dealing with the concepts of *nodes*, *edges*, and by induction, *paths* in a graph. As of the W3C SPARQL 1.1 Standard [155], *regular path expressions* are allowed as part of graph patterns, as a general case of *triple patterns*. These expressions can be either recursive or non-recursive, the former effectively employing a transitive closure on a

certain property or a combination of properties. The non-recursive kinds of expressions include *chaining*, *reversal*, *alternatives*, and *reflexive closure*.

We do not include the *negated property set* operator in the current version of SciSPARQL, due to the problems with its standard definition, explored in [93]. Though not theoretically ambiguous, together with *reversal* it introduces a certain counter-intuitive 'butterfly effect' in the set of query solutions.

Below with a set of rules R1 - R7 we define a translation function $tr(C)$ for a *path expression* condition C , the latter consisting of subject $C.s$, object $C.o$, and a path expression $C.p$. We list the translation function for each type of expression listed. Formally, at certain points we translate the new (constituent or equivalent) path expressions, introduced with $pe(s, p, o)$ constructor function.

R1. If $C.p$ is an RDF term or a variable, C is translated to a triple pattern, according to the definitions in the previous section (Section 5.4.5.2).

R2. If $C.p$ is a *chain* P/Q of two path expressions P and Q , an intermediate variable $seq = newvar(TC)$ is introduced, and C is translated a conjunction of two path expressions:

$$tr(C) = tr(pe(C.s, P, seq) \wedge pe(seq, Q, C.o))$$

R3. If $C.p$ is $\wedge P$, denoting the *reversed expression* P , it is translated by direct reversal of *subject* and *object* in the condition:

$$tr(C) = tr(pe(C.o, P, C.s))$$

R4. If C is an *alternative* $P|Q$ of two path expressions P and Q , it is translated as a disjunction of the conditions:

$$tr(C) = tr(pe(C.s, P, C.o) \vee pe(C.s, Q, C.o))$$

R5. If C is a *reflexive closure* $P?$ of the path P , it is translated as a disjunction of the original path expression condition, and the equality of subject with object:

$$tr(C) = tr(pe(C.s, P, C.o) \vee (C.s = C.o))$$

The recursive path expressions are translated using a *transitive closure* function defined in AmosQL. There is a `tclosen()` function for the *non-reflexive* case, and `tclose()` for the *reflexive-transitive closure*. These correspond to Q^+ and Q^* syntax respectively. The first argument to either of these is a bag-valued function $f(x)$ with the same domain and range, and the second argument is a starting point x . The transitive closure $tclosen(f, x)$ function returns all distinct values acquired by computing $f(x)$, $f(f(x))$, $f(f(f(x)))$, and so on. The finite amount of such results for each x is assumed. A reflexive-transitive closure $tclose(f, x)$ additionally includes x in the result.

A reverse application is also an option: given $y \in \text{tclose}(f, x)$ expression, it is possible to determine the set of all possible x values if y value is already bound. The function $f(x)$ should be reversible - if it were a foreign function, the corresponding predicate $(F X Y)$ would have both '-+' and '+-' binding patterns. All stored functions are fully *multidirectional*, and derived functions are 'flattened' by Amos II query processor to the logical expression of their constituent predicates. If a derived function consists only of stored predicates, it is *multidirectional* by induction¹³.

A *non-recursive* path expression P can be regarded as a derived function $p(x) = y$, connecting all possible pairs (x, y) of nodes, connected by paths satisfying P . Such $p(x)$ is fully multidirectional - since it ultimately calls the same stored predicate $(\text{GRAPH } \text{GID } \text{S } \text{P } \text{O})$. In particular, $p(x)$ can be used to find all possible bindings for y given a binding for x , and all possible binding for x given a binding for y . In AmosQL this can be expressed as a function with surrogate name $\text{pathfn} = \text{newfname}(TC)$, and where block formed by the translation of condition $\text{pe}(x, P, y)$:

$$TR_W = \text{tr}(\text{pe}(x, P, y), TC)$$

```
trfn(P) = create function pathfn (Literal x) -> Bag of Literal y as
select y from trD(TC.newvars) where TRW
```

and if P contains no *chain* operators, no new variables were created during the translation process, and thus the `from` clause is omitted. The function $\text{tr}_D()$ given a set of variables, constructs a comma-separated list, prefixing each name by `Literal` type specifier.

R6. If $C.p$ is a *transitive closure* P^+ of a path expression P , an internal function pathfn is defined as $\text{tr}_{fn}(P)$, and the condition C is translated as:

$$\text{tr}(C) = \text{tr}(C.o) \text{ in } \text{tclosen}(\text{pathfn}, \text{tr}(C.s))$$

R7. If $C.p$ is a *reflexive-transitive closure* P^* , similarly:

$$\text{tr}(C) = \text{tr}(C.o) \text{ in } \text{tclose}(\text{pathfn}, \text{tr}(C.s))$$

Note that such translations of *recursive* path expressions are also fully multidirectional, as functions connecting pairs (x, y) of nodes: they call only the stored predicate `GRAPH` and the foreign predicates `TCLOSE` or `TCLOSEN`. Either of the latter has both '--+' and '-+-' binding patterns, and their first argument is always bound to the function named *pathfn*. Hence, a path expression P , used under the transitive closures P^+ or P^* does not need to

¹³ A condition for a derived function $f(x)$ being multidirectional is actually much weaker: it requires there should be a predicate binding x in every disjunctive branch, with no restriction on any other predicates in branches. Having only multidirectional (e.g. stored) predicates is just a simple particular case.

be non-recursive for multidirectionality of the closure expression. Nested recursion is generally supported.

The following example illustrates the translation rules defined in this section:

Q13 (W3C SPARQL 1.1): Select names of all ancestors of Alice:

```
PREFIX : <http://example.org/>
SELECT ?n
WHERE { ?x (:fatherOf|:motherOf)+/:name "Alice" ;
          :name ?n }
```

This is translated by recursively applying R2, R6, R3, and R1 in the terminal cases, resulting in:

```
select n
from Literal x, Literal n, Literal seq:1
where seq:1 in tclosen('#'path:1', x)
and (seq:1, URI('http://example.org/name'), USTR('Alice'))
in GRAPH(0)
and (x, URI('http://example.org/name'), n) in GRAPH(0);
```

where function path:1() is defined on-the-fly as:

```
create function path:1(Literal x) -> Bag of Literal y as
select x
where (x, URI('http://example.org/fatherOf'), y) in GRAPH(0)
or (x, URI('http://example.org/motherOf'), y) in GRAPH(0);
```

5.4.5.4 Translating expressions

Currently, SciSPARQL supports the following kinds of expressions, listed here with their translation rules and examples:

- a *variable*, e.g. ?x - translated to an AmosQL variable, e.g. x
- a *full URI*, e.g. <http://udbl.uu.se/g1> - translated to a URI constructor call in AmosQL, e.g. URI('http://udbl.uu.se/g1')
- an *abbreviated URI*, e.g. udbl:g1 - translated similarly, using the prefix lookup first in *TC.prefixes* and then in session-prefixes, in order to get the full URI form
- a *numeric or logical literal*, e.g. 3.14 or true - translated to the same numeric literal in AmosQL (textual representations fully comply)
- a *string literal*, e.g. "Cat" or "Katz"@de - translated to a Unicode string constructor call in AmosQL, e.g. USTR('Cat') or USTR('Katz', 'de')
- a *typed literal*, e.g. "Katt"^^udbl:Djur - translated to a typed literal constructor call in AmosQL, e.g. TypedRDF('Katt', URI('http://udbl.uu.se/Djur')) - this translation excludes the standard XMLS types for numbers, Boolean values and text strings date literals, where one of the above translations is used instead.

- a user-specified *blank node*, e.g. `_:r1` - translated to an AmosQL variable, e.g. `b:r1`
- a parser-generated *blank node* resulting from square brackets syntax (containing a unique id supplied by the parser) - translated to an AmosQL variable, e.g. `g:324`
- a unary or binary *arithmetic operation*, e.g. `x + y`, where `x` and `y` are expressions - translated to a call to the corresponding function, accepting the arguments of generic type *Literal* - e.g. `rdf:plus(tr(x), tr(y))`
- a binary dot-prefixed *arithmetic operation*, e.g. `x .+ y`, where `x` and `y` are expressions - translated to a call to a specialized Amos function - e.g. `rdf:aplus(tr(x), tr(y))`, where `rdf:aplus()` has the same general implementation as `rdf:plus()`, but a higher cost estimate
- a *comparison operation*, e.g. `x >= y` - translated to the corresponding comparison expression in AmosQL (since the comparison operators in AmosQL accept arguments of any type), e.g. `tr(x) >= tr(y)`. In order to enforce the strict adherence to the SPARQL standard, (as controlled by `_sq_strict_` flag) additional condition `comparable(tr(x), tr(y))` will be added to `TC.newconds` - this ensures that comparing e.g. a number to a string never returns *true*
- a dot-prefixed *comparison operation*, e.g. `x .>= y` - translated to the specialized Amos function, performing the element-wise comparison and producing a *Boolean* array (unless both operands happen to be scalar) - e.g. `rdf:agte(tr(x), tr(y))`. Specialized operations on *Boolean* arrays are also supported: `x .& y` and `x .| y` for the element-wise logic
- a logical *conjunction* or *disjunction* operation, e.g. `x && y` - translated to the corresponding logical operator in AmosQL, e.g. `tr(x)` and `tr(y)`
- a logical *negation* operation `!x` - the translation depends on the kind of the immediate subexpression `x`: if it is found in the pairs of opposite expressions (e.g. `!=` and `=` comparisons, or `false` and `true` literals, `bound()` and `notbound()` functions), then the opposite expression is translated instead, otherwise, the `rdf:not(EBV(x))` translation is used, where `EBV()` implements the *Effective Boolean Value*, as explained in Section 3.3.3, and `rdf:not()` negates the logical value
- a block with an EXISTS or NOT EXISTS quantifier, e.g. `EXISTS B` - translated to an AmosQL subquery inside `some()` or `notany()` quantifier respectively, e.g. `some(select true from trD(declare(B)) where tr(B))`, where `declare(B)` builds a list of variables to be declared for the translation of block `B`, as specified in Section 5.4.5.6 below.
- a typecasting expression, e.g. `xsd:integer("314")` - translated to the corresponding built-in typecasting function call, e.g.

`rdf:toInteger('314')` for the numeric, Boolean, and string types, otherwise to a *typed literal* constructor call, as shown above

- a call to a built-in function with variable number of arguments, e.g. 1D array construct `A(x, y)` - translated to a call to the implementing Amos function, with translated arguments packed into a vector, e.g. `a({tr(x), tr(y)})`
- a call to `ARGMIN()` or `ARGMAX()` built-in second-order function - translation is explained in Section 5.4.5.9
- a call to any other non-aggregate function (either built-in or user-defined), e.g. `round(x)` - translated to a call to an Amos function with the same name prefixed with `rdf:`, e.g. `rdf:round(x)`
- a call to an aggregate function, e.g. `SUM(x)` - not translated as part of the expression, such aggregate function calls are collected into *Q.agg* during the preprocessing phase, as explained in Section 5.4.5.7, and are replaced with expressions under the call to an aggregate function, to be translated as part of the *select* list of the *inner query*
- a call to any non-aggregate first-order function with some of the arguments replaced by *asterisk*, e.g. `power(*, 2)` - translated to a pair of consecutive arguments to a second-order function, together implementing a lexical closure - a function name and a partial tuple constructor call - e.g. `#'power', make_partial_tuple({2},{0})`, with the first vector containing all non-asterisk arguments, and the second vector containing the positions of asterisks. Such lexical closures are only used as arguments to built-in second-order functions, like the `ARRAY()` constructor or the `MAP()` array mapper, and represent function calls with some arguments bound, while other arguments are free.
- an *array dereference* operation, e.g. `x[1,4:2:8]` - translated to a superposition of calls to `aref()` and `asub()` for each referred dimension: `aref()` implements *projections* and `asub()` implements *range selections* - both functions take a dimension index, e.g. `asub(aref(tr(x), 1, 1), 1, 4, 2, 8)` - note that the dimension index for the second dimension becomes decremented after applying the projection.

5.4.5.5 Translating blocks of conditions

The `WHERE` block of a query or update statement lists a sequence of conditions, appearing in conjunction (though not strictly a conjunction with commutative property in the case of not well-formed queries). Such blocks are translated to AmosQL conjunctions of translations of the respective conditions, the order being preserved, together with any additional conditions introduced by the translation of expressions. Such conjunctions are fully commutative in the absence of the `optional()` operator, which, if

present, divides their sequence into sections that cannot be reordered, as explained in Section 5.4.4.

Section 5.4.5.2 above has already introduced the translation of triple patterns, within and without GRAPH blocks. The following table summarizes the translation used for all kinds of conditions. The original conditions are shown in SciSPARQL's syntax for simplicity, even though the translator operates on the internal data structure representations of conditions and their components - nested blocks, expressions, etc.

Table 3. Translation of conditions in WHERE block

Type of condition	Condition <i>C</i> (SciSPARQL syntax)	Symbols	Translation $tr(C, TC)$ (AmosQL syntax)
Triple pattern outside GRAPH block	$s p o$	RDF terms	$(tr(s), tr(p), tr(o))$ in $TC.source$
Triple pattern inside GRAPH block	$GRAPH g \{s p o\}$	RDF terms	$(tr(s), tr(p), tr(o))$ in $tr_{Source}(g)$
Path patterns	$s P o$ $GRAPH g \{s P o\}$	g, s, o - RDF terms, P - path expression	explained in Section 5.4.5.3
Filter	$FILTER e$	e - expression	$tr(e)$
Explicit binding	$BIND (e AS v)$	e - expression, v - variable	$tr(v) = tr(e)$
	$VALUES v \{e_1, e_2, \dots\}$	v - variable e_1, e_2, \dots - expressions	$tr(v) = tr(e_1)$ or $tr(v) = tr(e_2)$ or ...
Optional block	$OPTIONAL B$	B - block	$optional(tr(B))$
Union	$B_1 UNION B_2$ $UNION B_3 \dots$	blocks	$(tr(B_1) \text{ or } tr(B_2) \text{ or } tr(B_3)) \dots$
Subquery	Q	Q - query	$(tr(var(Q.what)))$ in $(tr(Q))$

The translation $tr(B)$ of a whole condition block is done by putting the translations of the individual conditions $B.conds$ into a conjunction:

$$tr(B) = andify(tr(B.conds))$$

where $andify()$ combines the given translated conditions into a single conjunction, effectively interleaving them with the `and` keyword. When translating the basic block of a query, as shown next in Section 5.4.5.6, additional translated conditions from $TC.newconds$ are added into the conjunction.

A note on the translation of *subqueries*: the function $var(Q.what)$ returns the list of variables projected from a SELECT query Q - this includes variables appearing in the $Q.what$ list and names of the named expressions appearing there. All variables in $var(Q.what)$ become bound and referenced in the host query, according to the definitions given in Section 5.4.1.3

An aggregate subquery is translated in two layers, however, its outer layer is effectively merged into the *host query*, with e.g. *Q.having* conditions added to *TC.newconds*. No ordering or segmentation is allowed in subqueries, however, some additional variables on the outer layer might arise from cross-referencing of expressions, as explained in Section 5.4.5.7.

5.4.5.6 Translating basic SELECT queries

Here and below, the term *basic query* is used as shorthand for a non-aggregate SELECT query (and justifies the 'B' enumeration for the translation steps listed below). This can also be a subquery or a host query containing subqueries as conditions.

Given the above definitions for translations $tr(e)$ of an expression e , and $tr(B)$ of a condition block B , the translation $tr(Q)$ of a basic query Q becomes straightforward, as described by steps B1 - B9 below. These steps are given for illustrative purpose only, since determining whether Q is an aggregate query or not is done after the second step, as shown in the next section.

B1. The *Q.where* block and all its nested blocks are preprocessed in order to compute *bound*, *semibound*, *referenced*, and *blanks* sets of variables for each block.

B2. The translation context *TC* is created, with fields initialized as described in Section 5.4.5.1. If Q is a *subquery*, the *host query* translation context is cloned to *TC* instead, with *newconds* and *newvars* fields emptied, and *TC.bound* set including all variables bound in the parent query.

B3. A check for cross-referenced named expressions is performed, as described in Section 5.4.3.1, any such expressions are rewritten to their names, and additional BIND conditions are added as AmosQL translations to *TC.newconds*. If Q is a subquery, all named expressions in *Q.what* are considered to be cross-referenced.

B4. The translation TR_S of the *Q.what* list of a query is computed, with *TC* possibly updated. This also includes any additional variables used for ordering:

$$TR_S = tr(Q.what + Q.select-extra, TC)$$

B5. The translation TR_W of the *Q.where* block is computed, with *TC* possibly updated. This translation includes any extra conditions added to *TC.newconds* along the way.

$$TR_W = tr(Q.where, TC)$$

B6. The $declare(B)$ set of variables is computed for the basic block B as follows (with definitions of $Ref(B)$ and $B.blanks$ in Section 5.4.3.1):

$$\text{declare}(B, TC) = (\text{Ref}(B) \cup B.\text{blanks} \cup TC.\text{newvars}) \setminus TC.\text{bound}$$

B7. The *core translation* $tr_C(Q)$ of the query is constructed as

$$tr_C(Q) = \text{select } TR_S \text{ from } tr_D(\text{declare}(B, TC)) \text{ where } TR_W$$

or with `distinct` option added after `select` if $Q.\text{distinct}$ flag is set. (The $tr_D()$ function is defined in Section 5.4.5.3)

B8. If $Q.\text{orderby}$ list is empty the translation $tr_O(Q)$ is left unchanged

$$tr_O(Q) = tr_C(Q) \quad \text{iff } Q.\text{orderby} = \emptyset$$

otherwise, a facility to order the query results is added to the translation. For this purpose, the (possibly rewritten) expressions in $Q.\text{orderby}$ are looked up in the extended select list.

Formally, each entry in $Q.\text{orderby}$ contains an expression, and a direction specifier, to be translated to either 'inc' or 'dec' representation in AmosQL. Function $\text{expr}(Q.\text{orderby})$ returns the list of such expressions, and $\text{dir}(Q.\text{orderby})$ returns the aligned list of directions. Also, the function $\text{lookup}(e, \text{list})$ returns a 0-based position of expression e in the list . When applied to a list of expressions (in the first argument), it returns a list of such positions.

$$tr_O(Q) = \text{sortbagby}(tr_C(Q), \{ \text{lookup}(\text{expr}(Q.\text{orderby}), Q.\text{what} + Q.\text{select-extra}) \}, \{ tr(\text{dir}(Q.\text{orderby})) \})$$

B9. The *final translation* is constructed, by applying the segmentation facility. If both $Q.\text{offset}$ and $Q.\text{limit}$ are empty, the translation is unchanged, $tr(Q) = tr_O(Q)$ otherwise the `bsection1()` function call is wrapped around the previously constructed translation

$$tr(Q) = \text{bsection1}(tr_O, tr(Q.\text{offset}), tr(Q.\text{limit}))$$

where the translations of $Q.\text{offset}$ and $Q.\text{limit}$ default to 0, in order to indicate the absence of such bounds to the `bsection1()` function, as explained in Section 5.4.3.3.

Note that while `ORDER BY` is meaningless in subqueries, `OFFSET` and especially `LIMIT` clauses can prove quite useful (e.g. in formulating a *top-k* selection). The expressions in $Q.\text{offset}$ and $Q.\text{limit}$ are typically constant, but might as well depend on variables external to the query Q - i.e. those in the $TC.\text{bound}$ set.

5.4.5.7 Translating aggregate queries

Substantially more preprocessing is required for aggregate queries, with the first step being taken to determine if Q is an aggregate query.

A1 - A2. Same as B1 - B2 in the previous section.

A3. The expressions in $Q.what$, $Q.orderby$ and $Q.having$ are scanned for aggregate functions (not necessarily at the top level). The presence of $Q.groupby$ or $Q.having$ per se does not make Q an aggregate query - instead, these fields are only allowed in aggregate queries. Named aggregate expressions are put into $Q.agg$ together with their names. Other (not top-level) calls to aggregate functions are assigned surrogate names, like $agg:1$, $agg:2$ etc., and are also placed into $Q.agg$, with new variables added to $TC.newvars$. The places where aggregate function calls were found are rewritten with their mentioned names (as variables).

Only unique aggregate function calls are added to $Q.agg$ - if an expression is already found there, its name from $Q.agg$ is used for rewriting. Currently, equality of expressions is done by comparing the parse trees, which is simple but certain equivalent expressions might not be detected. For example $\max(?a + ?b)$ and $\max(?b + ?a)$ would be treated as different expressions. A more thorough comparison, e.g. one based on rewriting such expressions to a canonic form and sorting the commutative operands, awaits its implementation in the future versions of SSDM.

If $Q.agg$ remains empty then Q is not an aggregate query, and Steps B3 - B7 for the simple query translation are performed.

A4. The translation context TC will be used for the outer query, and the new TC_{INNER} context is cloned for the *inner* query, with *newvars* and *newconds* fields emptied.

If Q is an aggregate *subquery*, the already-cloned TC will be used as TC_{INNER} for the inner query and the original translation context of the *host query* will be used as TC for the outer query.

A5. The check for cross-referenced named expressions is performed, similarly to Step B3. However, named expressions already collected into $Q.agg$ are already rewritten to their names in place, and do not require extra BIND conditions to be translated into $TC.newconds$.

A6. Each named expression e in $Q.agg$, contains an aggregate function call at the top level, with a single argument. We denote the argument expression to the top-level function as $arg(e)$, that function name as $fn(e)$, and the naming variable as $var(e)$. The same functions applied to the $Q.agg$ list denote the lists of the respective objects. First we are going to compute the translations of the arguments to the aggregate functions, possibly updating the inner context:

$$TR_A = tr(arg(Q.agg), TC_{INNER})$$

A7. Similarly to B5, we translate the *where* block of the inner query, possibly updating TC_{INNER} and including $TC_{INNER}.newconds$ into the conjunction:

$$TR_W = tr(Q.where, TC_{INNER})$$

A8. The *select* list of the inner AmosQL query is constructed of two parts: (i) variables listed in $Q.groupby$, packed into a vector (unless grouping on a single variable) and (ii) the translations TR_A of arguments to the aggregate functions computed at step A6. The translation of the inner block is:

$$TR_{INNER} = select \{tr(Q.groupby, TC_{INNER})\}, TR_A \text{ from } tr_D(declare(B, TC_{INNER})) \text{ where } TR_W$$

Additionally, if $Q.groupby$ is empty, a short version of the inner query translation is made, which may be used in next step A9, and also for the purpose of translating function bodies:

$$TR_{INNER}^s = select TR_A \text{ from } tr_D(declare(B, TC_{INNER})) \text{ where } TR_W$$

The *distinct* option will be added into both translations after *select* if $Q.inner-distinct$ flag is set, i.e. if the *DISTINCT* keyword was encountered under at least one aggregate function. The set $declare(B, TC_{INNER})$ is computed in the same way as in B6, except with a different translation context - having more variables in $TC.bound$.

Note that the general-case translation TR_{INNER} is viable even if there was no *GROUP BY* clause in Q - the first element in *select* then becomes an empty vector, and the `groupby()` call introduced in step A10 is used to invoke the single-pass evaluation of multiple aggregate functions, effectively grouping on a constant empty group $\{\}$.

A9. In case of a single aggregate expression in $Q.agg$, empty $Q.groupby$ set, absence of $Q.having$, empty $TC.newconds$ list, and a single expression in $Q.what$ being rewritten to the only variable defined in $Q.agg$:

$$|Q.agg| = 1 \wedge Q.groupby = \emptyset \wedge Q.having = nil \wedge TC.newconds = \emptyset \wedge Q.what = var(Q.agg)$$

the outer query can be translated directly as a call to that aggregate function. This results in the following *core translation*, (possibly followed by applying a segmentation facility at step A15):

$$tr_C(Q) = tr(fn(Q.agg))(TR_{INNER}^s)$$

Otherwise, steps A10-A13 are preformed to construct a general-case *core translation* of aggregate query Q .

A10. The central condition of the outer query is the one calling `groupby()`. This condition is formed as:

$$TR_G = (\{tr(Q.groupby, TC)\}, tr(var(Q.agg), TC)) \text{ in } groupby(TR_{INNER}, \{tr_{\#}(fn(Q.agg))\})$$

The function $tr_{\#}()$ annotates the translated function names with syntactic features used for the functional arguments in AmosQL, e.g. `#'rdf:sum'` for the `sum()` aggregate function.

A11. If Q is a subquery, TR_G and $tr(Q.having, TC)$ conditions are simply added to $TC.newconds$ in the host query (sharing the translation context TC). The variables in $Q.groupby$ and $var(Q.agg)$, if not already referenced in the host query block, are added to $TC.newvars$.

If Q is not a subquery, the remaining steps proceed.

A12. The expressions in $Q.what$ (where certain subexpressions have been rewritten to variables at steps A3 and A5), are translated to a *select* list, possibly updating the translation context, and including any extra variables used for ordering:

$$TR_S = tr(Q.what \cup Q.select-extra, TC)$$

A13. The core translation of the outer query is now constructed as:

$$tr_C(Q) = \text{select } TR_S \text{ from } tr_D((Q.groupby \cup var(Q.agg) \cup TC.newvars) \setminus TC.bound) \text{ where } \text{andify}(TR_G, tr(Q.having, TC), TC.newconds)$$

or with `distinct` option added after `select` if $Q.distinct$ flag is set.

A14 - A15 The final translation $tr(Q)$ is constructed in the same way as for a basic query, following the steps B8 - B9.

5.4.5.8 Translating function definitions

The translation of a function definition statement S to its AmosQL equivalent $tr(S)$ depends on whether $S.body$ is a basic or an aggregate query.

For the *top-level-aggregate (TLA) functions* (i.e. ones containing aggregate queries in $Q.body$) it is important to separate *inner* and *outer* function definitions, which are similar to the concepts of inner and outer queries. This is required to correctly translate `ARGMIN()` and `ARGMAX()` to subqueries, as shown in the next section. However, not all kinds of functions can be passed as arguments to these second-order functions, leading us to the definition of an *argmax-compatible* function.

An *argmax-compatible* function has result *width* of 1, and is known to return a single result solution. The latter is not an enforced requirement, but a simple criterion used to disqualify certain classes of SciSPARQL functions from the need of being translated in the *argmax-compatible* way. For example, functions with a `GROUP BY` clause are not *argmax-compatible*, neither are the functions which require ordering or segmentation. The `HAVING` clause, designed for filtering the resulting solutions, also assumes their multiplicity, hence showing the lack of *argmax-compatibility*. For

simplicity, we will use the following criterion to identify *argmax-compatible* functions:

$$|S.body.what| = 1 \wedge S.body.groupby = \emptyset \wedge S.body.having = nil \wedge S.body.orderby = \emptyset \wedge S.body.offset = nil \wedge S.body.limit = nil$$

Note that the class of *argmax-compatible* functions might include both basic (i.e. non-aggregate) functions, and functions with any number of aggregate expressions collected into $Q.agg$. The `groupby()` function might be used in their translations, albeit then performing no actual grouping - only expressing single-pass computation of multiple aggregate expressions.

In the case of a TLA function, this criterion rules out any extra conditions that can be added to the $TC.newconds$ when translating the outer query. Such an outer query includes a single expression in $Q.what$, depending on a number of aggregate function calls collected into $Q.agg$. In AmosQL this will be translated either to a query with single condition in *where* clause, calling `groupby()` on a constant group `{}`, or a direct call to the aggregate function, as shown in A9.

The following steps outline the process of obtaining the translation $tr(S)$ of the function definition S .

F1. First, the translation $tr(S.body)$ of the function body (always being a SELECT query) is constructed following the steps A1 - A15 described above. At step A2 (same as B2) the $TC.bound$ set of the newly created translation context TC will contain $S.params$.

F2. If $S.body$ is a basic query, i.e. $S.body.agg = \emptyset$, or if S is not *argmax-compatible* function, a single function definition in AmosQL translates S :

$$tr(S) = \text{create function } tr(S.name) (tr_D(S.params)) \rightarrow \text{Bag of Literal as } tr(S.body)$$

Otherwise, the remaining steps are performed to translate an *aggmax-compatible* TLA function in two separate parts: an *inner* function translation $tr_{INNER}(S)$ and an *outer* function translation $tr_{OUTER}(S)$.

F3. The *inner* function is based on the *short* translation TR_{INNER}^s of the inner query, as defined in step A8. The inner function is defined as:

$$tr_{INNER}(S) = \text{create function } tr(S.name) : inner (tr_D(S.params)) \rightarrow \text{Bag of } (rptq('Literal', |Q.body.agg|)) \text{ as } TR_{INNER}^s$$

The function $rptq()$ makes a comma-delimited list of repeated code fragments, given a number or repetitions in the second argument.

F4. The translated body TR_{OUTER} of the outer function may be constructed in the simplified form:

$$TR_{OUTER} = tr(fn(Q.agg)) (select tr(S.name) : inner (tr(S.params)))$$

if the conditions from A9 hold:

$$|S.body.agg| = 1 \wedge TC.newconds = \emptyset \wedge S.body.what = var(Q.agg)$$

Otherwise a general-case translation, similar to the one defined in steps A10, A12, A13, under certain simplifications arising from *argmax-compatibility* criterion stated above:

$$TR_G = (\{\}, tr(var(S.body.agg), TC)) \text{ in } groupby(select \{\}, tr(S.name):inner(tr(S.params)), \{tr_{\#}(fn(S.body.agg))\})$$

$$TR_S = tr(S.body.what, TC)$$

$$TR_{OUTER} = select TR_S \text{ from } tr_D((var(S.body.agg) \cup TC.newvars) \setminus Q.params) \text{ where } andify(TR_G, TC.newconds)$$

F5. The outer function translation $tr_{OUTER}(S)$ extends the translated body by supplying a header:

$$tr_{OUTER}(S) = \text{create function } tr(S.name)(tr_D(S.params)) \rightarrow \text{Bag of Literal as } TR_{OUTER}$$

F6. The overall result of translating an *argmax-compatible* aggregate function is a pair of function definitions

$$tr(S) = tr_{INNER}(S); tr_{OUTER}(S)$$

The rewritten SELECT expression $S.body.what$ and the list of collected aggregate functions $S.body.agg$ are saved in SSDM in the TLA hash table, with $S.name$ serving as a key:

$$agg(S.name) = S.body.agg$$

$$expr(S.name) = S.body.what$$

This information is needed for the ARGMIN() and ARGMAX() translations introduced in the next section.

5.4.5.9 Translating ARGMIN and ARGMAX

A call to ARGMIN() or ARGMAX() is a kind of expression, as listed in Section 5.4.5.4. The only argument is a function of a single parameter (or a lexical closure with a single free argument), which is *argmax-compatible*, as defined in the previous section, and returns a single value for each binding of its parameter. Another fundamental requirement (beyond the scope of the translator) is that the function needs to have a finite domain, i.e. it can be evaluated without externally binding its argument.

SSDM defines the `rdf:argmin()` and `rdf:argmax()` aggregate functions in AmosQL with identical signatures: both of them take a bag of (*arg*, *res*) tuples, and return the encountered *arg* values where *res* was found at its minimum or maximum. A simple derived Amos function is internally

flattened after its translation to ObjectLog, so that the chain of predicates connecting its argument and finiteness conditions (e.g. a graph pattern) can be reversed by the optimizer towards the set of all possible argument values. The same is, unfortunately, not technically possible with functions involving top-level aggregate operations, which confine their finiteness conditions in a separate 'inner' calculus expression, and consequently, a separate execution plan, so that any external arguments can only be passed inwards, but not outwards.

Though this can be regarded as purely architectural restriction with ObjectLog; there is an interesting workaround, which relies on directly addressing the 'inner' part of such a function in order to iterate over all possible argument values. The subqueries translating `ARGMIN()` and `ARGMAX()` calls are constructed based on this idea, described in the following steps.

Without any loss of generality, let e be an expression calling a second-order function $e.fn$ with a single argument $a = e.args$. This argument is a *closure*, consisting of a function name $a.fn$ and list of its arguments $a.args$ containing exactly one *asterisk*. Though SciSPARQL allows omitting the lexical closure syntax when passing a unary function to `ARGMIN()` or `ARGMAX()`, at the parsing phase such a lexical closure is constructed, with $a.args$ consisting of a single *asterisk*.

M1. First, the information on $a.fn$ is looked up in the *TLA* hash table in SSDM - whether it was a *TLA* function translated to the *inner* and *outer* function definitions on steps F3 - F6, with additional information $agg(a.fn)$ and $expr(a.fn)$ available, or as a basic function on step F2, in which case $agg(a.fn)$ is empty.

M2. The closure a is translated to a function call, with *asterisk* substituted to a newly-generated variable $arg = newvar(TC)$. For this purpose, an alternative translation function $tr_1(e)$ is defined, differing from the expression translation specified in Section 5.4.5.4 in two ways: (i) *asterisk* is translated to arg , postfixed with `:i`, in order to avoid possible collision with the outer level of the query, and (ii) the closure function $a.fn$ is translated to its inner name, unless $a.fn$ is a basic function:

$$tr_1(asterisk) = arg:i$$

$$tr_1(a.fn) = tr(a.fn):inner \quad \mathbf{iff} \quad agg(a.fn) \neq \emptyset$$

M3. Even if the `groupby()` operator was not used in the translation of the *TLA* function $a.fn$, evaluating our second-order function involves grouping: for each possible argument value a number of *inner function* solutions is generated and passed to the aggregate function(s) used in $a.fn$.

$$TR_G = groupby((TR_U, \{tr_{\#}(fn(agg(a.fn)))\}) \quad \mathbf{iff} \quad agg(a.fn) \neq \emptyset$$

where TR_U is a subquery returning tuples of $a.fn$ argument and the under-
aggregate values to be grouped:

$$TR_U = \text{select } arg:i, tr_I(a) \text{ from Literal } arg:i$$

For the basic functions, TR_U needs neither grouping nor aggregation, and
comes as a direct argument to e.g. `rdf:argmax()`.

M4. If the outer function $a.fn$ was translated in the simplified form, i.e.
the SELECT expression $expr(a.fn)$ is just a name $var(agg(a.fn))$ of the only
aggregate function call, then

$$tr(e) = tr(e.fn)(TR_G)$$

The same works for the basic functions, formally achieved by $TR_G = TR_U$.
In all other cases, the next step concludes the process.

M5. Since the outer function consists of a single expression and `groupby()`
condition, reproducing it is a simple task. The expression $expr(a.fn)$ is
translated exactly in the same way as it was in the outer function definition,
and bag of (arg, res) pairs is fed to e.g. `rdf:argmax()`:

$$tr(e) = tr(e.fn)(\text{select } arg, tr(expr(a.fn)) \text{ from Literal } arg, \\ tr_D(var(agg(a.fn))) \text{ where } (arg, tr(var(agg(a.fn)))) \text{ in } TR_G)$$

5.4.5.10 Examples

The translations defined in the four previous sections are illustrated by the
following examples of *aggmax-compatible* functions. For simplicity, all the
following definitions use a common prefix declaration:

PREFIX : <http://example.org/data/#>

A basic function f_0 selecting value of property `:x` from a given graph
node

```
DEFINE FUNCTION f0(?a) AS
SELECT ?x
WHERE { ?a :x ?x }
```

is translated as

```
create function rdf:f0(Literal a) -> Bag of Literal
as select x
from Literal x
where (a, URI('http://example.org/data/#x'), x) in GRAPH(0);
```

and `ARGMAX(f0(*))` in SciSPARQL translates to:

```
rdf:argmax(select arg:1:i, rdf:f0(arg:1:i) from Literal arg:1:i);
```

Function *f1*, computing the sum of all such :x values

```
DEFINE FUNCTION f1(?a) AS
SELECT (sum(?x) AS ?res)
WHERE { ?a :x ?x }
```

is translated to the pair of definitions

```
create function rdf:f1:inner(Literal a) -> Bag of Literal
as select x
  from Literal x
  where (a, URI('http://example.org/data/#x'), x) in GRAPH(0);

create function rdf:f1(Literal a) -> Bag of Literal
as rdf:sum((select rdf:f1:inner(a)));
```

and ARGMAX (f1 (*)) also uses a simplified translation:

```
select rdf:argmax(groupby((select arg:1:i, rdf:f1:inner(arg:1:i)
                           from Literal arg:1:i), #'rdf:sum'));
```

The translation of the example ARGMAX () call from Section 4.3 is similar to this case:

```
select rdf:argmax(groupby((select arg:1:i,
                               rdf:sum_diag_positive:inner(arg:1:i)
                           from Literal arg:1:i), #'rdf:sum'));
```

where the sum_diag_positive () function, defined in Section 4.2, is translated to the outer and inner parts:

```
create function rdf:sum_diag_positive:inner(Literal r)
-> Bag of Literal
as select aref(a,0,rdf:minus(i,1)),0,rdf:minus(i,1)
  from Literal i, Literal a
  where (r, URI('http://example.org/data/result'), a) in GRAPH(0)
        and aref(a,0,rdf:minus(i,1)),0,rdf:minus(i,1)>0;

create function rdf:sum_diag_positive(Literal r) -> Bag of Literal
as rdf:sum((select rdf:sum_diag_positive:inner(r)));
```

Function *f2*, computing a numeric range of :x properties for the given node

```
DEFINE FUNCTION f2(?a) AS
SELECT (max(?x) - min(?x) AS ?range)
WHERE { ?a :x ?x }
```

is translated as follows:

```
create function rdf:f2:inner(Literal a) -> Bag of (Literal, Literal)
as select x, x
  from Literal x
  where (a, URI('http://example.org/data/#x'), x) in GRAPH(0);

create function rdf:f2(Literal a) -> Bag of Literal
as select rdf:minus(agg:1, agg:2)
  from Literal agg:1, Literal agg:2
  where ( {}, agg:2, agg:1)
```

```

in groupby((select {}, rdf:f2:inner(a)),
           {'#rdf:min', '#rdf:max'});

```

and ARGMAX(f2(*)) reproduces the expression in the *outer* function:

```

select rdf:argmax(select arg:1, rdf:minus(agg:1, agg:2)
                 from Literal agg:1, Literal agg:2, Literal arg:1
                 where (arg:1, agg:2, agg:1) in
groupby((select arg:1:i, rdf:f2:inner(arg:1:i) from Literal arg:1:i),
        {'#rdf:min', '#rdf:max'}));

```

5.4.5.11 Translating CONSTRUCT queries

A CONSTRUCT query returns a new RDF graph in form of triples. For each solution of a query, exactly the same number of RDF triples is created, as there are *triple templates* in *Q.what*. The triple templates are quite similar to *triple patterns*, except that they are used for constructing, not for matching.

The translation is an AmosQL query of width 3, effectively returning a *union* of results specified by each triple template for each solution corresponding to the WHERE block of the CONSTRUCT query. Since no *unbound* values are allowed in the result graph, the `rdf:bound()` check needs to be passed using a triple template with a *semibound* variable. The following example illustrates this approach:

Q14 (W3C Standard SPARQL): Extract all realizations of all experiments into a new graph as nodes of type `ex:OldRealization` and the properties `ex:a` and `ex:c` containing these two parameters

```

PREFIX ex: <http://udbl.uu.se/ex#>
CONSTRUCT { ?r a ex:OldRealization ; ex:a ?a ; ex:c ?c }
WHERE { ?r a ex:OurExperimentRealization ;
        ex:parameter_A ?a .
OPTIONAL { ?r ex:parameter_C ?c } }

```

The translation constructs each triple template as an alternative solution, and performs additional check on the *semibound* variable `?c`:

```

select c:s, c:p, c:o
from Literal r, Literal c, Literal a,
     Literal c:s, Literal c:p, Literal c:o
where (r, URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
       URI('http://udbl.uu.se/ex#OurExperimentRealization'))
in GRAPH(0)
and (r, URI('http://udbl.uu.se/ex#parameter_A'), a) in GRAPH(0)
and optional((r, URI('http://udbl.uu.se/ex#parameter_C'), c)
             in GRAPH(0))
and ((c:s = r and
      c:p = URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#type')
      and c:o = URI('http://udbl.uu.se/ex#OldRealization'))|
     or (c:s = r and c:p = URI('http://udbl.uu.se/ex#a') and c:o = a)
     or (c:s = r and c:p = URI('http://udbl.uu.se/ex#c') and c:o = c
         and rdf:bound(c)));

```

Formally, the translation process can be defined as $tr(Q)$, where *Q.what* is a set of triple templates *tt*, each having *tt.s*, *tt.p* and *tt.o* fields. The query

always uses the same variables for selection, binding them alternatively, according to each triple template.

C1. A new translation context TC is created.

C2. The translation TR_W of the $Q.where$ block is computed, with TC possibly updated. This translation includes any extra conditions added to $TC.newconds$ along the way:

$$TR_W = tr(Q.where, TC)$$

C3. The basic translation $tr_B(tt)$ of triple template tt is defined as

$$tr_B(tt) = c:s = tr(tt.s) \text{ and } c:p = tr(tt.p) \text{ and } c:o = tr(tt.o)$$

and the additional binding checks are added for all semibound variables $var(tt)$, used in the given triple template:

$$tr(tt) = andify(tr_B(tt), tr_{bound}(var(tt) \cap Q.semibound))$$

where

$$tr_{bound}(v) = rdf:bound(v)$$

C4. The final translation $tr(Q)$ is constructed:

$$tr(Q) = \text{select } c:s, c:p, c:o \text{ from } tr_D(\text{declare}(Q, TC)), \text{ Literal } c:s, c:p, c:o \text{ where } TR_W \text{ and } orify(tr(Q.what.conds))$$

where $orify()$ function builds disjunction of a given list of translated conditions.

5.4.5.12 Translating SPARQL updates

In all the above translations, the Amos function $GRAPH()$ was ultimately addressed for matching the triple patterns. Though in most recommended settings, as discussed in the following chapters, the triples reside in the main memory in the SSDM server, its extensible architecture allows any external API, to be invoked for the purpose of querying $GRAPH()$, including formulation of foreign queries to the external storage systems. The query-only access to $GRAPH()$ assumes that different mechanisms are required for inserting and removing the triples.

For this purpose SSDM defines the $rdf:insert()$ and $rdf:remove()$ functions, which are generic and encapsulate the extensibility mechanisms, similarly to $GRAPH()$. Since these functions contain side effects, they cannot be called from an AmosQL query. Fortunately, Amos II allows hybrid semantics, combining a query and a procedure: for each solution of a given query, a number of operations are performed (update operations in our case). In order to avoid inserting *unbound* values, the *procedural if* syntax can be used. The following example illustrates this approach:

Update1 (W3C standard): rename `ex:parameter_A` and `ex:parameter_C` properties to `ex:a` and `ex:c` respectively.

```

PREFIX ex: <http://udbl.uu.se/ex#>
DELETE { ?r ex:parameter_A ?a ; ex:parameter_C ?c }
INSERT { ?r ex:a ?a ; ex:c ?c }
WHERE { ?r a ex:OurExperimentRealization ;
        ex:parameter_A ?a .
        OPTIONAL { ?r ex:parameter_C ?c } }

```

The translation is rather straightforward, providing a remove or insert operation for each triple pattern in DELETE and INSERT clause respectively, with additional binding check, needed only for insert operation.

```

for each Literal r, Literal c, Literal a
  where (r, URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
        URI('http://udbl.uu.se/ex#OurExperimentRealization'))
    in GRAPH(0)
    and (r, URI('http://udbl.uu.se/ex#parameter_A'), a) in GRAPH(0)
    and optional((r, URI('http://udbl.uu.se/ex#parameter_C'), c)
                in GRAPH(0))
begin
  rdf:remove(0, r, URI('http://udbl.uu.se/ex#parameter_A'), a);
  rdf:remove(0, r, URI('http://udbl.uu.se/ex#parameter_C'), c);
  rdf:insert(0, r, URI('http://udbl.uu.se/ex#a'), a);
  if rdf:bound(c) then
    rdf:insert(0, r, URI('http://udbl.uu.se/ex#c'), c);
end;

```

Formally, an update statement S contains an $S.where$ block, $S.delete$ and $S.insert$ lists of patterns, and $S.with$ URI for the graph to perform the updates on. The latter is translated to integer id by $GDict(S.with)$, defaulting to 0. Additionally, as with queries $S.from$ might contain the list of graphs listed in the USING clause and addressed in the WHERE clause.

U1. A new translation context TC is created. If $S.from$ is omitted but $S.with$ is present, $TC.source$ stores the translation $GRAPH(GDict(S.from))$.

U2. Similarly to C2 and A7, the translation TR_W of the $Q.where$ block is computed, with TC possibly updated. This translation includes any extra conditions added to $TC.newconds$ along the way.

$$TR_W = tr(Q.where, TC)$$

U3. The *delete* and *insert* patterns are translated in a similar way:

$$tr_{del}(tp) = \text{rdf:remove}(GDict(S.with), tr(tp.s), tr(tp.p), tr(tp.o));$$

$$tr_{ins0}(tp) = \text{rdf:insert}(GDict(S.with), tr(tp.s), tr(tp.p), tr(tp.o));$$

However, the insert calls should be skipped if not all used variables are bound, so the final translation of an *insert* pattern includes an `if` condition:

$$tr_{ins}(tp) = \text{if } \text{andify}(tr_{bound}(var(tp) \cap Q.semibound)) \text{ then } tr_{ins0}(tp)$$

iff $var(tp) \cap Q.semibound \neq \emptyset$

If there are no *semibound* variables in the *insert* pattern then $tr_{ins}(tp) = tr_{ins0}(tp)$

U4. The final translation $tr(S)$ of the update statement S is constructed:

```
tr(S) = for each trD(declare(S, TC)) where TRW begin
concat(trdel(S.delete), trins(S.insert)) end
```

Note that the `INSERT DATA` and `DELETE DATA` statements are handled differently, in accordance with W3C Recommendations [156]: the corresponding `rdf:insert()` and `rdf:remove()` calls are made as the syntactic parsing stage, so that the triple patterns (where no variables are allowed) are not accumulated in the parse tree. This approach allows processing arbitrarily long statements for bulk updates of *RDF with Arrays* datasets.

5.5 Polymorphic Properties Problem

In Section 5.3 we state that SSDM supports backwards-compatibility when handling collections of numbers as arrays. This assumes that along with the new way to address e.g. the first element `?A[1]` of a collection `?A`, the old way, using a triple pattern is also supported:

```
?A rdf:first ?x
```

In the current version of SSDM, this is guaranteed by translating triple patterns with `rdf:first` and `rdf:rest` as disjunctions, e.g.

```
(rdf:first(A) = x
or (A, URI('http://www.w3.org/1999/02/22-rdf-syntax-ns#first'), x)
in GRAPH(0))
```

where the Amos function `rdf:first()` makes a run-time type checking, and if the argument is an array, returns the first element, which, in the case of a multidimensional array will be the first $(n-1)$ -dimensional slice

Effectively, the OR branches are mutually exclusive: if `A` is not an array, `rdf:first()` yields no result, but a triple pattern might have some bindings for `x`. If `A` is an array, the RDF triple pattern will never bind, since array value can never be a *subject* in a triple - only an *object*.

5.5.1 Directionality Problem

As illustrated by dataset on Figure 4 in Section 2.3.5.1, a triple pattern

```
?x rdf:first 2
```

can easily be matched, putting the blank node `_:c` into the solution. Even though the graph representing an array can be quite big, dedicated RDF

stores with powerful indexing techniques, such as RDF-3X [112], will find the match in logarithmic time - equivalent to matching e.g.

```
_:c rdf:first ?y
```

In our case the graph in Figure 4 is stored as a numeric array. So there is actually no blank node `_:c`, and a 1D array consisting of single element 2 can be regarded as an equivalent replacement of `_:c`. Such an array value can easily be created on demand as a derived array, without copying any array data. Similarly, the whole array is a replacement for the `_:a` node.

However, while finding the first element of a given array is simple and takes constant time, finding an array (or any 1D subarray) with a given first element would involve a linear scan through every array in the graph, resulting in a linear complexity w.r.t. the total volume of array data stored.

Solution: Currently, `rdf:first()` and `rdf:rest()` functions are implemented as uni-directional, effectively forbidding queries with triple patterns like the first one in this section, unless `?x` can be bound otherwise. This results in a certain limitation of the backwards-compatibility feature, motivated by the intended use of SSDM - storing and querying massive array data as part of *RDF with Arrays* datasets.

5.5.2 Normalization Problem

The typical use of `rdf:first` and `rdf:rest` predicates is chaining. For example, in order to address `?A[2,2]`, in standard SPARQL one would use a graph pattern

```
?A rdf:rest [ rdf:first [ rdf:rest [ rdf:first ?x ] ] ]
```

or, in SPARQL 1.1 one might prefer the path expression syntax

```
?A rdf:rest/rdf:first/rdf:rest/rdf:first ?x
```

both of which effectively translate to four triple patterns. Since in our case, each such triple pattern is translated to a disjunction of two alternatives in AmosQL, further query transformations always involve normalization to *disjunctive normal form (DNF)*. A conjunction of n binary alternatives (of unique expressions, as in our case) is transformed to a disjunction of 2^n branches, each containing a conjunction of n terms, thus resulting in a combinatorial explosion of the execution plan w.r.t. the number of `rdf:first` and `rdf:rest` patterns in the query.

Solution: In our case, we effectively get a union of 16 branches - however, only 5 of them are theoretically capable of yielding any solutions. This is due to the fact that `rdf:first()` and `rdf:rest()` functions may only return arrays, scalar values, or `rdf:nil`, and neither of these values, according to the RDF standard, can be a subject in an RDF triple. Hence, any union branch containing a chain of predicates where the result of such a function is used as a subject in a triple pattern can be safely ruled out. The remaining 5 alternatives contain chains of triple patterns followed by a chain of function calls, either chain consisting of 0 to 4 elements. This set is illustrated by the following diagram, where *tp* stands for a triple pattern with an `rdf:first` or `rdf:rest` predicate, and *fn()* stands for a corresponding `rdf:first()` or `rdf:rest()` function call:

$?A \rightarrow tp \rightarrow tp \rightarrow tp \rightarrow tp \rightarrow ?x \cup$
 $?A \rightarrow tp \rightarrow tp \rightarrow tp \rightarrow fn() \rightarrow ?x \cup$
 $?A \rightarrow tp \rightarrow tp \rightarrow fn() \rightarrow fn() \rightarrow ?x \cup$
 $?A \rightarrow tp \rightarrow fn() \rightarrow fn() \rightarrow fn() \rightarrow ?x \cup$
 $?A \rightarrow fn() \rightarrow fn() \rightarrow fn() \rightarrow fn() \rightarrow ?x$

With the number of viable branches being $n+1$, such reduced normalization would rearrange $2n$ original terms to $n(n+1)$ terms in the normalized expression, resulting in only a linear complexity increase.

We implement this reduction as a customized behavior of the ObjectLog normalization algorithm for predicated expressions. The normalizer will rule out any resulting union branches, where the same variable appears both in the place where only a URI is allowed (which is: *subject* and *property* of a translated triple pattern) and in a place where only non-URI values (or `rdf:nil`) may appear - e.g. the result of unidirectional functions `rdf:first()` and `rdf:rest()`. Other unidirectional functions, known to never return any URIs, such as `adim()`, `aref()`, `asub()` implementing array functionality, or standard SPARQL functions operating only on string or numerical values, such as `concat()` or `round()`, also contribute to this list.

A union branch *e*, if it is a conjunction of predicates, is viable **iff**:

$$uriOnlyVars(e) \cap nonUriVars(e) = \emptyset$$

where the set *uriOnlyVars()* contains variables used on *subject* and *property* positions in graph patterns inside the conjunction, and the *nonUriVars()* set contains the variables used on the respective places in the functions mentioned above.

This solution completely solves the task of preventing combinatorial explosion of the execution plans due to normalization of graph patterns, containing chains of `rdf:first` and `rdf:rest`, and is generally useful reducing the sets of possible alternatives when executing disjunctive queries.

6 External Storage of RDF with Arrays

Scientific SPARQL is designed to help scientists and engineers in the tasks of storing, annotating, and querying large amounts of numeric data. Storage and query scalability is of a major concern, and is addressed by a wide range of storage alternatives for *RDF with Arrays* data.

As Figure 8 in the beginning of the previous chapter shows, SSDM includes a *generic storage back-end/wrapper interface*. This interface is used for partially translating SciSPARQL queries (in the form of ObjectLog predicates) to the API calls of the respective storage system. This might be file access, client-server communication (with systems like Chelonia [114, 166] or Rasdaman [16]), or SQL queries sent over JDBC to any relational DBMS.

There are two classes of application configurations where this mechanism is utilized:

Wrapper configuration - the data is already stored in a certain form in some storage system, which might be an RDBMS, an array store, or just a collection of files. We support *mappings* of this native data model to *RDF with Arrays*. This might be a fully standard mapping, like RDB-to-RDF [127], or a highly ad-hoc mapping, e.g. one involving extraction of the metadata from file names, and converting it to RDF (as we did in [6]). Section 2.3 above presents an overview of the general data mappings available. As a result, we are able to query the data with SciSPARQL, bypassing any bulk-loading steps.

Back-end configuration - we use SSDM as a primary service to store the data, in the form of *RDF with Arrays*. Internally, SSDM delegates the storage (either entirely, or arrays-only) to a back-end storage. This might either be a dedicated array store (like Rasdaman), a collection of files on the server, or any RDBMS. The translation of SciSPARQL queries to the back-end API calls is then governed by the chosen storage schema or convention. The key difference from the wrapper scenario is that SSDM controls how the imported data is going to be stored in the back-end system, instead of just mapping the data which is already stored in a certain way to the *RDF with Arrays* model.

When it comes to array processing, the *generic back-end/wrapper interface* makes no difference between the scenarios. Exactly the same techniques are used to accumulate array operations, and then to retrieve the array data in a lazy fashion - Section 6.1 describes these techniques in a way independent of the particular storage system. We call this part *array storage extensibility interface* (ASEI).

For each interfaced storage system there might be a variety of choices regarding how exactly ObjectLog predicates should be translated to the API calls (e.g. JDBC calls), and how the array subsets should be retrieved from the external storage. Different strategies need to be compared, so that the optimal ones could be chosen in each particular case. Section 6.2 presents the alternatives of storing arrays in a conventional RDBMS and of processing array queries in such a configuration. Experimental comparison is presented using a mini-benchmark for array queries in Section 6.3, and on the real-life application in Section 6.4, where we compare the performance of SSDM with different RDBMS back-ends to the original manual implementations of the same computational tasks in Matlab.

Even though some storage systems (like RDBMSs, as we show in Section 6.2) are well-suited to store *RDF with Arrays*, and SSDM is capable to translate whole SciSPARQL queries to SQL [182], here we mainly concentrate on the optimizations for array data retrieval. The reason is simple: SSDM includes the highly efficient main-memory database engine of Amos II [136], and can certainly handle the classical RDF processing using its native main-memory data structures. Deployed as a server process, SSDM can be instructed to cache the RDF part of the dataset completely, leaving only the arrays for on-demand access. In our target applications, it is the array data that offers a scalability challenge, while the metadata in the form of RDF graph fits into main memory. Again, this is not a requirement or limitation, just the prospective usage scenario we build our evaluations upon.

6.1 Array Storage Extensibility Interface

We will refer to arrays stored in files, array stores or DBMS back-ends as *externally-stored* arrays, in contrast to *resident arrays* stored in the main memory of SSDM. Externally-stored arrays are represented in an *RDF with Arrays* graph with *array proxy* objects. For the query user, array proxies are indistinguishable from the resident arrays, as SSDM takes care of *resolving* the array proxies to resident arrays on demand.

In the beginning of Chapter 5 we presented a *main-memory scenario*, where *file links* like `<file://realization_1.mat#Res>` were converted to

memory-resident arrays at the data loading stage. In order to save memory and data loading time, it is possible to read the same *extended Turtle* file also in the *wrapper scenario*, so that the `GRAPH()` function would internally store array proxies, each containing a file name, e.g. `realization_1.mat` and a label (i.e. Matlab variable name), e.g. `Res`, as a way to identify an array in the specific storage system.

Another important piece of information is the *kind* of *array proxy*, identifying the external storage system itself. The same RDF graph might refer to arrays stored in files and different databases or array stores SSDM is connected to. Since most array operations work exactly in the same way on memory-resident arrays and externally stored ones, the main-memory *array descriptor* is considered a particular case of array proxy, with a reserved *kind* value, telling that SSDM's own in-memory storage is used for storing the array.

The Array Storage Extensibility Interface (ASEI) thus consists of three custom methods that need to be registered with SSDM for each kind of array storage:

- A custom *array loader* for loading a memory-resident array into the external array storage. It returns a new array proxy representing the loaded array. For example, if SSDM is configured to store the arrays `.mat` files, the array proxy will refer to the name of the file and a label inside it indentifying the array.
- A custom *URI decoder* method that constructs the array proxy for a given URI (file link). If a URI does not have all the information about the array shape and element type (as in the example in the Section 5.1.1), the decoder will have to access the storage system in order to retrieve this necessary information.
- A custom *proxy resolver*, which creates a memory-resident arrays from a proxy object by accessing the storage system corresponding to the proxy kind. The resolver is called by `APR()`, when it needs to materialize an array proxy.

Registering these three methods with ASEI generates a new array proxy kind, which is then used as an identifier for the interfaced array storage system.

Additionally, the `_sq_resolve_file_links_` flag governs whether array proxies should be eagerly resolved after creation from file links, at the stage of data import (the *main-memory scenario* presented in Section 5.1), or retained for possible later retrieval on-demand (the *rapper scenario* in Chapter 7). In a pure *back-end scenario*, file links are eagerly resolved, and the array loader is immediately called to transfer the array to the configured back-end array storage system, resulting in array proxies of the corresponding *kind* (as we did in Section 6.4).

Embeddings of SciSPARQL into algorithmic languages open a way to explicitly create array proxies, before inserting them into an *RDF with Arrays* graph using the SPARQL Update syntax (explained in Section 3.9). For example, in the Matlab integration described in Chapter 7, the Matlab function `store()` is used to store the array in a new `.mat` file on the server file system, and obtain an array proxy for subsequent insertion and access.

The array proxy lifecycle during the query execution can be described in few words as follows: *original* array proxies are retrieved from an *RDF with Arrays* graph, just like any other nodes, at the stage of graph pattern matching. *Derived* array proxies might be produced when applying operations such as array *range selection* or *projection*, in the same way as derived memory-resident arrays are produced (Section 5.2.2). Finally, when the actual array data is required for computation, array proxies are *resolved* to memory-resident arrays, by calls to the `APR()` function.

The next two sub-sections describe the changes to translated queries introduced by the need to resolve the possible proxies, and the internal structure of `APR()` function. A discussion of the approach follows.

6.1.1 Placing APR Calls into the Translation

Array proxies are in most respects identical to *array descriptor* objects, and thus serve to accumulate array operations without actually accessing the underlying array data, thus implementing a lazy approach to array data loading. Except for the generalized transposition, all array operations described in Section 5.2.2 produce *derived arrays* that are smaller than the original ones, so that a lazy approach typically results in lesser amounts of data read into main memory.

A single-element access to an externally stored array, also results in a derived 0-dimensional array proxy, pointing to that element. We will refer to these as *single-element proxies*, in contrast to *(sub)array proxies* referring to either original arrays or array subsets, and use the term *array proxy* as a union of those.

In case of any external array proxy *kind* registered (and thus ASEI is considered to be active), the SciSPARQL translator will insert calls to the `APR()` function in the AmosQL query translations. For this purpose, the translator traverses the expressions leaf-to-root during the translation (described in Section 5.4.5.4), guided by the following rules:

- a query variable, participating only in *value* position in triple patterns **may be** bound to a *(sub)array proxy*;
- a parameter to a functional view **may be** bound to an *array proxy*;
- the result of array *range selection* or *transposition* **may be** a *(sub)array proxy*, **if** the operand **may be** an *array proxy*;

- result of array projection **may be** an *array proxy*, **if** the operand **may be** an *array proxy*;
- **if** an expression whose value **may be** an *array proxy* is used as an argument to any internal function accepting arrays (including aggregate functions like `SUM()` and the overloaded arithmetic operations like '+'), an `APR()` call **should be** inserted to wrap it;
- **if** an expression whose value **may be** a *single-element proxy* is used as an argument to an internal function expecting a number (or used in an array dereference expression), an `APR()` call **should be** inserted to wrap it;
- **if** an expression whose value may be an array proxy is used as an argument to a user-defined foreign function, an `APR()` call **should be** inserted to wrap it;
- **if** the `_sq_resolve_results_` flag is set, and a top-level `SELECT` expression in a query **may be** an *array proxy*, an `APR()` call **should be** inserted to wrap it.

Additionally, SSDM keeps track of functions defined as parameterized SciSPARQL queries, which **may** return a proxy.

For example, query *Q15*, retrieving an average value of the second column of `ex:result` matrix, corresponding to the realization with `id = 1` of `ex:Experiment1` (from the dataset *GI*), here given in a reduced form:

```
SELECT (array_avg(?A[:,2]) AS ?col2_avg)
WHERE { [] ex:id 1 ; ex:result ?A }
```

will be translated to the following AmosQL query:

```
select rdf:array_avg(APR(aref(a,1,1)))
from Literal a, Literal g:0
where (g:0, URI('http://udbl.uu.se/ex#id'), 1) in GRAPH(0)
and (g:0, URI('http://udbl.uu.se/ex#result'), a) in GRAPH(0);
```

so that the array proxy resulting from projection of matrix `?a` to the second column gets resolved before applying the `array_avg()` computation.

As a not-so-trivial example of how the above rules are applied, consider a query:

```
SELECT (transpose(?A)[?B] + round(f(?C)) AS ?result)
WHERE { _:x :a ?A ; :b ?B ; :c ?C }
```

Here, SSDM has to track the `SELECT` expression leaf-to-root, considering whether each intermediate result may be a proxy, and what type of proxy. Figure 17 illustrates the process: expressions whose values may be proxies (either type of) are shown in gray color - same with the arrows showing the dependencies.

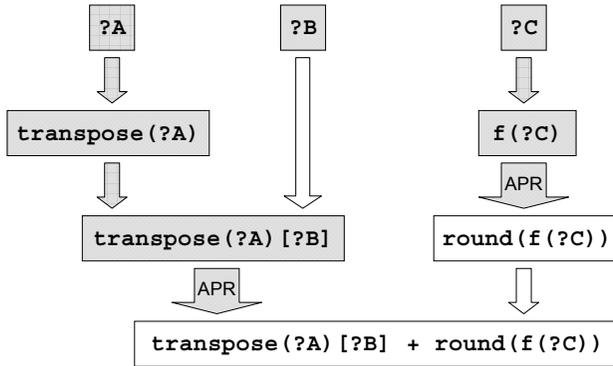


Figure 17. Placement of `APR()` calls into an expression tree

All three variables are only bound by their appearance on the *value* positions in triple patterns - hence each of them **may be** a *sub(array) proxy*, though not a *single-element proxy*¹⁴. If they are, the `APR()` function will return an array of one or more dimensions. However, variable `?B` is used in a way only a scalar *Integer* value may be used. If `?B` happens to be a proxy, the expression `'transpose(?A) [?B]'` will not return due to the invalid subscript type (array). Hereby, SSDM assumes that in all valid query solutions `?B` is not an array proxy (neither it is array or any other non-integer value), so there is no need to insert `APR()` call around it.

In this example we also assume that `f()` is a function defined as a parameterized SciSPARQL query, and is listed among those which **may** return a proxy. If it were not, we would not need an `APR()` call around its result. The `round()` function is applicable both to scalar values and arrays, and hence needs its argument to be materialized.

Overall, as Figure 17 shows, SSDM prefers to keep the possibility of proxies as far towards the root of an expression as possible, since there is always a chance that the amount of relevant array data will be reduced to a subset, or the retrieval will be skipped altogether due to filtering.

6.1.2 APR Implementations

Since SciSPARQL is a purely dynamic-typed language, whatever is passed to the `APR()` function may be an array proxy, or may be any other RDF term, or a memory-resident array. For this reason, `APR()` performs a type check first, and then a check for array proxy *kind*. The argument is returned without changes unless it is an external array proxy that needs resolving.

¹⁴ We do not store scalar values as proxies in an *RDF with Arrays* graph. If a whole graph is stored in a back-end (as in Section 6.2), the scalar values are retrieved/cached in the process of graph pattern matching or caching, so the lazy retrieval does not apply to *RDF Literals*.

In the latter case, `APR()` looks up the respective *proxy resolver*, registered for the given *array proxy kind*, and calls it, passing the array proxy object as argument. Besides the information about the accumulated selection and projection operations (as explained in Section 5.2.2 for *array descriptors*), an array proxy holds the information sufficient to identify the original array in the specific storage system, and to retrieve its relevant subset.

For example, the `.mat` file *proxy resolver* would invoke the HDF/Matlab API to access the particular part of the array specified, and read the array elements into to a newly allocated memory-resident array. The flexible nesting order of dimensions, supported by SSDM, allows optimizing this process, by matching the order to the one exposed by the storage system's API, and thus allowing to transfer the array content in large fragments. We describe the process of discovering such fragments below in Section 6.2.4.2, where it is critical for identifying the array chunks to retrieve.

Currently, SSDM has extensions to address or store arrays in:

- binary files - `.mat` format (*wrapper* and *back-end*)
- Chelonia [114, 166, 6]] distributed data store (*wrapper only*)
- Rasdaman [16] array database (*wrapper* and *back-end*)
- relational databases supporting SQL - see Section 6.2 (*back-end only*)

In all cases, the array proxy kinds corresponding to the storage systems are registered with SSDM, and APR implementations are provided to retrieve the specified array subsets. Different APIs or communication techniques are used in each case. A *wrapper-only* interface effectively means that the access is restricted to read-only. A *back-end only* interface means that we do not map native array representations to SSDM, due to the absence of the former - only arrays originating from SSDM are stored.

6.1.3 Problems and Solutions

The described approach of calling `APR()` whenever an array proxy is possible and its resolving might be needed, effectively introduces certain aspects of *lazy evaluation* for the purpose of materializing external arrays. The entire implementation is contained within the query translation layer, as opposed to a perhaps more obvious direction of incorporating the logic of lazy evaluation into the query execution runtime. Our translator-based implementation has proven to be sufficiently simple and robust, but still has a couple of technical shortcomings that need to be addressed.

6.1.3.1 Reduced directionality problem

The array-proxy-resolve function is defined as uni-directional: it is not generally possible to reconstruct an array proxy based on a memory-resident array. However, in those cases where `APR()` returns its argument without

changes, certain optimization opportunities are lost. This effect of reduced directionality might limit the freedom of SSDM query optimizer at reordering ObjectLog predicates for an optimal execution plan, compared to the freedom of evaluating SciSPARQL query translations without `APR()` calls.

Consider the following query **Q16**, selecting those realization ids where parameters *A* and *B* are numerically equal:

```
SELECT ?id
WHERE { [] ex:id ?id ;
          ex:parameter_A ?a ;
          ex:parameter_B ?b .
          FILTER (?a = ?b) }
```

When evaluated with an external array storage system connected, the AmosQL translation would look like:

```
select id
from Literal id, Literal a, Literal b, Literal g:0
where (g:0, URI('http://udbl.uu.se/ex#id'), id) in GRAPH(0)
and (g:0, URI('http://udbl.uu.se/ex#parameter_A'), a) in GRAPH(0)
and (g:0, URI('http://udbl.uu.se/ex#parameter_B'), b) in GRAPH(0)
and APR(a) = APR(b);
```

Since SSDM does not know that `?a` and `?b` are scalar parameters in the dataset (because it does not have any kind of schema that might contain such type information), it assumes that they may be array proxies, and thus prefers to resolve them before checking for equality (which is defined for arrays in Section 4.1.6). Without these `APR()` calls, SSDM query optimizer could potentially use the equality filter to e.g. infer the value of `?b` based on the known value of `?a`, and then simply check for its existence in the RDF graph (which is sometimes faster than looking up a value).

This missed optimization would of course be invalid if `?a` and `?b` were bound to array proxies: equality of the arrays per se does not entail equality of array proxies that point to them. So, the problem is in the lack of type inference mechanisms in the current implementation of SSDM. One option is using RDF Schema documents, which will provide specification of type constraints. This will allow to infer the types of variables. However, this will only work when RDF Schemas are provided.

In a *wrapper scenario*, e.g. when mapping from a relational data model, there might be no cost at all, as the *RDF Schema* is mandatory due to the RDF view definition, and its specification comes for free as part of RDB-to-RDF mapping [127, 51, 123], as suggested in [97].

6.1.3.2 Delegating more array operations

Specialized array stores like Rasdaman [16] are capable of performing most array computations on their own. In query **Q15**, for example, if *GI* arrays

were stored in Rasdaman, it would be possible to delegate the `array_avg()` computation to the back-end. In fact, most general array computations, as those supported by second-order functions like `MAP()` and `CONDENSE()` were introduced into SciSPARQL for compatibility with Rasdaman, and for the purpose of easy delegation of computations.

However, array proxies only accumulate array *selection*, *projection* and *transposition* operations, so without additional optimizations, `array_avg()` function in **Q15** would still be performed in SSDM, after the required array subset is transferred from Rasdaman over TCP connection.

The solution at hand is using Amos query mediator capabilities, available to SSDM. The ObjectLog predicates corresponding to the creation of a derived proxy would be grouped together with the predicates performing computations on such derived proxies. For example, the whole expression `array_avg(?A[:,2])` would correspond to a single Rasdaman API call, based on two ObjectLog predicates grouped together, and given there is an array proxy binding for `?A`.

The capabilities of any connected array storage system can thus be taken into account. Some storage systems accept a greater range of delegated operations than others: e.g. with Rasdaman it would be typically possible to delegate entire array expressions (free of foreign UDFs). In contrast, with `.mat` files only the access to array subsets would be delegated.

6.2 Relational Back-end

One of the configurations of SSDM relies on a relational back-end DBMS for persistent storage of *RDF with Arrays* datasets. Any relational database supporting SQL queries, JDBC interface, and storage of large binary objects (BLOBs) may be utilized for this purpose.

The relational schema for storing *RDF with Arrays* in an RDBMS and the query mapping process are explained in the following sub-section. Since the array proxies are resolved by means of sending SQL queries to retrieve the relevant array chunks, it becomes important to avoid sending too many queries, in order to save on the amount of round-trips to the RDBMS. Retrieving as little irrelevant data as possible is another optimization goal.

For this purpose the *Aggregate APR* function is defined in Section 6.2.4, which groups array proxies, buffers the data transfer operations, and generates SQL queries that are capable to serve the aggregated retrieval of array data under complex access patterns. Different strategies for formulating such SQL queries are introduced. Experimental comparison follows in Section 6.3.

6.2.1 Storage Schema

For the purpose of simplicity and good normalization, we have chosen to partition the set of *RDF with Arrays* triples into three subsets, based on the *value* type:

- URIs,
- RDF literals, including numbers, strings, any custom-typed values,
- arrays.

Figure 18 below shows the ER-diagram modelling the storage schema we use with the relational back-end databases. All URI values are normalized to the *URI* table, serving as a dictionary. The three 'triples' tables additionally store a *g* property - a URI identifier of a named graph the particular triple belongs to, or a reserved value for the default graph. The common *s*, *p*, and *v* attributes correspond to the *subject*, *property*, and *value* of an RDF triple.

While in *URITriple* table the *value* of a triple refers to the *URI* dictionary, in *LiteralTriple* it is the type of an *RDF literal* which is identified by the URI, and *v* attribute stores the string representation, with an optional language and locale tag in the *lang_loc* attribute. Upon creation, the *URI* dictionary is initialized with the standard types for common *RDF literals*, including strings, numbers, temporal and logical values. A limited space is allocated for an *RDF literal* by default, as we expect most of them to contain numbers or other short values. However, in order to provide space e.g. for larger pieces of text, a special *LongString* table is introduced, to accommodate string values without size limit.

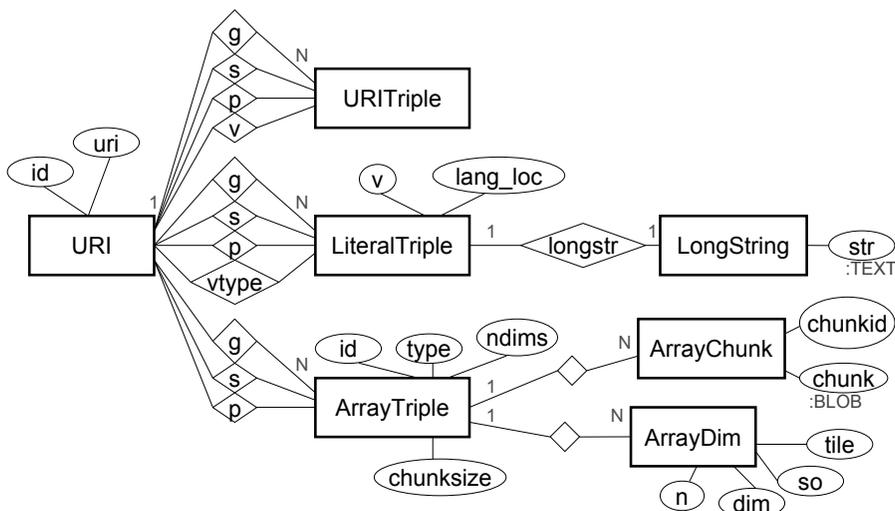


Figure 18. Relational storage schema for *RDF with Arrays*, shown as ER-diagram

Each array-valued triple is stored in the *ArrayTriple* table, and receives a unique *id* for its array value. The element type and the number of dimensions are stored in *type* and *ndims* attributes, and the information about each dimension is normalized out to the *ArrayDim* table. Since the logical and physical (nesting) order of dimensions are independent, the corresponding sequential numbers are stored in attribute *n* for the logical order, and *so* for the storage order. The size of array in the given dimension is stored in the *dim* attribute.

There are two basic ways to partition a multidimensional array into the limited-size chunks: either splitting its logical multidimensional form into chunks of the same dimensionality, or splitting its linearized 1-dimensional form into linear chunks. We will refer to the first kind of partitions as *multidimensional chunks*, or *tiles* and to the second kind of partitions as *linear chunks*. Figure 19 in Section 6.2.2 shows the same array-valued triple with different partitionings of its array value.

Partitioning to linear chunks can be fully defined by a single scalar value - the chunk size limit, stored in *chunksizelimit* attribute. If linear chunking is not used, this attribute is set to 0. In contrast, multidimensional partitioning is defined by a tile size limit in each dimension. If the array is stored in tiles, the *tile* attributes in *ArrayDim* table store the tile size in the corresponding dimension. Note that the actual chunk or tile sizes might be less for the remainder instances.

The array chunks are stored in table *ArrayChunk*, having three attributes: *arrayid*, *chunkid*, and a BLOB value representing the *chunk* contents. This table is the only one queried by `APR()`, and this process benefits from the optimizations explained in Sections 6.2.3 - 6.2.5 and evaluated in Sections 6.3 and 6.4. A clustered index is defined for (*arrayid*, *chunkid*), since *arrayid* is always known when resolving an array, and chunk ids might either be scanned starting from a known chunk id, or retrieved using index lookups when chunk ids are listed in the query. In both cases, the *physical locality* is important, i.e. records with contiguous chunk ids are physically stored close.

6.2.1.1 SQL access to the triples

In order to address the complete set of triples in a specific graph, an SQL view employing the relational *UNION* operator is defined across the three triple tables. *GeneralView(g, s, p, vtype, value)* exposes five attributes, first three of which are taken directly from the corresponding tables of triples.. For the *URITriple* and *ArrayTriple* tables the reserved URI ids, identifying *URI* type and the introduced *Array* type respectively, are returned for *vtype*, while *LiteralTriple* table stores *vtype* explicitly. Technically, *g*, *s*, *p*, and *vtype* are integer URI ids, referring to the *URI* dictionary

The fifth attribute returned by *GeneralView* is a string representation of triple's *value*. It is taken directly from *LiteralTriple* or *URITriple* (as a stringified integer). In case of array-valued triples, this *value* attribute encodes a glued-together textual representation of the *ArrayTriple* row and the group of connected *ArrayDim* rows, ordered by *n*. We use a common *group concatenation* operator to pack this information about an arbitrary number of dimensions into a single string value. This representation is sufficient to construct an array proxy in SSDM, as described below in Section 6.2.1.2.

The *GeneralView* can be employed as an *imported table* [85] under Amos II federated query framework utilized by SSDM. For instance, mediated by some simple conversions and URI cache lookups, a call to GRAPH predicate with '+---' binding pattern would translate to an SQL query:

```
SELECT s, p, vtype, v FROM GeneralView WHERE g = ?
```

Amos II mediator facilities are quite adept at translating the predicate calls to SQL queries under the different binding patterns, so this is certainly a viable solution, should the amount of RDF data (not counting the arrays) exceed the main memory limit. Self-joins, filters and arithmetic operations would be delegated as well into an SQL query.

However, since in our targeted scientific and engineering applications the majority of data is contained in arrays, the RDF triples proper contribute only to a small fraction of the total dataset size. Hence, e.g. the pure SPARQL queries can be processed entirely in main memory, without the need of addressing the persistent storage past the initial caching phase. In our back-end scenario with RDBMS, we cache all RDF triples (and create all original array proxies) in main memory when the SSDM server is started. Any consequent updates are applied to the cache and the storage back-end within the same transaction.

6.2.1.2 Specifics of the relational array proxies

While the database-unique array *id* value is sufficient to address an array in our relational back-end, resolving an array proxy would first require the *chunk size* or *tile size* information in order to compute the relevant chunk ids for the specified array subset. In order to avoid this extra round-trip to the back-end RDBMS, we choose to cache this partitioning information on the proxy object, created from the string returned by *GeneralView* as a *value* of an array-valued triple.

As shown in Figure 13 (d) and (e) in Section 5.2.2, a projection operation over an *array descriptor* produces the *derived array descriptor*, with a reduced number of dimension components (DADs). Since we are going to need the original array shape (i.e. the dimension sizes across all original

dimensions) and the complete tile size, in order to compute the relevant tile ids, we cannot afford to drop this information when producing derived array proxies. Hence the information about the original array dimensions and tile sizes is not stored in DADs, but is contained in a proxy-specific part, together with array id, and is passed along to the derived proxies without changes.

In summary, the storage-specific information of an array proxy corresponding to the relational back-end consists of array id, chunk size (if defined), and a list of original array dimensions and the corresponding tile sizes (if defined).

6.2.2 The Problem of Retrieving Array Content

Along with desinging a relational back-end storage for *RDF with Arrays*, we are going to focus on the problem of efficient retrieval of array content by resolving the array proxies. We illustrate the context and the task with the following example: let us consider the following SciSPARQL query **Q17**. The query selects equally spaced elements from a single column of a matrix, which is found as a value of the `:result` property of the `:Experiment1` node.

```
SELECT (?A[2:2:, 5] AS ?result)
WHERE { :Experiment1 :result ?A }
```

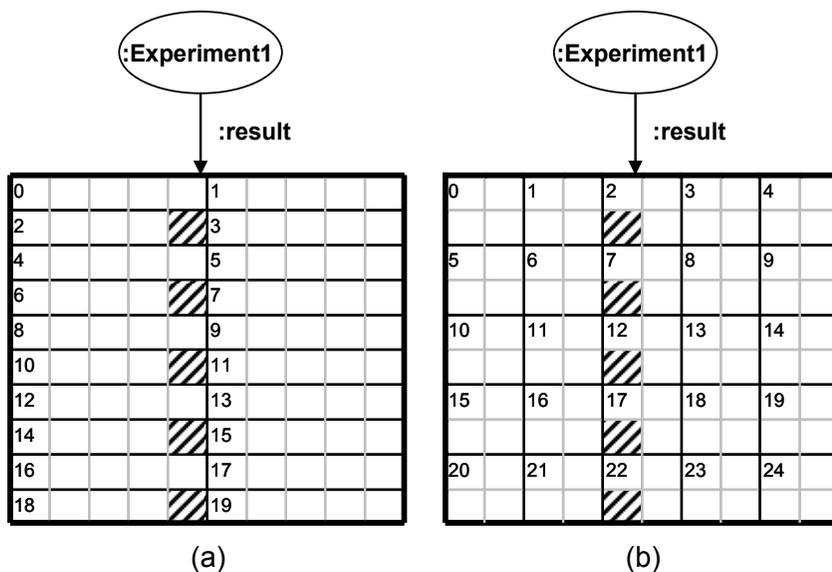


Figure 19. An example *RDF with Arrays* dataset using (a) linear partitioning and (b) multidimensional partitioning

We assume our example dataset **G3** includes the following *RDF with Arrays* triple, containing a 10x10 matrix as its *value*, as shown in Figure 19 (a), with the subset retrieved by **Q17** is shown hatched.

In our relational back-end the matrix is stored in 20 *linear chunks*, containing 5 elements each (chunk ids shown on the picture). Figure 19b shows a variant of the same dataset, where the array is stored in 25 2x2 non-overlapping square *tiles*. The example (a) is used through the rest of this section, and we compare the two storage approaches in Section 6.3.

In this toy example, our back-end relational database would be populated by an entry in *ArrayTriple*, two entries in *ArrayDim*, with the array data residing in *ArrayChunk*. In general, while querying the big *ArrayChunk* table, we would like to:

- minimize the number of SQL queries to *ArrayChunk*, and
- minimize the amount of irrelevant data returned.

There is a number of steps to be performed before the back-end will be queried for the real array data:

- Identifying the set of array elements that are going to be accessed while processing a SciSPARQL query. Such sets of elements are described, in general, with bags of *array proxy* objects.
- The array proxies accumulate array subsetting operations. Enumerable set of array proxies can be generated using free index variables, as shown in example queries **Q18** and **Q18a** below.
- Identifying fragments of this subset, that are contiguous in the linearized representation of the original array in order to save on the number of data-transfer operations. This step is explained in Section 6.2.4.2.
- Identifying array chunks needed to be retrieved and formulating *data transfer operations* for each chunk, as explained in Section 6.2.4.3. Buffering these chunk ids and operations, as explained in Section 6.2.4.1.
- Formulating SQL queries to the back-end RDBMS, as explained next in Section 6.2.3.
- If the query was prediction-based, switching between the phases of (I) simulation and buffering, (II) performing the buffered operations, and (III) performing the further (unbuffered) operations, as long as the prediction-based query yields the relevant chunks. This includes taking care of false-positives and false-negatives, as explained in Sections 6.2.4.4 - 6.2.4.6.

The example in Section 6.2.4.8 illustrates the complete process.

6.2.3 Strategies for Formulating SQL Queries during APR

There is a number of possible strategies to translate sets of chunk ids in the buffer to SQL queries retrieving the relevant chunks:

- **NAIVE**: send a single SQL query for each chunk id. This proves to be unacceptably slow in realistic data volumes, due to interface and query processing overheads.
- **IN (single)**: combine all the required chunk ids in a single IN list, sending a query like

```
SQL:
SELECT chunkid, chunk FROM ArrayChunk
WHERE arrayid = 1
AND chunkid IN (2,6,10,14,18)
ORDER BY chunkid
```

This would work well until the SQL query size limit is reached.

- **IN (buffered)**: an obvious workaround is to buffer the chunk ids (and the description of associated data copying to be performed, as described in Section 6.2.4), and send a series of queries containing limited-size IN lists.
- **SPD** (Sequence Pattern Detection) : sending a query like

```
SQL:
SELECT chunkid, chunk FROM ArrayChunk
WHERE arrayid = 1 AND chunkid >= 2
AND mod(chunkid - 2, 4) = 0
ORDER BY chunkid
```

Here the condition expresses a certain *cyclic pattern*. Such a pattern is described by *origin* (2 in the example above), *divisor* (4 in the example above), storing the total periodicity of repetitions, and the *modulus list* (consisting of single 0 in the example above), containing the repeated offsets. The *size* or *complexity* of a pattern is the length of its *modulus list*. Section 6.2.5 describes our algorithm for detecting such patterns.

In most cases the **SPD** strategy will allow to send a single query retrieving all desired chunks. If the pattern was too complex to be inferred from the buffer (e.g. there was no cyclic pattern at all), some extra chunks might also be retrieved.

Still, there are two problems with a straightforward application of **SPD**: (1) in cases when there actually is a cyclic pattern it is unnecessary to identify all the relevant chunk ids first - a small sample list of chunk ids is

enough; and (2) in case of an acyclic (random) access, like query **QT6** defined in Section 6.3, the detected pattern might be as long as the list of chunk ids, thus making it a similar problem as for **IN (single)**.

- **SPD (buffered)**: solving the two above problems by computing a small sample sequence of the needed chunk ids, and then formulating and sending an SQL query with the detected pattern. If the pattern covers all the chunks to be retrieved, the single SQL query does all the work. Otherwise (on the first false-negative, or when the false-positives limit is reached), the SQL query is stopped and the buffering process is restarted. In the worst case (when there is no cyclic pattern), it will work similarly to **IN (buffered)**, otherwise, fewer queries will be needed to return the same set of chunks.
- **SPD-IN (buffered)**: the difference between **IN** and **SPD**-generated SQL queries is that in **IN**, the `chunkid` values are explicitly bound to a list, which allows most RDBMSs to utilize the `(arrayid, chunkid)` composite index directly. As we have discovered in our experiments, neither MS SQL Server nor MySQL are utilizing an index when processing a query with `mod` condition.

However, by comparing a *pattern size* (i.e. length of the *modulus list*) to the number of distinct chunk ids in the buffer, we can easily identify if a realistic pattern was really discovered, or should we generate an **IN** query instead. We currently use the following rules to switch between **IN** and **SPD** buffer-to-SQL query translations:

- (A) If the pattern size is less than half the number of distinct chunk ids, then the cycle is not completely repeated, and is probably not detected at all.
- (B) If the sample size is less than the buffer limit - then we have buffered the last chunk ids for the query, so there is no advantage of using **SPD** either.

6.2.4 Resolving Bags of Array Proxies

The array proxies are limited to expressing the subsets of arrays which can be formulated as a superposition of array operations supported by SciSPARQL syntax. Thus, the execution of **Q17** above will include resolving a single proxy defining the array subset. Other interesting kinds of subsets cannot be defined by a single proxy, so a query returning a (multi-)set of elements can be formulated instead. For example **Q18** selects the elements of the main diagonal:

```

SELECT ?i (?A[?i, ?i] AS ?e)
WHERE { :Experiment1 :result ?A }

```

One unique feature of SciSPARQL is that whenever an (otherwise unbound) variable used as array subscript, it assumes all valid values for that subscript (Section 4.1.2). So, the first appearance of `?i` binds it to all valid row indices, and the query generates an array proxy for each such binding. For the dataset in Figure 19 this query will generate and resolve 10 array proxies, each pointing to a single element, i.e. 10 result tuples will be returned.

Our framework is capable to resolve such bags of proxies, as well as proxies referring to multiple elements. In general, bags of proxies can be filtered before resolving, and post-processed after resolving. Query *Q18a* contains these additional steps: it retrieves every second diagonal element, and returns the sum:

```

SELECT (SUM(?A[?i, ?i]) AS ?result)
WHERE { :Experiment1 :result ?A .
FILTER (mod(?i, 2) = 0) }

```

Figure 20 shows a fragment of the execution plan for *Q18a*, containing proxy generation, filtering, aggregated resolving (AAPR), and post-processing (SUM). Parallel arrows indicate the relative cardinalities of the intermediate results, i.e. the amounts of iterations in the corresponding nested loops.

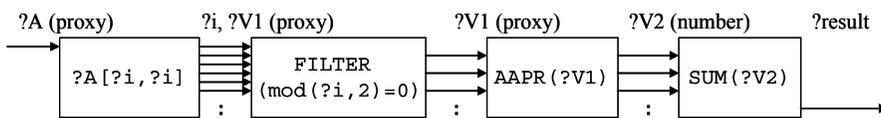


Figure 20. A fragment of *Q18a* execution plan

For the query *Q18a* (and the dataset in Figure 19) ten (proxy and `?i` value) pairs will be generated first, 5 of these intermediate results will be filtered out, 5 remaining proxies will be resolved together as a bag, and the corresponding array elements will be aggregated by the `SUM()` function into a single result.

In order to invoke the aggregated resolving of a bag of array proxies, the `AAPR()` function call is inserted instead of `APR()` call immediately under an aggregate function call, like `SUM()` in the translation of *Q18a*:

```

rdf:sum(aapr(select aref(aref(a,0,rdf:minus(i,1)),0,rdf:minus(i,1))
  from Literal i, Literal a
  where (URI('http://udbl.uu.se/ex#Experiment1'),
    URI('http://udbl.uu.se/ex#result'), a)
  in GRAPH(0)
  and rdf:mod(i, 2) = 0));

```

Technically, `AAPR()` is a *combiner* type function in AmosQL terms, i.e. it iterates over a bag of inputs, and emits a bag of results, maintaining an internal state during the whole process. This state consists of a buffer of pending data transfer operations, pre-allocated results, and a running SQL query to the back-end.

As a general approach, for each array proxy being resolved, a set of required chunk ids is computed, and for each chunk the set of data transfer operations is determined. If the proxy refers to a single element, the single reading operation results in a number, otherwise, a memory-resident array is allocated first, and a write position is associated with each such operation. The following subsections describe this process in detail, with Figure 22 summarizing the flow of information.

6.2.4.1 Buffer

The buffer is designed to store the description of work to be done when the respective chunks will be retrieved. It is organized as a hash table, with chunk id serving as a key, and the value being a list of data transfer operations described with the following fields:

- reading position in the chunk,
- the number of bytes to read,
- a reference to the allocated memory-resident array (none if single-element proxy), and
- a writing position in that array.

The buffer is primarily limited by the number of distinct chunk ids (i.e. hash table records) - the `_sq_buffer_size_` parameter. For the purpose of SQL query generation, only the set of distinct chunk ids is extracted from the buffer. For the queries with non-overlapping proxies, the number of buffered operations is thus limited by the amount of possible fragments per chunk times the number of distinct chunks to be retrieved. However, in case of e.g. random access queries like *QT6* in Section 6.3, this upper bound does not hold. Another 'technical' limit on the number of data transfer operations in the buffer is included as part of the SSDM settings.

6.2.4.2 Fragment mapper

Since we also minimize the number of data transfer operations, we first identify the largest possible contiguous fragments in the stored array (the algorithm is presented in Section 6.2.4.7) and the intersections of such fragments and the chunks become these operations. Besides this purpose, fragments are generally useful for e.g. copying as many elements as possible with a single memory operation, and offer a potential of parallelized (SIMD) array processing.

Figure 21 shows the number of fragments for different selections from a 2D array stored row-by-row. The discovery of the fragments involves the discovery if *innermost broken dimension* and *fragment size*. The dimensions are analysed in their *storage (nesting) order*, starting from the innermost one.

The innermost dimension is *broken* **iff** (i) its access multiplier am_k is not equal to 1 (when the proxy resulted from a projection from an array of higher dimensionality and the original inner dimension was projected out) or (ii) its *stride* is not equal to 1. In Figure 21 the innermost (column) dimension is *unbroken* in all three examples. If the innermost dimension is broken, the fragment size is 1, otherwise the incremental fragment size $fsize_k$ is equal to the derived array size in that dimension e.g. 6, 4, and 2 in the Figure 21 examples.

Provided the dimension k (in *storage order*) is *unbroken*, dimension $k-1$ is *broken* **iff** (i) its access multiplier am_{k-1} is not equal to $fsize_k$ (when some intermediate dimensions of the original array were projected out), or (ii) its stride is not equal to 1. If the dimension $k-1$ is *unbroken*, the incremental fragment size is multiplied by the derived array size in that dimension. In example (a) in Figure 21 the outer (row) dimension remains unbroken, and since $am_1 = 6$ in all cases, the condition (i) holds for examples (b) and (c).

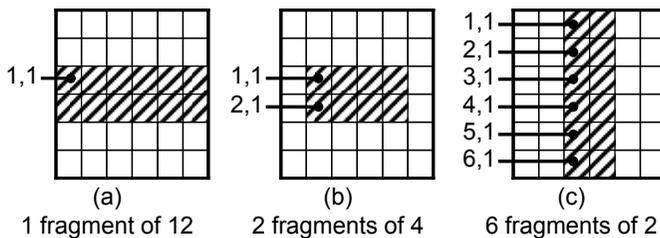


Figure 21. Derived array fragments discovery and iteration

The fragments' starting points are then defined by a nested iteration of logical subscripts (for the derived array) up to the *innermost broken dimension*, while padding the subscripts for unbroken dimensions with 1. Figure 21 shows these logical subscripts for each fragment, with no iteration happening in case (a). These starting points are translated into *storage indices* as described in the next section.

6.2.4.3 Identifying the relevant chunks

First, for any logical subscript (i_1, \dots, i_n) in the derived array (denoting a fragment start) a storage index $a(i_1, \dots, i_n)$ in the original array is computed using the equation (1) in Section 5.2.1.

Under the *linear partitioning* approach (Figure 19a), the first chunk id and the corresponding read position are given as the quotient and the remainder of the division of storage index by the chunk size. If the fragment size is greater than $(chunksize - read_pos)$ subsequent chunks are also included with $read_pos = 0$, until the fragment size is exhausted.

Under the *multidimensional partitioning* approach (Figure 19b), the logical subscripts j_1, \dots, j_N in the basic array need to be reconstructed from the storage index a first. Let $k(x)$ return the dimension k such that storage order $so_k = x$. The original array subscript (*0-based*) in the outmost dimension $j_{k(0)}$ and in the nested dimensions $j_{k(x+1)}$ are found as

$$j_{k(0)} = \left\lfloor \frac{a}{am_{k(0)}} \right\rfloor, \quad j_{k(x+1)} = \left\lfloor \frac{a_x}{am_{k(x+1)}} \right\rfloor$$

Here a_x is the remainder of the division while computing $j_{k(x)}$.

The multidimensional indexes t_k of the fragment's first tile are computed by dividing every component j_k by the tiles sizes s_k in the respective dimensions, and intra-tile logical indexes i_k as remainders of that division:

$$t_k = \left\lfloor \frac{j_k}{s_k} \right\rfloor, \quad i_k = j_k \bmod s_k$$

The linear chunk index (i.e. chunk id in this case) Ti and the linear chunk position pos are computed similarly to equation (1), using the number of tiles per dimension, and the actual tile size $s_k(Ti)$ as the dimensions, and N being the dimensionality of the original array:

$$Ti = \sum_{k=1}^N t_k \prod_{\substack{m=1 \\ so_m > so_k}}^N \left\lfloor \frac{\dim_m(A)}{s_m} \right\rfloor, \quad pos = \sum_{k=1}^N i_k \prod_{\substack{m=1 \\ so_m > so_k}}^N s_m(Ti)$$

The second product depends on a particular tile, as the last tiles in each dimension might have different sizes in that dimension.

When the *multidimensional chunking* is used, no generated fragment can go beyond the array range in the original array's innermost dimension - in Figure 21 example (a) would have two row-sized fragments. Once the first tile and chunk position of a fragment are determined, the remaining tiles are found by iterating the chunk index $t_{k(N)}$ along the innermost dimension, and resetting intra-tile index $i_{k(N)}$ in that dimension to 0.

6.2.4.4 Switching between the phases

The general case of resolving a bag of proxies includes switching between three phases: (I) buffering -> (II) processing the buffer -> (III) continuing

beyond the buffer, The formulation of an SQL query is done at the beginning of phase II, and its termination is done at the end of phase III.

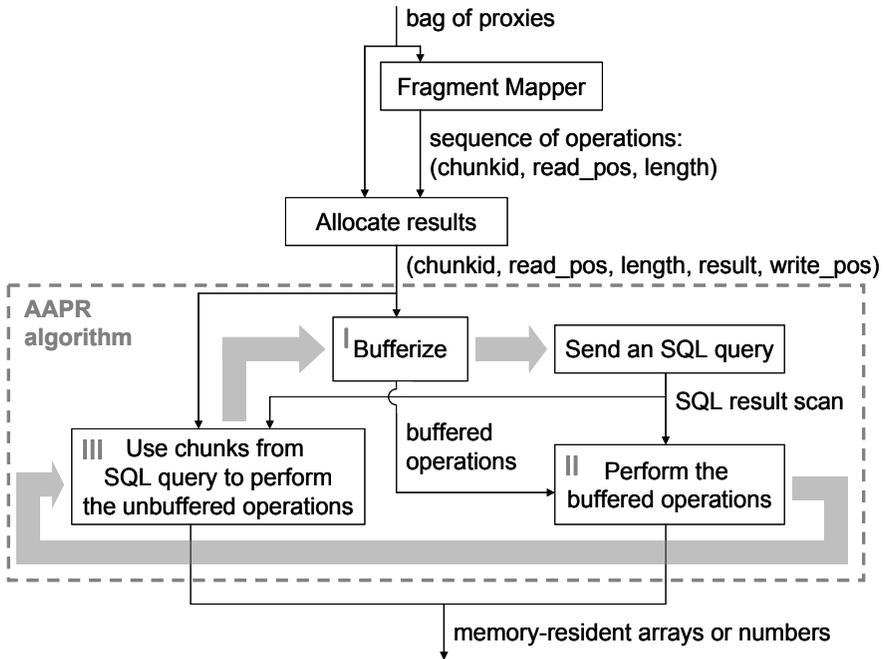


Figure 22. Three phases of the AAPR algorithm

Figure 22 shows an overview of the process of resolving a bag of proxies to a bag of memory-resident arrays or numbers. Broad grey arrows show the transitions between the three phases. Thin black arrows show the data flow, as annotated. The process starts at phase I, which ends when the buffer is full, or there are no more operations to be performed for the same stored array. A proxy referring to a different array effectively restarts the process. Phase II empties the buffer and proceeds to phase III.

If the SQL query was formulated according to the **IN** strategy, phase III is limited to processing any remaining operations for the last chunk retrieved. Once the next chunk id is required, the query is stopped, and the new buffering phase is started. Under the **SPD** strategy, if the detected pattern covers all the chunk retrieval needed for the query, phase III does the rest of the work in a completely streamed fashion. Section 6.2.4.7 presents a more formal description of the AAPR algorithm.

6.2.4.5 Emitting

For each result, a counter of pending data transfer operations is incremented at phase I and decremented at phase II, where the result is emitted if there are no more operations to be performed (and if it's not the result where the

buffering process has stopped). Numbers resulting from single-element proxies always involve a single data transfer operation, so they are emitted immediately after that operation is performed. At phase III, the current result is emitted when the operations for the next allocated result start arriving.

6.2.4.6 Chunk cache

The chunk cache is not shown in Figure 22 for simplicity. In fact, every new chunk id in phases I and III is first looked up in the cache. If present, the relevant data transfer operations are performed immediately, and no buffering or advancing of the SQL query scan is needed.

This cache is mainly designed for inter-query speedup, and in the experiments described in Section 6.3 the cache is reset before each query, to ensure independent statistics. However, the chunk cache is also helpful inside a single query when chunk ids do not come sorted, like e.g. for the random multi-proxy query *QT6* in Section 6.3.

6.2.4.7 AAPR algorithm pseudocode

Below is the formal description of AAPR algorithm which is illustrated in Figure 22. Phases I and III are implemented with the `aapr()` function, which accepts a bag of array proxies and returns a bag of memory-resident arrays.

Phases I and III. The below pseudocode outlines `aapr()`. The details like caching the chunks, freeing the resources, and error handling are omitted for brevity. Note that `aapr()` is a single-pass function, and involves no lookahead in its input bag. Instead, it uses a limited-size buffer to serve its needs.

```
function AAPR(Bag of Literal bx) -> Bag of Literal
{
  buffer = ();
  arrayid = nil;
  For each x in bx
  {
    if (x is not a proxy) emit(x);
    if (x.amd.arrayid != arrayid)
    {
      if (buffer is not empty)
        resetBuffer(); //resume phase II for previous array
      scan = closed; //start in phase I
      arrayid = x.amd.arrayid;
    }
    chunkid = invalid;
    if (x is a single-element proxy) result = nil;
    else result = allocateNewArray(x.amd);
      //to accommodate elements referred in x

  For each f in fragments(x)
  {
    TransferData
      (chunkid, readpos, writepos, length) td;
```

```

For each td in computeTransferData(f, x.amd)
{
  if (scan is open) //processing in phase III
  { //advance scan to the desired chunk
    advanceScan(scan, td.chunkid);
    if (scan.chunkid != td.chunkid)
      //SQL query is no longer useful
      scan.close(); //switch back to phase I
    else
      writeFragment(td, scan.chunk, result);
  }
  if (scan is closed) //processing in phase I
  {
    buffer.pushSorted(td, result);
    result.pendingOps++;
    // will be ready to emit when this counter is back at 0
    if (buffer is full)
      scan = resetBuffer(buffer, amd, result);
    // resume phase II, then switch to phase III
  }
} //of TransferData cycle
} //of fragment cycle
if (result.pendingOps == 0) emit(result);
} //of input cycle

if (buffer is not empty)
  resetBuffer(); // resume phase II
if (scan is open) scan = closed;
}

```

There are three nested cycles in this function, and the code inside the innermost cycle runs along either phase I (buffering) or phase III (emitting beyond the buffer) branches, or both when switching from phase III to phase I, depending on the state of the scan.

If input *x* is a proxy, then all the information about the array is stored on its `ArrayMetadata` *x.amd* property. Except for the `arrayid` field, `ArrayMetadata` is used inside the `allocateNewArray()` and `computeTransferData()` functions. The function

```

computeTransferData(Fragment f, ArrayMetadata amd)
    -> Bag of TransferData

```

returns a bag of transfer operation descriptions for the given fragment and array metadata. Each transfer operation description consists of *chunk id* and *read position* (computed as specified in Section 6.2.4.3), *write position*, selected sequentially in the allocated result, and *data length* - the fragment size adjusted to the chunk size.

Phase II. The following function is called when the buffer is full, or when the work (on a particular stored array) is done. It resumes the pending data transfer operations, emits the result and clears the buffer:

```

function resetBuffer(buffer, amd, result) -> scan
{
  //generate SQL query here
  query = bufferToSQLQuery(buffer, amd.arrayid);
  scan = openScan(query);

  //perform the buffered operations and emit results when ready
  For each (td, result) in buffer
  {
    advanceScan(scan, td.chunkid); //always successful
    writeFragment(td, scan.chunk, result);
    result.pendingOps--;
    if (result.pendingOps == 0
        and result is not the one where Phase I stopped)
      emit(result); //emitting from aapr()
  }
  buffer = (); //empty the buffer, leave the scan open
  return scan;
}

```

Note the difference on how `advanceScan()` function is called: in phase II we know that any `bufferToSQLQuery()` translation will construct a query guaranteeing that at least for all chunk ids in the buffer the chunks will be returned. However, in phase III we are always checking that the required chunk was returned by the query, and if not - we close the scan and start buffering again, in order to make a new SQL query.

It is also worth noting though `aapr()` returns exactly one result per input, in general, the order of results might not be the same, due to the buffering, and sorting the buffer contents by chunk id. However, in many simple cases (like the diagonal access with **Q18** elaborated below) the bag of proxies is generated using a variable ranging over the valid array subscripts, so the sequence of relevant chunk ids is naturally ordered.

6.2.4.8 AAPR example: diagonal access

Q18 (diagonal access) provides a nontrivial case, under the **SPD** translation strategy. With the dataset from Figure 19a and the buffer size limited to 3 distinct chunk ids, the first pattern detected will result in the SQL query:

```

SQL:
SELECT chunkid, chunk FROM ArrayChunk
WHERE arrayid = 1
AND mod(chunkid, 2) = 0
ORDER BY chunkid

```

The first 3 elements will be emitted in phase II, and the next 2 elements (from chunks 6 and 8) in phase III. However, the next required chunk would be 11, and the SQL query will continue returning chunks 10 and 12. The switching to phase II will be performed on the first false negative, that is, when chunk 12 is retrieved. Thus two "false positive" chunks will be retrieved before the query is stopped, and the new buffering process begins.

The next iteration of the buffer will contain chunk ids 11, 13, 15, and the new SQL query

```
SQL:
SELECT chunkid, chunk FROM ArrayChunk
WHERE arrayid = 1 AND chunkid >= 11
AND mod(chunkid - 11, 2) = 0
ORDER BY chunkid
```

will retrieve the remaining 5 chunks, which will allow returning the last 5 results. Figure 23 illustrates the process:

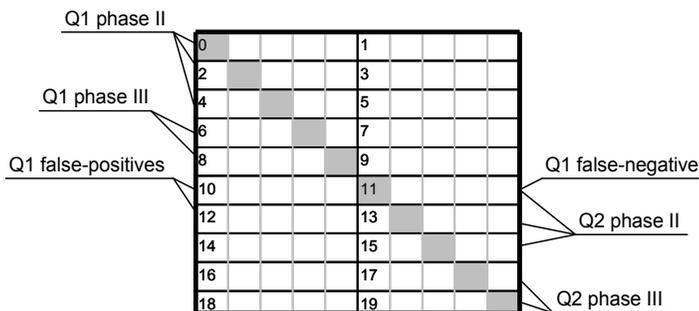


Figure 23. *Q18* chunk retrieval with SPD, buffer size 3

6.2.5 Sequence Pattern Detector (SPD) Algorithm

Once the buffer is filled at the end of phase I of the AAPR algorithm, an SQL query needs to be generated based on the buffer contents. An IN query is simple to generate, and the list of chunk ids does not even need to be sorted (as we have discovered, the RDBMS performs this sorting if using a clustered index). In order to generate an SPD query, we first extract and sort the list of distinct chunk ids from the buffer.

The following algorithm operates on an increasing sequence of numbers - in our case - sorted chunk ids. Since we are detecting a cyclic pattern, we are not interested in the absolute values of the numbers in the sequence, we will only store the first number as the point of origin, and the input values to the algorithm are the positive differences between the subsequent chunk ids.

Each input is processed as a separate step, as shown in Figure 24. The state of the algorithm is stored with the history and pattern lists, (initialized empty), and the next pointer into the pattern list (initialized to an invalid pointer which will fail any comparison operation).

The general idea is that each input either conforms to the existing pattern or not. In the latter case the second guess for the pattern is the history of all inputs. The input either conforms to that new pattern, or

the new `pattern` (which is now equal to `history`) is extended with the new input. In either case, `input` is appended to `history`, and `count` is incremented.

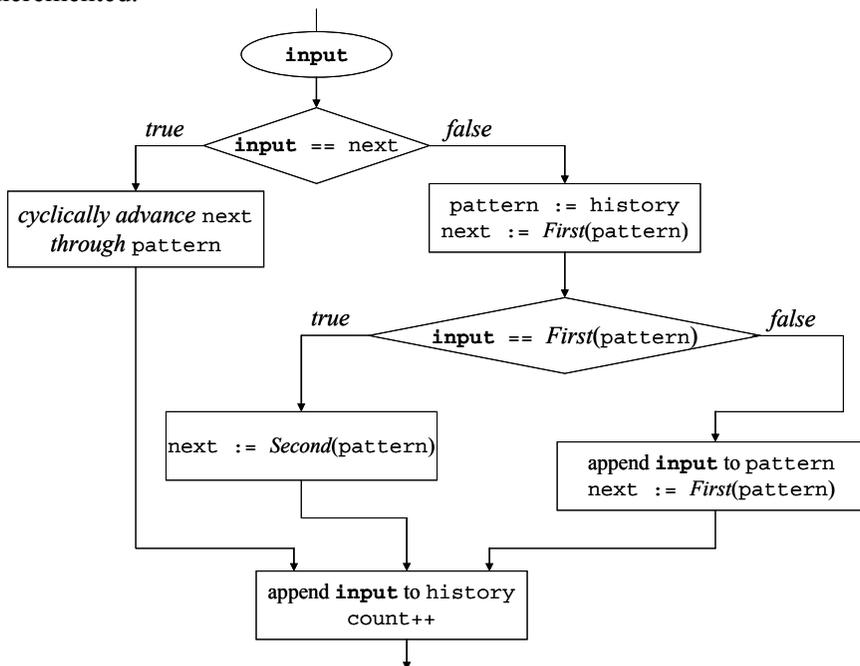


Figure 24. A step in the SPD algorithm

The resulting pattern will have the form:

$$x \geq x_0 \quad \wedge \quad \text{mod}(x - x_0, d) \in \{0, m_1, \dots, m_{n-1}\}$$

where x is the `chunk id` value to retrieve, x_0 is the first `chunk id` value generated (i.e. "reference point"), d is the *divisor*, and m_1, \dots, m_{n-1} is the *modulus list*. The generated `pattern` is the sequence of offsets $P = \{p_1, \dots, p_n\}$. We will compute the *divisor* as the total offset in the pattern:

$$d = \sum_{i=1}^n p_i$$

Each element in the *modulus list* is the partial sum of offsets:

$$m_k = \sum_{i=1}^k p_i, \quad k = 1, \dots, n-1$$

In the next section we compare this strategy of formulating an SQL query with the more straightforward approach of sending IN lists that was presented in Section 6.2.3.

6.3 Comparing the Storage and Retrieval Strategies

For evaluation of the different storage approaches and query processing strategies we first use synthetic data and query templates for the different access patterns where parameters (or additional data in the RDF graph) control the selectivity. Since our concern here is minimizing data accesses, the performance is independent of the array element values. Thus the synthetic arrays are populated with random values.

For simplicity and ease of validation, we use two-dimensional square arrays throughout our experiments. More complex access patterns may arise when answering similar queries to arrays of larger dimensionality. Still, as s below, the two-dimensional case already provides a wide spectrum of access patterns, sufficient to evaluate and compare our array storage alternatives and query processing strategies. The parameterized SciSPARQL queries (listed in Table 5) we use for our experiments involve typical access patterns, such as: accessing elements from one or several rows, one or several columns, in diagonal bands, randomly, or in random clusters.

The efficiency of query processing thus can be evaluated as a function of parameters from four different categories: data properties, data storage options, query properties, and query processing options, as summarized in Table 4. A plus sign indicates that multiple choices were compared during an experiment, and a dot sign corresponds to a fixed choice.

Table 4. Summary of performance evaluation axes

Axis	Experiment		
	1	2	3
<i>Data properties</i>			
• array shape and element type	•	•	•
<i>Data storage options</i>			
• partitioning: linear / multidimensional	+	•	+
• chunk size	•	•	+
• nesting order of dimensions	•	•	•
<i>Array query properties</i>			
• logical access pattern	+	•	•
• intra-array selectivity	+	•	•
• logical locality	+	•	•
<i>Query processing options</i>			
• strategy: SPD / IN / hybrid	+	•	•
• buffer size	+	+	•

The structure of the data remains the same throughout the experiments. Namely, it is the dataset shown on Figure 19 (Section 6.2.2), containing a single 100 000 x 100 000 array of integer (4-byte) elements, with total size ~40Gb. The logical nesting order is also the same (row-major), as changing

it would effectively swap row query *QT1* and column query *QT2* while having no impact on the other query types from Table 5. The rest of the axes are explored during our experiments, as Table 4 indicates.

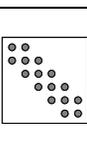
Experiment 1 compares the performance of different query processing strategies (including different buffer sizes), as introduced in Section 6.2.3, for different kinds of queries. For each kind of query, cases of different selectivity are compared under either data partitioning approach.

Experiment 2 explores the influence of buffer size on the total query response time, using the **IN** strategy for a simple single-column query and a tiled partitioning of the array. This combination of data storage access pattern is chosen because it is evenly balanced between best- and worst-case.

Experiment 3 explores the influence of chunk size on the query performance. There is obviously a trade-off between retrieving too much irrelevant data (when the chunks are big) and forcing the back-end to perform too many lookups in a chunk table (when the chunks are small).

For all experiments, the selectivity is shown both as the number of array elements accessed and the number of the relevant chunks retrieved. We expect the latter quantity to have higher impact on overall query response time, in other words, more time is going to be spent on the communication with RDBMS back-end. These expectations are confirmed in Section 6.3.2.

Table 5. Query patterns

Query type	SciSPARQL query	Parameters	Access diagram	Asymptotic selectivity
<i>QT1</i> : single row	<code>SELECT (?A[a, c:d:] AS ?res) WHERE { :Experiment1 :result ?A }</code>	<i>a</i> - first (or single) row <i>b</i> - row		$\frac{n-c}{dn^2}$
<i>QT2</i> : single column	<code>SELECT (?A[a:b:, c] AS ?res) WHERE { :Experiment1 :result ?A }</code>	stride <i>c</i> - first (or single) column		$\frac{n-a}{bn^2}$
<i>QT3</i> : regular grid (generalization of <i>QT1- QT2</i>)	<code>SELECT (?A[a:b:, c:d:] AS ?res) WHERE { :Experiment1 :result ?A }</code>	<i>d</i> - column stride		$\frac{(n-a)(n-c)}{bdn^2}$
<i>QT4</i> : diagonal band (main diagonal)	<code>SELECT ?i ?j (?A[?i, ?j] AS ?e) WHERE { :Experiment1 :result ?A . FILTER (mod(?i - 1, b) = 0 && abs(?i - ?j) <= w) }</code>	<i>b</i> - row and column stride <i>w</i> - diagonal band width		$\frac{1+2w}{bn}$
<i>QT5</i> : uniform random	<code>SELECT ?i ?j (?A[?i, ?j] AS ?e) WHERE { :Experiment1 :result ?A .</code>	<i>s</i> - amount of random elements to return		$\frac{s}{n^2}$
<i>QT6</i> : clustered random	<code>?e a :ElementIndices ; :i ?i ; :j ?j } LIMIT ?s</code>			

All experiments were run with SSDM and the back-end MS SQL Server 2008 R2 deployed on the same HP Compaq 8100 workstation with Intel Core i5 CPU @ 2.80 GHz, 8Gb RAM and running Windows Server 2008 R2 Standard SP1. The communication was done via MS SQL JDBC Driver version 4.1 available from Microsoft.

6.3.1 Query Generator

Similarly to the examples above, in each query template we identify an array-valued triple directly by its subject and property, thus including a single SPARQL triple pattern:

```
:Experiment1 :result ?A .
```

Each time we retrieve a certain subset of an array and return it either as a single small array (*QT1* - *QT3*) or the single elements accompanied by their subscript values (other queries). The templates listed in Table 5 differ only in the array access involved, including extra conditions on variables used as array subscripts.

For the random access patterns, the main parameters are the random array subscripts. Nodes of type `:ElementIndices` with `:i` and `?j` properties are added into the RDF dataset, with the following parameterized update:

```
DEFINE PROCEDURE AddCoordinates(?b ?i ?j) AS
  INSERT { ?b rdf:type :ElementIndices .
            ?b :i ?i .
            ?b :j ?j }
```

where `?b` is a unique blank node generated for each random coordinate pair.

6.3.2 Experiment 1: Comparing the Retrieval Strategies

We compare the different query processing strategies and the impact of buffer sizes for each query presented in Table 5, with different parameter cases resulting in the varying selectivity (and, in case of *QT3*, logical locality). We are interested to see how well the different buffer-to-query translation strategies fit to these use cases in terms of number of SQL queries generated and the total measured response time.

Each query and parameter case is run against two stored instances of the dataset, differing in array partitioning method:

- **linear chunks**: the array is stored in row-major order, in chunks of 40 000 bytes (10 chunks per row, 10 000 elements per chunk, 1 000 000 chunks total) using *linear partitioning*

- **square tiles:** the array is stored in 100x100 chunks, occupying 40 000 bytes each (10 000 elements per tile, 1 000 000 tiles total - same as above) using *multidimensional partitioning* (2-dimensional in our case).

We pick the strategies among **SPD**, **IN**, **SPD-IN** (as described in Section 6.2.3), buffered variants. The buffer size is also varied for the **IN** strategy, with values picked among 16, 256, and 4096 distinct chunk ids. The **SPD** strategy is not affected by the buffer size in our cases - it either discovers the cyclic pattern with the buffer size of 16 or does not.

The query parameters are chosen manually, to ensure different selectivity (for all query patterns) and absence of data overlap between the parameter cases (for *QT1-QT3*). Each query is repeated 5 times, and the average time among the last 4 repetitions is shown on the diagrams below.

6.3.2.1 Query *QT1*

Accessing a single row in a matrix stored in row-major order is obviously an example of best-case workload, since max 10 linear chunks are involved. Hence, all **IN** strategies issue the same SQL query, listing the 10 chunk ids, regardless of buffer size. Since the buffer is never filled completely, **SPD-IN** always chooses the **IN** strategy following rule (B) specified in Section 6.2.3

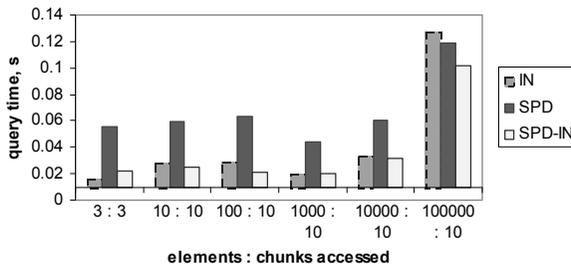


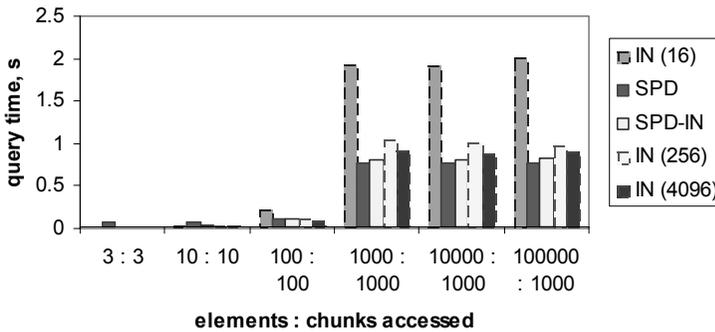
Figure 25. *QT1* run time (s) for linear chunks

Six parameter cases were used, first one iterating across 3 elements in 3 different chunks, others accessing all 10 chunks in the row, with different number of row elements copied into the resulting memory-resident array. Figure 25 shows that the amount of data copied from chunks to memory has clear impact on the response time only when going from 40 kB to 400 kB resulting array size. On smaller results the query time is dominated by the execution of a single query to the back-end, and the fluctuations may only result from OS/DBMS cache states and background activity. The difference between first and second case for the **IN** strategy, though smaller than fluctuation range, might still correspond to the cost of transferring different

number of chunks from the back-end to SSDM. However, queries *QT2-QT6*, retrieving greater amounts of chunks, certainly provide a better clue.

Another important result one can already notice: processing a single **SPD** query is apparently more expensive than processing a single **IN** query, with the total amount of chunks retrieved staying small and the same.

In the case of *multidimensional partitioning*, the maximum is 1000 square tiles being retrieved, and Figure 26 shows that the difference in element copying operations is playing little role compared to retrieving the chunks:



QT1 Parameters			time, s				
a	c	d	IN(16)	SPD	SPD-IN	IN(256)	IN(4096)
1	50001	20000	0.019	0.068	0.025	0.020	0.018
101	1	10000	0.032	0.074	0.030	0.031	0.030
201	1	1000	0.234	0.104	0.114	0.117	0.102
301	1	100	1.945	0.780	0.803	1.058	0.916
401	1	10	1.909	0.776	0.812	1.024	0.886
501	1	1	2.007	0.769	0.825	0.979	0.904

Figure 26. *QT1* run time (s) for square tiles

While the first case retrieves only three chunks - with ids 500, 700, and 900, the second and third case retrieve every 100th and every 10th chunk among the first thousand. The response time rises slower than proportional, implying there is a constant cost of sending an SQL query, and a per chunk cost of transferring the results back.

In the last three cases, in order to retrieve 1000 chunks, the **IN(16)** strategy sends 63 SQL queries to the back-end. The different number of SQL queries sent determines the clearly seen low performance of **IN(16)**, compared to **IN(256)** and **IN(4096)** in these cases.

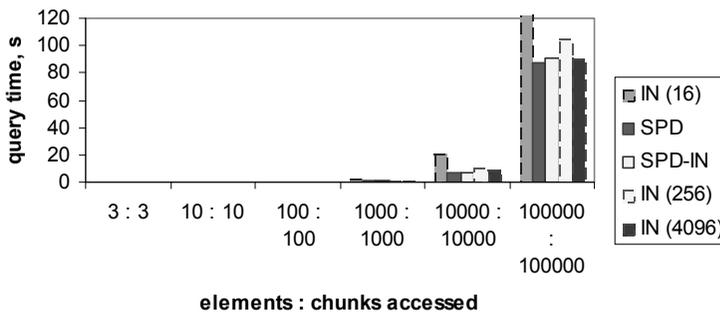
The set of retrieved chunks, with ids 0, 1, ... 999 in the last three cases, have the best possible physical locality, i.e. they are stored in a contiguous range of clustered index values in the ArrayChunks table. Because of this, a single SQL query sent under the **SPD** strategy performs better than a single

SQL query under **IN(4096)**, the latter containing an **IN** list of 1000 items. As the execution plans reported by MS SQL Server show, answering an **SPD** query inside the DBMS involves a scan through a range of rows, while **IN** queries invoke index lookups.

The conclusion drawn from experiments with **QT1** is that the **SPD** strategy becomes slightly better than **IN(4096)** in a situation of extremely good physical locality (retrieving every chunk among the first 0.1% of the chunks in the database). Under more sparse access, **IN** with a big enough buffer, also sending a single SQL query, is preferable.

6.3.2.2 Query **QT2**

This query represents worst case workload for the linear partitioning, given the row-major nesting order. Each element accessed belongs to a different chunk, and accessing 100 000 chunks with **IN** queries listing 16 chunk ids each, takes 680 s on average, with 6250 SQL queries generated with **IN(16)** strategy.



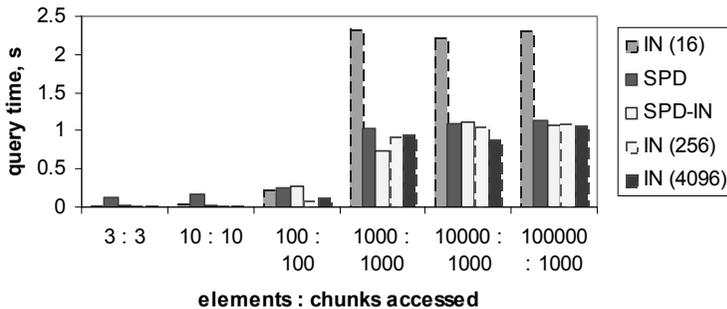
QT2 Parameters			time, s				
a	b	c	IN(16)	SPD	SPD-IN	IN(256)	IN(4096)
50001	20000	1	0.016	0.241	0.016	0.029	0.029
1	10000	10001	0.029	0.177	0.020	0.035	0.034
1	1000	20001	0.188	0.200	0.277	0.123	0.120
1	100	30001	2.298	1.054	1.049	0.992	0.925
1	10	40001	20.770	6.777	7.323	10.376	9.043
1	1	50001	680.570	87.782	90.976	105.116	91.089

Figure 27. **QT2** run time (s) for linear chunks

The **SPD** strategy provides an order-of-magnitude speedup (for the same buffer size of 16) by sending a single query. However, sending 25 **IN** queries listing up to 4096 chunks each proves only slightly slower. Figure 27 shows the expected linear rise, and the serious disadvantage of sending too many SQL queries.

One unexpected phenomenon is the superlinear rise in response time - mainly for **IN(16)** strategy, when retrieving 100 000 chunks. We discuss this for **QT4**, where it appears to be even more prominent.

As expected, the performance is roughly the same as for **QT1** in the case of *multidimensional partitioning*, with the same maximum of 1000 square tiles being retrieved (this time, making up for one column of tiles). Figure 28 shows the performance figures, similar to those on Figure 26 in most aspects. This includes the different performance of **IN** strategies,



QT2 Parameters			time, s				
a	b	c	IN(16)	SPD	SPD-IN	IN(256)	IN(4096)
50001	20000	1	0.021	0.116	0.019	0.021	0.023
1	10000	10001	0.037	0.177	0.020	0.021	0.019
1	1000	20001	0.221	0.256	0.269	0.093	0.123
1	100	30001	2.324	1.040	0.727	0.927	0.939
1	10	40001	2.227	1.096	1.110	1.054	0.874
1	1	50001	2.307	1.144	1.081	1.089	1.064

Figure 26. **QT2** run time (s) for square tiles

The **SPD** strategy performs differently for **QT2**, as clearly seen in the last three cases: the **IN** strategies are sending 63 SQL queries this time, with worse physical locality retrieving e.g. chunks 3, 1003, ... 999003, instead of 0, 1, ... 999 when processing **QT1**. The comparison of these results to **QT1** shows the slight slowdown introduced by this effect, and the **SPD** strategy is no longer the best one.

The long and sparse range of rows, given by **QT2** access pattern, thus incurs *slower-than-index SPD* performance (e.g. slower than the index lookups used by **IN** strategies). In contrast a short condensed range (as for **QT1**) is *faster-than-index* - as a non-selective scan is generally faster than index lookups.

6.3.2.3 Query *QT3*

The 'regular grid' query generates more complex access patterns, and with considerable sparsity touches a relatively small amount of chunks. *Linear chunk* partitioning, both row-major and column-major, works equally well for 'isotropic' grids, (i.e. if grid density is the same in different dimensions).

Table 6. Parameter cases used for *QT3*

case	QT3 parameters				grid coverage
	a	b	c	d	
A	1	20000	1	20000	100%
B	2	10000	1	10000	100%
C	90001	100	90001	100	1%
D	4	1000	1	1000	100%

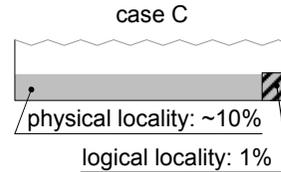
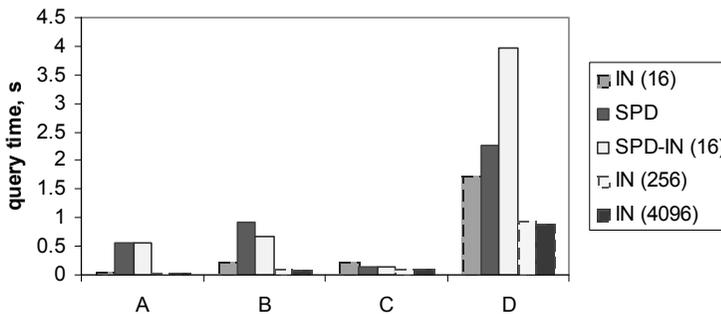


Table 6. shows the parameter cases used. Cases A, B, and D cover the whole array, retrieving elements at certain intervals (*strides*), as specified by the parameters *b* and *d* - i.e. 5x5, 10x10 and 100x100 elements in total. The 100x100 elements retrieved in case C are co-located in one corner of the array, taking up to the last 1/10 of the array in each dimension. Although logically this corresponds to 1% of the array 'area', the first accessed linear chunk or square tile marks the beginning of the last ~10% range of the chunks stored (sequentially) for the array, a fact that determines the actual physical distribution of the chunks, as the figure next to the Table 6 shows.



case	elements accessed	chunks accessed	time, s				
			IN(16)	SPD	SPD-IN(16)	IN(256)	IN(4096)
A	25	25	0.052	0.554	0.553	0.022	0.018
B	100	100	0.224	0.918	0.681	0.101	0.078
C	10000	100	0.236	0.132	0.137	0.113	0.121
D	10000	1000	1.738	2.252	3.963	0.955	0.902

Figure 29. *QT3* run time (s) for linear chunks

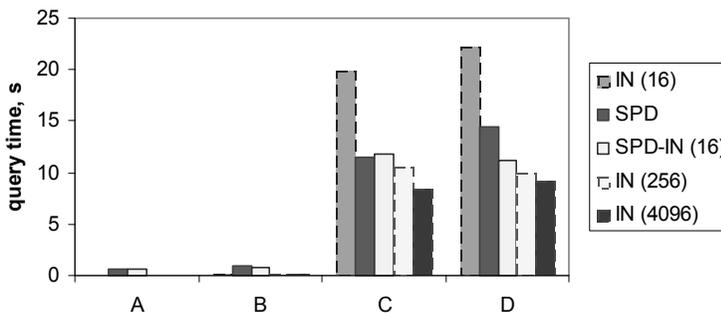
Figure 29 shows that **SPD** does not perform well in cases A, B, and D, where the relevant chunks are distributed across the whole array. Ten times better physical locality in case C already puts **SPD** on par with **IN** strategies,

which also send a single SQL query when the buffer size allows to collect all 100 chunk ids first. These chunks are then selected among the last 10%, i.e. 100 000 chunks stored. This supports our earlier observation that **SPD** favors short and condensed ranges.

Another clearly visible problem is poor performance of **SPD-IN**. In case A it clearly chooses **SPD**, which is not optimal. In case D chunks 40...49, 1040...1049, ... are retrieved, so that pattern of size 10 is detected under **SPD** with buffer size 16, but pruned under **SPD-IN**, since it is not repeated within this small buffer. As a result, **SPD-IN** sends mainly **IN** queries, but in certain cases (e.g. when the buffer contains chunk ids 3042..3049, 4040..4047) an incorrect cyclic pattern (in this case 'retrieve 8 - skip 990') is detected, so an **SPD** query is occasionally sent, having a false negative right after switching to phase III (chunk id 4048), thus providing no benefits over **IN(16)** query, but performing much slower (as explained in 6.3.21).

A similar misdetection happens for **SPD-IN** in case B, however, with smaller amount of retrieved chunks, the benefit of sending at least some **IN(16)**, outweighs the pure **SPD** approach.

In order to show that a bigger buffer for **SPD-IN** would solve this problem, we also run it with buffer sizes 256 and 4096 for **QT3** and linear chunks. Under buffer size 256 **SPD-IN** chooses to run a single **SPD** query to retrieve 1000 chunks, while under buffer size 4096 a single **IN** list is always sufficient (**SPD-IN** with these buffer sizes is not shown, since it is equivalent to **SPD** and **IN** respectively).



case	elements accessed	chunks accessed	time, s					
			IN(16)	SPD(256)	SPD-IN	IN(256)	IN(4096)	
A	25	25	0.052	0.549	0.633	0.022	0.023	
B	100	100	0.205	0.890	0.758	0.080	0.100	
C	10000	10000	19.858	15.023	11.832	10.608	8.421	
D	10000	10000	22.229	14.428	11.224	9.98275	9.093	

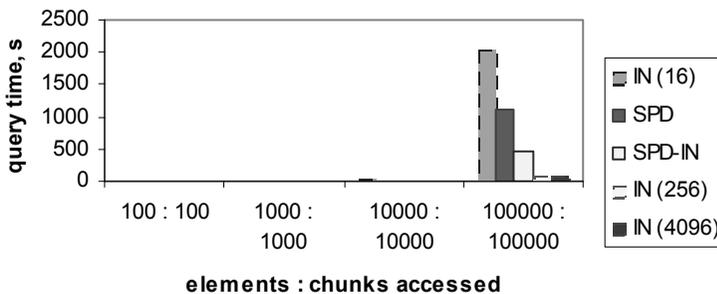
Figure 30. **QT3** run time (s) for square tiles

In the case of *multidimensional* array partitioning and under certain grid densities, this query is becoming a worst case workload - retrieving a single element from every tile. In cases C and D the query has to retrieve one tile per element, effectively, 10000 square tiles (compared to 1000 linear chunks). While in case C these tiles are selected from the last 10% of the chunk range, in case D the query retrieves an element from every 10th tile vertically and every 10th tile horizontally:

In cases A, B, and C for *linear chunks* (Figure 29), the difference in buffer size above 100 does not matter, as the strategies **IN(256)** and **IN(4096)** send identical SQL queries - the differences in response times allow us to assess the accuracy of our measurements, which depend on the state of the whole software stack involved. Figures 29 and 30 show that the **IN** strategy with a large buffer is the best choice in all cases, regardless of the partitioning scheme.

6.3.2.4 Query QT4

Similarly to query *QT2*, this one is the worst case workload for linear chunk partitioning, and, as shown in Section 6.2.4.8, the chunk access pattern, as detected by **SPD** changes along the diagonal, initiating re-buffering and cyclic phase switching in the AAPR framework. We experiment with diagonal width of 1 and different strides. Every element accessed belongs to a different linear chunk.



QT4 Parameters			time, s				
b	w		IN(16)	SPD	SPD-IN	IN(256)	IN(4096)
1000	1		0.265	0.979	0.758	0.102	0.087
100	1		1.825	2.484	1.419	1.006	0.846
10	1		17.768	7.190	11.000	7.986	8.097
1	1		2043.404	1107.296	470.062	99.467	86.335

Figure 31. *QT4* run time (s) for linear chunks

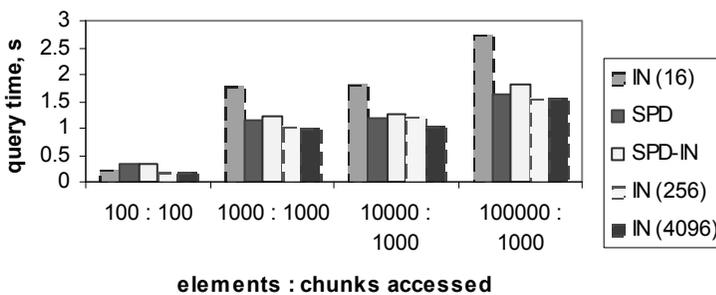
Here **IN** strategies show superlinear rise in response time which is most severe for the small buffer cases. Retrieving 10 times more chunks (and thus sending 10 times more SQL queries) entails 115 times longer response time

(when sending 6250 SQL queries as in the last case), however, this is alleviated down to the factor of 10.6 in the case of long SQL queries. We attribute this to a critical performance bottleneck inside the back-end DBMS, which has to be investigated by the DBMS engineers. This loss of throughput is similar to the one presented for *QT2* (Figure 27), but is measured to be more significant in the diagonal access pattern tested here.

The **SPD** strategy sends only 10 SQL queries (or a single query in case of $b=1000$, where it captures the complexity of the whole pattern with a single access pattern), and **SPD-IN** always chooses **SPD**. However, the variance in query response times measured throughout the test is greater than factor of 10, which suggests unstable or nondeterministic (border case?) query execution by the DBMS, in addition to the superlinear cost (w.r.t. to selectivity) when executing such queries.

Such a drastic slowdown is typically associated with a wrong choice of execution plan for certain expected cardinalities, as noted e.g. in [83]. Our investigation has shown that the actual execution plans reported by MS SQL Server are identical for the different parameter cases. For **SPD** queries they all include index seek for the beginning of the chunk id range, and then a filter based on the MOD condition for every chunk id in that range. For **IN** queries they include index seek for each chunk id provided in the list. We have to conclude that the observed border case behaviour does not arise at the stage of query optimization.

The tiled partitioning is much better for this access pattern, grouping each 100 of diagonal elements within one tile, thus resulting in the retrieval of maximum 1000 tiles for "thin" diagonal queries, and just a few more tiles if we were testing wider diagonal bands:



QT4 Parameters		time, s					
b	w	IN(16)	SPD	SPD-IN	IN(256)	IN(4096)	
1000	1	0.233	0.318	0.329	0.179	0.177	
100	1	1.767	1.140	1.239	1.032	0.983	
10	1	1.826	1.192	1.263	1.214	1.023	
1	1	2.747	1.624	1.819	1.566	1.556	

Figure 32 *QT4* run time (s) for square tiles

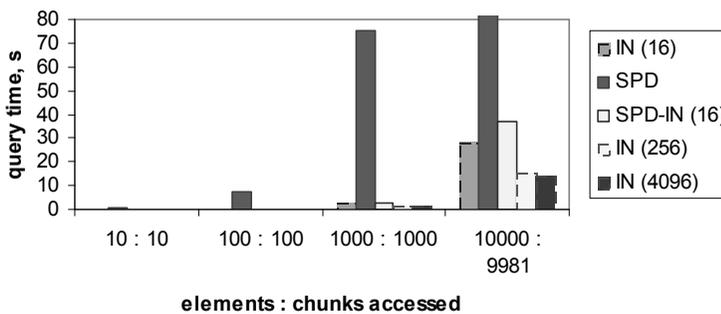
The conclusion so far is that **IN** queries with long lists are better handled by the DBMS, in cases when the access pattern presents a worst case for the given array partitioning, and the relevant chunks are not stored close to each other. On the other hand, multidimensional partitioning helps to avoid worst cases for diagonal queries, helping to speed up the execution by factor of 55.4 (for the unselective queries)

6.3.2.5 Query *QT5*

The `:ElementIndices` nodes are inserted into the database, containing independent uniformly distributed `:i` and `:j` integer values within $N \times N$ matrix domain. The selectivity is varied by using `LIMIT` clause of SciSPARQL query.

Though some elements happen to be in the same chunk, it takes a big buffer to discover this fact and save on communication. The SSDM-side chunk cache also solves the problem of the repeating chunks.

The number of chunks accessed is shown on Figures 33-34 below for the buffer size of 4096. As expected, it takes the selectivity of 0.01% for the different random array elements to be found in the same chunks, thus decreasing the overall number of chunks retrieved.



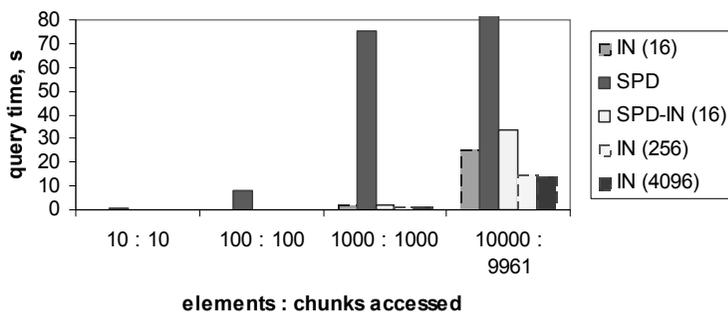
QT5 Parameter s	time, s				
	IN(16)	SPD	SPD-IN	IN(256)	IN(4096)
10	0.025	0.787	0.021	0.023	0.030
100	0.334	7.613	0.297	0.193	0.181
1000	2.564	75.312	2.597	1.413	1.565
10000	28.015	753.215	36.956	15.594	14.312

Figure 33. *QT5* run time (s) for linear chunks

Another important observation is that **SPD** obviously detects wrong patterns (since there are no patterns to detect), leading to a serious slowdown. However, **SPD-IN** is able to discard most (but not all) of these patterns as unlikely, almost restoring the default **IN** performance. And, by the

way, **SPD** is sending the same amount of 625 SQL queries as **IN** strategy does (for the buffer size of 16).

Since the distribution is uniform, there is practically no difference between chunked (Figure 33) and tiled (Figure 34) partitioning, because of the same estimate in the number of elements per chunk and hence roughly the same number of distinct chunks accessed.



QT5 Parameter s	time, s				
	IN(16)	SPD	SPD-IN	IN(256)	IN(4096)
10	0.054	0.820	0.020	0.017	0.028
100	0.262	7.733	0.156	0.122	0.146
1000	1.992	74.961	2.337	1.419	1.383
10000	25.871	755.269	33.782	14.675	14.132

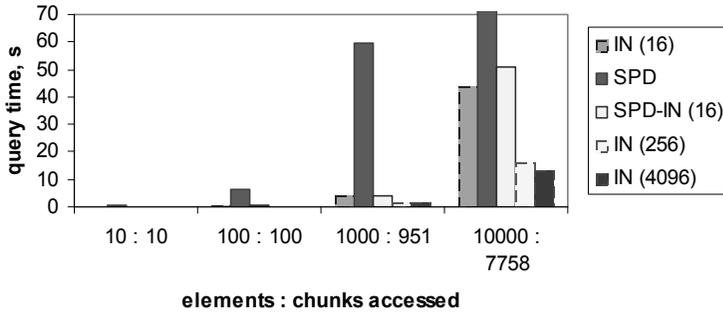
Figure 34. *QT5* run time (s) for square tiles

This part of the experiment shows that **SPD** is certainly not suited for random access patterns, and **SPD-IN** rules help (though not completely) to avoid misdetections (leaving a small amount of false positives). As for the **IN** strategies, extensive pre-buffering of chunk ids, and caching of the retrieved chunks on the SSDM side helps to avoid repeated retrievals.

6.3.2.6 Query QT6

This time the matrix coordinates are generated in clusters. For the test purposes we generate 3 clusters, with centroids uniformly distributed inside the matrix space. The probability of a sample being assigned to the cluster is uniform. Samples are normally distributed around the centroids with variance $0.01*N - 0.03*N$ (randomly picked for each cluster once). We deliberately use such highly dispersed clusters, as the effects of logical locality already become visible at certain selectivity threshold. Samples produced outside the $N \times N$ rectangle are discarded, thus effectively decreasing the weight of clusters with centroids close to the border.

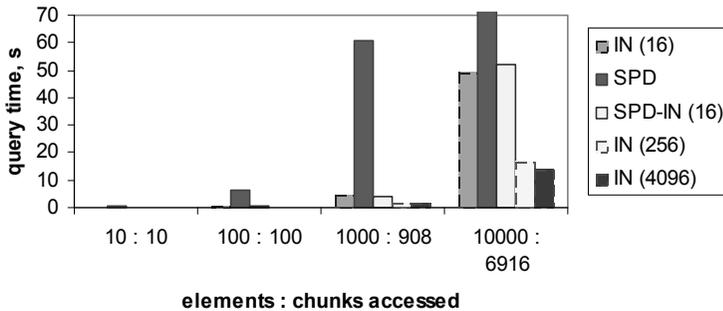
Similarly to *QT5*, in Figures 35-36 we show the minimal number of chunks accessed (i.e. given the biggest buffer).



QT6 Parameter s	time, s				
	IN(16)	SPD	SPD-IN	IN(256)	IN(4096)
10	0.077	0.646	0.047	0.044	0.048
100	0.432	6.092	0.347	0.168	0.170
1000	3.849	59.816	4.201	1.732	1.520
10000	44.068	619.773	50.710	16.426	13.493

Figure 35. *QT6* run time (s) for linear chunks

We see that the effect of logical locality starts to play a role already when selecting 0.001% of the array elements. At the selectivity of 0.01% the number of chunk to access is just 78% to the number of elements in the case of linear chunks, and 69% in case of square tiles. We see that the square tiles better preserve the logical query locality, especially for the unselective queries. We expect this effect to be even greater for more compact clusters w.r.t. the tile size, and Experiment 3 below (where we vary the chunk size) supports this idea.



QT6 Parameter s	time, s				
	IN(16)	SPD	SPD-IN	IN(256)	IN(4096)
10	0.071	0.637	0.043	0.043	0.048
100	0.494	6.194	0.399	0.202	0.158
1000	4.427	60.679	3.845	1.709	1.555
10000	49.145	625.696	52.054	16.715	13.632

Figure 35. *QT6* run time (s) for square tiles

One important conclusion here is that we are able to achieve sub-linear increase of query response w.r.t. the amount of array data retrieved - using extensive chunk id pre-buffering and SSDM-side caching of the retrieved chunks. We are able to show this effect on rather low overall densities. We save on the amount of retrieved chunks thanks to the cluster characteristic of the access pattern.

6.3.2.7 Comparing linear chunks vs. square tiles

In this experiment, we have gathered empirical proof to a common intuition [41, 60, 105, 142, 172] that for every data partitioning scheme there is a possible worst-case workload. Furthermore, our theoretical expectations regarding best and worst case access patterns for each array partitioning found full support. These can be summarized by the following table, listing *QT1* - *QT4* as representative access patterns:

Table 7. Partitioning/workload best and worst cases

access pattern	linear partitioning		multidimensional partitioning
	row-major	column-major	
QT1	best	worst	
QT2	worst	best	
QT3			worst*
QT4	worst	worst	

The multidimensional partitioning has its only worst case (when a separate chunk needs to be retrieved for each element) on sparse enough regular grids. Also, as shown by *QT6*, the multidimensional partitioning is still more advantageous for random access patterns, with even a small degree of clustering. Overall, it can be regarded as more robust, though having fewer best-case matches. Compact enough clusters that can be spanned by a small number of tiles would obviously be a near-best-case access pattern.

6.3.2.8 Comparing SPD vs. IN strategies

The **SPD** approach in most cases allows packing the sequence of all relevant chunk ids into a single SQL query, and thus skipping all the subsequent buffering. However, we have discovered that the **SPD** queries are generally do not perform so well in the back-end DBMS, as queries with IN lists. The last two cases show very clearly that in the case where there is no pattern, so that we have to send the same amount of **SPD** and **IN** queries (for the same buffer size), the difference in query response time is greater than order of magnitude.

The obvious explanation to this is the index utilization. A query with an IN list involves index lookups for each chunk id in the list, while a query with `mod` condition, as generated with **SPD** strategy, is processed straightforwardly as a scan through the whole *ArrayChunk* table.

We believe it could be highly advantageous to implement a simple rewrite on a `mod()` function. A condition like ' $X \bmod Y = Z$ ' with Z and Y known, and X being an attribute in a table (and thus having a finite set of possible bindings), could be easily rewritten to generate a sequence of possible X values on the fly (thus making `mod()` effectively a *multidirectional function* [58]).

This, however, would require a facility to generate sequences of values during the execution plan. In Amos II [136] generators are used for all bag-valued functions. We have run additional experiments with other RDBMSs, including PostgreSQL [125], MySQL, Mimer [106], and found that even though some of these support table-valued UDF, only the recent versions (tested 9.4.4) of PostgreSQL are capable of avoiding the generation of complete sequences before use. We see this as an important improvement in the present-day RDBMS architecture.

6.3.3 Experiment 2: Varying the Buffer Size

In this experiment we explore the impact of the buffer size on the query response time with the **IN** strategy. We use the same dataset as in Experiment 1, and query *QT2* as a model query because it is the simplest query retrieving a large amount of chunks. We retrieve 10 000 *linear chunks* each time (*QT2* parameters result in accessing an element from a single column and every 10th row). Chunk sizes are varied with finer resolution than in the respective case of Experiment 1.

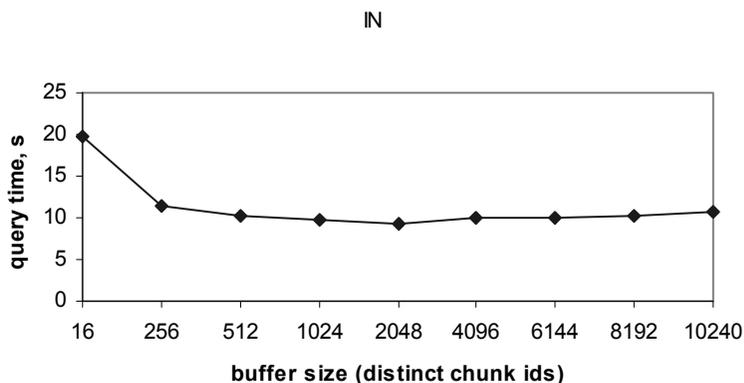


Figure 37. *QT2* run time (s) for linear chunks, **IN** strategy, with buffer size varied

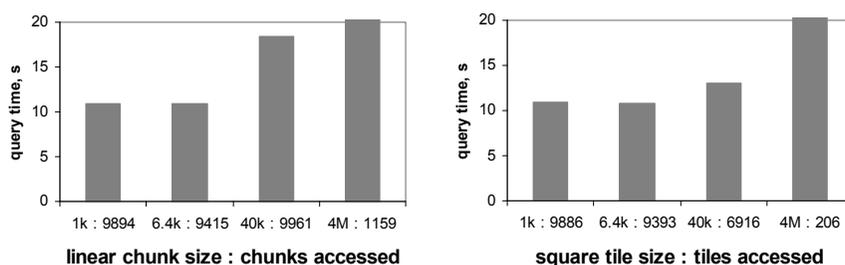
The results shown on Figure 37 confirm our hypothesis that extremely small buffers, producing lots of chunk retrieval queries under the **IN** strategy, result in unnecessary query processing overhead. However, after a

certain threshold the amount of SQL queries sent is low enough (1-10 queries), so this overhead is not significant anymore.

6.3.4 Experiment 3: Varying the Chunk Size

In this experiment we evaluate the trade-off between the need to retrieve many small chunks in one extreme case, and few big chunks (with mostly irrelevant data) in the other extreme case. We chose *QT6* as a query with certain degree of spatial locality. The effect of this locality is greater for the square tiles, which are aligned to the logical dimensions, than for the linear chunks.

We also study how well our back-end DBMS handles the requests for numerous small or big binary values, thus using the *IN* strategy with buffer size set to 4096 distinct chunks. In each case, we retrieve 10000 array elements, arranged into three clusters, with variance chosen in range $0.01 * N \dots 0.03 * N$.



chunk size	linear chunks		square tiles	
	accessed	response time, s	accessed	response time, s
1k	9894	10.923	9886	10.945
6.4k	9415	10.897	9393	10.846
40k	9961	18.396	6916	13.022
4M	1159	612.269	206	101.333

Figure 38. *QT6* run time (s) for linear chunks, *IN* strategy, with chunk size varied

Figure 38 shows the results for both partitioning cases: even though big chunk size (around 4 megabytes) results in a much smaller amount of chunks retrieved (only 1 SQL query is sent), the overall query time rises superlinearly to 612 s. Besides that, smaller chunks result in slightly better performance in this case, since the amount of "small" chunks retrieved stays approximately the same for the same sparse random access.

Using square tiles helps to leverage the access locality even better. However, big tiles do not seem to pay off at this level of sparsity: retrieving 206 4-megabyte tiles results in a factor of 81.4 larger binary data retrieval

than 9886 1-kilobyte tiles, and contributes to a factor of 9.26 longer query time (101 s).

This experiment shows that for the given access selectivity (10^{-6} of the total number of array elements selected randomly in clusters), small chunks perform better than big chunks, and the choice between linear chunks or square tiles is not important for small chunk/tile sizes. However, there is a significant overhead in retrieving separate chunks, as a factor of 81.4 gross data transfer increase contributes to only a factor of 9.26 query time increase.

Analytically, we would model the query response time as a function $T(s)$ of chunk size s :

$$T(s) = P(s)N(s)$$

where $P(s)$ is the cost of transferring one chunk (given a fixed total number of SQL calls), and $N(s)$ is the amount of relevant chunks to be retrieved. Figure 39 shows our qualitative expectations of $P(s)$ and $N(s)$. It illustrates that $P(s)$ is basically linear after some 'efficient chunk size' threshold, while $N(s)$ should experience a steep fall, corresponding to the logical locality of the query, which is saturated at some point.

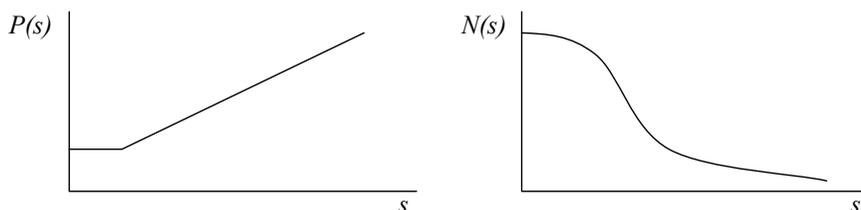


Figure 39. Query response time factors shown qualitatively as functions of chunk size

While the quantitative properties of $P(s)$ depend largely on the underlying DBMS, the middleware, and the operating system used (along with hardware configurations), $N(s)$ is purely statistical, and can be easily computed by simulation, as presented below.

6.3.4.1 Amount of distinct chunks as a function of chunk size

Figures 40 and 41 below show the simulation results of *QT6* retrieving 10 000 random elements, with clusters of element coordinates having average variance of $0.2 \cdot N$ (*very dispersed*) to $0.0002 \cdot N$ (*very condensed*). Figure 40 presents $N(s)$, given the linear chunks of varying size, and Figure 41 presents $N(s)$ for square tiles.

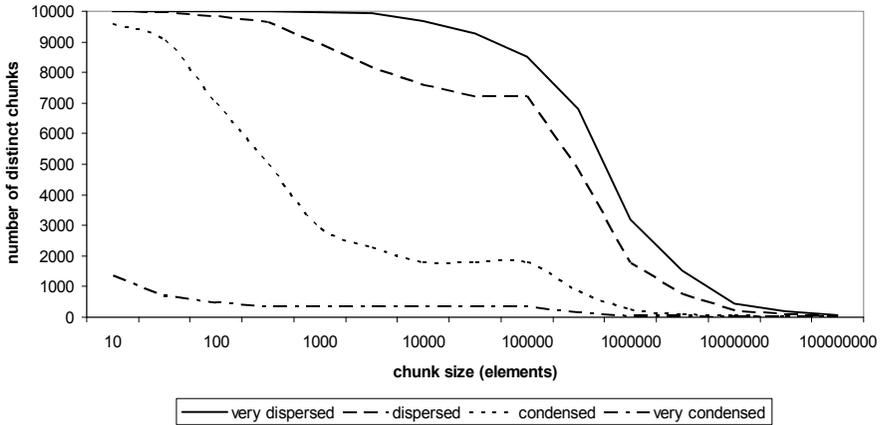


Figure 40. Amount of distinct linear chunks as a function of chunk size, results of simulating *QT6* retrieving 10 000 elements clustered with different density.

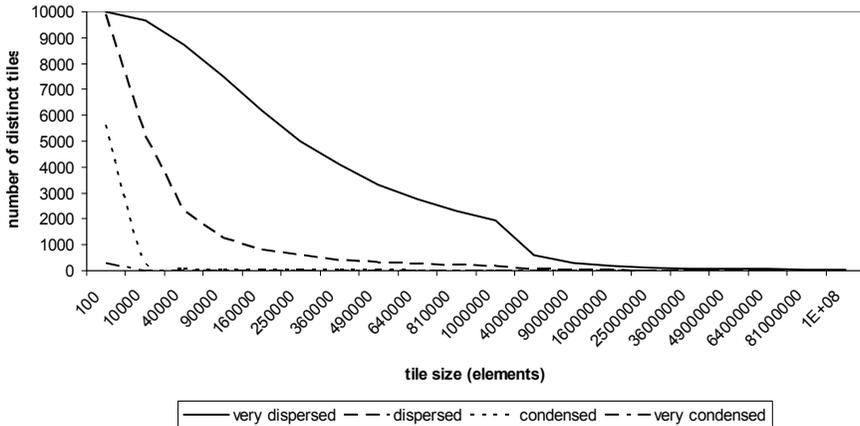


Figure 41. Amount of distinct square tiles as a function of tile size, results of simulating *QT6* retrieving 10 000 elements clustered with different density.

As we can see, the linear chunk case clearly exhibits a top 'plateau' for most of our cases, and thus confirms our expectations above. This feature is not visible for the square tiles case (Figure 41), as the square tiles utilize the query locality much better. In order to see the plateau, we have to re-run the simulation with a greater sparsity (so that there is a greater probability of having single selected element per tile retrieved). Figure 42 shows the result of such simulation, with *QT6* retrieving this time only 1000 random elements.

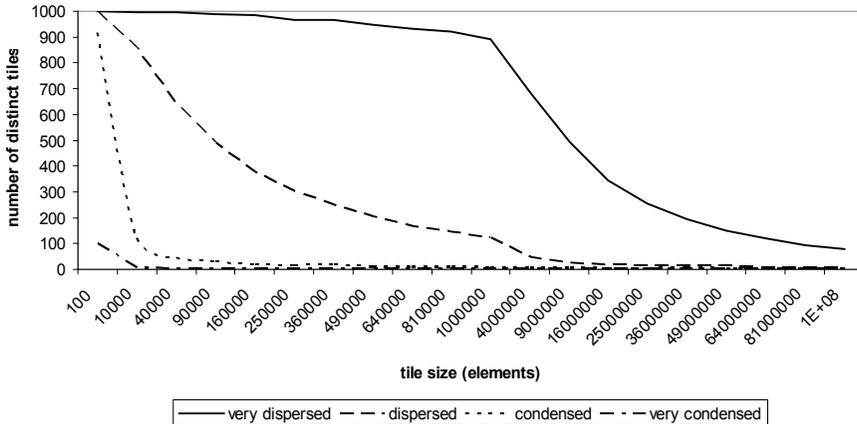


Figure 42. Amount of distinct square tiles as a function of tile size, results of simulating *QT6* retrieving 1000 elements clustered with different density.

Another interesting feature on Figures 40 and 42 is a 'middle plateau' for the (not very) dispersed access patterns. The beginning of such plateau should be considered as one of the sweet spots when choosing the partitioning granularity, where chunk/tile size is adequate to the distribution of access densities. Of course, assuming the statistical properties of the workload are known before the array data is partitioned.

Similarly, the earlier observations (Figure 38) suggest that there is always a threshold in access density after which the bigger chunks become more favorable. For example, we expect 4 MB square chunks to be on par with 1 kB chunks, when the gross data transfers for each case relate as $\sim 8:1$. In other words, it still pays off to transfer 8 times more gross data from a back-end, if it results in retrieving correspondingly lesser amount of chunks.

6.3.5 Summary of the Comparison Experiments

We have compared two pure and one hybrid strategy for generating SQL queries based on the buffered set of chunk ids to be retrieved. One is putting a long IN list into the query, and the other is creating an expression for a cyclic chunk access pattern discovered. It turned out that even though the second approach allows accessing an entire array with a single SQL query, and skipping further buffering in most cases; it only pays off for very unselective queries, retrieving a large percentage of array's chunks. Apparently, modern RDBMS optimization algorithms do not rewrite the kind of conditional expressions we were using in order to utilize existing indexes. Hence, the general advice is to use long IN lists for best performance of a contemporary RDBMS as a back-end.

We have also investigated two distinct partitioning schemes - linear and multidimensional - used to store large numeric arrays as binary objects in a relational database back-end. Our mini-benchmark consists of six distinct parameterized query patterns, and it becomes clear that for each partitioning scheme one can easily define best-case and worst-case queries. For example, a diagonal access pattern works much better with square tiles than with any linear chunking, where linear chunks in an array stored row-by-row are perfect for single-row queries and worst for single-column queries. As for the chunk size, we have empirically found a proportion when the overhead of transferring more gross data balances out the overhead of retrieving more chunks.

The conclusion is that choosing the right partitioning scheme and chunk size is crucial for array query response time, and the choices made should be workload-aware whenever possible. Though it might not be possible to know the expected workload for long-term storage of scientific data, such knowledge can certainly be inferred for materializations of intermediate results in cascading array computation tasks. As one direction of a future work, a query optimizer that makes choices on materializing intermediate results (e.g. common array subexpressions) should be enabled to choose the storage options based on the downstream operations.

Buffering array access operations and formulating aggregated queries to the back-end has proved to be essential for performance. We put the relational back-end scenario to a real-life test in the next section, comparing the performance with purely manual Matlab implementations of the same scientific computing tasks.

6.4 Real-Life Query Performance Evaluations

The previous section explored the SSDM performance using synthetic (ultimately simplified) data and queries implementing a variety of typical access patterns. Though we have used SciSPARQL to formulate parameterized array queries, the presented results are largely language-independent, and can be extrapolated to any setting involving a chunked array access. It is only a real-life application that can put a wide range of SciSPARQL features and SSDM architectural decisions to a realistically integrated test.

In this section we demonstrate the expressivity of the SciSPARQL language for an application-representative set of queries and present response times with different relational storage back-ends. We show that our approach results in comparable performance to hand-written Matlab scripts reading files directly from disk, which is the data processing approach

previously employed by the users of the BISTAB application. BISTAB is a stochastic simulation in the field of computational biology we use for evaluation of SciSPARQL and SSDM.

In Section 6.4.1 we first define the BISTAB application, together with some scientific computing background. This part of the work is the outcome from a collaboration with A.Hellander and B.Drawert at the University of California Santa Barbara. The results were published in [6] and Section 6.4.1 is based on the part of the paper describing the BISTAB experiment written by A.Hellander. The computational and data management problems are put in focus, and the typical data post-processing tasks are formulated. The BISTAB application has motivated our following steps to move the computation towards the data, so the results presented here are an important reference point.

Next, in Section 6.4.2 we define the *RDF with Arrays* schema, capturing BISTAB data and metadata together. Since the metadata (parameter cases, grid properties, etc.) were stored separately - partially in arrays in separate Matlab files, partially encoded into file names - we had to do a pre-loading step. In order to build an *RDF with Arrays* dataset to be queried, certain ad-hoc parsing and other data retrieval operations were performed, which is not covered here. Instead, in Chapter 7 we demonstrate a more general and simple way for application users to provide metadata annotations for their numeric results.

The prepared Turtle files (with *file links* for the computation results) were loaded into SSDM with a relational back-end configured to store the arrays, as reported in Section 6.4.3. Section 6.4.4 presents the BISTAB queries formulated in SciSPARQL. The queries are answered using different relational back-ends and under different cache states. In Section 6.4.5 the performance is compared to running the equivalent scripts in Matlab.

As expected, employing SSDM with a relational database back-end for storage of large array data results in comparable performance to using Matlab natively, however, at a cost of the initial data-loading phase. As presented in Chapter 7 we can avoid paying this cost altogether, by introducing a tight integration of SciSPARQL queries into a Matlab-based scientific computing workflow, and retrieving the array data on demand directly from `.mat` files.

6.4.1 BISTAB: an Application from Computational Biology

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)

[63]. 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 [56, 52].

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.

6.4.1.1 The URDME framework

BISTAB is implemented using the URDME framework for stochastic simulation of reaction-diffusion processes on unstructured meshes [48, 53]. 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.

6.4.1.2 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 [184]. 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.

6.4.1.3 Model problem

The BISTAB dataset is a model of a bistable system [52], 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.

6.4.1.4 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 here are:

- **BQ1**: Compute the number of molecules over the whole spatial domain of a certain species as a function of time.
- **BQ2**: Compute the number of certain species at a certain time point for all the realizations that have kinetic rate constants in a certain range.
- **BQ3**: Retrieve the identifier of the trajectory that resulted in the maximal result for **BQ2**.

The operation in **BQ1** is typical for visualization of the realizations and is for example needed to produce the time series plots in [52, 56]. While simple array slicing and aggregate functions like that done on a single matrix in **BQ1** can be expressed easily and efficiently in a scripting language such as Matlab, already simple queries such as **BQ2** and **BQ3** 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.

6.4.2 BISTAB Data Model as *RDF with Arrays*

We have developed a database schema for the BISTAB dataset, as described by the ER-diagram shown on Figure 43. It is used for generating an *RDF with Arrays* dataset to be stored in SSDM with a relational database backend. The dataset is loaded as the default graph in SSDM. In the BISTAB schema..To test SQL-based storage, we use a sample of 100 *Task* instances.

All BISTAB data and metadata are contained in the properties of *Experiment* and *Task* instances. 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, as shown on the Figure 44a. The elements of U matrix are the populations of given species in a given mesh cell at a given time point.

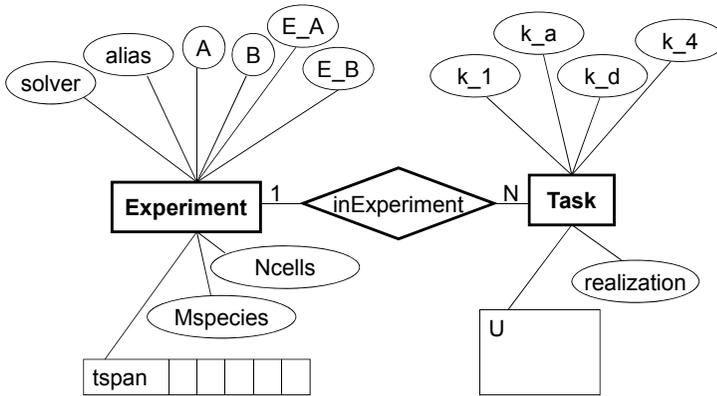


Figure 43. Entity-relationship diagram of the BISTAB dataset

The number of cells (*Ncells*), species (*Mspecies*), and the time values for every time point (*tspan* vector) are part of *Experiment* metadata. The types of species *A*, *B*, *E_A*, *E_B* and four others are modeled as properties of the :*Experiment* instance. Their values are the row offsets used to access rows corresponding to these species types within a row range of a given cell.

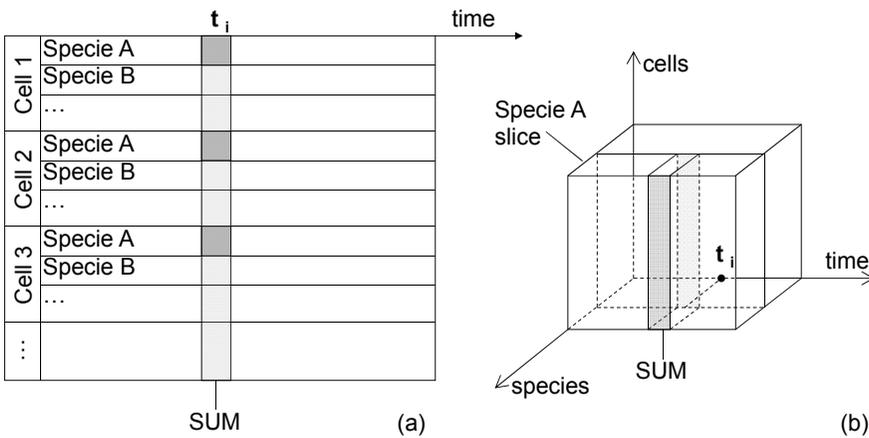


Figure 44. Simulation results stored in *U* matrix

A more natural representation of *U* would be a 3D array, as shown on Figure 44b, with the *cells* and *species* dimensions logically unnested (though physically all dimension are always nested, both for chunk-based and main-memory storage). However, the BISTAB stochastic simulation outputs the results as 2D arrays, and though it would be simple to re-shape these arrays, we prefer to keep the array expressions in our queries similar to those in the original Matlab scripts we compare our approach to. With the flexibility of SciSPARQL we do not really need to restructure the data before querying.

Together with each U matrix, a set of simulation parameters k_I , 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.

6.4.3 Experiment Setup and Data Loading

We have deployed both SSDM and the back-end DBMSs on a single HP Compaq 8100 workstation with Intel Core i5 CPU @ 2.80 GHz, 8 GB 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, together with *file links* to the binary (.mat) data files containing the experimental results (U matrices). We used a dataset containing 100 :Task instances, each representing a realization of the U matrix, containing an integer element for each of (11107 cells \times 8 specie types \times 201 time points). This amounts to about 71.5 MB per matrix, and the total of ~ 7.15 GB array data in our sample dataset.

As for the, 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 *linear chunks* of 1608 bytes each, so that a chunk contains two successive rows of a U matrix, stored row-by-row. This amounts to 44 428 chunks per matrix, and 4 442 801 array chunks in total (one chunk stores *tspan*).

The choice of the partitioning scheme was deliberately made data-aware, but not workload-aware, in order to test both natural and worst-case workloads. For example, each chunk contains the complete time series per mesh cell for two types of species, whereas **BQ1** and **BQ2** are only interested in one certain kind of species (while **BQ3** may benefit from chunk caching). At the same time, all queries summarize the populations across all mesh cells, so storing the matrix U column-by-column would benefit **BQ2** and **BQ3**, which are currently the worst-case workloads, retrieving only 1 or 2 elements per chunk. Below in Section 6.4.4 we formulate **BQ1** in a way that it retrieves the time series per mesh cell first, and then applies a vector sum, thus becoming a near-best case (50% relevant data per chunk). Automatic query rewriting based on the partitioning scheme remains a challenging direction of the future work.

6.4.3.1 Bulk-loading performance

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 8. 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 (Figure 43) 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 8. 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.

6.4.4 BISTAB Application Queries

In this section we define the queries **BQ1-BQ3** outlined in Section 6.4.1.4 as SciSPARQL queries, using a number of distinctive features of the query language, introduced in Chapter 4.

BQ1: 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. **BQ1** 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 matrix:

```

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 Figure 44a. 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 Figure 44b.

Query **BQ1** 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 (i) the *U* matrix associated with the experiment task instance named `:Task1`, (ii) the corresponding experiment instance (property `:inExperiment`), and (iii) other metadata associated with the experiment. The query returns pairs of (*timepoint*, *sum*) values that can be directly used for plotting the wanted function of time.

BQ2: 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. **BQ2** selects just one column of the *U* matrix, at the column index `?j` which is looked up in 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 Figure 44a). 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()`.

BQ3: Find the task that has the maximal total population of species *A* or *B* for any time point. **BQ3** 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: summed up populations of species *A* and *B*, and their maximum element is returned. The function is similar to **BQ1** and **BQ2** 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 the query **BQ3**:

```
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 (introduced in Section 6.2.4.6) prevents double retrieval of the same chunks. A very small cache, capable of storing 25% chunks of a matrix (less than 18 MB), completely eliminates this problem. The matrices are accessed one at a time, so that the chunk cache is automatically refreshed (according to the least-recently-used (LRU) replacement strategy) when function `max_AB_sum()` proceeds to the next matrix.

6.4.5 Query Performance

Once the data was loaded into SSDM configured with the relational back-end storage, we ran the queries **BQ1-BQ3** and measured the execution time and the number of resulting tuples emitted. We used the **SPD** strategy for retrieving the chunks (since all our patterns are fairly regular) with the buffer size of 16 distinct chunks. With the results from Section 6.3 in mind, all three queries are a suitable case for the **SPD** strategy, since the chunk access pattern is fairly dense (in fact, all queries retrieve every 4th chunk in a matrix).

As a comparison, we also made Matlab scripts to perform the equivalent computations on a set of 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 9 shows 'cold cache' run times where all data reside on disk before the query is executed:

Table 9 Query execution time (in seconds) with cold cache

task	U matrices	results	MySQL	MS SQL Server	MATLAB
BQ1	1	201	1.748	2.15	1.826
BQ2	36	36	80.703	44.512	30.042
BQ3	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 10 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 chunk cache. Due to massive amounts of data processed, **BQ3** does not benefit from any of these in repeated runs (though intra-query caching is still essential), and the results are the same as in Table 9. In contrast, for **BQ1** there is an interesting case possible when all array data processed fit entirely into the SSDM chunk 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 **BQ1***.

Table 10. Query execution time (in seconds) with warm OS/DBMS level cache

task	U matrices	results	MySQL	MS SQL Server	MATLAB
BQ1	1	201	0.434	0.526	0.157
BQ1*	1	201	0.138	0.152	N/A
BQ2	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, **BQ2** reads just a single column from every matrix, but has to retrieve the same amount of chunks from the back-end. This makes it significantly slower than a system with any other partitioning scheme than row-wise linear chunking, used in this experiment. Single-column access can be regarded as particular worst case for row-based array storage, as the useful data load is relatively small during the array retrieval operations.

7 Integration of SciSPARQL into Matlab

In many branches of science and engineering, researchers accumulate large amounts of experimental data [162, 154] and use widely recognized libraries of algorithms to analyze and refine that data. Tools such as Matlab or similar serve as integrated environments that provide basic file management, extensibility with algorithmic libraries, visualization and debugging tools, and are generally oriented towards single-user scenario.

What is typically missing is the infrastructure for storing the descriptions of experiments, including parameters, terminology mappings, provenance records and other kinds of metadata. At best, this information is stored in a set of variables in the same files that contain large numeric arrays of experimental data, and thus is prone to duplication and hard to update.

RDF with Arrays and SciSPARQL provide an infrastructure where both metadata and data are represented in a database using the RDF data model accessible from the Matlab environment.

7.1 Usage Scenario

The SSDM configuration presented in this chapter assumes a multi-user client-server environment. Users interact with Matlab clients and the SSDM server acts as a central repository for all kinds of data and metadata that needs to be stored between the sessions or shared among the users.

The main challenge, as expected, is interoperability - Matlab has only arrays as a data model. We address this by allowing users to provide metadata annotations for the array data they generate, within the Semantic Web paradigm, so that the Matlab arrays become part of an *RDF with Arrays* graph on the server, which is queryable and updatable with SciSPARQL. SSDM as an *RDF with Arrays* store participates in two kinds of interactions with a Matlab client, which we refer to as *phases*:

- Phase 1: **generate & store** - populating an *RDF with Arrays* graph on the SSDM server with data and metadata generated on the client.
- Phase 2: **query & postprocess** - searching an *RDF with Arrays* graph based both on metadata and data properties, shipping the results back to the Matlab client to perform any Matlab-specific postprocessing.

The first phase is exemplified in Figure 45. First, a Matlab array A is created on the client. A call to the Matlab function `store()` ships the array to the server and returns an *array proxy object*. This array proxy object is used in RDF triples later sent to SSDM when populating the *RDF with Arrays* graph describing the experiment.

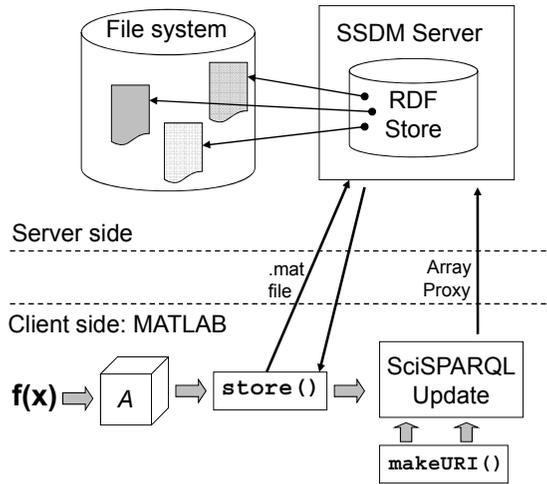


Figure 45. Storing client-generated data and metadata on SSDM server

At the query phase (Figure 46), a subset of A , now stored on the server in a `.mat` file is selected, processed (e.g. fed to an aggregate function), and the result (e.g. a single number) is shipped back to the Matlab client for post-processing and visualization.

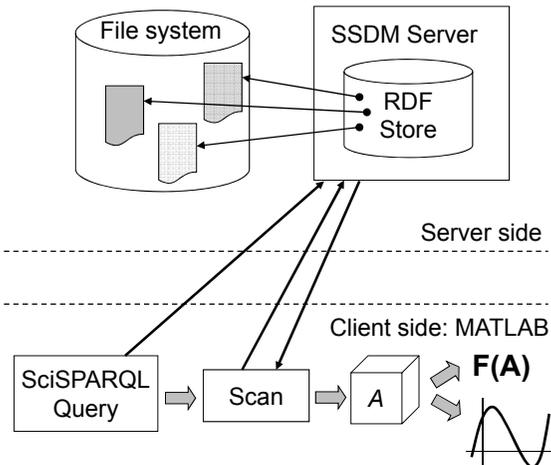


Figure 46. Querying data and metadata on SSDM server from MATLAB client

In the next section we demonstrate both of these phases using the example BISTAB experiment introduced in Section 6.4.

7.2 A Workflow Example

As an example¹⁵ of a typical workflow, we first create the *RDF with Arrays* dataset for the BISTAB experiment using a remote Matlab client. As soon as both RDF data and arrays are stored in the SSDM server, we are able to query them with SciSPARQL, receive the results using the *Scan* functionality, and perform visualization in Matlab.

First, we load the SSDM client library into Matlab, initialize it, and establish a connection to the server:

```
addpath('./embeddings/MATLAB/M');  
sparqlInit('mat');  
c = newConnection('udbl64.uu.se');
```

The 'mat' option to the initializer indicates that we are going to use .mat files for storing the arrays on the server, so that the corresponding MCR libraries are loaded at that point.

We use the prefix <http://udbl.uu.se/bistab#> for the URIs we construct for our *RDF with Arrays* dataset, both on the client and the server:

```
c.usePrefix('', 'http://udbl.uu.se/bistab#');
```

Phase 1. For simplicity, we begin the construction from an empty graph, and first insert the :Experiment1 instance with constant parameters (see Figure 45):

```
c.sparqlDo('CLEAR()');  
c.sparqlDo('INSERT { :Experiment1 :Mspecies 2; :Ncells 8; :A 1; ' ...  
                    ':B 2; :E_A 3; :E_B 4; :tlen 5 }');
```

The Matlab concatenation operator '...' is used to pass multi-line textual representations of SciSPARQL queries and updates.

In order to insert an array, we use the Matlab-native array value for the :tspan property. The URIs representing the *subject* and *property* of the inserted triple are also created in Matlab.

```
uriExperiment = c.makeURI('', 'Experiment1');  
c.insert(uriExperiment, c.makeURI('', 'tspan'), [0 0.5 1 1.5 2]);
```

¹⁵ In the Matlab code below we show the integration-related functions in bold italic, with keywords (both Matlab and SciSPARQL) in bold.

Now we populate the BISTAB realizations data. We have these data in variables in a set of `.mat` files, so we load the file with BISTAB parameter cases first:

```
load('input.mat');
```

This file contains the `parameters` variable, with rows corresponding to the different realization parameters, and the columns corresponding to the parameter cases. By contrast, in Section 6.4 we read a pre-generated *Turtle* file with these values, while here we populate an *RDF with Arrays* graph online. For each parameter case we create a `:TaskN` node. Since we are building a graph, first we record that our new `:TaskN` node belongs to `:Experiment1`. The rows in the `parameters` array are now assigned meaningful names, like `:k_1` or `:k_d`, serving as part of metadata annotation:

```
for i = 1:size(parameters,2)
    uriTask = c.makeURI('', ['Task', int2str(i)]);
    c.insert(uriTask, c.makeURI('', 'inExperiment'), uriExperiment);
    c.insert(uriTask, c.makeURI('', 'k_1'), parameters(1,i));
    c.insert(uriTask, c.makeURI('', 'k_a'), parameters(2,i));
    c.insert(uriTask, c.makeURI('', 'k_d'), parameters(3,i));
    c.insert(uriTask, c.makeURI('', 'k_4'), parameters(4,i));
    load(['realization_',int2str(i),'_ltoy.mat']);
    c.insert(uriTask, c.makeURI('', 'U'), c.store(U));
end
```

As the last step, for each parameter case we populate the server database with the massive numeric data, produced as a result of a computer simulation. We could have run the simulation itself at this point, but it was run before and the results were stored as in a set of `.mat` files on the client machine, with parameter case and realization numbers encoded into the filename (a very common practice in the absence of databases!). Now we integrate our massive arrays into the RDF graph by shipping them to the server and connecting them as `:U` properties of our `:TaskN` nodes.

This is all data migration efforts needed to convert the `.mat` files on the client into an *RDF with Arrays* database on the server. We can now save the database on the server:

```
c.save();
```

Phase 2. The array data stored on the SSDM server is now available for querying from Matlab clients. For example, we can select row 15 from the `:U` matrix corresponding to `:Task1` and send it back to the client:

```
s = c.sparql('SELECT (?U[15] AS ?res) WHERE { :Task1 :U ?U }');
s.getElement(1)
```

In the process we created a *Scan* object containing a single row with a single value in it, which is available in Matlab as a 1D array.

We can now execute **BQ1** defined in Section 6.4.4, by first defining its reusable part as a `total_species()` function:

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

This function will be stored on the SSDM server, together with the dataset - similarly to SQL stored procedures.

Executing **BQ1** will return a *Scan* containing a time value and the sum of *A* species in each row:

```
s = c.sparql([
  '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 }'])
```

For the purpose of plotting, we need to collect the results from the *Scan* into a 2-column array. This is done by the following Matlab code:

```
i = 1;
while not(s.endOf())
  for j=1:s.width()
    res(i,j) = s.getElement(j);
  end
  i = i + 1;
  s.nextRow();
end
```

Here the `nextRow()` method to advances through the *Scan*, and `endof()` checks if there are any more results to retrieve. The number of elements in the row is available via `width()` and `getElement()` returns the row element with the specified index as a Matlab value.

Finally, the Matlab plotting functionality is invoked in order to visualize **BQ1** results:

```
figure;plot(res(:,1),res(:,2))
```

The complete demo, also featuring **BQ2** and **BQ3** queries is available on SciSPARQL homepage [146].

7.3 Matlab Interface to SSDM

The interface to Matlab includes two main Matlab classes: *Connection* and *Scan*. In addition there are a number of classes used to represent RDF types, e.g. URIs and typed literals. Matlab constructors, and field accessors are

defined for these classes. A special class *MatProxy* is used on the client-side to represent an array stored in a `.mat` file in the SSDM server.

The Matlab class *Connection* encapsulates a connection to an SSDM server, including methods for:

- executing SciSPARQL queries and obtaining a result as a *Scan* – method `sparql()`;
- executing non-query SciSPARQL statements, e.g. updates and function definitions – method `sparqlDo()`, separate triples may also be inserted into an *RDF with Arrays* graph with the `insert()` method;
- defining URI prefixes to be used both on client and server side – method `makeURI()`;
- shipping Matlab arrays from client to server for bulk loading – method `store()`;
- managing data persistence on the server – method `save()`.

The Matlab class *Scan* encapsulates the result set of a query. The data is not physically retrieved, stored or shipped anywhere before it is explicitly accessed as a row in the scan. *Scan* includes methods `nextRow()` etc. for iterating through the result sets of SciSPARQL queries: the arrays and scalar numbers become represented by native Matlab arrays and numbers while other RDF values get represented by objects defined in the Matlab client.

In the above workflow example the SSDM server is configured to store RDF triples in main memory, while array data is stored in a file directory of native `.mat` files. Reading and writing `.mat` files on the server side is done via freely distributed MCR libraries, so this configuration requires no additional Matlab installation.

The SSDM server processes SciSPARQL queries and updates. As part of an update, the `store()` function can be called from the client. A numeric multidimensional array value in the Matlab client will be shipped to the server as a binary `.mat` file and saved under a server-managed name in the server's file system. The *Array Proxy* object pointing to the value in that `.mat` file will be returned to the client, and may be used as a replacement for the actual array, e.g. as a parameter to SciSPARQL queries and updates. Once stored in the database, the *Array Proxy* object serves as a link from the metadata RDF graph to the numeric data stored in a `.mat` file on the server.

If the file is already on the server, and its name is known, an alternative `link()` function can be used to obtain an equivalent *Array Proxy* object persisted on the server side.

7.4 Discussion

The use of standard query languages for bringing remotely stored data into computational environments is becoming increasingly popular as the data become more distributed. One obvious benefit is simplicity and reusability of data retrieval operations. For example, Matlab already has a facility to execute SQL. Similarly, the R statistical environment recently gained a simple SPARQL package [138]. We take the next step, by extending the standard query techniques (with arrays, functional views and other SciSPARQL features), aiming to make the database connections even more useful and efficient.

The approach with linking matrices to the data on the server instead of downloading and storing them locally is beneficial. There is a number of efficient binary storage formats around, and our approach can be easily extended to any of them, as long as it is possible to address stored data in terms of string or symbolic identifiers, and read specified parts of the arrays. Even when data are generated locally, it's still better to upload it once to the server, rather than distributing the massive datasets across the workstations on a regular basis.

The main benefit, however, is integrating the Semantic Web metadata management approach (RDF and SPARQL) into an environment that misses it so obviously (i.e. currently using arrays for everything). We show that the creation of an *RDF with Arrays* graph to represent both metadata and data on the server is simple, and may serve as a good annotation of experimental data. The Matlab users can now take advantage of remote and centralized repositories for both massive numeric data and metadata, send queries that combine them both, retrieve exactly as much data as required for the task, and do any further processing the way they already do.

A similar integration of SciSPARQL into Numeric Python [172] is underway. We believe that introducing SciSPARQL queries into the traditionally procedural scientific computing workflows enables a convenient and minimum-effort annotation of numeric datasets in science and engineering, using the Semantic Web approach. This, in turn, opens a way to greater interoperability and fosters wider collaboration among the users and interlinking of the open scientific data.

8 Summary and Future Work

In this Thesis we presented the design, implementation and evaluation of *Scientific SPARQL* - a language for querying data and metadata represented using the RDF graph model extended with numeric multidimensional arrays as node values - *RDF with Arrays*. The techniques used to store *RDF with Arrays* in a scalable way and process Scientific SPARQL queries and updates are implemented in our prototype software - Scientific SPARQL Database Manager, SSDM, and its integrations with back-end data storage systems and computational frameworks.

In *RDF with Arrays*, arrays are used to model massive numeric data, typically ordered along a number of orthogonal axes. The rest of the RDF graph serves to represent different kinds of metadata, for example, a formalized description of an experiment, tools and methods used, parameter cases, provenance, etc. Scientific SPARQL allows combining metadata and numeric data conditions in one query, making it expressive and self-contained, eliminating the need for extra round trips to the server, and giving more freedom to the optimizer in order to build better execution plans.

The ability to process Scientific SPARQL queries involves suitable main memory representations for numeric multidimensional arrays, and efficient implementation of operations over such arrays (e.g. selecting array subsets). Whenever possible the SciSPARQL query processor accumulates such array operations and accesses the array content in a lazy fashion.

For scalability, arrays can be physically stored in a variety of external storage systems, including files, relational databases, and specialized array data stores - SSDM offers a simple and flexible Array Storage Extensibility Interface. The array data is retrieved from these storage systems only on demand, and only in relevant subsets, thus minimizing both network usage and memory footprint.

One option is storing *RDF with Arrays* in a relational DBMS supporting SQL and JDBC. We studied the different optimization strategies for the retrieval of array content under a variety of partitioning approaches and access patterns - the performance evaluation we present is based on our mini-benchmark for array queries. The conclusions suggest a preferred way to formulating SQL queries to the back-end, and also carry certain advice for choosing a partitioning approach, if the expected workload is known.

In scientific applications, numeric computations are often used for filtering or post-processing the retrieved data, and may typically be expressed in a functional way. Existing computational libraries (many of which became de-facto standards in scientific computing and are often referred for reproducibility of results) can be interfaced and invoked from the query language as *foreign functions*. Cost estimates and alternative directions of evaluation can be additionally specified, in order to aid the construction of better execution plans.

As we expect complex tasks to be formulated as complex queries, good query modularity becomes as important for scalability as good data design and annotation. SciSPARQL allows expressing common query sub-tasks as *functional views*, i.e. SciSPARQL functions defined as parameterized queries. This flexibility is further strengthened by functional language abstractions such as *second-order functions* and *lexical closures*. When it comes to array processing tasks, SciSPARQL offers *array constructors*, *mappers*, and *condensers* as second-order functions.

An integral real-life evaluation is presented, where SciSPARQL queries accessing array data stored in an RDBMS back-end are compared to the equivalent manually written scripts run in pure Matlab - resulting in comparable performance in the general cases. Besides, the unification of array data and Semantic Web styled metadata makes the queries shorter and much easier to write than the equivalent procedural scripts.

SciSPARQL queries are easy to integrate into the common 'sequential' scientific and engineering workflows, involving generation, storage, retrieval, and post-processing of the numeric data, typically based on programs in Java, Python, or C, or scientific computing environments like Matlab. One important benefit is the communication saved, by pushing to the server all the costly processing that can be expressed in a query, e.g. filtering and aggregation. We also demonstrate how such integration helps to supply and use the descriptive metadata, opening a way to interoperability and collaboration, while in all other aspects the users may keep doing their work the way they already do.

SciSPARQL is a proper superset of the W3C SPARQL 1.1 standard, and its query processor is implemented on the basis of Amos II - a functional object-oriented DBMS. The successful implementation of SPARQL constitutes an important part of this work, and proves the viability of such an approach in general, along with certain semantic mismatches discovered and extensions made. The SSDM system is tested, documented, and available on the project homepage [146].

There is a number of directions for future work, aimed at further improving the performance and usability of Scientific SPARQL.

When it comes to the query processing, for example, a greater freedom for the query optimizer may be achieved by conveying the bound/semibound status of query variables to the optimization stage (Section 5.4.4.2). A deeper comparison of expressions, e.g. by using the canonic forms may lead to a saved amount of computations in very complex aggregate queries (Section 5.4.5.7). Also implementing polymorphic predicates at the algebra level may further reduce the need for the disjunctive execution plans (Section 5.5).

Mastering the RDF Schema information for the purpose of type inference offers a totally new direction of SciSPARQL development. As RDF Schemas may come as a by-product of RDF views over stricter data models [97], an originally RDBMS-based scientific or engineering application ported to use SciSPARQL may initiate this line of research.

A completely separate direction of query optimization arises when building execution plans for computing array expressions. At the high data scales, careful tile-by-tile pipelining is essential e.g. for common matrix operations. The storage choices for the materialized intermediate results can be made automatically with the results from our array query benchmark in mind. The need for automating the physical design (or co-optimizing queries and storage) is manifested in [41], and correlates to the dataflow programming with array data structures [64, 118]. This would complement the automatic choice of array function implementations presented in [120], the work which is based on the same DBMS infrastructure.

The ongoing research, however, is focused on the techniques necessary to delegate larger parts of array expressions, including second-order function calls, to array databases like Rasdaman, offering rich array processing APIs. The potential benefit of delegating the computation of an aggregate function to the back-end is the transfer cost for a scalar number compared to a transfer cost for a (potentially huge) array.

Deploying SSDM as a (Scientific) SPARQL endpoint on the Web, deep integration with other scientific computing environments, cloud-based distribution and other technical improvements are also on our wish list, and await their motivating applications.

Summary in Swedish

Mängden vetenskapliga och tekniska data har ökat explosionsartat under de senaste årtiondena. Även antalet sätt att representera denna information har ökat avsevärt. Detta inkluderar hur data beskrivs och representeras såsom vilka begrepp som används, vilken detaljnivå som valts, och hur data lagras fysiskt. Denna tillväxt förväntas fortsätta då nya sätt att producera och använda data utvecklas hela tiden. Detta medför att det blir alltmer kritiskt att utveckla metoder för att integrera och kombinera olika sorters information. Semantiska Webben (Semantic Web) och Länkad Data (Linked Open Data) är lovande ansatser för att generellt beskriva och representera olika sorters information på ett lättbegripligt sätt i form av grafer av noder sammankopplade med länkar. För att representera dessa grafer används en samling tekniker utvecklade av WWW-konsortiet som kallas RDF (Resource Description Framework). Dessa tekniker omfattar bland annat frågespråket SPARQL med vilket man kan söka efter information i RDF-grafer.

Denna avhandling utreder hur RDF och SPARQL kan användas för att representera, söka och bearbeta olika sorters vetenskapliga och tekniska data. RDFs styrka är att grafer är mycket naturliga för att beskriva information i form av egenskaper hos olika objekt som personer, företag, webbsidor, etc., och hur de relaterar till varandra. Emellertid har RDF haft begränsad användning för att lagra och hantera vetenskapliga och tekniska data beroende på att det är onaturligt och ineffektivt att använda RDF för att representera numeriska data i form av vektorer, matriser, tensorer, dvs. multidimensionella arrayer. Vetenskapliga och tekniska tillämpningar kräver ofta att mätvärden lagras och bearbetas i form av arrayer och avsaknaden av arrayer i RDF och SPARQL har varit en begränsning. Vidare har det saknats möjlighet att definiera egna funktioner i SPARQL och att integrera RDF frågor med existerande beräkningssystem.

I avhandlingen presenteras design, implementering och utvärdering av *Scientific SPARQL (SciSPARQL)*, ett språk för att söka bland både data och beskrivningar av data (metadata) representerade som RDF-grafer utvidgade med numeriska multidimensionella arrayer, benämnt *RDF with Arrays*. Tekniker för att skalbart lagra *RDF with Arrays* och att därefter bearbeta SciSPARQL frågor över lagrade data har utvecklats och implementerats i SSDM-systemet (Scientific SPARQL Database Manager). SSDM är ett

öppet system som har integrerats med olika sorters databas- och beräkningssystem för att åstadkomma skalbara system för lagring av multidimensionella arrayer och operationer på dessa. Ett generellt gränssnitt gör det möjligt att fysiskt lagra arrayer i många olika datalagringssystem, inklusive primärminne, filer, relationsdatabaser, och speciella array-orienterade databassystem.

Beräkningar som används i vetenskapliga och tekniska tillämpningar kan formuleras med funktioner. I SciSPARQL-frågor kan man använda dessa funktioner för både filtrering och efterbearbetning. Dessa funktioner kan definieras i form av parametriserade frågor. Existerande beräkningsbibliotek kan transparent anropas från SciSPARQL med hjälp av så kallade främmande funktioner implementerade i olika lämpliga programmeringsspråk. Kostnadsuppskattning och alternativa sökalgoritmer kan specificeras för att göra det möjligt för SSDM att generera effektiva sökstrategier för en given fråga i SciSPARQL. Dyrbara operationer över arrayer, såsom filtrering, aggregering och vanliga matristransformationer, utförs på SSDM-servern där arrayerna är lagrade, vilket minimerar kommunikationskostnaden mellan tillämpningsprogram och SSDM. SSDMs prestanda och kraftfullhet har utvärderats för en praktisk vetenskaplig tillämpning och där jämförts med den traditionella lösningen att göra all bearbetning i ett beräkningssystem som Matlab. Vidare har SSDMs skalbarhet utvärderats med hjälp av en uppsättning syntetiska data.

Följande forskningsfrågor besvaras delvis av avhandlingen:

1. Hur kan RDF och SPARQL utvidgas för att vara lämpliga att representera, söka och analysera kombinationer av data och metadata?
2. Hur kan frågebearbetning för SciSPARQL implementeras med hjälp av existerande databassystem? I synnerhet:
 - a. Vilka utvidgningar behövs av den bearbetning som utförs och algebra som används för att representera och transformera frågor i ett databassystem för skalbart besvarande av SciSPARQL frågor?
 - b. Hur kan olika existerande system för permanent lagring av data (filsystem, relationsdatabaser, arraydatabaser, etc.) utnyttjas för skalbar representation av *RDF with Arrays*?
 - c. Hur kan SciSPARQL-frågor integreras i existerande omgivningar och arbetsflöden för vetenskaplig och teknisk dataanalys?
 - d. Hur mäter vi effekten av olika designbeslut när det gäller hur *RDF with Arrays* data skall lagras effektivt och hur frågor i SciSPARQL över lagrade data skall besvaras så snabbt som möjligt?

Acknowledgement

First and foremost I would like to thank my supervisor Tore Risch for sharing his knowledge and enthusiasm, and my co-supervisors Kjell Orsborn and Ruslan Fomkin for the fruitful discussions and the inspiration for improvements.

I would also like to thank my former and current colleagues Cheng, Erik, Lars, Khalid, Minpeng, Sabesan, Silvia, Sobhan, and Thanh for their support, and our partners Peter Baumann and Dimitar Misev from Jacobs University Bremen, Andreas Hellander and Brian Drawert from University of California Santa Barbara, for the joyful collaboration and shared achievements.

Finally, I would like to thank my friends and family for their great patience and for always being there to support me.

This project is supported by eSENCE and the Swedish Foundation for Strategic Research, grant RIT08-0041.

References

- [1] M. Acosta, M-E. Vidal, T. Lamp, J. Castillo, and E. Rickhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. *Proc. 10th International Semantic Web Conference (ISWC'11)*, Bonn, Germany, October 2011
- [2] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki: NoDB: Efficient Query Execution on Raw Data Files. *Proc. 2012 ACM SIGMOD/PODS Conference*, Scottsdale AZ, USA, June 2012
- [3] M. I. Ali, N. Lopes, O. Friel, and A. Mileo: Update Semantics for Interoperability among XML, RDF and RDB. *Proc. 15th Asia-Pacific Web Conference (APWeb 2011)*, Sydney, Australia, April 2013
- [4] AllegroGraph. <http://franz.com/agraph/allegrograph/>
- [5] A. Andrejev and T. Risch. Scientific SPARQL: Semantic web queries over scientific data. *Proc. Third International Workshop on Data Engineering Meets the Semantic Web (DESWEB)*, Washington DC, USA, April 2012
- [6] A. Andrejev, S. Toor, A. Hellander, S. Holmgren, and T. Risch. Scientific Analysis by Queries in Extended SPARQL over a Scalable e-Science Data Store. *Proc 9th IEEE International Conference on e-Science*, Beijing, China, October 2013
- [7] A. Andrejev, X. He, T. Risch. Scientific data as RDF with Arrays: Tight integration of SciSPARQL queries into Matlab. *Proc. 13th International Semantic Web Conference (ISWC'14)*, Riva del Garda, Italy, October 2014
- [8] A. Andrejev, D. Misev, P. Baumann, and T. Risch. Spatio-Temporal Gridded Data Processing on the Semantic Web. *Proc. IEEE International Conference on Data Science and Data-Intensive Systems (DSDIS)*, Sydney, Australia, December 2015
- [9] M. Arenas, A. Bertails, E. Prud'hommeaux, J. Sequeda. A Direct Mapping of Relational Data to RDF. 2012, <http://www.w3.org/TR/rdb-direct-mapping/>
- [10] M.van Assem, A.Gangemi, and G.Schreiber. RDF/OWL Representation of WordNet. *W3C Working Draft* 19 June 2006. <https://www.w3.org/TR/wordnet-rdf/>
- [11] S.Auer, C.Bizer, G.Kobilarov, J.Lehmann, R.Cygniak, and Z.Ives. Dbpedia: A nucleus for a web of open data. *The Semantic Web*, pp.722-735, Springer, 2007
- [12] A. R. van Ballegooij. RAM: A Multidimensional Array DBMS. *Proc. 9th International Conference on Extending Database Technology*, Heraklion, Greece, March 2004
- [13] A. van Ballegooij and R. Cornacchia. Distribution Rules for Array Database Queries. *Proc. 16th International Conference on Database and Expert Systems Applications (DEXA)*, Copenhagen, Denmark, August, 2005

- [14] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. Querying RDF Streams with C-SPARQL. *Proc. 2010 ACM SIGMOD/PODS Conference*: Indianapolis IN, USA, June 2010
- [15] R. Battle and D. Kolas. GeoSPARQL: Enabling a Geospatial Semantic Web. *Semantic Web Journal* 3(4) pp. 355-370, 2011
- [16] P. Baumann. On the Management of Multidimensional Discrete Data. *VLDB Journal* 4 (3), *Special Issue on Spatial Database Systems*, pp. 401-444, 1994
- [17] P. Baumann: A Database Array Algebra for Spation-Temporal Data and Beyond. *Proc. 4th International Workshop on Next Generation Information Technologies and Systems*, Zikhron-Yaakov, Israel, July 1999
- [18] P. Baumann, S. Holsten: A Comparative Analysis of Array Models for Databases. *T.-h. Kim, H. Adeli, A. Cuzzocrea, T. Arslan, Y. Zhang, J. Ma, K.-i. Chung, S. Mariyam, and X. Song (eds.): Database Theory and Application, Bio-Science and Bio-Technology, volume 258 of Communications in Computer and Information Science*, pp. 80-89, 2011
- [19] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. OWL Web Ontology Language Reference. 2004, <http://www.w3.org/TR/owl-ref/>
- [20] D. Beckett. A line-based syntax for an RDF graph. <https://www.w3.org/TR/n-triples/>
- [21] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. Terse RDF Triple Language. <https://www.w3.org/TR/turtle/>
- [22] G. Bell, T. Hey, and A. Szalay. Beyond the Data Deluge, *Science*, 323 pp. 1297-1298, March 2009
- [23] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284 (5) pp. 34-43, May 2001
- [24] T. Berners-Lee and D. Connolly. Notation3 (N3): A readable RDF syntax. <https://www.w3.org/TeamSubmission/n3/>
- [25] N. Bikakis, C. Tsinaraki, N. Gioldasis, I. Stavrakantonakis, and S. Christodoulakis. *The XML and Semantic Web Worlds: Technologies, Interoperability and Integration. A survey of the State of the Art. Semantic Hyper/Multi-media Adaptation: Schemes and Applications*, Springer, 2013
- [26] N. Bikakis, C. Tsinaraki, I. Stavrakantonakis, N. Gioldasis, and S. Christodoulakis. *The SPARQL2XQuery Interoperability Framework. World Wide Web Journal* 18 (2) pp. 403-490, Springer, 2014
- [27] N. Bikakis, C. Tsinaraki, I. Stavrakantonakis, and S. Christodoulakis. Supporting SPARQL Update Queries in RDF-XML Integration. *Proc. 13th International Semantic Web Conference (ISWC'14)*, Riva del Garda, Italy, October 2014
- [28] S. Bischof, S. Decker, T. Krennwallner, N. Lopes, and A. Polleres. Mapping between RDF and XML with XSPARQL. *Journal on Data Semantics* 1 (3) pp. 147-185, Springer-Verlag, 2012
- [29] C. Bizer, T. Heath, T. Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, 5 (3) pp. 1-22, 2009

- [30] A. Björkelund., L. Edström, M. Haage, J. Malec, K. Nilsson, P. Nugues., S. Gestegård Robertz, D. Störkle, A. Blomdell, R. Johansson, M. Linderöth, A. Nilsson, A. Robertsson, A. Stolt, H. Bruyninckx. On the integration of skilled robot motions for productivity in manufacturing, *Proc. IEEE International Symposium on Assembly and Manufacturing*, Tampere, Finland, May 2011.
- [31] E. Blomqvist. Ontology Patterns - Typology and Experiences from Design Pattern Development. *Proc. SAIS Workshop 2010*, Uppsala, May 2010
- [32] D. Brickley and L. Miller. FOAF Vocabulary Specification 0.99. <http://xmlns.com/foaf/spec/>
- [33] D. Brickley and R. V. Guha. RDF Schema. 2014, <http://www.w3.org/TR/rdf-schema/>
- [34] J. Broekstra and A. Kampman. SeRQL: A Second Generation RDF Query Language. *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, Amsterdam, Netherlands, November 2003
- [35] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. *Proc. 2010 ACM SIGMOD/PODS Conference*, Indianapolis IN, USA, June 2010
- [36] 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) pp. 81–86, April 1997
- [37] J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. *Proc. 13th International Conference on World Wide Web*, New York NY, USA, May 2004
- [38] S. Chaudhuri and K. Shim: Optimizing Queries with Aggregate Views. *Proc. 5th International Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996
- [39] S. Chaudhuri: An Overview of Query Optimization in Relational Systems. *Proc. 18th ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems (PODS'98)*, Seattle WA, USA, June 1998
- [40] A. Chebotko, S. Li, and F. Fotouhi. Semantics Preserving SPARQL-to-SQL Translation. *Data & Knowledge Engineering Journal*, 68 (10) pp. 973-1000, October 2009
- [41] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. *Proc. 35th International Conference on Very Large Data Bases (VLDB'09)*, Lyon, France, August 2009
- [42] R. Cornacchia, A. van Ballegooij, and A. P. de Vries. A Case Study on Array Query Optimization. *Proc. 1st International Workshop on Computer Vision meets Databases (CVDB'04)*, Paris, France, June 2004
- [43] CouchBase. <http://www.couchbase.com/>
- [44] P. Cudré-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of SciDB: a science-oriented DBMS. *Proc. VLDB Endowment* 2 (2) pp. 1534-1537, 2009

- [45] P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. Sequeda, and M. Wylot. NoSQL Databases for RDF: An Empirical Evaluation. *Proc. 12th International Semantic Web Conference (ISWC'13)*, Sydney, Australia, October 2013
- [46] R. Cyganiak. A Relational Algebra for SPARQL. *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, 2005
- [47] R. Cyganiak, C. Bizer, J. Garbers, O. Maresch, C. Becker. The D2RQ Mapping Language. 2012, <http://www4.wiwiw.fu-berlin.de/bizer/d2rq/spec/>
- [48] B. Drawert, S. Engblom, and A. Hellander. URDM 1.1: User's manual. *Technical Report 003, Department of Information Technology, Division of Scientific Computing, Uppsala University*, 2010
- [49] L. Dobos, A. Szalay, J. Blakeley, T. Budavári, I. Csabai, D. Tomic, M. Milovanovic, M. Tintor, and A. Jovanovic. Array Requirements for Scientific Applications and an Implementation for Microsoft SQL Server. *Proc. EDBT/ICDT Workshop on Array Databases, Uppsala, Sweden*, March 2011
- [50] V. Dritsou, P. Constantopoulos, A. Deligiannakis, and Y. Kotidis. Optimizing Query Shortcuts in RDF Databases. *The Semantic Web: Research and Applications Volume 6644 pp 77-92*, Springer, 2011
- [51] Dublin Core Metadata Initiative. <http://dublincore.org/>
- [52] J. Elf and M. Ehrenberg. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Systems Biology*, 1 (2) pp. 230-236, IET, 2004
- [53] S. Engblom, L. Ferm, A. Hellander, and P. Lotstedt. Simulation of stochastic reaction-diffusion processes on unstructured meshes. *SIAM Journal on Scientific Computing* 31 (3) pp. 1774-1797, 2009
- [54] O. Erling. Declaring RDF views of SQL Data. *Proc. W3C Workshop on RDF Access to Relational Databases*, Cambridge, MA, USA, October 2007
- [55] O. Erling and I. Mikhailov: RDF Support in the Virtuoso DBMS. *Studies in Computational Intelligence*, 221 pp. 7-24, Springer, 2009
- [56] D. Fange and J. Elf. Noise induced Min phenotypes in E. coli. *PLoS Computational Biology*, 2 (6) p. e80, 2006
- [57] S. Flodin and T. Risch. Processing Object-Oriented Queries with Invertible Late Bound Functions. *Proc. 21st International Conference on Very Large Data Bases (VLDB'95)*, Zurich, Switzerland, September 1995
- [58] S. Flodin, K. Orsborn, and T. Risch: Using Queries with Multi-Directional Functions for Numerical Database Applications. *Proc. 2nd East-European Symposium on Advances in Databases and Information Systems (ADBIS'98)*, Poznan, Poland, September 1998
- [59] R. Fomkin. Optimization and Execution of Complex Scientific Queries. *Uppsala Dissertations from the Faculty of Science and Technology, No. 80, ISBN 978-91-554-7382-2 Acta Universitatis Upsaliensis*, 2009
- [60] P. Furtado and P. Baumann. Storage of Multidimensional Arrays Based on Arbitrary Tiling. *Proc. 15th IEEE International Conference on Data Engineering (ICDE'99)*, Sydney, Australia, March 1999
- [61] L. Galarraga, K. Hose, and R. Schenkel. Partout: A Distributed Engine of Efficient RDF Processing. *Proc. 23rd International Conference on World Wide Web*, Seoul, Korea, April 2014

- [62] J. H. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S. W. Tu. The evolution of Protégé: an environment for knowledge-based systems development. *Human-Computer Studies* 58 (2003) pp. 89–123, 2003
- [63] D. T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reacting systems. *Journal of computational physics*, 22(4) pp. 403-434, Elsevier, 1976
- [64] C. Glitia, P. Dumont, and P. Boulet. Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*. 21 (2) pp. 105-131, Springer, June 2010
- [65] F. Goasdoue, K. Karanasos, J. Leblay, and I. Manolescu. View Selection in Semantic Web Databases. *Proc. 38th International Conference on Very Large Data Bases (VLDB'12)*, Istanbul, Turkey, August 2012
- [66] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab and Sub-Totals. *Proc. 12th IEEE International Conference on Data Engineering (ICDE'96)*, New Orleans, LA, February 1996
- [67] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, G. Heber, and D. DeWitt. Scientific Data Management in the Coming Decade. *ACM SIGMOD Record*, 34 (4), 2005
- [68] R. V. Guha: Light at the End of the Tunnel. *ISWC 2013 Keynote. 12th International Semantic Web Conference*, Sydney, Australia, October 2013
- [69] W. R. van Hage, M. van Erp, and V. Malaisé. Linked Open Piracy: A Story about e-Science, Linked Data, and Statistics. *Journal on Data Semantics*, 1 (3) pp 187-201, Springer, September 2012
- [70] H. Halpin and P. J. Hayes. When owl: sameAs isn't the Same: An Analysis of Identity Links on the Semantic Web. *Proc Linked Data on the Web (LDOW)*, Raleigh NC, USA, April 2010
- [71] L. Han, T. Finin, C. Parr, J. Sachs, and A. Joshi. RDF123: from Spreadsheets to RDF. *Proc. 7th International Semantic Web Conference (ISWC'08)*, Karlsruhe, Germany, October 2008
- [72] M. Hansson. Wrapping External Data by Query Transformations. *Uppsala Master's Theses in Computing Science No. 243, ISSN 1100-1836*, July 2003
- [73] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K-U. Sattler, and J. Umbrich: Data Summaries for On-Demand Queries over Linked Data. *Proc. 19th International Conference on World Wide Web*, Raleigh NC, USA, April 2010
- [74] Apache HBase. <http://hbase.apache.org/>
- [75] J. M. Hellerstein. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *ACM SIGMOD Record* 39 (1), March 2010
- [76] T. Hey, S. Tansley, and K. Tolle (eds): *The Fourth Paradigm: Data-Intensive Scientific Discovery*. ISBN 978-0-9825442-0-4, Microsoft Research, 2009
- [77] F. Holzschuher and R. Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. *Proc. Joint EDBT/ICDT Workshops*, Genoa, Italy, 2013
- [78] I. Horrocks and P. F. Patel-Schneider. A Proposal for an OWL Rules Language. *Proc. 13th International Conference on World Wide Web*, New York NY, USA, May 2004

- [79] B. Howe, K. Tanna, P. Turner, and D. Maier. Emergent Semantics: Towards Self-Organizing Scientific Metadata. *Proc. International Conference on Semantics of a Networked World*, Paris, France, June 2004
- [80] Y. E. Ioannidis, and Y. C. Kang, Y. C. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. *ACM SIGMOD Record*, 20 (2) pp. 168-177, 1991
- [81] A. Isaac, S. Schenk, and A. Scherp: Semantic Web Languages. R. Troncy, B. Huet, S. Schenk (eds.): *Multimedia Semantics: Metadata, Analysis and Interaction*, Wiley, 2011
- [82] K. E. Iverson. A programming language. *Proc. AIEE-IRE '62 (Spring)* pp. 345-351, 1962
- [83] A. Jacobs. The Pathologies of Big Data. *ACMQueue*, 7 (6), July 2009
- [84] Apache Jena. <http://jena.apache.org/>
- [85] V. Josifovski. Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration. *Linköping University Dissertation No 582*, 1999
- [86] V. Josifovski, T. Risch. Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations. *Proc. 25th International Conference on Very Large Data Bases (VLDB'99)*, Edinburgh, UK, September 1999
- [87] M. Kamdar, D. Zeginis, A. Hasnain, S. Decker, and H. Deus. ReVealD: A User-Driven Domain-Specific Interactive Search Platform for Biomedical Research. *Journal of Biomedical Informatics* 47 (1) pp. 112-130, Elsevier, 2014
- [88] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, and M. Magiridou, A. Papadakis-Pesaresi. Atlas: Storing, Updating and Querying RDF(S) Data on top of DHTs. *Web Semantics: Science, Services and Agents on the World Wide Web* 8 (4) pp. 271-277, Elsevier, 2010
- [89] V. Karuaskas and M. Sileikis. Wrapping Persistent ROOT Framework Objects in an Object-Oriented Mediator System. *Uppsala Master's Theses in Computing Science* 304, ISSN 1100-1836, 2006
- [90] G. Karvounakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. *Proc. 11th international conference on World Wide Web*, Honolulu HI, USA, May 2002
- [91] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes: SciQL, a query language for science applications. *Proc. EDBT/ICDT Workshop on Array Databases*, Uppsala, Sweden, March 2011
- [92] M. Kifer. Rule Interchange Format: The Framework. *Proc. Web Reasoning and Rule Systems. Lecture Notes in Computer Science*, Springer, 2008
- [93] E. Kostylev, J. Reutter, M. Romero, and D. Vrgoč. SPARQL with Property Paths. *Proc. 14th International Semantic Web Conference (ISWC'15)*, Bethlehem PA, USA, October 2015
- [94] S. Kotoulas and J. Urbani. SPARQL Query Answering on a Shared-nothing Architecture. *Proc. VLDB Workshop on Semantic Data Management (SemData)*, Singapore, September, 2010
- [95] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM* 34 (10), pp. 50-63, October 1991

- [96] A. Langegger and W. Wöß. XLWrap - Querying and Integrating Arbitrary Spreadsheets with SPARQL. *Proc. 8th International Semantic Web Conference (ISWC'09)*, Chantilly VA, USA, October 2009
- [97] G. Lausen, M. Meier, and M. Schmidt. SPARQLing Constraints for RDF. *Proc. 11th International Conference on Extending Database Technology (EDBT'08)*, Nantes, France, March 2008
- [98] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable Multi-Query Optimization for SPARQL. *Proc. IEEE International Conference on Data Engineering (ICDE'12)*, Arlington VA, USA, April 2012
- [99] L. Libkin, R. Machlin, and L. Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996
- [100] W. Litwin, and T. Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4 (6) pp. 517-528, 1992
- [101] K. Mahmood, T. Risch, and M. Zhu. Utilizing a NoSQL Data Store for Scalable Log Analysis, *Proc. 19th International Database Engineering & Applications Symposium*, Yokohama, Japan, July 2015
- [102] D. Maier and B. Vance. A Call to Order. *Proc. 12th ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems (PODS'93)*, Washington DC, USA, May 1993
- [103] G. M. Manipon, B. D. Wilson, and H. Hua. Publishing NASA Metadata as Linked Open Data for Semantic Mashups. *Proc. American Geophysical Union, Fall Meeting*, San-Francisco CA, USA, December 2013
- [104] A. Marathe and K. Salem. Query processing techniques for arrays. *The International Journal on Very Large Data Bases 11 (1)* pp. 68-91, August 2002
- [105] P. Marques, P. Furtado, and P. Baumann. An Efficient Strategy for Tiling Multidimensional OLAP Data Cubes. *Proc. Workshop on Data Mining and Data Warehousing (Informatik'98)*, Magdeburg, Germany, September 1998
- [106] Mimer SQL. <http://www.mimer.com>
- [107] D. Misev and P. Baumann. Extending the SQL Array Concept to Support Scientific Analytics. *Proc. 26th International Conference on Scientific and Statistical Database Management (SSDBM)*, Aalborg, Denmark, June 2014
- [108] D. Misev and P. Baumann. Homogenizing Data and Metadata Retrieval in Scientific Applications. *Proc. 18th International Workshop On Data Warehousing and OLAP*. Melbourne, Australia, October 2015
- [109] Community Cleverness Required. *Nature, editorial*, 455 (7209) p. 1, 2008
- [110] W. Neidl, B. Wolf, C. Qu, S. Decker, M. Sinek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. EDUTELLA: A P2P Networking Infrastructure Based on RDF. *Proc. 11th international conference on World Wide Web*, Honolulu HI, USA, May 2002
- [111] NetCDF. <http://www.unidata.ucar.edu/software/netcdf/>
- [112] T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. *Proc. 34th International Conference on Very Large Data Bases (VLDB'08)*, Auckland, New Zealand, August 2008

- [113] T. Neumann and G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. *Proc. IEEE International Conference on Data Engineering (ICDE'11)*, Hannover, Germany, April 2011
- [114] J. K. Nilsen, S. Toor, Zs. Nagy, and A. Read. Chelonia: A self-healing, replicated storage system. *Journal of Physics: Conference Series*, 331 (6), 2011
- [115] NitroBase. <http://nitrobase.com/>
- [116] NorduGrid Collaboration. <http://www.nordugrid.org/>
- [117] N. F. Noy and D. L. McGuinness. *Ontology Development 101: A Guide to Creating Your First Ontology*. Knowledge Systems Laboratory, March, 2001
- [118] P. de Oliveira Castro, S. Louise, and D. Barthou. A Multidimensional Array Slicing DSL for Stream Programming. *Proc. 4th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'10)*, Krakow, Poland, February 2010
- [119] C. Ordonez and J. Garcia-Garcia. Vector and matrix operations programmed with UDFs in a relational DBMS. *Proc. Conference on Information and Knowledge Management (CIKM'06)*, Arlington, VA, USA, November 2006
- [120] K. Orsborn, T. Risch, and S. Flodin: Representing Matrices Using Multi-Directional Foreign Functions. *P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, ISBN 3-540-00375-4* Springer, 2004
- [121] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *Proc. 5th International Semantic Web Conference (ISWC'06)*, Athens, GA, USA, November 2006
- [122] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34 (3) pp. 16:1-16:45, 2009
- [123] J. Petrini. Querying RDF Schema Views of Relational Databases, *Uppsala Dissertations from the Faculty of Science and Technology, No. 75, ISBN 978-91-554-7202-3 Acta Universitatis Upsaliensis*, 2008
- [124] J. Petrini and T.Risch. Processing queries over RDF views of wrapped relational databases. *Proc. 1st International Workshop on Wrapper Techniques for Legacy Systems (WRAP)*, Delft, Netherlands, November 2004
- [125] PostgreSQL. <http://www.postgresql.org/>
- [126] F. Prasser, A. Kemper, and K. A. Kuhn. Efficient Distributed Query Processing for Autonomous RDF Databases. *Proc. 15th International Conference on Extending Database Technology (EDBT'12)*, Berlin, Germany, March 2012
- [127] R2RML: RDB to RDF Mapping Language. <http://www.w3.org/TR/r2rml/>
- [128] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *The journal of logic programming* 23 (2) pp. 125-149, Elsevier, 1995
- [129] Resource Description Framework (RDF). <https://www.w3.org/RDF/>
- [130] RDF 1.1 XML Syntax. <https://www.w3.org/TR/rdf-syntax-grammar/>
- [131] RDFa Core 1.1 - Third Edition Syntax and processing rules for embedding RDF through attributes. <https://www.w3.org/TR/rdfa-syntax/>
- [132] RDFBeans Framework. <http://rdfbeans.sourceforge.net/>
- [133] RDF Data Cube. <http://www.w3.org/TR/vocab-data-cube/>

- [134] B. R. K. Reddy and P. S. Kumar. Optimizing SPARQL queries over the Web of Linked Data. *Proc. VLDB Workshop on Semantic Data Management (SemData)*, Singapore, September, 2010
- [135] T. Risch and V. Josifovski. Distributed Data Integration by Object-Oriented Mediator Servers. *Concurrency and Computation: Practice and Experience*, 13 (11), John Wiley & Sons, September 2001
- [136] T. Risch, V. Josifovski, and T. Katchaounov: Functional Data Integration in a Distributed Mediator System. *P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, ISBN 3-540-00375-4* Springer, 2004.
- [137] M. Rodriguez-Muro, M. Rezk, J. Hardi, M. Slusnys, T. Bagosi, and D. Calvanese: Evaluating SPARQL-to-SQL Translation in Ontop, *Proc. Owl Reasoner Evaluation Workshop (ORE 2013)*, Vienna, Austria, July 2013
- [138] R-SPARQL. <http://cran.r-project.org/web/packages/SPARQL/index.html>
- [139] S.Sakr and G. Al-Nayat. Relational Processing of RDF Queries: A Survey. *Proc. ACM SIGMOD/PODS Conference*, Providence RI, USA, June 2009
- [140] M. Saleem, S. Padmanabhuni, A. Ngomo, J. Almeida, and S. Decker. Linked Cancer Genome Atlas Database. *Proc. 9th International Conference on Semantic Systems*, Graz, Austria, September 2013
- [141] SAP HANA. <http://hana.sap.com>
- [142] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. *Proc. 10th IEEE International Conference on Data Engineering (ICDE'94)*, Houston TX, USA, February 1994
- [143] S-B. Scholz. Single Assignment C - efficient support for high-level array operations in a functional setting. *Journal of Functional Programming* 13 (6), November 2003
- [144] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. *Proc. 10th International Semantic Web Conference (ISWC'11)*, Bonn, Germany, October 2011
- [145] M. Sköld and T. Risch. Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions. *Proc. 12th IEEE International Conference on Data Engineering (ICDE'96)*, New Orleans LA, USA, February 1996
- [146] Scientific SPARQL. <http://www.it.uu.se/research/group/udbl/SciSPARQL/>
- [147] Statistical Data and Metadata eXchange. <http://sdmx.org/>
- [148] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database System. *Readings in Database Systems*. Morgan Kaufman, 1979
- [149] J. Sequeda and D. Miranker. Ultrawrap: SPARQL Execution on Relational Data, *Tech. Report, Univ. of Texas at Austin*, 2013. http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2078.pdf
- [150] M. Shaw, L. T. Detwiler, N. Noy, J. Brinkley, and D. Suci. vSPARQL: A View Definition Language for the Semantic Web. *Journal of Biomedical Informatics* 44 (1) pp. 102-117, Elsevier, 2010
- [151] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6 (1) pp. 140-173, 1981

- [152] M. Sintek and S. Decker. TRIPLE - An RDF query, inference, and transformation language. *Proc. Deductive Databases and Knowledge Management (DDL'2001)*, Tokyo, Japan, October 2001
- [153] SKOS Vocabulary. <http://www.w3.org/2004/02/skos/>
- [154] E.Soroush, M.Balazinska, and D.L.Wang. Arraystore: a storage manager for complex parallel array processing. *Proc. ACM SIGMOD/PODS Conference*. Athens, Greece, June 2011
- [155] SPARQL 1.1 Query Language. <http://www.w3.org/TR/sparql11-query/>
- [156] SPARQL 1.1 Update. <https://www.w3.org/TR/sparql11-update/>
- [157] Starcounter. <http://starcounter.com/>
- [158] Stardog. <http://stardog.com/>
- [159] S. Stefanova, and T. Risch. Optimizing Unbound-property Queries to RDF Views of Relational Databases. *Proc. 7th International workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*, Bonn, Germany, October 2011
- [160] S. Stefanova T. Risch: Scalable Recreation of RDF-Archived Relational Databases, *Proc. 5th International Workshop on Semantic Web Information Management (SWIM 2013)*, New York NY, USA, June 2013
- [161] S. Stefanova and T. Risch. Scalable Long-term Preservation of Relational Data through SPARQL queries. *Semantic Web Journal*, 2015
- [162] M. Stonebraker, J. Becla, D. J. DeWitt, K-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and SciDB. *Proc. Conference on Innovative Data Systems Research (CIDR'09)*, Pacific Grove CA, USA, January 2009
- [163] A. S. Szalay and J. Gray. 2020 Computing: Science in an Exponential World. *Nature*, 440 (7083) pp. 413–414, 2006
- [164] A. R. Thakar, A. S. Szalay, P. Z. Kunszt, and J. Gray. The Sloan Digital Sky Survey Science Archive: Migrating a Multi-Terabyte Astronomical Archive from Object to Relational DBMS. *Computer Science and Engineering*, 5 (5), pp. 16–29, September 2003.
- [165] A. R. Thakar. The Sloan digital sky survey: Drinking from the fire hose. *Computing in Science & Engineering* 10 (1) pp. 9–12, 2008
- [166] S. Toor, M. Sabesan, S. Holmgren, and T. Risch, A Scalable Architecture of Distributed Storage by Employing Databases for e-Science Applications. *Proc. 12th International Semantic Web Conference (ISWC'13)*, Sydney, Australia, October 2013
- [167] T. Truong and T. Risch. Transparent inclusion, utilization, and validation of main memory domain indexes. *Proc. 27th International Conference on Scientific and Statistical Database Management*, San Diego CA, USA, June 2015
- [168] P. Tsialiamanis, L. Sigiougros, I. Fundulaki, V. Christophides, and P. Boncz. Heuristic-based Query Optimization for SPARQL. *Proc. 15th International Conference on Extending Database Technology (EDBT'12)*, Berlin, Germany, March 2012
- [169] J. D. Ullman. Principles of Database and Knowledge Base Systems, vol. I,II. C.S. Press, 1989
- [170] The Uniprot Consortium: Ongoing and future developments at the Universal Protein Resource. *Nucleic Acids Research*, 39 (Database issue) pp.214-219, 2010

- [171] S. van der Walt, S. C. Colbert, and G. Vaouquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science and Engineering*, 13 (2), March 2011
- [172] Y. Wang, A. Nandi, and G. Agrawal. SAGA: Array Storage as a DB with Support for Structural Aggregations. *Proc. 26th International Conference on Scientific and Statistical Database Management (SSDBM)*, Aalborg, Denmark, June 2014
- [173] J. Webber. A programmatic introduction to Neo4j. *Proc. 13rd ACM Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH'12)*, Tucson AZ, USA, October 2012
- [174] Versa. <http://xml3k.org/Versa>
- [175] VoiD Vocabulary. <http://www.w3.org/TR/void/>
- [176] XSPARQL Language Specification. <http://www.w3.org/Submission/xsparql-language-specification/>
- [177] C. Xu, D. Wedlund, M. Helguson, and T. Risch. Model-based Validation of Streaming Data. *Proc. 7th ACM International Conference on Distributed Event-Based Systems (DEBS)*, Arlington TX, USA, June 2013.
- [178] E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. *Proc. VLDB Endowment*, 4(11), pp. 1181-1188, 2011
- [179] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-Efficient Numerical Computing without SQL. *Proc. Conference on Innovative Data Systems Research (CIDR'09)*, Pacific Grove CA, USA, January 2009
- [180] Y. Zhang, K. Munagala, and J. Yang. Storing Matrices on Disk: Theory and Practice Revisited. *Proc. VLDB Endowment*, 4 (11) pp. 1075-1086, 2011
- [181] M. Zhu and T. Risch. Querying Combined Cloud-Based and Relational Databases, *Proc. 2011 International Workshop on Data Cloud (D-CLOUD)*, Hong Kong, December 2011
- [182] M. Zhu, S. Stefanova, T. Truong, and T. Risch. Scalable Numerical SPARQL Queries over Relational Databases. *Proc. 4th International Workshop on Linked Web Data Management (LWDM)*, Athens, Greece, March 2014
- [183] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL Queries via Subgraph Matching. *Proc. 37th International Conference on Very Large Data Bases (VLDB'11)*, Seattle WA, USA, August 2011
- [184] P-O Östberg, A. Hellander, B. Drawert, E. Elmroth, S. Holmgren, and L. Petzold. Reducing Complexity in Management of eScience Computations. *Proc. 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Ottawa, Canada, May 2012

Glossary

array - same as *NMA*

array descriptor - a data structure that allows to address the certain subset of array elements, as a result of *slicing*, *projection* and *generalized transposition* operations. Operations involving different dimensions can be applied in independent order. One or more *descriptors* may refer to the same *storage object*. Different *descriptors* identify different arrays. A *descriptor* is also part of any *array proxy*

array fragment - a tuple of *storage index* and *size* that can be used to access certain physically contiguous sequence of array elements in a *storage object*, or used to address certain *chunks* representing that *storage object*.

array proxy - an object that allows to address a (subset of) *array* stored in a particular storage system. Contains *kind*, associated with certain storage system and access routines, storage-specific array identifier (e.g. file and variable names for *.mat file proxies*), element type and *array descriptor* data. Can additionally contain information about array partitioning.

array subscript - see *logical index*

back-end - see *storage system*

binding pattern - a certain way to evaluate a *predicate* in the *execution plan*, with certain variable *bound*, and possibly certain variables *free*. The evaluation will result in zero or more results, with values provided for the free variables.

chunk - a binary object representing a part of the *storage object* of an array. Chunks of the same array have sequential *chunkid* identifiers. See also *partitioning scheme*.

derived array - an array resulting from *slicing*, *projection*, or *generalized transposition* operation of another array. *Derived arrays* are typically represented by non-original *descriptor object* and a storage object that is also referred to by other *descriptor(s)*. However, that might not always be the case, as other *descriptors* might already be garbage-collected. Also e.g. two mutually reversing *transposition* operations on an *original descriptor* will result in the same *descriptor*.

descriptor, descriptor object - same as *array descriptor*

execution plan - a final stage of query optimization, specifying the exact operations to be performed in order to answer a query. Consists of predicates with binding patterns assigned, combined with *join*, *union*, and other specialized operators.

extended Turtle (file) - a file in *Turtle* format containing URIs that are interpreted as *file links* by SSDM, and as normal URIs by the standard *Turtle* readers.

file link - a URI in a turtle file that is interpreted by SSDM data loader as an array value. An *array proxy* is created based on information contained in *file link* (and the information from *linked file* it refers to), which can be resolved immediately, or later on demand.

generalized transposition - an operation that results in a *derived array* of same size and number of dimensions but of different shape. Involves specifying the new logical order of dimensions. A simple matrix *transposition* involves swapping the two dimensions, so that their new order is always $(1, 0)$.

linked file - see *file link*

logical index - a vector of integer *array subscripts*, one for each *array dimension*, that identifies an element in an array. Given *array descriptor*, a *logical index* can be translated into a *storage index* (Section 5.2.1). *Array subscripts* are 1-based by default, but an alternative SciSPARQL dialect that supports Python notation for array operations uses 0-based *array subscripts*.

memory-resident array - see *resident array*

NMA - *Numeric Multidimensional Array*, one of the extensions introduced in this work to the basic *RDF* model. NMAs can contain arbitrary number of dimensions, and always have rectangular shapes. The supported element types are: *Boolean*, *Integer*, *Double*, and *Complex*.

ObjectLog - a dialect of Datalog used in SSDM to internally represent SciSPARQL queries

original descriptor / proxy - an *array descriptor* or *array proxy* that refers to the complete array in the corresponding *storage object* so that a single *array fragment* can be used to access the entire *storage object*. An *original descriptor* object is always created with a new *storage object*. The *array proxies* are created with *original descriptor* data.

partitioning scheme (of array, either *linear* or *multidimensional*) - a way to split the array contents into *chunks*: either *linear chunks* or *multidimensional tiles*, defined by the corresponding *chunk* or *tile* size.

predicate (in Datalog / ObjectLog) - a constituent part of a query or expression, can be put into the execution plan and evaluated with a certain binding pattern. *Predicates* can be *stored* (corresponding to tables in-memory or mapped), or *foreign* (corresponding to a certain computable function, possibly multidirectional - i.e. one with different *binding patterns*).

projection (of array) - an operation that produces a *derived array* with lesser number of dimensions. Involves specifying a single *subscript* for a certain dimension(s).

range selection (of array) - an operation that produces a smaller *derived array* with the same dimensionality. Involves specifying explicit or implicit *ranges* for all array dimensions.

RDF view - a mapping defined from non-RDF data model to RDF, allowing to query (and, possibly, update) the underlying data with SPARQL.

RDF with Arrays - a data model combining RDF graph and numeric multidimensional arrays as possible *values*.

resident array - an array with contents stored in main memory

storage index - an integer value addressing a particular element (or beginning of *array fragment*) in a *storage object*, either existing in memory or represented by *chunks*. Storage indexes are always 0-based.

storage object - a main-memory object that physically contains the elements of a *resident array*. Can be serialized to binary *chunks*. The element type is also stored here with *storage object*, to avoid redundancy.

storage system - a software system interfaced with SSDM that provides persistent storage for *RDF with Arrays* data - either completely (like relational database back-ends) or partially (arrays-only) (as *.mat* files on the server file system).

(sub)array proxy - an *array proxy* pointing to an array of one or more dimensions, in contrast to a *single-element proxy*, pointing to a particular element in an externally stored array.

subscript - see *logical index*

tile - a multidimensional array *chunk*, specified by its *size* in the logical dimensions of the corresponding array, see also *partitioning scheme*.

triple (of RDF graph) - a (*subject, property, value*) tuple, constituent part of an RDF graph.

Acta Universitatis Upsaliensis

Uppsala Dissertations from the Faculty of Science

Editor: The Dean of the Faculty of Science

1–11: 1970–1975

12. *Lars Thofelt*: Studies on leaf temperature recorded by direct measurement and by thermography. 1975.
13. *Monica Henricsson*: Nutritional studies on *Chara globularis* Thuill., *Chara zeylanica* Willd., and *Chara haitensis* Turpin. 1976.
14. *Göran Kloow*: Studies on Regenerated Cellulose by the Fluorescence Depolarization Technique. 1976.
15. *Carl-Magnus Backman*: A High Pressure Study of the Photolytic Decomposition of Azoethane and Propionyl Peroxide. 1976.
16. *Lennart Källströmer*: The significance of biotin and certain monosaccharides for the growth of *Aspergillus niger* on rhamnose medium at elevated temperature. 1977.
17. *Staffan Renlund*: Identification of Oxytocin and Vasopressin in the Bovine Adenohypophysis. 1978.
18. *Bengt Finnström*: Effects of pH, Ionic Strength and Light Intensity on the Flash Photolysis of L-tryptophan. 1978.
19. *Thomas C. Amur*: Diffusion in Dilute Solutions: An Experimental Study with Special Reference to the Effect of Size and Shape of Solute and Solvent Molecules. 1978.
20. *Lars Tegnér*: A Flash Photolysis Study of the Thermal Cis-Trans Isomerization of Some Aromatic Schiff Bases in Solution. 1979.
21. *Stig Tormod*: A High-Speed Stopped Flow Laser Light Scattering Apparatus and its Application in a Study of Conformational Changes in Bovine Serum Albumin. 1985.
22. *Björn Varnestig*: Coulomb Excitation of Rotational Nuclei. 1987.
23. *Frans Lettenström*: A study of nuclear effects in deep inelastic muon scattering. 1988.
24. *Göran Ericsson*: Production of Heavy Hypernuclei in Antiproton Annihilation. Study of their decay in the fission channel. 1988.
25. *Fang Peng*: The Geopotential: Modelling Techniques and Physical Implications with Case Studies in the South and East China Sea and Fennoscandia. 1989.
26. *Md. Anowar Hossain*: Seismic Refraction Studies in the Baltic Shield along the Fennolora Profile. 1989.
27. *Lars Erik Svensson*: Coulomb Excitation of Vibrational Nuclei. 1989.
28. *Bengt Carlsson*: Digital differentiating filters and model based fault detection. 1989.
29. *Alexander Edgar Kavka*: Coulomb Excitation. Analytical Methods and Experimental Results on even Selenium Nuclei. 1989.
30. *Christopher Juhlin*: Seismic Attenuation, Shear Wave Anisotropy and Some Aspects of Fracturing in the Crystalline Rock of the Siljan Ring Area, Central Sweden. 1990.

31. *Torbjörn Wigren*: Recursive Identification Based on the Nonlinear Wiener Model. 1990.
32. *Kjell Janson*: Experimental investigations of the proton and deuteron structure functions. 1991.
33. *Suzanne W. Harris*: Positive Muons in Crystalline and Amorphous Solids. 1991.
34. *Jan Blomgren*: Experimental Studies of Giant Resonances in Medium-Weight Spherical Nuclei. 1991.
35. *Jonas Lindgren*: Waveform Inversion of Seismic Reflection Data through Local Optimisation Methods. 1992.
36. *Liqi Fang*: Dynamic Light Scattering from Polymer Gels and Semidilute Solutions. 1992.
37. *Raymond Munier*: Segmentation, Fragmentation and Jostling of the Baltic Shield with Time. 1993.

Prior to January 1994, the series was called *Uppsala Dissertations from the Faculty of Science*.

Acta Universitatis Upsaliensis

Uppsala Dissertations from the Faculty of Science and Technology

Editor: The Dean of the Faculty of Science

1–14: 1994–1997. 15–21: 1998–1999. 22–35: 2000–2001. 36–51: 2002–2003.

52. *Erik Larsson*: Identification of Stochastic Continuous-time Systems. Algorithms, Irregular Sampling and Cramér-Rao Bounds. 2004.
53. *Per Åhgren*: On System Identification and Acoustic Echo Cancellation. 2004.
54. *Felix Wehrmann*: On Modelling Nonlinear Variation in Discrete Appearances of Objects. 2004.
55. *Peter S. Hammerstein*: Stochastic Resonance and Noise-Assisted Signal Transfer. On Coupling-Effects of Stochastic Resonators and Spectral Optimization of Fluctuations in Random Network Switches. 2004.
56. *Esteban Damián Avendaño Soto*: Electrochromism in Nickel-based Oxides. Coloration Mechanisms and Optimization of Sputter-deposited Thin Films. 2004.
57. *Jenny Öhman Persson*: The Obvious & The Essential. Interpreting Software Development & Organizational Change. 2004.
58. *Chariklia Rouki*: Experimental Studies of the Synthesis and the Survival Probability of Transactinides. 2004.
59. *Emad Abd-Elrady*: Nonlinear Approaches to Periodic Signal Modeling. 2005.
60. *Marcus Nilsson*: Regular Model Checking. 2005.
61. *Pritha Mahata*: Model Checking Parameterized Timed Systems. 2005.
62. *Anders Berglund*: Learning computer systems in a distributed project course: The what, why, how and where. 2005.
63. *Barbara Piechocinska*: Physics from Wholeness. Dynamical Totality as a Conceptual Foundation for Physical Theories. 2005.
64. *Pär Samuelsson*: Control of Nitrogen Removal in Activated Sludge Processes. 2005.

65. *Mats Ekman*: Modeling and Control of Bilinear Systems. Application to the Activated Sludge Process. 2005.
66. *Milena Ivanova*: Scalable Scientific Stream Query Processing. 2005.
67. *Zoran Radovic*: Software Techniques for Distributed Shared Memory. 2005.
68. *Richard Abrahamsson*: Estimation Problems in Array Signal Processing, System Identification, and Radar Imagery. 2006.
69. *Fredrik Robelius*: Giant Oil Fields – The Highway to Oil. Giant Oil Fields and their Importance for Future Oil Production. 2007.
70. *Anna Davour*: Search for low mass WIMPs with the AMANDA neutrino telescope. 2007.
71. *Magnus Ågren*: Set Constraints for Local Search. 2007.
72. *Ahmed Rezine*: Parameterized Systems: Generalizing and Simplifying Automatic Verification. 2008.
73. *Linda Brus*: Nonlinear Identification and Control with Solar Energy Applications. 2008.
74. *Peter Naucclér*: Estimation and Control of Resonant Systems with Stochastic Disturbances. 2008.
75. *Johan Petrini*: Querying RDF Schema Views of Relational Databases. 2008.
76. *Noomene Ben Henda*: Infinite-state Stochastic and Parameterized Systems. 2008.
77. *Samson Keleta*: Double Pion Production in $dd \rightarrow \alpha\pi\pi$ Reaction. 2008.
78. *Mei Hong*: Analysis of Some Methods for Identifying Dynamic Errors-invariables Systems. 2008.
79. *Robin Strand*: Distance Functions and Image Processing on Point-Lattices With Focus on the 3D Face-and Body-centered Cubic Grids. 2008.
80. *Ruslan Fomkin*: Optimization and Execution of Complex Scientific Queries. 2009.
81. *John Airey*: Science, Language and Literacy. Case Studies of Learning in Swedish University Physics. 2009.
82. *Arvid Pohl*: Search for Subrelativistic Particles with the AMANDA Neutrino Telescope. 2009.
83. *Anna Danielsson*: Doing Physics – Doing Gender. An Exploration of Physics Students' Identity Constitution in the Context of Laboratory Work. 2009.
84. *Karin Schöning*: Meson Production in pd Collisions. 2009.
85. *Henrik Petré*: η Meson Production in Proton-Proton Collisions at Excess Energies of 40 and 72 MeV. 2009.
86. *Jan Henry Nyström*: Analysing Fault Tolerance for ERLANG Applications. 2009.
87. *John Håkansson*: Design and Verification of Component Based Real-Time Systems. 2009.
88. *Sophie Grape*: Studies of PWO Crystals and Simulations of the $\bar{p}p \rightarrow \bar{\Lambda}\Lambda, \bar{\Lambda}\Sigma^0$ Reactions for the PANDA Experiment. 2009.
90. *Agnes Rensfelt*: Viscoelastic Materials. Identification and Experiment Design. 2010.
91. *Erik Gudmundson*: Signal Processing for Spectroscopic Applications. 2010.
92. *Björn Halvarsson*: Interaction Analysis in Multivariable Control Systems. Applications to Bioreactors for Nitrogen Removal. 2010.
93. *Jesper Bengtson*: Formalising process calculi. 2010.
94. *Magnus Johansson*: Psi-calculi: a Framework for Mobile Process Calculi. Cook your own correct process calculus – just add data and logic. 2010.
95. *Karin Rathsmann*: Modeling of Electron Cooling. Theory, Data and Applications. 2010.

96. *Liselott Dominicus van den Bussche*. Getting the Picture of University Physics. 2010.
97. *Olle Engdegård*. A Search for Dark Matter in the Sun with AMANDA and IceCube. 2011.
98. *Matthias Hudl*. Magnetic materials with tunable thermal, electrical, and dynamic properties. An experimental study of magnetocaloric, multiferroic, and spin-glass materials. 2012.
99. *Marcio Costa*. First-principles Studies of Local Structure Effects in Magnetic Materials. 2012.
100. *Patrik Adlarson*. Studies of the Decay $\eta \rightarrow \pi^+ \pi^- \pi^0$ with WASA-at-COSY. 2012.
101. *Erik Thomé*. Multi-Strange and Charmed Antihyperon-Hyperon Physics for PANDA. 2012.