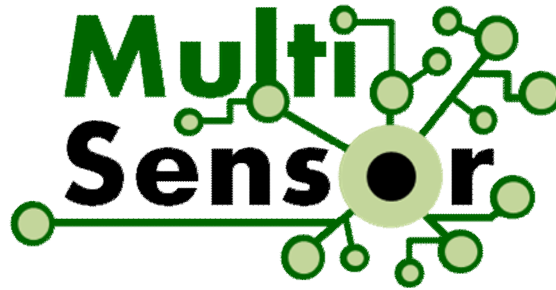

MULTISENSOR

Mining and Understanding of multilingual content for Intelligent Sentiment Enriched context and Social Oriented interpretation

FP7-610411

D5.2

Hybrid Reasoning Techniques

Dissemination level:	Public
Contractual date of delivery:	Month 18, 30 April 2015
Actual date of delivery:	Month 18, 30 April 2015
Workpackage:	WP5 Semantic Reasoning and Decision Support
Task:	T5.3 Hybrid Reasoning
Type:	Report
Approval Status:	Final Draft
Version:	1.0
Number of pages:	36
Filename:	D5.2_HybridReasoningTechniques_2015-04-30_v1.0.pdf

Abstract

The goal of this document is to describe the technical details and motivation behind the implementation of four different types of reasoning strategies - Hybrid reasoning (Forward plus Backward), Parallel multi-threaded reasoning, GeoSPARQL and SPARQL-MM. It explains how these approaches cover the specific requirements of the main use cases expressed in D8.2.

The information in this document reflects only the author's views and the European Community is not liable for any use that may be made of the information contained therein. The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Co-funded by the European Union

History

Version	Date	Reason	Revised by
0.1	10/04/2015	First draft sent to partners for comments	Dimitar Manov (ONTO)
0.2	15/04/2015	First integrated draft	Dimitar Manov (ONTO)
0.8	28/04/2015	Document for internal review	Dimitar Manov (ONTO), Boyan Simeonov (ONTO)
0.9	29/04/2015	Review	E. Kamateri (CERTH), Stefanos Vrochidis (CERTH)
1.0	30/04/2015	Final updated document correcting minor errors	Dimitar Manov (ONTO), Boyan Simeonov (ONTO)

Author list

Organization	Name	Contact Information
Ontotext AD	Dimitar Manov	dimitar.manov@ontotext.com
Ontotext AD	Boyan Simeonov	boyan.simeonov@ontotext.com
Ontotext AD	Kiril Simov	kiril.simov@ontotext.com
Ontotext AD	Pavel Mihaylov	pavel.mihaylov@ontotext.com
OntotextAD	Vladimir Alexiev	vladimir.alexiev@ontotext.com

Executive Summary

The objective of this deliverable is to develop inference methods that support efficient information selection from heterogeneous data pools. They have to be implemented on top of GraphDB – an RDF graph database engine. The final goal is to enable efficient retrieval of data, considering different criteria and implementing mechanisms, which go beyond the capabilities of today's database and search engines. Technically, these mechanisms can be implemented in two ways (i) as expert-systems-style reasoning techniques, which allow new facts to be inferred or (ii) as query language extensions, which allow for more efficient handling of specific data constraints.

For the purposes of MULTISENSOR, Ontotext has developed four different types of inference:

- Hybrid reasoning: a combination of forward- and backward-chaining;
- Parallel multi-threaded reasoning;
- SPARQL-MM;
- GeoSPARQL.

The first section of this document provides an introduction. Section 2 describes hybrid reasoning that involves partial forward-chaining materialization, where new facts are derived upon insertion of data and stored in the database, with query-time backward-chaining or query re-writing. Such support is important in cases where none of the reasoning strategies can do the job alone. For instance, in scenarios with very complex reasoning over big datasets that would take too long time to pre-compute using forward-chaining. Forward-chaining is also impractical when big parts of the dataset are deleted or updated on a regular basis.

The third section describes the implementation of parallel multi-threaded reasoning. It allows for a serious speed up of the forward-chaining and materialization – the main method used for inference in GraphDB. Effectively, this results in loading times that are between 50% and 200% faster compared to loading data with a serial inference engine. This is an important advantage for a number of application scenarios involving very large datasets (in the range of billions of statements). Thus, speeding the inference up makes the use of GraphDB feasible for applications where loading data with non-trivial inference can take several days or even weeks. Ontotext has multiple clients in the publishing and pharmaceutical sectors, which deal with datasets larger than 20 billion triples and for them parallel inference is a must.

Section 4 is about SPARQL-MM, which allows for efficient retrieval of multimedia content, based on annotation of images, and movies, using both temporal constraints and positioning of objects within the video scenes. The SPARQL-MM support in GraphDB implements the required additional specific index structures and the handling of SPARQL syntax extensions. The additional indexing structures allow SPARQL-MM queries to be executed in real time. In news and media organizations this means that authors and editors will be able to search content archives in a much more efficient manner.

Section 5 presents GraphDB's support for GeoSPARQL – an extension of SPARQL that defines a powerful language for expressing geo-spatial constraints, e.g. overlap or proximity of two

geographic regions. Such queries enable much more efficient querying and visualization of datasets where geographic location is of high importance, such as news and content archives.

Abbreviations and Acronyms

LOD	Linked Open Data
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	SPARQL Protocol and RDF Query Language
WP	Work Package
BCWI	Backward-Chaining Wrapping Iterator
SI	Statement Iterator
W3C	World Wide Web Consortium
URI	Uniform resource identifier
GDP	Gross domestic product
OGC	Open Geospatial Consortium
GML	Geography Markup Language
WKT	Well-known Text
RCC	Region Connection calculus
QSQ	Query-subquery
OWL	Web Ontology Language
SPB	Semantic Publishing Benchmark
FC	Forward Chaining
BC	Backward Chaining

Table of Contents

1 INTRODUCTION 9

1.1 Requirements 9

2 COMBINATION OF FORWARD AND BACKWARD-CHAINING 12

2.1 Approaches overview 12

2.2 Implementation in MULTISENSOR 13

2.2.1 Supporting indexes for each added statement 14
2.2.2 Answering a SPARQL Query 14
2.2.3 owl:inverseOf 14
2.2.4 owl:SymmetricProperty 14
2.2.5 owl:equivalentProperty 14
2.2.6 Combining inverse/symmetric and equivalent properties for efficiency 15
2.2.7 owl:equivalentClass 15
2.2.8 rdf:type and rdfs:subClassOf 15
2.2.9 owl:Transitive and owl:TransitiveOver 15

2.3 Evaluation 15

3 PARALLEL MULTI-THREADED REASONING 17

3.1 Approaches overview 17

3.2 Implementation in MULTISENSOR 18

3.2.1 GraphDB storage background 18
3.2.2 GraphDB parallel data loading pipeline 18
3.2.3 Implementation details 20

3.3 Evaluation 20

4 SPARQL-MM 22

4.1 Approaches overview 22

4.1.1 SPARQL multimedia functions 24
4.1.1.1 Spatial Relations 24
4.1.1.2 Spatial Aggregations 26
4.1.1.3 Temporal Relations 26
4.1.1.4 Temporal Aggregations 27
4.1.1.5 Combined Aggregations 27

4.2 Implementation in MULTISENSOR 28

4.3 Evaluation 28

5 GEO-SPARQL 29

5.1 Approaches overview 29

5.1.1 GeoSPARQL ontology 29

5.1.2 Rules for query transformation 30

5.2 Implementation in MULTISENSOR 31

5.2.1 Query optimization 33

5.2.2 Custom literals-in-relations extension 33

5.2.3 Third-party libraries used 33

5.3 Evaluation 34

6 CONCLUSIONS 35

7 REFERENCES 36

1 INTRODUCTION

The objective of this deliverable is to develop inference methods that support efficient information selection from heterogeneous data pools. They have to be implemented on top of the GraphDB – an RDF graph database (a type of NoSQL graph database engine), also known as triplestore[1]. The final goal is to enable efficient retrieval of data, considering different criteria and implementing mechanisms, which go beyond the capabilities of today's database and search engines.

This deliverable explores the development of four different types of reasoning. Technically, these mechanisms can be implemented in two ways (i) as expert-systems-style reasoning techniques, which allow new facts to be inferred or (ii) as query language extensions, which allow for more efficient handling of specific data constraints. Broadly speaking, inference can be characterized by discovering new relations. On the Semantic Web, data is modeled as a set of (named) relations between resources. "Inference" means that automatic procedures can generate new relations based on the data and some additional information in the form of a vocabulary - a set of rules. Whether the new relations are explicitly added to the set of data or returned at query time is matter of implementation. Inference is a tool of choice for improving the quality of data integration by discovering new relations, automatically analyzing the content of data, or managing knowledge in general. Inference-based techniques are also important for discovering possible inconsistencies in the data.

The reasoning techniques will support the semantic search service (i.e. semantic reasoning in the knowledge base) and the decision support service. In Figure 1 (taken from D7.2 deliverable describing the overall architecture) we highlight the exact role of these services in the whole MULTISENSOR architecture.

1.1 Requirements

In this section, we present the relation of the four different reasoning techniques to the purpose of the project.

Hybrid reasoning combines the strengths of forward-chaining (the default reasoning strategy in GraphDB) and backward-chaining. Such combined strategies are useful for reasoning scenarios where large amounts of data are combined with the need for goal-driven reasoning, particularly in: decision support systems, semantic data integration from multiple sources and involving multiple ontologies.

GraphDB already implements a kind of limited backward-chaining, namely the *owl:sameAs* optimization. With this optimization, statements are stored against clusters of sameAs-equivalent URIs, rather than directly against URIs. Thus, all statements of a URI are "inherited" by all equivalent URIs. When solving queries, any constant URI is treated as the equivalent cluster, thus ensuring completeness. A special mode allows enumerating all equivalent URIs, if needed. In MULTISENSOR, we extend this approach to other mechanisms for semantic data integration, namely through the OWL mapping properties *owl:equivalentProperty* and *owl:equivalentClass*. Currently, these constructs generate a large

[1] Triplestores are built around a range of W3C standards for representation and querying of data in the Semantic Web: RDF (data representation, sort of alternative to XML), RDFS (schema language), OWL (ontology language), SPARQL (query language)

number of "duplicate" inferences (e.g. each statement for a property P is repeated for all other properties Q in its cluster of equivalent properties). We also address symmetric, inverse, and transitive reasoning by implementing special reasoning strategies. Implementing efficient hybrid reasoning is a complex problem as it always involves a tradeoff between load-time pre-computation (forward-chaining) and storage consumption on one hand, and the speed of query answering on the other. The criteria when to use one or the other are not always clear and are hard to estimate. With this initial implementation in MULTISENSOR, we are only starting to explore this area.

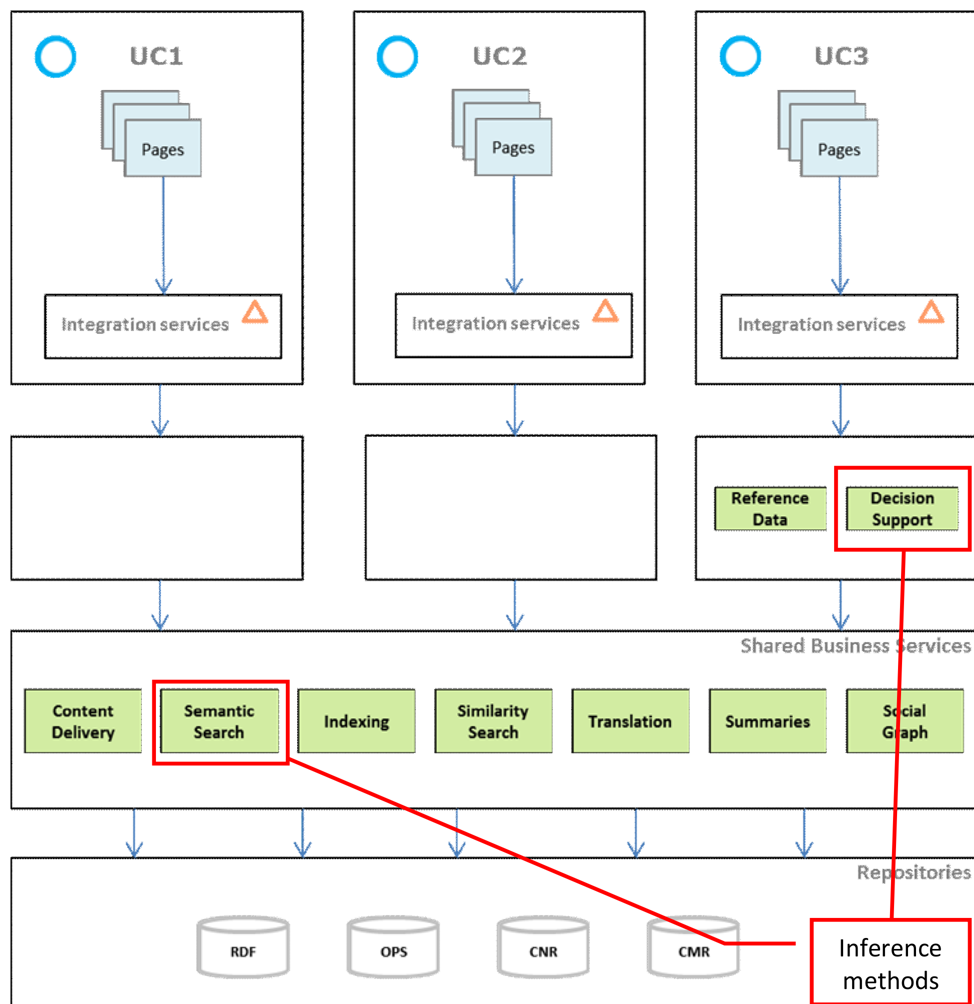

Figure 1: Reasoning services in MULTISENSOR architecture.

One of the important technical problems that the GraphDB development team had to solve in 2014 was to increase data loading speed. This is an important requirement for very large or often-changing datasets. We have implemented parallel loading, where the major challenge was how to ensure the correct work of the *Entity Pool* (a central storage of URIs and literals) given the multiple loading threads. This has increased the loading speeds from 50% to 200%. While the parallel loader made the storage process work in parallel, it did not do so for the reasoning. The next improvement requested by our clients was to increase the speed of inferencing during loading. This requirement is important for even moderately big datasets, which involve a significant amount of inferencing (over 20%). MULTISENSOR also benefits a lot from this as the ontologies that it uses (NIF and related) involve large amounts

of inferencing. Guaranteeing the accuracy of inferencing when it is performed in parallel is a challenging technical task, which we are able to tackle as part of our MULTISENSOR work.

SPARQL-MM (Multimedia) is a proposed SPARQL extension that enables reasoning and query answering over media files (e.g. videos). The query extensions include exploring spatio-temporal relations in semantically annotated multimedia, e.g. finding particular objects or types of objects in a spatial relation to each other (e.g. left-right), or a temporal relation to each other (e.g. one appearing immediately after the other). Given MULTISENSOR's goals to work with varied media content (including images and videos), SPARQL-MM reasoning is a natural requirement. Such relations can be very useful for precise querying of large multimedia collections. The power of querying shows synergies with semantic enrichment and classification. For example, if a video asset is annotated with DBpedia URLs of Barack Obama and Gen. Martin Dempsey, we know their professions/offices, and have reasonable classification of military professions, then the asset can be found through a query like "find shots of the US President standing next to a Soldier".

Geographical reasoning and querying is very relevant for the news and media domain, especially for local news. Many events happen in a specific locality, and the ability to query by geography provides an important dimension. In addition, geography is very important in a trade context. Therefore, GeoSPARQL reasoning is also relevant for Use Case 3[2]. GraphDB supported only rudimentary geospatial reasoning, based on some custom extensions. It can find points within a rectangle, points nearby a center, and compute distances. However, GeoSPARQL is an emerging standard that provides much deeper functionality, based on sophisticated "algebras of regions". These allow various spatial questions to be answered, and various geometries and features to be addressed.

[2] D8.2 Requirements Use Cases Validation Plan

2 COMBINATION OF FORWARD AND BACKWARD-CHAINING

2.1 Approaches overview

The logical inference over RDF datasets allows for two principle strategies for rules application:

- *Forward-chaining*: to start from the known facts (the explicit statements) and to perform inference in an inductive fashion. Typically, the goal is to compute the *Inferred Closure*.
- *Backward-chaining*: to start from a particular fact or a query, and to verify it or get all possible results. In a nutshell, the reasoner decomposes (or transforms) the query (or the fact) into simpler (or alternative) facts, which are available in the knowledge base or can be proven through further recursive transformations.

The forward-chaining strategy applies the rules over the available facts in order to infer new facts, which are added to the dataset, and then recursively applies the rules over the new dataset. The result is the so-called inferred closure: an extension of a knowledge base (the RDF dataset or the graph of RDF triples) with all implicit facts (RDF triples) that can be inferred from it. The notion *Materialization* is defined as a procedure that keeps an up-to-date inferred closure of the knowledge base. Materialization is known as a technique for applying inference before query evaluation. This allows for many query optimization approaches to be implemented as querying is realized by lookups in the database. The main drawback of materialization is that the database changes, additions, and updates are generally slow operations. In many scenarios, the materialization of such frequent changes does not affect the querying process, as many of the materialized facts are not used in the answers.

In such cases, an alternative to forward-chaining is a backward-chaining strategy for inferencing over knowledge bases. Here, answering a query requires only partial materialization over the knowledge base. Unfortunately, backward-chaining is inefficient for large knowledge bases, as many optimizations for the materialization of a knowledge base are not possible.

In the MULTISENSOR user scenarios, we come across heterogeneous knowledge bases, including a stable core set of RDF datasets from a LOD cloud such as DBpedia, Geonames, etc., as well as dynamic datasets, extracted by the services from different sources (text and multimedia documents), or hypothesis testing for decision support systems. Therefore, to support the necessary functionality, we have implemented a hybrid approach to rule application in the project. In order to gain from both strategies, one needs to understand very well their positive and negative sides.

Pros and Cons of materialization:

- ✓ Fast query evaluation:
  - o No deduction, satisfiability testing, or other kind of reasoning are required at query time;
  - o RDBMS-like query optimization techniques are applicable.

– Slower insertion of new facts (the repository extends the inferred closure after each transaction for modification);

– Slower deletion of facts (the repository has to remove all facts that are no longer true from the inferred closure);

– The maintenance of the inferred closure requires additional space (RAM and disk) to manage the extra statements;

Pros and Cons of backward-chaining:

✓ No impact on the insert and update performance (no work is required, when the schema changes);

– Queries easily become 10x slower:

  - If pattern-level re-writing is used, query optimization is impossible, making working with big datasets with query optimization impractical;
  - If query re-writing is used, one has to evaluate tens of query variants (to mitigate this, some engines using query re-writing have strategies that cut-off query re-writing at a certain point, which leads to incomplete results).

– Query-time reasoning is OK only if:

  - It is not recursive AND
  - It does not introduce multiple alternatives at pattern level (this is the case when there is a simple "syntactic transformation", i.e. If it is equivalent to custom SPARQL functions);
  - OR if data is very small (for instance, less than 1M statements), query performance is acceptable even without optimization.

In the next section, we present the main elements of the backward-chaining strategy implemented in the project.

2.2 Implementation in MULTISENSOR

The Graph DB workbench already supports a state-of-the-art forward-chaining strategy, which is additionally extended with parallel reasoning in the MULTISENSOR project. Thus, here we present mainly the elements of backward-chaining. We follow one of the most advanced approaches to implementing hybrid reasoning for a fraction of OWL in RDF databases as presented in the work of Urbani et al. (2014). They implement backward-chaining based on the QSQ (query-subquery) algorithm for Datalog databases modified to support reasoning over OWL RL. In the application of the algorithm, the facts are divided in two sets: one over which the forward-chaining is applied and the materialization over the set is stored in GraphDB in an optimal way. The other is used to support a backward-chaining strategy. It applies the materialization only when it is necessary.

The paper defines the following properties, which can use backward-chaining:

- TYPE *rdf:type*
- SCO *rdfs:subClassOf*
- SPO *rdfs:subPropertyOf*

- EQC *owl:equivalentClass*
- EQP *owl:equivalentProperty*
- INV *owl:inverseOf*
- SYM *owl:SymmetricProperty*
- TRANS *owl:TransitiveProperty*

We have implemented some of these, and want to add support for *ptop:transitiveOver* – a property that specifies that one property is transitive with respect to another. E.g. *rdf:type* is transitive-over *rdfs:subClassOf*.

2.2.1 Supporting indexes for each added statement

On each added statement, we perform checks for the backward-chaining properties and keep in-memory structures relevant to each of them:

- *owl:inverseProperty* and *owl:equivalentProperty* keep a mapping between each property and the list of its inverse (or equivalent) properties.
- *rdf:type* and *rdfs:subClassOf* keep the explicit types and the whole subclass hierarchy.

2.2.2 Answering a SPARQL Query

Generally speaking, all SPARQL queries are answered using iterators *It(?subject, ?predicate, ?object)*, where each of the variables can be a wildcard (the contexts, limit/offset are mostly irrelevant to the inference). Therefore, in order to provide an answer for the query, we need to explain how we define the iterator for each backward-chained property. The implementation introduces a new class *BackwardChainingWrappingIterator* (*BCWI*), which uses the default GraphDB *StatementIterator* (*SI*) to provide extended results.

2.2.3 owl:inverseOf

We keep track of the list of inverse properties for each property (there might be more than one, although all will be equivalent). There are two cases depending on whether the property of the iterator (?p) is bound or not:

1) *BCWI(?s, ?p=P, ?o) = Restrict(UnionOf {SI(?o, Pi1, ?s), …, SI(?o, PiN, ?s)))* where *Pi1…PiN* is the set of the inverse properties of the property *P* and *Restrict()* is an operator filtering the results in case *?s* or *?o* are bound.
2) *BCWI(?s, ?p, ?o) = UnionOf(BCWI(?s, ?p=P1, ?o), …, BCWI(?s, ?p=PN, ?o))*

   Where *P1…PN* is the list of all properties in the database.

2.2.4 owl:SymmetricProperty

The symmetric properties are handled as simplified inverse properties.

2.2.5 owl:equivalentProperty

There are again two cases depending on whether the predicate is bound or not:

1) *BCWI(?s, ?p=P, ?o) = Restrict(UnionOf {SI(?s, Pi1, ?o), …, SI(?s, PiN, ?o)))* where *Pi1…PiN* is the set of all the equivalent properties of the property *P*.

2) *BCWI(?s, ?p, ?o) = UnionOf(BCWI(?s, ?p=P1, ?o), …, BCWI(?s, ?p=PN, ?o))*

   Where *P1…PN* is the list of all properties in the database.

2.2.6 Combining inverse/symmetric and equivalent properties for efficiency

Because all the three properties are handled similarly, it is more efficient to process them together.

2.2.7 owl:equivalentClass

This has not been implemented yet.

2.2.8 rdf:type and rdfs:subClassOf

We keep the explicit types and the subclass hierarchy in memory. It takes 17 bytes for each entity (with 32-bit entity IDs) that has a type statement.

The queries for type are expressed as a join between two iterators:

BCWI(?s type ?o)=SI(?s type ?x) join SI(?x subClassOf ?o).

2.2.9 owl:Transitive and owl:TransitiveOver

We plan to handle them in a similar way to *type/subClassOf*.

2.3 Evaluation

As explained, the trade-off between FC and BC is the loading speed improvement for query speed decrease. As shown in the next table, the loading speed has improved nearly 4 times:

Reasoning	Load time, s	Total statements	Speed, statements/s
Forward-chaining	905	64M	70,718
Backward-chaining	3551	64M	18,023

We have executed two very simple queries to get an initial idea of the query speed decrease:

Query	Forward-chaining	Backward-chaining
select count (*) {?x a cwork:CreativeWork}	11,283	15,851
select count (*) {?x owl:inverseOf ?y}	153	172

So, it seems that for simple queries the decrease is OK (negligible for non-transitive properties and quite acceptable for *type/subClassOf* or transitive properties).

In order to get a feel how the backward chaining performs in a real world scenario, we run the basic interactive query mix LDBC's SPB on this data set. The setup is multithreaded and the test ends when each query from the mix has been executed 10 times. The results are presented in Table 1.

Query	FC, Time, ms	BC, Time, ms	Times slower
Q1	171	2,448	14.32
Q2	6	14	2.33
Q3	50	46	0.92

Q4	43	44	1.02
Q5	111	104	0.94
Q6	120	5,969	49.74
Q7	45	43	0.96
Q8	250	210	0.84
Q9	211	219	1.04
Q10	454	422	0.93
Q11	28	789,499	28,196.39
Q12	70	63	0.90

Table 1: The execution time of 12 queries

Table 1 shows that there are some queries that are tens of times slower in the BC scenario, compared to the FC one. A notable exception is the query Q11 which is very slow. The reason for that is it uses the pattern “*?e a owl:Thing”*, and *owl:Thing* is the root of the hierarchy.

On the other hand there are queries t(Q3, Q8 and Q10) that are faster in the BC scenario. This could be attributed to the fact that there are less statements in the repository and some queries become slightly faster.

3 PARALLEL MULTI-THREADED REASONING

3.1 Approaches overview

Parallel architecture and multi-threaded reasoning provide very appealing techniques for processing RDF knowledge bases consisting of an enormous amount of statements (usually several billions). The main reasoning strategy for RDF knowledge bases - materialization, faces two problems: (i) maintenance of huge number of URIs, and (ii) inferring new RDF statements via inference rules applied to existing, in the knowledge base, RDF statements. The problem of handling URIs is related to their size. In order to define a more compact representation of RDF statements, the URIs are represented in dictionaries, where each URI is identified by a numeric value, which is then used for the internal representation of the RDF statements. One of the URI terms' characteristics in an RDF knowledge base is their uneven distribution, i.e. many URI terms appear only a few times. This compression approach allows for using parallel processing. Urbani, Maassen and Bal 2010 (Urbani, et al. 2010) describes a MapReduce architecture performing the creation of a URI dictionary in three steps.

The first algorithm identifies the popular terms that appear in knowledge bases above some given threshold. It randomly reads portions of the knowledge bases and counts the URI terms that appear above the threshold. A dictionary of these most popular terms is created and loaded in the memory.

The second algorithm simultaneously processes all URIs of the statements in the knowledge base and constructs dictionaries for the more rare terms. In this way, the algorithm constructs also the compact representation of the RDF statements.

The third algorithm performs the reconstruction of the original statements. A similar technique is already implemented in GraphDB and maximally optimized for materialization reasoning. Thus, in the MULTISENSOR project, we concentrate on parallel optimization of inference of new statements during the materialization of the knowledge base.

The parallel reasoning cannot manage the global characteristics of RDF reasoning for very expressive ontological languages. For example, Oren et. al (2009) shows that partitioning of an RDF data base into independent parts is very difficult because it is hard to guarantee the soundness and completeness of the reasoning. Random partitioning of the knowledge base results in communication overload between the different partitions. Another approach is grouping the statements based on the URI terms, participating in the corresponding statements. This results in uneven balance between the different groups because of the different popularity (as we mentioned above) of the different terms.

In order to cope with this problem, some authors impose additional restrictions on the language expressivity and the type of statements in the knowledge base. For example, Priaya et. al (2014) defines an ABox independent partitioning, which supports reasoning in OWL Lite knowledge bases. They assume that the knowledge base is represented as a TBox, containing the ontology, and an ABox, containing the statements about the world. The independent ABox partitioning is defined for a knowledge base *K=(T,A)*, where *T* is the TBox and *A* is ABox of the knowledge base. Then $A = A1 \cup A2 \ldots \cup Ai$ in such a way that if *(T,A) |=* F then there is a partition *Aj* such that *(T,Aj) |= F*. Using this definition, they define an incremental algorithm for constructing an independent ABox partitioning of *K*. After building the partitioning, the inference is performed on different machines in such a way that different inferences on sub-

queries are done in the appropriate partition. In MULTISENSOR, we have also used knowledge base partitioning but we do not separate the ontology statements from the world statements, and we support a more expressive ontology language as well. Our approach is tuned to the overall architecture of GraphDB. It is presented it in the next section 3.2.

3.2 Implementation in MULTISENSOR

3.2.1 GraphDB storage background

GraphDB stores the statements in the following manner:

- First the entities (URIs, literals, or blank nodes) are resolved into internal IDs (32- or 40-bit long values), which are stored in a dictionary file-based structure, called *Entity Pool*;
- Then each statement is stored in several disk-based structures, called *Collections*. The structures differ by their sort order and are named after it. They are called POS, PSO, CPSO, CPSO (the last two are optional). The letters C, P, S and O stand for “context”, “predicate”, “subject” and “object” respectively. Thus, the statements are multiplied and the advantage is faster query times.

3.2.2 GraphDB parallel data loading pipeline

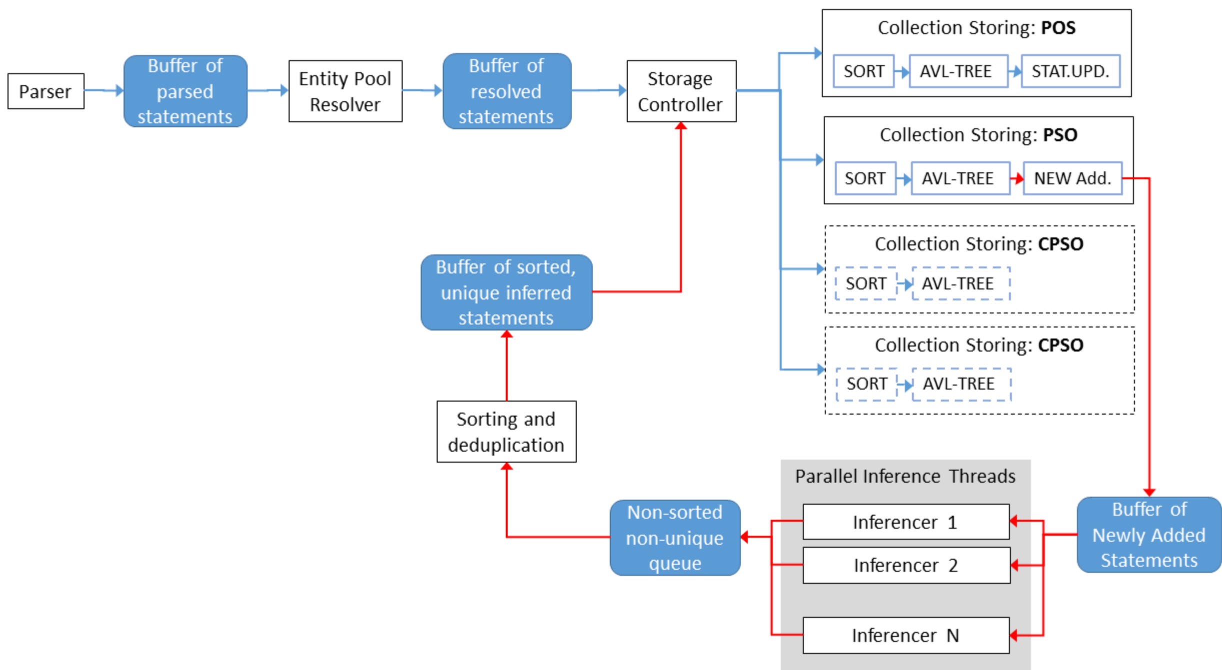

Figure 2: The GraphDB parallel data loading pipeline

In Figure 2, the workflow for fast loading is indicated with blue arrows and the one for reasoning is indicated with red arrows. What follows is a brief description of the mechanism:

1. The thread that stores statements in the PSO collection is modified to put newly added statements (i.e. not existing in the repository before) in a buffer. There we collect the statements that will be used as an input for the inference. The buffer is attached to the PSO collection, because (i) unlike C... collections, it is always there

and (ii) the POS collection already takes care to update the statistics. Therefore, it is a sort of distribution of the load between POS and PSO storing threads.

2. There are N identical threads that do inference, each of which consuming statements from the buffer, populated on the previous step. This process starts when the storage of all statements from the buffer that is currently being loaded is completed in all collections. If inference starts beforehand, there is a chance of missing valid inferences.
3. Each of the inferencers feeds newly inferred statements in a queue (eventually, blocking queue). This simple structure has to allow multiple threads to put statements into it with minimal contention.
4. There is a separate thread that consumes statements from the inference results queue and produces a sorted buffer of statements without duplications.
5. One "epoch" of inference is done, when (i) the inferencers have exhausted all the statements from the buffer of newly added statements, (ii) all the inference threads are done and (iii) the sorter and deduplication thread have exhausted the inference results queue and have completed the generation of the buffer of sorted unique inference results.
6. The Storage Controller starts a new iteration of storage and inference epoch as it starts to consume statements from the Buffer of sorted unique inferred statements.
7. The inference process is done when at the end of an inference epoch the Buffer of sorted unique inferred statements is empty.

Notes:

- The Storage Controller does not start processing the next Buffer of resolved statements until the inference for the previous one is completed. In other words, storage and inference work in shifts and never run in parallel.
- There is a space for further parallelization here and there but we consider the implementation of this algorithm to be the right starting point. Because there is a good balance between parallelization and simplicity, it looks manageable to implement and debug. It serves as a formal proof that this type of parallel inference can generate correct results. It also provides a baseline for speed.
  - One such parallelization is not to process the statements in epochs but in a continuous manner, in which the queue for the deduplicated inferred statements will not wait for all the inferencers to finish before feeding the statements in the storage controller;
  - Another one is to postpone the updates for context indexes, CPSO, CPOS until the whole data is loaded and then update them from one of the other indexes. This will reduce the time inferencers wait for the storage to be updated.

3.2.3 Implementation details

- *sameAs* optimization not supported (we still support *sameAs* but we keep all statements as if they were fully materialized);
- Currently, the LRU Object Cache has a locking issue, which severely limits the speed of the parallel load, especially with big data sets;
- Update and delete transactions cannot be handled in such a parallel mode;
- The functionality is currently implemented in the *LoadRDF tool*, which is used for initial loading of data but the pipeline works for normal commits as well;
- There are detailed performance statistics allowing the comparison between two runs with different number of threads.

3.3 Evaluation

We have evaluated the parallel inference with three datasets and with different number of threads. These are Wordnet, and the SPB-50 million and SPB-1 billion, artificially generated datasets from the LDBC's Semantic Publishing Benchmark (SPB, 2015). As a baseline for each dataset, we take the single threaded inference.

Dataset	Wordnet 2.2
Explicit statements	2.2M
Total statements	13M
Rule set	Owl-max

Table 2: Description of the dataset Wordnet 2.2

According to the LDBC Website *"**The Semantic Publishing Benchmark v2.0** (SPB) is a LDBC benchmark for RDF database engines inspired by the Media/Publishing industry, particularly by the BBC's Dynamic Semantic Publishing approach."*

The application scenario considers a media or a publishing organization that deals with large volume of *streaming content,* namely news, articles or "media assets". This content is enriched with *metadata* that describes it and links it to *reference knowledge* – taxonomies and databases that include relevant concepts, entities and factual information. This metadata allows publishers to efficiently retrieve relevant content, according to their various business models. For instance, some, like the BBC, can use it to maintain rich and interactive web-presence for their content, while others, e.g. news agencies, would be able to provide better-defined content feeds, etc.

From a technology standpoint, the benchmark assumes that an RDF database is used to store both the reference knowledge (mostly static) and the metadata (that grows constantly, to stay in synch with the inflow of streaming content). The main interactions with the repository are (i) *updates*, that add new metadata or alter it, and (ii) *queries*, that retrieve content according to various criteria."[3]

[3] See more at: http://ldbcouncil.org/developer/spb

Dataset	SPB-50m
Explicit statements	64M
Total statements	133M
Rule set	Custom: RDFS + owl:sameAs + owl:inverseOf

Table 3: Description of the dataset SPB-50m

Dataset	SPB-1B
Explicit statements	1B
Total statements	1.3B
Rule set	Custom: RDFS + owl:sameAs + owl:inverseOf

Table 4: Description of the dataset SPB-1B

Only the 4-thread parallel results are presented in the Table 5 below. The 2- and 8-thread are skipped because they are not good enough due to a locking issue.

The SPB-1B dataset has not been successfully loaded in a parallel mode but the expected increase is around 20%.

Dataset	**Serial**		**Parallel**		**Increase**
	time, s	**speed, st/s**	**time, s**	**speed, st/s**	
Wordnet	730	3,727	628	4,338	16%
SPB-50M	6,818	7,764	1,345	39,351	407%
SPB-1B	~48h	5,787	n/a	n/a	n/a

Table 5: The 4-thread parallel results

4 SPARQL-MM

Over the last decade, we have witnessed the incredible growth of multimedia technologies. The video content is all around us – television, Internet, mobile applications, etc. The video content takes a big part of our lives. However, armies of journalists, engineers, and media specialists are working behind the scene to make it possible for all this content to reach us. One of the MULTISENSOR project's goals is to deliver a platform and services that will support and help these people to do their job more efficiently, and provide a better media content as a result.

The first Pilot Use Case covered by MULTISENSOR describes journalism and media monitoring. To tell a story or to present a story from a different angle, a journalist needs to analyze a big amount of data from different sources – newspapers, blogs, web sources, social media, video materials, etc. All of these different types of data have their own style and specifics. It is difficult for a single person, in our case a journalist or a media monitoring specialist, to select and find all the data he needs for a limited period.

The MULTISENSOR project will deliver an infrastructure that will lead to a better data quality and reduce the time significantly. Parts of the services that will be provided are speech recognition, entity recognition, data enrichment, semantic and faceted search over the data, and a decision support system. As we are talking about journalistic work and media monitoring, one of the first things that come to mind is video material. This is because today video is one of the main means of information in this area. However, it is not an easy job to process many video materials and get the relevant information for your task.

For example, we are working on a story about Barack Obama and the military policy represented by his cabinet. We are searching for video content where the US president is seen next to a US soldier or officer. Usually, the search will start from the video archives. If they are stored in a computer database and we use a full-text search over the titles of the materials such as YouTube, this will facilitate our task. Now we retrieve 10 results and every video material is between 30 min to an hour. It will take a long time to watch all of it and find the content and the scenes that we need for our material. But what happens if a title does not represent the content and the most important material never appears in our result set? The answer of this question is SPARQL-MM. It gives us the ability to retrieve structural data from annotated video content.

4.1 Approaches overview

Over the last years, multimedia content and semantic technologies have come closer together than ever. Different projects and initiatives have contributed to this process such as the W3C recommendations for media annotations and fragmented URIs. The main goal of these initiatives is to standardize the representation of multimedia metadata by semantic technologies. SPARQL-MM is an extension of the SPARQL query language, which provides spatio-temporal filters and aggregated functions and can be valuable for journalists and media analyzers.

In the Barack Obama example, by using SPARQL-MM we can ask with a structured query "Give me all video fragments from my archive, where Barack Obama is next to a US soldier or a US officer." The results with the annotated content are shown on video 1 and 2. We use Media Fragment URIs to connect the annotations with specific spatio-temporal parts of the

video. These specifications provide a standardized media format, which helps to address specific media fragments on the web by URIs. Twisted pairs are supported – name-value-pairs, for example *s=start, end* for temporal and *xywh=x, y, width, height* for regions of a given fragment.

In the first video, Barack Obama is next to a US soldier from sec 29 to sec 33 and in the second, for the better part of the video. If you want to retrieve a spatio-temporal snippet that shows the US president next to a US soldier or officer, you have to ask the database. Currently SPARQL does not support queries of this kind because the required information is not represented in RDF format but is hidden in media fragmented URIs. SPARQL does not support functions of the type *rightBeside*, *remporalOverlap* and *boudingBox*, either.

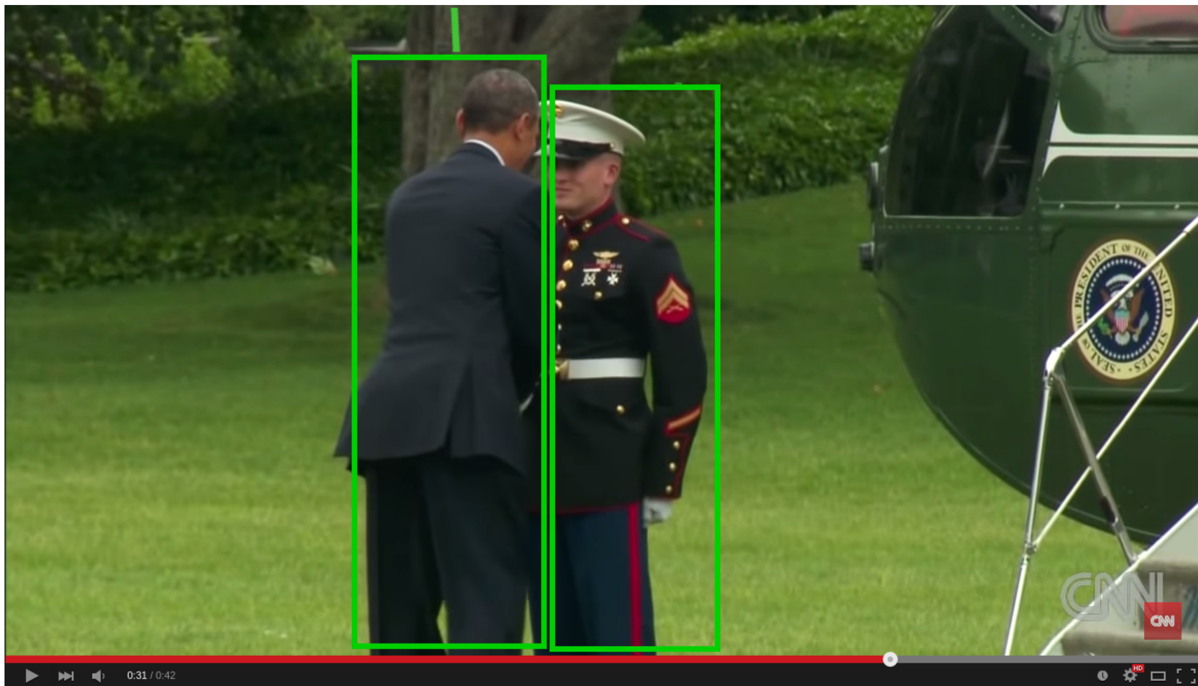

Figure 3: A snapshot from Video 1.

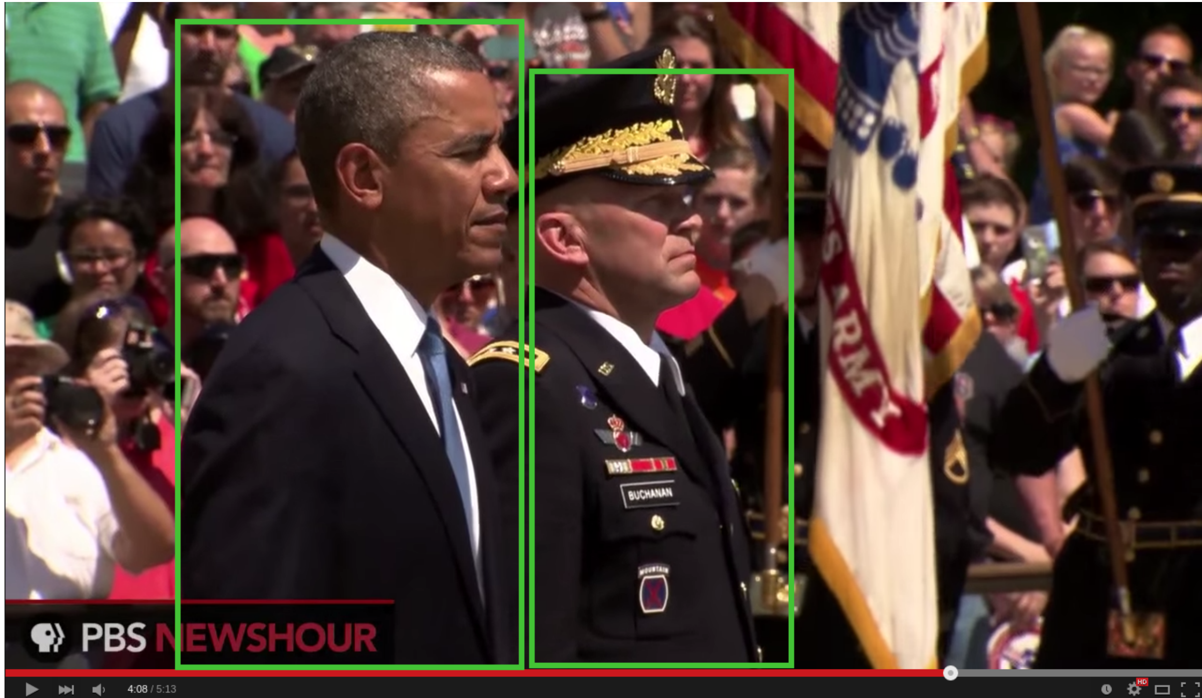

Figure 4: A snapshot from Video 2.

Thanks to SPARQL-MM, you can ask the database by the following query:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mmf: <http://linkedmultimedia.org/sparql-mm/ns/1.0.0/function#>
SELECT ?t1 ?t2 WHERE {
    ?f1 rdfs:label "Barack Obama".
    ?f2 rdfs:label "US soldier".
    FILTER mmf:rightBeside(?f1,?f2)
} ORDER BY ?t1 ?t2
```

4.1.1 SPARQL multimedia functions

SPARQL-MM supports a set of functions and all of them are following the DE-9IM standard describing the topological and temporal relations. The following tables present the name of a given function and a short description of the returned results.

4.1.1.1 Spatial Relations

Relation name	**Description**
mf:bottom(resource1,resource2)	returns true if resource.y >= 50%.
mf:isAbove(resource1,resource2)	returns true if resource1.y + resource1.h <= resource2.y, else false.
mf:isBelow(resource1,resource2)	returns true if resource2.y + resource2.h <= resource1.y, else false.

Relation name	**Description**
mf:left(resource1,resource2)	returns true if resource.x + resource.w <= 50%.
mf:leftBeside(resource1,resource2)	returns true if resource1.x + resource1.w <= resource2.x, else false.
mf:right(resource1,resource2)	returns true if resource.x >= 50%.
mf:rightBeside(resource1,resource2)	returns true if resource2.x + resource2.w <= resource1.x, else false.
mf:spatialCovers(resource1,resource2)	returns true if all points of resource1.box are points of resource2.box, else false.
mf:spatialDisjoint(resource1,resource2)	returns true is resource1.box has no common points with resource2.box, else false.
mf:spatialEqual(resource1,resource2)	returns true if resource1.box == resource2.box, else false.
mf:spatialIntersects(resource1,resource2)	returns true if resource1.box has at least one common point with resource2.box, else false.
mf:spatialTouches(resource1,resource2)	returns true if resource1.box.edge has at least one common point with resource2.box.edge and resource1.box.interior has no common point with resource2.box.interior, else false.
mf:top(resource1,resource2)	returns true if resource.y + resource.h <= 50%.

4.1.1.2 Spatial Aggregations

Relation name	Description
mf:spatialBoundingBox(resource1,resource2)	returns new MediaFragmentURI with spatial fragment out of existing resources A and B, so that x = min(A.x, B.x) and y = min(A.y, B.y) and w = max(A.x + A.w, B.x + B.w) and h = max(A.y + A.h, B.y + B.h).
mf:spatialIntersection(resource1,resource2)	returns new MediaFragmentURI with spatial fragment out of existing resources A and B, so that x = max(A.x, B.x) and y = max(A.y, B.y) and w = min(A.x + A.w, B.x + B.w) - max(A.x, B.x) and h = min(A.y + A.h, B.y + B.h) - max(A.y, A.x)

4.1.1.3 Temporal Relations

Relation name	Description
mf:after(resource1,resource2)	Returns *true* if resource1.start >= resource2.end, else *false*.
mf:before(resource1,resource2)	returns true if resource1.end <= resource2.start, else false.
mf:finishes(resource1,resource2)	returns true if resource1.end == resource2.end and resource1.start > resource1.start , else false.
mf:temporalMeets(resource1,resource2)	returns true if resource1.start = resource2.end or resource1.end = resource2.start, else false.
mf:starts(resource1,resource2)	returns true if resource1.start == resource2.start and resource1.end < resource2.end , else false.
mf:temporalContains(resource1,resource2)	returns true if resource1.start <= resource2.start and resource1.end >= resource2.end, else false.
mf:temporalEqual(resource1,resource2)	returns true if resource1.start == resource2.start and resource1.end == resource2.end, else false.

Relation name	Description
mf:temporalOverlaps(resource1,resource2)	returns true if resource1.start < resource2.start < resource1.end < resource2.end or resource2.start < resource1.start < resource2.end < resource1.end, else false.

4.1.1.4 Temporal Aggregations

Relation name	Description
mf:temporalBoundingBox(resource1,resource2)	returns new MediaFragmentURI with temporal fragment (Min(resource1.start, resource2.start), Max(resource1.end, resource2.end)).
mf:temporalIntermediate(resource1,resource2)	returns new MediaFragmentURI with temporal fragment (Min(resource1.end, resource2.end), Max(resource1.start, resource2.start)) if intersection not exists, else null.
mf:temporalIntersection(resource1,resource2)	returns new MediaFragmentURI with temporal fragment (Max(resource1.start, resource2.start), Min(resource1.end, resource2.end)) if intersection exists, else null.

4.1.1.5 Combined Aggregations

Relation name	Description
mf:boundingBox(resource1,resource2)	returns new MediaFragmentURI with spatial and temporal fragment. It it works like spatialFunction:boundingBox, temporalFunction:boundingBox or both together.
mf:intersection(resource1, resource2)	returns new MediaFragmentURI with spatial and temporal fragment. It works like spatialFunction:boundingBox, temporalFunction:intersection and both.

4.2 Implementation in MULTISENSOR

Support for SPARQL-MM in MULTISENSOR is provided through a set of GraphDB SPARQL functions, based on a code written by Thomas Kurz (Kurz et al. 2014), one of the principal authors of the SPARQL-MM specification.

This initial release has no SPARQL-MM indexing or optimizations but it is fully functional.

4.3 Evaluation

No further evaluation besides conformance testing has been done.

5 GEO-SPARQL

The second main Pilot Use Case in the MULTISENSOR project is about expanding the market of small and medium-sized enterprises outside the country where they operate. The reasons for this can be different – searching for new markets, revenue growth, business expansion, etc. When making such a decision, many steps have to be considered. For example, studying the laws of the chosen country can be challenging, as most of the documents are available only in the native language.

For this purpose, MULTISENSOR offers a platform and services that will provide important information about the country's GDP, corruption, unemployment, purchasing power, geopolitical information and specification in a human-readable form. The geopolitical information also includes geographical areas, settlements, nested regions, etc. This requires the implementation of a single standard for working with this type of data – GeoSPARQL.

5.1 Approaches overview

GeoSPARQL represents a standard for retrieving and inserting geospatial RDF data in a wide range of geospatial cases – from simple point of interest knowledge bases to detailed authoritative geospatial data sources for transportation. It is designed to accommodate systems based on qualitative spatial reasoning as well as systems based on quantitative spatial computations. This means that a simple knowledge base implementation intended for simple use cases does not need to implement all advanced reasoning capabilities of GeoSPARQL such as quantitative reasoning or query re-writing.

There are several terminology sets connected with the topological relations between geometries. Every triplestore implementation can choose which terminology set to use and support.

The GeoSPARQL specification contains three main components:

- Vocabulary definitions, which represent the geometries, relations, and connections between them;
- Aggregation of specific domain functions, which are used in the SPARQL queries;
- Set of rules for query re-writing.

5.1.1 GeoSPARQL ontology

The ontology for representation of features and geometries allows writing queries and retrieving spatial data. It is based on the OGC Simple Features model with some adaptations for RDF and consists of the *geo:SpatialObject* class, which can be linked to any ontology of interest, and its two subclasses (*geo:Features* and *geo:Geometry*). *geo:Features* and *geo:Geometry* are connected via the *geo:hasGeometry* property.

For example, the entity "airport" is represented as *geo:Feature* and its coordinates as *geo:Geometry*. Depending on the purpose and function, there are different approaches for measuring coordinates - from a single point denoting the center of the area to a very detailed rectangle describing all points along the borders of an airport. In most of the cases, the used geometry will be represented as *geo:defaultGeometry*.

GeoSPARQL includes two different approaches for representing geometry literals and their associated hierarchy types – WKT and GML. The default coordinate system for WKT is WGS84. The implementation of a given triple store can support either one or both of these types. GeoSPARQL provides different OWL classes of the geometry hierarchies associated with these geometry representations. Classes for many different types of geometries such as points, rectangles, curves, and arcs are provided. The *geo:asGML* property connects geometry entities to the geometry representation literals. The property values use the *geo-sf:WKTLiteral* and *geo-gml:GMLLiteral* types.

GeoSPARQL also supports a standard for representing topological relations such as overlapping between spatial entities. They come in the form of binary properties between entities and geospatial functions. The topological binary properties can be used in a SPARQL query like a normal property, mainly between objects of Geometry type. However, they can also be used between objects of type Features or between Features and Geometries, if GeoSPARQL query rewriting is supported. The properties can be represented by three different vocabularies – OGC's Simple Features, Egenhofer's 9-intersection model, and RCC8. Which of them is supported depends on the triple store implementation although usually they all are. Simple Futures topological relations include: equals, disjoint, intersects, within, contains, overlap, and crosses.

The filtering functions provide two different types of functionalities. The first type is an operator function that takes multiple geometries such as predicates and produces a new geometry or a new data type as a result. *ogcf:intersection* is an example of such a function. It takes two geometries and returns a new one representing a spatial intersection. Another function such as *ogcf:distance* produces *xsd:double* as a result. The second type of functions provide a Boolean topological test of geometries. They come with the same three vocabularies as binary topological properties – simple features, topological relations, Egenhofer relations, and RCC8 relations. These functions are partially redundant because of the topological binary properties but the topological functions take geometry literals as parameters, while the binary properties link Geometry and Features entities. This means that quantitative and qualitative applications can use topological functions. Comparison between concrete geometry provided in a query can also be done only by a function. One example of a topological function is *ogcf:intersects*, which is true, if two geometries overlap.

5.1.2 Rules for query transformation

The rules for query re-writing allow an additional abstraction layer in the SPARQL queries. While only the Geometry entities can be quantitatively compared, sometimes we have to consider whether there is a topological connection between two entities. In natural language this is represented by the question: "Is Reagan National airport in Washington, DC?" Although Reagan National is referred to as a Washington DC airport, it is actually across the Potomac River in Virginia. In GeoSPARQL, *geo:Feature* to *geo:Feature* and *geo:Geometry* to *geo:Feature* topological relations are represented as a combination of *geo:defaultGeometry* property and the rules for query re-writing. If *geo:Feature* is used as a subject or an object of the topological relation, the query is automatically rewritten to compare the geo:Geometry linked as a default, thus removing the abstraction for processing. For example:

#Before

```
ASK {
```

```
    ex:DC a geo:Feature;
    geo:within ex:WashingtonDC.
    ex:WashingtonDC a geo:Feature.
}
```

#After

```
ASK{
    ex:DCA a geo:Feature;
          geo:defaultGeometry ?g1.
    ex:WashingtonDC a geoFeature;
          geo:defaultGeometry ?g2.
    ?g1 geo:within ?g2

}
```

Thus, we provide a more intuitive approach for creating geospatial queries, especially in the cases where we do not need many different geometries.

5.2 Implementation in MULTISENSOR

In MULTISENSOR, support for GeoSPARQL has been implemented as a GraphDB plugin. The plugin provides support for the following GeoSPARQL conformance classes:

Conformance class	Description	Note
Core	Defines top-level spatial vocabulary components.	General SPARQL support through GraphDB. Support for the classes *geo:SpatialObject* and *geo:Feature*.
Topology vocabulary extension	Defines topological relation vocabulary.	Support for all Simple Features properties (*geo:sfxxx*). Support for all Egenhofer properties (*geo:ehxxx*). Support for all RCC8 properties (*geo:rcc8xxx*).

Conformance class	Description	Note
Geometry extension	Defines geometry vocabulary and non-topological query functions.	Support for the class *geo:Geometry*. Support for the *geof:getSRID* function. Support for WKT literals with default CRS http://www.opengis.net/def/crs/OGC/1.3/CRS84 and all CRSs defined in the EPSG Geodetic parameter Dataset. Support for the Geometry serialization property *geo:asWKT*. Partial support for the GML literals supporting a subset of GML 3.1.1 with the same CRSs as those for WKT. Support for the Geometry serialization property *geo:asGML*.
Geometry topology extensions	Defines topological query functions for geometry objects.	Support for the *geof:relate* function. Support for all Simple Features functions (*geof:sfxxx*). Support for all Egenhofer functions (*geof:ehxxx*). Support for all RCC8 functions (*geof:rcc8xxx*)
RDFS entailment extension	Defines a mechanism for matching implicit RDF triples that are derived based on RDF and RDFS semantics.	RDF and RDFS support through GraphDB. Support for WKT and GML geometry classes as defined by their respective ontologies.
Query rewrite extension	Defines query transformation rules for computing spatial relations between spatial objects based on their associated geometries.	Support for all Simple Features rules (*geor:sfxxx*). Support for all Egenhofer rules (*geor:ehxxx*). Support for all RCC8 rules (*geor:rcc8xxx*).
Custom literals-in-relations extension	Defines a custom extension that allows the use of WKT and GML literals in place of a *geo:SpatialObject*	GeoSPARQL requires *geo:SpatialObject* for queries with spatial relation properties. The extension allows for specifying a literal geometry serialization at the object position.

Table 6: GeoSPARQL conformance classes

5.2.1 Query optimization

The query re-write extension implementation uses an optimized GraphDB approach with an internal custom Lucene index to boost the performance with SPARQL queries that involve spatial relations properties. All RDF data that conforms to GeoSPARQL will be automatically synchronized with the Lucene index as part of the internal *INSERT* or *DELETE* operations.

In order to benefit from the Lucene index optimization, you have to use the spatial relations properties and not the spatial relations functions. For example, the index will be used in the query: "find all Features that are within <urn:Europe>"

```
select ?x where {
   ?x a geo:Feature .
   ?x geo:sfWithin <urn:Europe>
}
```

and will not be used in the query: "find all Features that are within <urn:Europe>", although it will return the same results.

```
select ?x where {
   ?x a geo:Feature .
   ?x geo:hasDefaultGeometry ?xg .
   ?xg geo:hasSerialization ?xgLit .
   <urn:Europe> geo:hasDefaultGeometry ?eg .
   ?eg geo:hasSerialization ?egLit .
   filter(geo:sfWithin(?xgLit, ?egLit))
}
```

5.2.2 Custom literals-in-relations extension

Standard GeoSPARQL supports only *geo:SpatialObject* (either *geo:Feature* or *geo:Geometry*) for the subject and object of spatial relations properties. The GeoSPARQL GraphDB plugin provides an extension to the standard that allows for Geometry WKT or GML literals to be used in the object position. This simplifies some queries and provides an easy way to use the Lucene index with geometries supplied at query time, for example:

find all Features that are *roughly* within Bulgaria (based on the supplied polygon that includes parts of neighboring countries too)

```
select ?x where {
   ?x a geo:Feature .
   ?x geo:sfWithin "Polygon((22 41, 29 41, 29 45, 22 45, 22
41))"^^geo:wktLiteral .
   }
```

5.2.3 Third-party libraries used

The GeoSPARQL plugins relies on the following third-party libraries:

- The JTS Topology Suite, an API of 2D spatial predicates and functions, version 1.13, LGPL;

- Geotoolkit.org, a library for developing geospatial applications, version 3.20, Apache license;
- OGC Tools GML-JTS, a GML parser for Java that supports JTS, version 1.0.3, a modified BSD license.

In addition, some code has been imported from uSeekM, a seemingly obsolete Java library that integrates JTS and Geotoolkit.org, version 1.2.2-SNAPSHOT, Apache License.

5.3 Evaluation

This release of the GeoSPARQL plugin has undergone only basic conformance testing.

6 CONCLUSIONS

In this report we have presented the development of four different types of reasoning – hybrid reasoning, parallel multi-thread reasoning, SPARQL-MM, and GeoSPARQL.

Hybrid reasoning combines the strengths of forward- and backward-chaining. It is useful for scenarios where a goal-driven reasoning is applied over a large amount of data. Hybrid reasoning is particularly common in decision support systems and semantic data integration from multiple sources, involving multiple ontologies. The results we achieved with the backward chaining prove that we can successfully trade query speed for loading speed (for ~50M datasets, at least), although some queries might become non-performant.

The parallel multi-threaded reasoning reduces the loading times from 50% to 200% compared to loading with a serial inference engine.

SPARQL-MM is a proposed SPARQL extension that enables reasoning and query answering over media files. Our implementation is very simple and does not integrate well with our Query Optimizer. Lack of available public data sets of annotated videos currently prevents the evaluation of our implementation.

GeoSPARQL defines a powerful language for expressing geo-spatial constraints, e.g. overlapping or proximity of two geographical regions. Our previous proprietary implementation supported only limited number of simple constructs for geospatial RDF querying and also it was not dynamic (i.e. inserting new data did not automatically update the indexes). This new GeoSPARQL implementation is dynamic, conforms to the standard and at the same time allows for efficient query execution plans.

The updates on the reasoning techniques including also evaluation will be reported in D5.3 and D5.4 including also assessment according to the performance indicators discussed in D1.1.

7 REFERENCES

Urbani, J., Maaseen, J., & Bal, H. (2010), Massive Semantic Web data compression with MapReduce, In Proceedings of the MapReduce workshop at HPDC '10.

Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., Teije, A., & van Harmelen, F. (2009), MARVIN: A platform for large scale analysis of Semantic Web data, In Proceedings of the WebSci'09: Society On- Line.

Kurz, T., Schaffert, S., Schlegel, K., Stegmaier, F., & Kosch, H. (2014). Sparql-mm-extending sparql to media fragments. In The Semantic Web: ESWC 2014 Satellite Events (pp. 236-240). Springer International Publishing.

Battle, R. & Kolas D. (2012) Enabling the Geospatial Semantic Web with Parlament and GeoSPARQL. Semantic Web 3(4): 355-370

SPB (2015), Semantic Publishing Benchmark - **http://ldbcouncil.org**

Urbani, J., Piro, R., van Harmelen, F., Bal, E. (2014) Hybrid reasoning on OWL RL. Semantic Web 5(6): 423-447