Big Data to Enable Global Disruption of the Grapevine-powered Industries

D4.2 - Methods and Tools for Distributed Inference

DELIVERABLE NUMBER	D4.2
DELIVERABLE TITLE	Methods and Tools for Distributed Inference
RESPONSIBLE AUTHOR	Nikola Tulechki (ONTOTEXT)

Co-funded by the Horizon 2020
Framework Programme of the European Union

GRANT AGREEMENT N.	780751
PROJECT ACRONYM	BigDataGrapes
PROJECT FULL NAME	Big Data to Enable Global Disruption of the Grapevine-powered industries
STARTING DATE (DUR.)	01/01/2018 (36 months)
ENDING DATE	31/12/2020
PROJECT WEBSITE	http://www.bigdatagrapes.eu/
COORDINATOR	Nikos Manouselis
ADDRESS	110 Pentelis Str., Marousi, GR15126, Greece
REPLY TO	nikosm@agroknow.com
PHONE	+30 210 6897 905
EU PROJECT OFFICER	Ms. Annamária Nagy
WORKPACKAGE N. | TITLE	WP4 | Analytics & Processing Layer
WORKPACKAGE LEADER	CNR
DELIVERABLE N. | TITLE	D4.2 | Methods and Tools for Distributed Inference
RESPONSIBLE AUTHOR	Nikola Tulechki (Sirma AI)
REPLY TO	nikola.tulechki@ontotext.com
DOCUMENT URL	http://www.bigdatagrapes.eu
DATE OF DELIVERY (CONTRACTUAL)	31 July 2020 (M31)
DATE OF DELIVERY (SUBMITTED)	31 July 2020 (M31)
VERSION | STATUS	3.0 | Final
NATURE	DEM (Demonstrator)
DISSEMINATION LEVEL	PU (Public)
AUTHORS (PARTNER)	Milena Yankova (ONTOTEXT), Boyan SImeonov (ONTOTEXT), Atanas Kiryakov (ONTOTEXT), Vladimir Alexiev (ONTOTEXT)
CONTRIBUTORS	Nicola Tonellotto (CNR), Raffaele Perego (CNR), Raffaele Perego (CNR), Pythagoras Karampiperis (Agroknow), Plamen Tarkalanov (Sirma AI)
REVIEWER	Giannis Stoitshs (Agroknow)

VERSION	MODIFICATION(S)	DATE	AUTHOR(S)
0.1	First draft	14/08/2018	Milena Yankova (ONTOTEXT)
0.2	Final draft for review	17/09/2018	Milena Yankova (ONTOTEXT)
0.3	Comments from CNR	18/09/2018	Nicola Tonellotto (CNR), Raffaele Perego (CNR), Raffaele Perego (CNR)
0.4	Added Implementation Section	19/09/2018	Milena Yankova (ONTOTEXT)
0.7	Input from partners	21/09/2018	Nicola Tonellotto (CNR), Raffaele Perego (CNR), Raffaele Perego (CNR), Pythagoras, Karampiperis (Agroknow)
0.9	Internal Review	24/09/2018	Antonis Koukourikos (Agroknow)
1.0	Final edits after internal review	28/09/2018	Milena Yankova (ONTOTEXT), Boyan SImeonov (ONTOTEXT), Atanas Kiryakov (ONTOTEXT), Vladimir Alexiev (ONTOTEXT)
2.0	Draft update V2	08/03/2019	Milena Yankova (ONTOTEXT), Aleksander Daskalov (ONTOTEXT), Yasen Marinov (ONTOTEXT)
2.1	Internal review	18/03/2019	Nicola Tonellotto (CNR)
2.2	Final V2	27/03/2018	Milena Yankova (ONTOTEXT)
2.8	Draft update V3	17/08/2020	Nikola Tulechki (Sirma AI)
2.9	Internal review	17/08/2020	Giannis Stoitshs (Agroknow)
3.0	Final Version V3	30/08/2020	Nikola Tulechki (ONTOTEXT)

PARTICIPANTS		CONTACT
Agroknow IKE (Agroknow, Greece)		Nikos Manouselis Email: nikosm@agroknow.com
Ontotext AD (ONTOTEXT, Bulgaria)		Todor Primov Email: todor.primov@ontotext.com
Consiglio Nazionale Delle Ricerche (CNR, Italy)		Raffaele Perego Email: raffaele.perego@isti.cnr.it
Katholieke Universiteit Leuven (KULeuven, Belgium)		Katrien Verbert Email: katrien.verbert@cs.kuleuven.be
Geocledian GmbH (GEOCLEDIAN Germany)		Stefan Scherer Email: stefan.scherer@geocledian.com
Institut National de la Recherché Agronomique (INRA, France)		Pascal Neveu Email: pascal.neveu@inra.fr
Agricultural University of Athens (AUA, Greece)		Katerina Biniari Email: kbiniari@aua.gr
Abaco SpA (ABACO, Italy)		Simone Parisi Email: s.parisi@abacogroup.eu
SYMBEEOSIS EY ZHN S.A. (Symbeeosis, Greece)		Konstantinos Rodopoulos Email: rodopoulos-k@symbeeosis.com

ACRONYMS LIST

BDG	BigDataGrapes
DL	Description Logic
LP	Logical Programming
OWL	Ontology Web LanguageQuery-subquery
QSQ	Query-Subquery
RDDs	Resilient Distributed Datasets
RDBMS	Relational Database Management Systems
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
MSC	The Most Significant Change Technique
SWRL	Semantic Web Rule Language
SPARQL	Symantec Protocol and RDF Query Language
W3C	World Wide Web Consortium

EXECUTIVE SUMMARY

The objective of this deliverable is to develop inference methods that support efficient information selection from heterogeneous data pools. There are many challenges in data reasoning and inference based on distributed data. The first one is addressing data security and access rights to both original data and inferred information. The second challenge is how the actual inference over distributed sources can be performed and implemented. We address the main principles applied to data inference and different types of inference – rule-based, query-based, model-based and fuzzy inference – and their application in BigDataGrapes project. The Final section is dedicated to state of the art with standard theoretical approach to inference from descriptive logic stand point, as well as related work in implementing those approaches.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The objective of this deliverable is to develop inference methods that support efficient information selection from heterogeneous data pools. Further specification will enable implementation on top of the BigDataGrapes database layer including a semantic graph database (a type of NoSQL 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.

There are many challenges in data reasoning and inference based on distributed data. The most prominent one is addressing data security and access rights to both original data and inferred information. To address data security, we follow the industry business need of building the missing piece is the universal semantic data layer. Dave Mariani, co-founder and CEO of startup AtScale and former vice president of development, user data and analytics at Yahoo formulates it:

"You can define security on the data lake itself … anyone who logs in and runs queries on the data lake is going to be secured at the data bit level rather than at the application that's using it. Now data is being secured as it's written as opposed to as it's used. You can't do that if you're sending data extracts out to the business and the business is dealing with it on its own."

The second challenge of how the actual inference over distributed sources can be performed, in BigDataGrapes project we do not limit ourselves to any specific reasoning technique. Approximate reasoning is a non-standard reasoning approach based on the idea of sacrificing soundness or completeness for a significant speed-up in reasoning. This is done in such a way that the loss of correctness is at least outweighed by the obtained speed-up. Parallel reasoning and distributed reasoning are considered to be essential for Web-scale reasoning to improve scalability. Stream reasoning provides the reasoning support in which memory overload is avoided by operating on streams of data instead of statically available sets. Granular reasoning is a non-standard reasoning approach in which multiple perspectives/views can be selected for reasoning by using knowledge at various levels of specificity and data at variable levels of granularity.

We aim to explore the state of the art and construct possibly several reasoning plug-ins, based on insights from both generic inference methods and non-standard reasoning, and invite third parties to contribute further components to the BigDataGrapes ecosystem.

2. TYPES OF INFERENCE

This section addresses the main principles applied to data inference and it is an attempt for drawing a roadmap including their major characteristics, related design and performance issues, the state of the art in the field and future directions. The major objectives are:

- to clarify the principles of operation of the inference and the potential of its distribution;
- to explain the facets of their performance, because we believe that their understanding is a key factor for the successful adoption of distributed inference.

The context in which we review types of inference and their distribution potential is addressed in one or more of the following goals:

- to handle efficiently larger volumes of data;
- to speed up the data loading and indexing and to improve the performance for updates;
- to lower the query evaluation time for complex queries (e.g. analytical Business Intelligence reports);
- to better handle concurrent query loads and large numbers of users and
- to ensure failover, e.g. to surmount failure of one or more nodes and repositories.

The reminder of this section provides discussion on the different approaches, their advantages and disadvantages and appropriateness with respect to different scenarios and goals.

2.1. RULE-BASED INFERENCE

Broadly speaking, inference can be characterised by discovering new relations (see Fig1.). On the Semantic Web, data is modelled 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 analysing the content of data, or managing knowledge in general. Inference-based techniques are also important for discovering possible inconsistencies in the data.

Inference is performed by semantic repositories - database management systems - which are capable of handling structured data, taking into consideration their semantics as well as rules for interpretation. To foster their realisation, the World Wide Web Consortium (W3C) developed a series of metadata, ontology, and query language standards. The standardisation efforts related to the Semantic technology, most notably RDF(S), OWL, and SPARQL, provided a solid ground for development and good minimal levels of interoperability. Following the enthusiasm and the wide adoption of the related standards, today, most of the semantic repositories are database engines, which deal with data represented in RDF, support SPARQL queries, and can interpret schemata and ontologies represented in RDFS and OWL. Naturally, such engines take the role of web servers of the Semantic Technology.

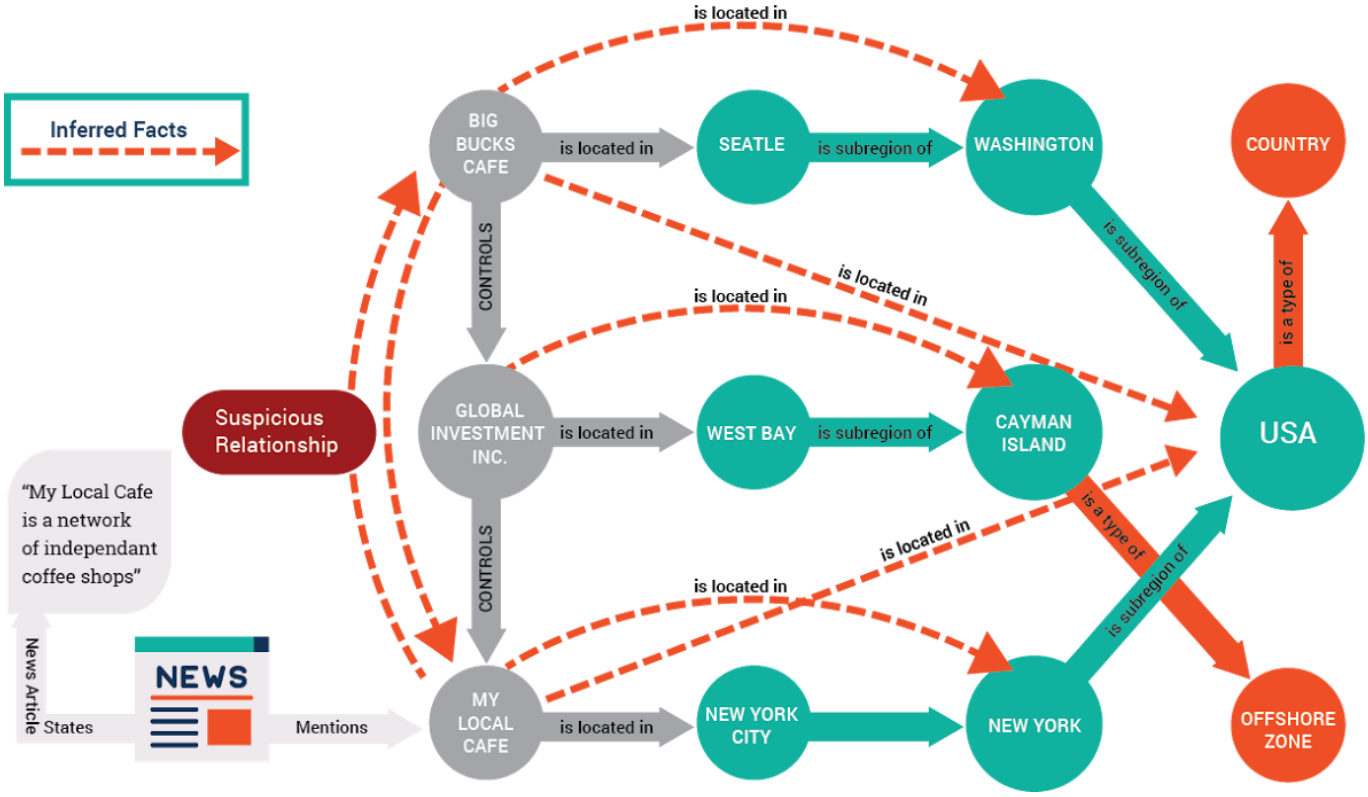


Figure 1. Rule based inference of transitive relations "is located in"

The logical inference over RDF datasets and their implementation in RDF triple stores or semantic graph databases follow one of the 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 materialisation is defined as a procedure that keeps an up-to- date inferred closure of the knowledge base.

Materialisation is known as a technique for applying inference before query evaluation. This allows for many query optimisations approaches to be forward-chaining as querying is realised by lookups in the database. The main drawback of materialisation is that the database changes, additions, and updates are generally slow operations. In many scenarios, the materialisation of such frequent changes does not affect the querying process, as many of the materialised 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 materialisation over the knowledge base. Unfortunately, backward-chaining is inefficient for large knowledge bases, as many optimisations for the materialisation of a knowledge base are not possible.

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 (2013). 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 materialisation over the set is stored in the semantic graph database in an optimal way. The other is used to support a backward-chaining strategy. It applies the materialisation only when it is necessary.

2.2. DISTRIBUTED RULE-BASED INFERENCE

Distributed 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 - forward-chaining, 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.

Some of the obstacles in distributing inference at scale come from the data volume. Data in the semantic representation paradigm are made of terms that are either URIs or literals. Since these terms usually consist of long sequences of characters, an effective compression technique must be used to reduce the data size and increase the application performance. 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. One of the best-known techniques for data compression is dictionary encoding and MapReduce algorithm efficiently compresses and decompresses a large amount of Semantic Web data, giving a compression ratio of about 1:6 to 1:8. This compression approach allows for using parallel processing.

The expressiveness of the ontology language and complexity of the rules is another challenging area for distribution. For example, Oren et. al (2009) shows that partitioning of an RDF database into independent parts is not trivial in regard to soundness and completeness of the reasoning or results in communication overload between the different partitions.

Some authors propose additional restrictions on the language expressivity to cope with the problem. For example, Priaya et. al (2014) define an ABox independent partitioning, which supports reasoning in OWL Lite knowledge bases. Further work in this direction by Shrinoshita et. al (2017) evaluates enhanced MSC method over random graph theory that results in very small tractable concepts provided that the number of role assertions are removed from consideration is large enough.

2.3. QUERY-BASED INFERENCE

The ability to abstract the query syntax from the data syntax bears important advantages in data access scenarios where one has to deal with complex relationships or with schema diversity. As long as the semantic repositories can interpret the semantics in a recursive fashion, one can enjoy interpretations of the data, which combine results from previous interpretations and explicit assertions. In other words, depending on the data patterns and the semantics, one can retrieve facts, which are results of multiple steps of interpretation, and this way to uncover relationships which would otherwise remain hidden.

The standardized way of distributed query inference is to use SPARQL 1.1 Federated Query extension for executing queries distributed over different SPARQL endpoints. The SERVICE keyword extends SPARQL 1.1 to support queries that merge data distributed across the Web, and the inference should follow the backward-chaining strategy implemented on the query level. This feature is very powerful and allows integration of RDF data from different sources using a single query. It is also possible to use the federation mechanism to do distributed querying over several repositories on a local server for managing security on data level.

The query-level inference is the most expressive mechanism for inference as it can use the full power of SPARQL and for defining rules (as SELECT) statements with filtering and exceptions. It can include custom functions and potentially wrap complex machine learning models as well.

2.4. MODEL-BASED INFERENCE

Scientists derive insights from models of complex systems by applying the models to address various types of prognostic queries. This can include, for example:

- Prediction: How will the system evolve in the near future?
- Conditional forecasting: How will the system respond if X changes?
- Counterfactual analysis: What would have happened if X had been Y?
- Comparative impact: What is the difference in utility between strategy X and strategy Y?
- Optimal planning: What is the optimal amount of X to introduce to maximise utility Y?
- Risk assessment: What is the risk of X?
- Outcome avoidance: What is the optimal action or intervention to reduce the risk of X decreasing more than Y?

Model-based inference can also be used diagnostically to test models against available data or knowledge through model checking, validation, and calibration. Automation of model-based inference procedures could increase the speed and accuracy with which these models can be used to address key questions of national security by orders of magnitude. Applications will include frequent update of user-specified queries as new data becomes available, rapid response to emerging natural disasters or other real-time threats, and even fully automated inference with machine-generated queries.

Model-based inference is predominantly based on machine learning techniques and depend very much on the available data features. As part of the initial research in BigDataGrapes will explore the available data sets in deliverable *D2.1 Use Cases & Technical Requirements Specification* before drawing any conclusions on the relevant techniques. This work is closely related to *D4.3 Methods and Tools for Scalable Distributed Processing*.

2.5. FUZZY INFERENCE

Fuzzy inference is the process of formulating the mapping from a given input to an output using fuzzy logic. It is classically applied in Fuzzy control systems to formalise the reasoning process of human language by means of fuzzy logic. It uses the "IF…THEN" rules along with connectors "OR" or "AND" for drawing essential decision rules.

Although alternative approaches such as genetic algorithms and neural networks can perform just as well as fuzzy logic in many cases, fuzzy logic casts to terms that human operators can understand and makes it easier to automate tasks that are already successfully performed by humans. State of the art implementation of Fuzzy Inference is provided by Mathworks.

It is still unclear if Fuzzy logic can be applied to any of the use cases in BigDataGrapes. Such decision can be made based on deliverable D2.1 Use Cases & Technical Requirements Specification and initial experiments using actual data provided by the use case partners.

3. STATE-OF-THE-ART

3.1. OWL DIALECTS SUITABLE FOR SCALABLE INFERENCE

In order to match the expectations for the next generation global Web of data, the Semantic Web requires scalable high-performance storage and reasoning infrastructure. One challenge towards building such an infrastructure is the expressivity of its schema and ontology definition standards RDFS and OWL. RDFS (Brickley and Guha, 2004) is the schema language for RDF, which allows for the definitions of subsumption hierarchies of classes and properties; the latter being binary relationships defined with their domains and ranges. While RDFS is generally a fairly simple knowledge representation language, implementing semantic repositories which support its semantics and provide performance and scalability comparable to those of relational database management systems (RDBMS) is very challenging.

The semantics of RDFS is based on Logical Programming (LP) – a declarative programming paradigm, in which the program specifies a computation by giving the properties of a correct answer. The LP languages like PROLOG emphasise the logical properties of a computation, using logic and proof procedures to define and resolve problems. Most logic programming is based on the Horn-clause logic with negation-as-failure to store the information and rule entailment to solve problems. Datalog is a query and rule language, a simplified version of PROLOG, meant to enable the efficient implementation of deductive databases. The semantics of RDFS is defined by means of rule entailment formalism, which is a simplification of Datalog.

OWL[1], (Dean and Schreiber, 2004) is an ontology language, which supports more comprehensive logical descriptions of the schema elements (see Fig.2), for instance: transitive, symmetric, and inverse properties; unions and intersections of classes; and property restrictions. The first version of the OWL specification, which was published as W3C standard in year 2004 has three dialects: OWL Lite, OWL DL and OWL Full. They range in their levels of expressivity. OWL Lite is a subset of OWL DL, and OWL DL is a subset of OWL Full. The OWL language is based on description logics (Baader et al, 2003).

The reasoning procedures of DLs are decision procedures that are aimed to always terminate – in mathematical logic terms this means that DLs are decidable. Compared to other logical languages DLs are relatively inexpressive. Still reasoning with DLs is based on satisfiability checking, which means that, in order to prove or to reject a specific statement, a DL reasoner needs to check whether it is possible or not to build a model of the world which satisfies a "theory" which includes this statement or its negation. For instance, suppose that there is a semantic repository which contains one billion statements and a client makes a query, checking whether specific resource is an instance of a specific class. In order to validate this, with respect to the semantics of OWL DL, a repository should add to its current contents the statement that the resource is not instance of the class and check whether the new state of the repository is consistent. It is clear that such semantics is impractical to implement for large volumes of data. Even the simplest dialect of OWL, OWL Lite is a DL formalism which does not support algorithms enabling efficient inference and query answering over reasonably large knowledge bases.

Logic programming and description logics support semantics and data interpretation capabilities of a different nature: LP uses rules to infer new knowledge, whereas DL employ descriptive classification mechanisms. None of these is more powerful or expressive than the other one – there are meaning aspects which can be expressed in each one of them, which cannot be expressed in a language from the other paradigm. As result, the semantics of OWL Lite and DL are incompatible with that of RDFS[2]. Although OWL was meant to be layered on top of RDFS in the Semantic Web specification stack, there is no "backward compatibility". In practical terms, this means that it may be impossible to "upgrade" an application that uses RDFS schemata to OWL, without replacing the schemata with OWL ontologies. The latter may require considerable changes in the semantics of the classes and the properties and in the data modelling principles used in the application.

1

2 The issues related to the interoperability and layering of the Semantic Web languages is also discussed in the introductory Chapter 1.

To bridge the gap of expressivity, compatibility and logical decidability and reach the goals of scalable inference, other dialects of OWL have been created which lay between RDF(S) and OWL Lite. o presents a simplified map of the expressivity or complexity of a number of these OWL-related languages together with their bias towards description logic and logical programming based semantics. The diagram provides a very rough idea about the expressivity of the languages, based on the complexity of entailment algorithms for them. A direct comparison between the different languages is impossible in many of the cases. For instance, Datalog is not simpler than OWL DL, it just allows for a different type of complexity.

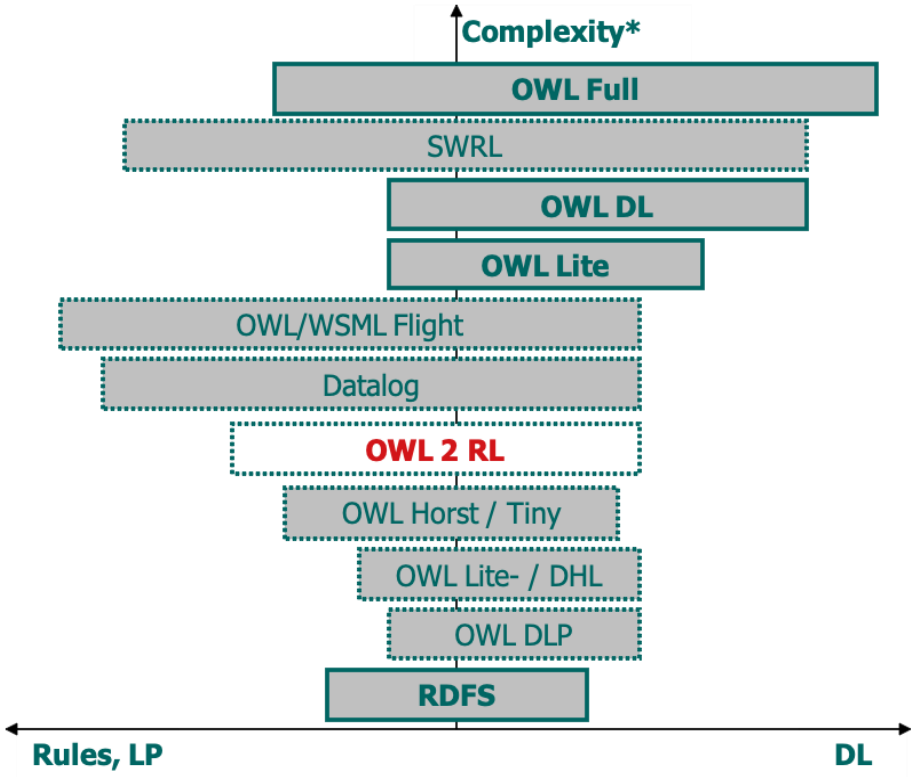


Figure 2. Diagram of expressivity of OWL dialects

OWL DLP is a non-standard dialect, offering a promising compromise between expressive power, efficient reasoning, and compatibility. It is defined in "Description Logic Programs: Combining Logic Programs with Description Logic" (Grosof et al, 2003) as the intersection of the expressivity of OWL DL and logical programming . In fact, OWL DLP is defined as the most expressive sub-language of OWL DL, which can be mapped to Datalog. OWL DLP is simpler than OWL Lite. The alignment of its semantics to the one of RDFS is easier, as compared to the Lite and DL dialects. Still, this can only be achieved through the enforcement of some additional modelling constraints and transformations. A broad collection of information related to OWL DLP can be found in "Ontology Logic and Reasoning at Semantic Karlsruhe"[3]. DLP has certain advantages:

- There is freedom to use either DL or LP (and associated tools and methodologies) for modelling purposes, depending on the modeller's experience and preferences.
- From an implementation perspective, either DL reasoners or deductive rule systems can be used. Thus, it is possible to model using one paradigm, e.g. a DL-biased ontology editor, and to use a reasoning engine based on the other paradigm, e.g. a semantic repository based on rules.

These features of DLP provide extra flexibility and ensure interoperability with a variety of tools. Experience with using OWL has shown that existing ontologies frequently use only very few constructs outside the DLP language.

[3] https://ieeexplore.ieee.org/document/7072815/

Ter Horst (2005) defines RDFS extensions towards rule support and describes a fragment of OWL, more expressive than OWL DLP. He introduces the notion of R-entailment of one (target) RDF graph from another (source) RDF graph on the basis of a set of entailment rules R. R-entailment is more general than the D-entailment used by Hayes (2004) in defining the standard RDFS semantics. Each rule has a set of premises, which conjunctively define the body of the rule. The premises are "extended" RDF statements, where variables can take any of the three positions. The head of the rule comprises one or more consequences, each of which is, again, an extended RDF statement. The consequences may not contain free variables, i.e. which are not used in the body of the rule. The consequences may contain blank nodes.

The extension of the R-entailment (as compared to the D-entailment) is that it "operates" on top of the so-called generalized RDF graphs, where blank nodes can appear as predicates. R-entailment rules without premises are used to declare axiomatic statements. Rules without consequences are used to imply inconsistency.

This extension of RDFS became popular as "OWL Horst". As outlined in "Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity" (ter Horst, 2005) this language has a number of important characteristics:

- It is a proper (backward-compatible) extension of RDFS. In contrast to OWL DLP, it puts no constraints on the RDFS semantics. The widely discussed meta-classes (classes as instances of other classes) are not disallowed in OWL Horst.
- Unlike the DL-based rule languages, like SWRL (Horrocks et al, 2005), R-entailment provides a formalism for rule extensions without DL-related constraints;
- Its complexity is lower than the one of SWRL and other approaches combining DL ontologies with rules of "Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity" (ter Horst, 2005).

OWL Horst is supported by GraphDB and ORACLE, which makes it the OWL dialect that has the largest industry support. An official OWL dialect with the same properties emerged recently under the name OWL 2 RL. The latter is one of the tractable profiles (dialects) defined in the specification of OWL 2 (Motik et al, 2009) – the next version of the OWL language that is currently in process of standardisation. OWL 2 RL is designed with the objective to be the most expressive OWL dialect which allows for efficient reasoning with large volumes of data in rule-based systems. OWL 2 RL was inspired by OWL Horst – its semantics is defined with rule language equivalent to R-entailment. However, OWL 2 RL is considerably more expressive than OWL Horst. Support for OWL 2 RL is provided by several reasoning engines, including GraphDB and ORACLE.

Recent research reported in "UniProt in RDF: Tackling Data Integration and Distributed Annotation with the Semantic Web" (Redaschi and the UniProt Consortium, 2009) evaluates the level of completeness of the inference supported by few inference engines (namely HAWK) and semantic repositories: IBM's Minerva, Sesame and GraphDB by ONTOTEXT. It demonstrates that although GraphDB supports sufficient reasoning to answer the LUBM[4] queries correctly, it is still not complete with respect to the semantics of the data and the queries.

3.2. Advantages of the Different Distribution Approaches

The general approaches for distribution of database management systems are:

- Data partitioning, where the information stored and accessed by the system is spread across multiple machines, so that none of them contains the entire dataset;
- Data replication, where the entire dataset resides on each of the machines.

Data replication is a traditional approach for boosting the read performance of a DBMS at the cost of redundancy and write propagation complexity. In a classic scenario, several slave nodes are assigned

[4] LUBM benchmark is introduced in D7.1

incoming read requests by a central master node that performs any sort of load balancing (e.g. round robin) to distribute the load evenly across the slaves. Writes are executed on the master node and updates are propagated to the slaves in the background. Such a setup is very appropriate in situations when a lot of read requests occur while write requests are rare or clustered together in large batches (for example if a large dataset is initially loaded in the repository). In such situations the resource-intensive replication procedure will not be necessary most of the time, while a theoretically linear scalability will take place on the reading side.

While data partitioning looks as the more promising schema, it is also the one which is most problematic to implement. In general, it enables the management of larger volumes of data and provides more space for in-memory data-structures. Each node can apply more efficient caching and optimisation with respect to the fraction of the data that it deals with. Data partitioning with redundancy also allows for failover support. Still, the major issue is that query evaluation against distributed data requires intensive communication between the nodes for exchanging of intermediate results; the most common variety of such communication is known as "remote join". Query optimisation schemata, which consider the communication costs, are far harder to implement, which triggers less-optimal query evaluation plans and larger overall numbers of index lookups. In large number of scenarios these effects neutralise the gains from the additional computing power gained from several machines. The same concerns are application for rule-based reasoning in repositories using data partitioning.

Two approaches to data partitioning appear in database systems from the established distributed DBMS research: horizontal and vertical partitioning. The horizontal data partitioning approach partitions a dataset across several repository nodes where no schema limitations apply to any of the nodes. A vertical partitioning approach would assign different parts of the data schema to different repository nodes, so, that later on requests for any type of data would be redirected to the respective repository node. This approach can be further extended and types of data that usually appear "close" together can be placed within a single node (when possible). In principle, such clustering would allow for whole sub-queries to be executed within a single node. It would therefore avoid the transfer of intermediate results between the repository nodes and the central query processing node only to complete the query.

As an overview of the two major distribution approaches we can summarise that:

- Data partitioning improves data scalability, however in most of the cases hampers the query evaluation performance due to high communication overheads. It can improve loading performance if there is no materialisation involved;

- Data replication allows for better handling of concurrent query loads and failover. It is neutral with respect to loading and inference performance.

None of these approaches provides a principal advantage for evaluation of complex queries. Under data replication one of the nodes can be off-loaded from concurrent queries, which would allow faster execution of a complex query. An approach known as "federated join" can in theory improve the performance of such queries in very specific data partitioning scenarios, where the communication costs can be minimized.

3.3. RELATED WORK

3.3.1. WEBPIE BY VU AMSTERDAM

"WebPIE (Web-scale Parallel Inference Engine)[5] is a MapReduce distributed RDFS/OWL inference engine written using the Hadoop framework. This engine applies the RDFS and OWL ter Horst rules and it materializes all the derived statements."

It's a stream of work starting from MSc thesis of Urbani (2009) supervised by Frank van Harmelen. Their notable achievement is performing materialization of RDFS on 100B statements using notable cluster of machines with high-speed connectivity funded as part of LaRCK[6] project.

The major concern about their approach is regarding the manually optimization of Map Reduce rules in a very special manner in order to avoid the pitfalls of remote joining, therefore implementing full OWL 2 RL this way is unfeasible.

The more-recent re-implementation of WebPIE (Kim and Parkis, 2015) based on SPARK with its Resilient Distributed Datasets (RDDs). They achieve lower scale (less than 1B) and proud of the fact that they doubled the speed 5 years later.

What evidence this provides for our deliverable:

- Distributed reasoning is only possible for logical languages with very limited expressivity
- Even for such languages, it requires tailor made inference flows crafted by human experts
- Although the results demonstrated in experimental setting were very impressive in terms of scalability, such systems appeared impractical to exploit in enterprise setting.
- This is way the project was abandoned and none of the commercial RDF engines with reasoning support adopted it.

3.3.2. RDFOX BY OXFORD / BORIS MOTIK

"RDFox[7] is a highly scalable in-memory RDF triple store that supports shared memory parallel datalog reasoning behind the founded in 2017 Oxford Semantic Technology[8] as university spin-off which aims to commercially exploit the technology. The most recent general article about the system (Kim and Park , 2015) is co-author by Zhe Wu – the architect of ORACLE's RDF support; ORACLE also have parallelized, but not distributed inference.

RDFox is not a fully-fledged triplestore as discussed in "RDFox: A Highly-Scalable RDF Store" (Motik et al, 2015), it still does not have full SPARQL support but uses owl:sameAs optimization and incremental delete of inferred statements.

Note: this is parallel inference, not distributed inference system.

What evidence this provides for our deliverable:

- Parallel inference is feasible and can speed up reasoning substantially, subject to very specific and careful implementation of lock-free data structures with low-level programming. This is why it is written in C++, not in Java

[5] http://few.vu.nl/~jui200/webpie.html

[6] https://cordis.europa.eu/project/rcn/85416_it.html

[7] https://cs.ox.ac.uk/isg/tools/RDFox/

[8] https://www.oxfordsemantic.tech/

- Parallel inference is way easier to implement than distributed reasoning. The core reason of this is that there is no data sharding and no "remote join" problem. In 2015 the RDFox team declared future work plans for implementation of distributed share-nothing reasoning system. 3 years later they haven't published any results in this direction

MULTI SENSOR write up on reasoning

Multisensor deliverable 5.4 (Simeonov et al, 2016) reports relevant results on parallel inference and provides evidence that a general-purpose commercial data store parallelization can speed up inference at least twice, both for small but tangled dataset like Wordnet and for 1B triples knowledge graph. Datasets used in Multisensor combine DBPedia, Bablenet, statistics and other business information and the implementation of parallel inference has become part of GraphDB since version 7.2. However, the practical results of GraphDB 8.x[9] show that on dataset like the one of LDBC SPB, the speed up is minimal, because in SPB's knowledge graph provides owl:sameAs mappings the reasoning implications of which limit the applicability of some of the parallelization techniques.

[9] https://ontotext.com/products/graphdb/benchmark-results/

4. IMPLEMENTATION PARADIGMS

In BigDataGrapes project we envision using GraphDB by ONTOTEXT as main semantic graph database. GraphDB is a highly-efficient and robust graph database with RDF and SPARQL support. GraphDB uses RDF4J[10] as a library, taking advantage of its APIs for storage and querying, as well as the support for a wide variety of query languages (e.g., SPARQL and SeRQL) and RDF syntaxes (e.g., RDF/XML, N3, Turtle).

The development of GraphDB is partly supported by SEKT[11], TAO, TripCom[12], LarKC[13], and other FP6[14] and FP7[15] European research projects[16].

Distributed inference engine for BigDataGrapes will be implemented as a set of external to GraphDB engines which use the APIs of the database to access the data in real time and synchronize inference indexes, power inference algorithms and provide provenance of the newly inferred fact.

4.1. GRAPHDB PLUGIN API

The most powerful access mechanism to GraphDB and its data layer is via GraphDB Plugin API. It is a framework and a set of public classes and interfaces that allow developers to extend GraphDB and build custom inference mechanism over distributed data space. These extensions are bundled into plugins, which GraphDB discovers during its initialisation phase and then uses to delegate parts of its query processing tasks. The plugins are given low-level access to the GraphDB repository data, which enables them to do their job efficiently. They are discovered via the Java service discovery mechanism, which enables dynamic addition/removal of plugins from the system without having to recompile GraphDB or change any configuration files.

Plugin API can be effectively used to modify, filter or enrich the final results of a request which is particularly suitable to query-based distributed inference. For each binding set that is to be returned to the GrapHDB client the implemented plugin modifies the binding set and return it. After a binding set is processed by a plugin, it is passed to the next plugin that has enabled post-processing. Finally, after all results are processed and returned, each plugin serves modified result set to the client.

Alternatively, Plugin API can be used for custom inference as it allows for processing of data update statement (forward-chaining inference) as it looks for patterns containing specific predicates. It works as during initialization the plugin register the predicates it is interested in and then GraphDB filters updates for statements using these predicates and notifies the plugin. Filtered updates are not processed further by GraphDB, but the particular implementation of the Plugin API must handle insert or delete operation.

Further documentation of how to implement Plugin API and provide plugin configuration can be found in GraphDB documentation[17].

4.2.RDF4J API EXTENSION OF GRAPHDB

Programmatically, GraphDB can be used via the RDF4J[18] Java framework of classes and interfaces. Documentation for these interfaces (including Javadoc[19]). Code snippets in the following sections are taken from (or are variations of) the developer-getting-started examples, which come with the GraphDB distribution.

[10] http://rdf4j.org/about/
[11] http://www.sekt-project.com/
[12] http://www.tripcom.org/
[13] http://www.larkc.org/
[14] http://cordis.europa.eu/fp6/
[15] http://cordis.europa.eu/fp7/
[16] http://ontotext.com/knowledge-hub/
[17] http://graphdb.ontotext.com/documentation/standard/plug-in-api.html#making-a-plugin-configurable
[18] http://rdf4j.org/about/
[19] http://docs.rdf4j.org/javadoc/latest/

RDF4J comprises a large collection of libraries, utilities and APIs. The important components for this section are:

- the RDF4J classes and interfaces (API), which provide a uniform access to the SAIL components from multiple vendors/publishers;
- the RDF4J server application.

With RDF4J 2, local repository configurations are represented as RDF graphs. To access remote repositories RDF4J server provides a Web application that allows interaction with repositories using the HTTP protocol. It runs in a JEE compliant servlet container, e.g., Tomcat, and allows client applications to interact with repositories located on remote machines. In order to connect to and use a remote repository, local repository manager must be replaced with a remote one.

The RDF4J HTTP server is a fully fledged SPARQL endpoint - the RDF4J HTTP protocol is a superset of the SPARQL 1.1[20] protocol. It provides an interface for transmitting SPARQL queries and updates to a SPARQL processing service and returning the results via HTTP to the entity that requested them.

4.3. GRAPHDB REMOTE NOTIFICATIONS

Remote notifications are powerful mechanism for maintaining distributed inference, where changes in the remote instances notify the inference engine for changes on data level which affect the inferred facts and indexes. GraphDB's remote notification mechanism provides filtered statement add/remove and transaction notifications for both local or remote GraphDB repositories. Subscribers for this mechanism use patterns of subject, predicate and object (with wildcards) to filter the statement notifications to only actionable ones in order to increase inference performance.

Apart from the native GraphDB notifications[21], RDF4J API provides such a mechanism implementing RepositoryConnectionListener which can be notified of changes in a NotifiyingRepositoryConnection. However, the GraphDB notifications API works at a lower level and uses the internal raw entity IDs for subject, predicate and object instead of Java objects. The benefit of this is that a much higher performance is possible. The downside is that the client must do a separate lookup to get the actual entity values and because of this, the notification mechanism works only when the client is running inside the same JVM as the repository instance.

[20] http://www.w3.org/TR/sparql11-protocol/

[21] http://graphdb.ontotext.com/documentation/standard/notifications.html

5. DISTRIBUTED INFERENCE WITH MONGODB

5.1. WHY TO CONSIDER MONGODB?

MongoDB is an open-source No-SQL database designed for ease of development and scaling, which provides high performance, high availability, and automatic scaling. In a nutshell, MongoDB is the distributed document database with one of the biggest developers and user community and it is part of the MEAN technology stack.

Similar to GraphDB, MongoDB uses a JSON and JSON-LD compatible data-interchange format - BSON - for modelling data as documents. The JSON-LD format is a hierarchical view of data, allowing more complex search queries for embedded/nested documents. It can be mapped to traditional RDF format and RDF use of namespaces fits well within MongoDB's BSON document storage model.

Furthermore, MongoDB allows for a more flexible approach to data and data changes that is closely aligned to the document and document annotation model, where a change in data models does not require re-indexing unless explicitly requested. MongoDB's general-purpose approach and explicit requirement to introduce indices for specific performance/query evaluations is a better modelling approach than classical RDF semantic representation for very large document/annotation data sets/models which are likely to change. This lowers the impact of schema changes and guarantees scalability and performance well beyond the throughput supported in GraphDB.

It should also be noted that MongoDB data modelling approach allows for simpler GraphDB mapping connector model/architecture than other connectors, e.g. ElasticSearch. Rather than defining explicit mapping of RDF property paths within GraphDB to JSON fields, as currently used within the GraphDB to ElasticSearch connector architecture, JSON-LD format of documents provides a simpler modelling approach where the graph references are provided as LD context within JSON files. RDF Quads can simply be converted to JSON-LD within the context of a named graph.

In BigDataGrapes we have use-cases (in WP8) with extreme scalability requirements and relatively simple data models (modelled in WP3) (i.e., tree representations of a document or image collection with its meta-data) that suggest usage of highly scalable document store in addition to GraphDB. Therefore, choosing the de facto state of-the-art distributed document store, our evaluation is focused on inference paradigms, instead of document store distributed implementation.

5.2. MAPPING APPROACHES FOR MONGODB DOCUMENTS IN JSON FORMAT TO RDF

We have identified three possible implementations:

1. By custom mappings via DocManager for each use case – this is the most powerful way since it poses virtually no limitation on the data replication.
2. By convention via DocManager – transforming documents to property graph model, which works well for simple documents only.
3. By convention via JSON-LD – this is probably the most elegant mapping since JSON-LD allows to annotate any JSON document with additional metadata used as tip for its transformation to RDF format. The tips become quite complex when namespace, context and URI patterns are combined altogether.

We have chosen to implement JSON-LD mapping approach. For example, a MongoDB document from the LDBC SPB benchmark (see 6 Evaluation of MongoDB Inference Scenarios) would look like the follows:

```
{
        "_id" : ObjectId("5bbde32067710d51290ef227"),
        "@graph" : [
                {
```

```
"@id" : "http://www.bbc.co.uk/things/1#id",
"@type" : "cwork:NewsItem",
"bbc:primaryContentOf" : [
        {
                "@id" : "bbcd:7#id",
                "bbc:webDocumentType" : {
                        "@id" : "bbc:HighWeb"
                }
        },
        {
                "@id" : "bbcd:8#id",
                "bbc:webDocumentType" : {
                        "@id" : "bbc:Mobile"
                }
        }
],
"cwork:about" : [
        {
                "@id" : "dbpedia:AccessAir"
        },
        {
                "@id" : "dbpedia:Battle_of_Bristoe_Station"
        },
        {
                "@id" : "dbpedia:Nicolas_Bricaire_de_la_Dixmerie"
        },
        {
                "@id" : "dbpedia:Bernard_Roberts"
        },
        {
                "@id" : "dbpedia:Bartolomé_de_Medina"
        },
        {
                "@id" : "dbpedia:Don_Bonker"
        },
        {
                "@id" : "dbpedia:Cornel_Nistorescu"
        },
        {
                "@id" : "dbpedia:Clete_Roberts"
        },
        {
                "@id" : "dbpedia:Mark_Palansky"
        },
        {
                "@id" : "dbpedia:Paul_Green_(taekwondo)"
        },
        {
                "@id" : "dbpedia:Mostafa_Abdel_Satar"
        },
        {
                "@id" : "dbpedia:Tommy_O'Connell_(hurler)"
        },
        {
                "@id" : "dbpedia:Ahmed_Ali_Salaad"
        }
],
"cwork:altText" : "thumbnail atlText for CW http://www.bbc.co.uk/context/1#id",
"cwork:audience" : {
        "@id" : "cwork:NationalAudience"
},
"cwork:category" : {
        "@id" : "http://www.bbc.co.uk/category/Company"
```

```
                        },
                        "cwork:dateCreated" : {
                                "@type" : "xsd:dateTime",
                                "@value" : "2011-03-19T13:02:55.495+02:00",
                                "@date" : ISODate("2011-03-19T11:02:55.495Z")
                        },
                        "cwork:dateModified" : {
                                "@type" : "xsd:dateTime",
                                "@value" : "2012-03-20T18:32:39.165+02:00",
                                "@date" : ISODate("2012-03-20T16:32:39.165Z")
                        },
                        "cwork:description" : " constipate meant breaking felt glitzier democrat's huskily breeding solicit
gargling.",
                        "cwork:liveCoverage" : {
                                "@type" : "xsd:boolean",
                                "@value" : "false"
                        },
                        "cwork:mentions" : {
                                "@id" : "geonames:2862704/"
                        },
                        "cwork:primaryFormat" : [
                                {
                                        "@id" : "cwork:TextualFormat"
                                },
                                {
                                        "@id" : "cwork:InteractiveFormat"
                                }
                        ],
                        "cwork:shortTitle" : " closest subsystem merit rebuking disengagement cerebrums caravans conduction
disbelieved might.",
                        "cwork:thumbnail" : {
                                "@id" : "bbct:1361611547"
                        },
                        "cwork:title" : "Beckhoff greatly agitators constructed racquets industry restrain spews pitifully
undertone stultification."
                }
        ],
        "@id" : "bbcc:1#id",
        "@context" : {
                "bbc" : "http://www.bbc.co.uk/ontologies/bbc/",
                "cwork" : "http://www.bbc.co.uk/ontologies/creativework/",
                "bbcc" : "http://www.bbc.co.uk/context/",
        }
}
```


Where:

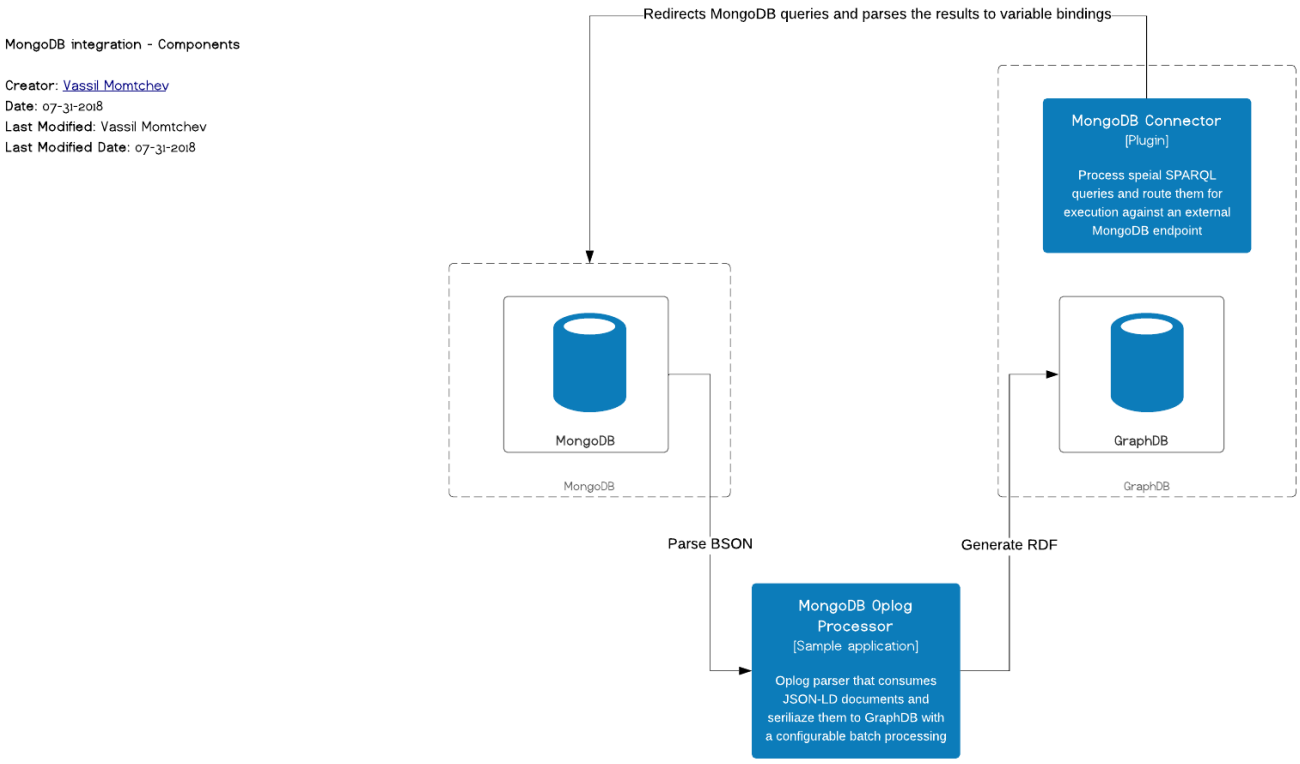
- @graph node represents the RDF context in the JSON-LD doc.
- _id key is a Mongo internal key.
- @type xsd:dateTime date has a @date key with a ISODate(...) value. This is not related to the JSON-LD standard and is ignored when the doc is parsed to RDF Model. The dates are extended for faster search/sorting. The ISODate in Mongo is its internal way to store dates and is optimized for searching. This step will make querying/sorting by this date field easier but is optional.

5.3. INTEGRATING MONGODB WITH GRAPHDB

Note: Querying MongoDB's JSON-LD serialized data using GraphDB plugin for MongoDB is publicly available since the release of GraphDB v. 8.0.0 (see http://graphdb.ontotext.com/documentation/standard/release-notes.html)

The integration between GraphDB and MongoDB is implemented as a GraphDB plugin (see 4.1 GraphDB Plugin API), which sends a request to MongoDB then transforms the result (which is expected to be a valid JSON-LD document) to RDF model (see the Architecture - component diagram below).

In order to be converted to RDF Models, the documents in Mongo should be represented as valid JSON-LD, where each document must be in separate context. This guarantees that the relationship between the statements in GraphDB and documents in MongoDB is preserved when selected documents are retrieved from MongoDB and made available for GraphDB in order to perform inference. Current implementation supports integration on the level of documents, while retrieval optimisation on selecting only relevant parts of documents is included in the future development plans.



5.4. CREATING AN INDEX

To configure GraphDB with MongoDB connection settings we need to set:

- The server where MongoDB is running;
- The port on which MongoDB is listening;
- The name of the database you are using;
- The name of the MongoDB collection you are using;

- The credetentials - (Optional unless you are using authentication) the username and password that will let you connect to the database.

Below is a sample query of how to create a MongoDB index:

```
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
INSERT DATA {
  inst:spb1000 :service "mongodb://localhost:27017" ;
    :database "ldbc" ;
    :collection "creativeWorks" .
}
```

Supported predicates:

- :service - MongoDB connection string;
- :database - MongoDB database;
- :collection - MongoDB collection;
- :user - (optional) MongoDB user for the connection;
- :password - (optional) the user's password;
- :authDb - (optional) the database where the user is authenticated;

5.5. Deleting an index

Deletion of an index is done using the following query:

```
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
INSERT DATA {
    inst:spb1000 :drop _:b .
}
```

5.6. Authentication

All types of authentication can be achieved by setting the credentials in the connection string. However, as it is not a good practice to store the passwords in plain text, the :user, :password and :authDb predicates are introduced. If one of those predicates is used, it is mandatory to set the other two as well. These predicates set credentials for SCRAM and LDAP authentication and the password is stored encrypted with a symmetrical algorithm on the disk. For x.509 and Kerberos authentication the connection string should be used as no passwords are being stored.

5.7. Querying MongoDB

Below is a sample query which returns the dateModified for docs with the specific audience:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>

SELECT ?creativeWork ?modified WHERE {
    ?search a inst:spb1000 ;
        :find '{"@graph.cwork:audience.@id" : "cwork:NationalAudience"}' ;
        :entity ?entity .
    GRAPH inst:spb1000 {
        ?creativeWork cwork:dateModified ?modified .
    }
}
```

	creativeWork	modified
1	http://www.bbc.co.uk/things/7#id	"2011-05-24T20:58:30.144+03:00"^^xsd:dateTime
2	http://www.bbc.co.uk/things/1#id	"2012-02-14T12:43:13.165+02:00"^^xsd:dateTime
3	http://www.bbc.co.uk/things/14#id	"2011-12-27T20:21:48.733+02:00"^^xsd:dateTime
4	http://www.bbc.co.uk/things/11#id	"2012-01-23T10:16:23.268+02:00"^^xsd:dateTime
5	http://www.bbc.co.uk/things/15#id	"2011-04-02T09:47:34.612+03:00"^^xsd:dateTime
6	http://www.bbc.co.uk/things/25#id	"2012-01-04T13:52:07.957+02:00"^^xsd:dateTime
7	http://www.bbc.co.uk/things/29#id	"2012-02-04T16:13:33.701+02:00"^^xsd:dateTime
8	http://www.bbc.co.uk/things/31#id	"2011-12-27T09:36:20.211+02:00"^^xsd:dateTime
9	http://www.bbc.co.uk/things/30#id	"2011-04-23T08:12:33.724+03:00"^^xsd:dateTime
10	http://www.bbc.co.uk/things/34#id	"2012-02-28T02:53:15.117+02:00"^^xsd:dateTime

In a query, use the exact values as in the docs. For example, if the full URIs are used instead of "cwork:NationalAudience" or "@graph.cwork:audience.@id" there wouldn't be any matching results.

The :find argument is a valid BSON document.

Note

The results are returned in a named graph to indicate when the plugin should bind the variables. This is an API plugin limitation. The variables to be bound by the plugin are in a named graph. This allows GraphDB to determine whether to bind the specific variable using MongoDB or not.

Supported predicates:

- :find - accepts single BSON and sets a query string. The value is used to call db.collection.find();
- :project - accepts single BSON. The value is used to select the projection for the results returned by :find. Find more info at MongoDB: Project Fields to Return from Query.
- :aggregate - accepts an array of BSONs. Calls db.collection.aggregate(). This is the most flexible way to make a MongoDB query as the find()method is just a single phase of the aggregation pipeline. The :aggregate predicate takes precedence over :find and :project. This means that if both :aggregate and :find are used :find will be ignored.
- :graph - accepts an IRI. Specifies the IRI of the named graph in which the bound variables should be. Its default value is the the name of the index itself.
- :entity - (**REQUIRED**) returns the IRI of the MongoDB document. If the JSON-LD has context, the value of @graph.@id is used. In case of multiple values, the first one is chosen and a warning is logged. If the JSON-LD has no context, the value of @id node is used. Even if the value from this predicate is not used, it is required to have it in the query in order to inform the plugin that the graph part of the current iteration is completed.
- :hint - specifies the index to be used when executing the query (calls cursor.hint())

5.8. MULTIPLE INDEX CALLS IN THE SAME QUERY

Multiple MongoDB calls are supported in the same query. There are two approaches:

- Each index call to be in a separate subselect (Example 1);

- Each index call to use different named graph. If querying different indexes, this comes out-of-the-box. If not, use the :graph predicate. (Example 2).

Example 1:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
SELECT ?creativeWork ?modified WHERE {
  {
    SELECT ?creativeWork ?modified {
      ?search a inst:spb1000 ;
        :find '{"@graph.@id" : "http://www.bbc.co.uk/things/1#id"}' ;
        :entity ?creativeWork .
      GRAPH inst:spb1000 {
        ?creativeWork cwork:dateModified ?modified ;
      }
    }
  }
  UNION
  {
    SELECT ?creativeWork ?modified WHERE {
      ?search a inst:spb1000 ;
        :find '{"@graph.@id" : "http://www.bbc.co.uk/things/2#id"}' ;
        :entity ?entity .
      GRAPH inst:spb1000 {
        ?creativeWork cwork:dateModified ?modified ;
      }
    }
  }
}
```

Example 2:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
SELECT ?creativeWork ?modified WHERE {
  {
    ?search a inst:spb1000 ;
        :graph :search1 ;
        :find '{"@graph.@id" : "http://www.bbc.co.uk/things/1#id"}' ;
        :entity ?creativeWork .
    GRAPH :search1 {
        ?creativeWork cwork:dateModified ?modified ;
    }
  }
  UNION
  {
    ?search a inst:spb1000 ;
        :graph :search2 ;
        :find '{"@graph.@id" : "http://www.bbc.co.uk/things/2#id"}' ;
        :entity ?entity .
    GRAPH :search2 {
        ?creativeWork cwork:dateModified ?modified ;
    }
  }
}
```

Both examples return the same result.

	creativeWork	modified
1	http://www.bbc.co.uk/things/1#id	"2012-02-14T12:43:13.165+02:00"^^xsd:dateTime
2	http://www.bbc.co.uk/things/2#id	"2011-11-27T13:17:29.243+02:00"^^xsd:dateTime

5.9. USING AGGREGATION FUNCTIONS

MongoDB has a number of aggregation functions such as: min, max, size, etc. These functions are called using the :aggregate predicate. The data of the retrieved results has to be converted to RDF model. The example below shows how to retrieve the RDF context of a MongoDB document.

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?s ?o {
  ?search a inst:spb1000 ;
    :aggregate '''[{"$match": {"@graph.@id": "http://www.bbc.co.uk/things/1#id"}},
        {'$addFields': {'@graph.cwork:graph.@id' :  '$@id'}}]''' ;
    :entity ?entity .
  GRAPH inst:spb1000 {
    ?s cwork:graph ?o .
  }
}
```

The $addFields phrase adds a new nested document in the JSON-LD stored in MongoDB The newly added document is then parsed to the following RDFstatement:

<http://www.bbc.co.uk/things/1#id> cwork:graph <http://www.bbc.co.uk/context/1#id>

We retrieve the context of the document using the cwork:graph predicate.

This approach is really flexible but is prone to error.

Let's examine the following query:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?g1 ?g2 {
  ?search a inst:spb1000 ;
    :aggregate '''[{"$match": {"@graph.@id": "http://www.bbc.co.uk/things/1#id"}},
        {'$addFields': {'@graph.inst:graph.@id' :  '$@id'}}]''' ;
    :entity ?entity .
  GRAPH inst:spb1000 {
    OPTIONAL {
      ?s inst:graph ?g1 .
    }
    ?s <inst:graph> ?g2 .
  }
}
```

It looks really similar to the first one except that instead of @graph.cwork:graph.@id we are writing the value to @graph.inst:graph.@id and as a result ?g1will not get bound. This happens because in the JSON-LD stored in MongoDB we are aware of the cwork context but not of the inst: context. In this way ?g2 will get bound instead.

5.10. CUSTOM FIELDS

Example:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>

SELECT ?size ?halfSize {
  ?search a inst:spb1000 ;
    :aggregate '''[{"$match": {"@graph.@type": "cwork:NewsItem"}},
         {"$count": "size"},
         {"$project": {"custom.size": "$size", "custom.halfSize": {"$divide": ["$size", 2]}}}]''' ;
    :entity ?entity .
  GRAPH inst:spb1000 {
    ?s inst:size ?size ;
    inst:halfSize ?halfSize .
  }
}
```

6. EVALUATION OF MONGODB INFERENCE SCENARIOS

We envision three scenarios for distributed inference in the context of integration of GraphDB with MongoDB:

- Scenario 1: Backward-chaining query rewrite
- Scenario 2: Forward-chaining materialisation in MongoDB
- Scenario 3: Forward-chaining materialisation in GraphDB

In the following sessions we describe each of the scenarios and discuss their advantages and disadvantages

6.1. EVALUATION METHODOLOGY

To comply with industry standards for performance measurement of implementation strategies , we have evaluated distributed inference using MondoDB in accordance to Linked Data Benchmark Council (LDBC) Semantic Publishing Benchmark. LDBS is established as an independent authority responsible for specifying benchmarks, benchmarking procedures and verifying/publishing results for software systems designed to manage graph and RDF data.

In particular we have chosen Semantic Publishing Benchmark v2.0 (SPB) of LDBC . SPB is inspired by the Media industry imitating the BBC's Dynamic Semantic Publishing approach. Its evaluation scenario considers large volume of streaming content, namely news, buts it is applicable to all forms of media assets. This content is provided as enriched asset with metadata which describes its links to reference data – rich semantic knowledge graph, ontologies and taxonomies - that include relevant concepts, entities and factual information. Actual content enrichment and generation of its metadata is out-of-scope of the benchmark and is provided primer to the evaluations.

Both screaming nature of the data and the rich semantic metadata of the SPB, as well as the fact that it is especially designed for RDF database engines, make it particularly relevant for evaluating integration scenarios of GraphDB and MongoDB in BigDataGrapes context.

The main interactions with the repository that are measured are (i) updates, that add new metadata or alter it, and (ii) queries, that retrieve content according to various criteria. Queries in SBP measure the efficiency of retrieving relevant content according to twelve pre- defined information questions that require inference.

The SBP is provided as synthetic data set generator (software component) and set of 12 queries to be executed against the generated data set. The original code base of 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).

To allow using the same data set and queries to measure distribution of data between GraphDB as RDF and MongoDB as JSON_LD and to finally measure the efficiency of inference our experiments required modifications in both: the dataset generator as well as on query syntax to query both GraphDB and MongoDB for retrieve the data for each information question (query).

Each of the scenarios we have evaluated required different approach for data and query distribution among GraphDB and MongoDB.

Evaluation is performed with a dataset of 256 million documents, written by 4 agents and queried by 8 agents. Initial set of tests – insert, update and delete are limited to 600 seconds.

6.2. SCENARIO 1: BACKWORD-CHAINING QUERY REWRITE

This is the simplest scenario of all the three ones in focus of this evaluation. Here there is a clear separation between knowledge graph and streaming data - knowledge graph is stored in GraphDB, while the streaming data is modelled as documents in MongoDB. The links between the two are implemented as using persistent URIs for concepts both in GraphDB and within documents. No inferences is materialised at the data insertion time as the links are virtual. Therefore, this scenario uses backword-chaining (see 2.3 Query-based inference). After the knowledge graph is loaded in GraphDB data insertion is performed on a single step:

Step 1. Each asset/streaming data object along with its metadata is modelled as JSON-LD document and it is inserted into MongoDB.

The **advantage** of this scenario is in data ingestion time. As there is no inference at ingestion time, and no duplication (de-normalisation) of any data, writes are extremely fast (see Evaluation results analysis).

While on the query time this scenario may perform as well as the other two, the main **disadvantage** is seen in queries that rely in inferenced data and data aggregations.

6.3. SCENARIO 1 EVALUATION RESULTS:

Seconds: 600
2019-03-22 11:43:08 (completed query mixes: 8)

Editorial:

4 agents

143629 inserts (avg : 6 ms, min : 1 ms, max : 606 ms)
17887 updates (avg : 34 ms, min : 6 ms, max : 3776 ms)
17950 deletes (avg : 32 ms, min : 4 ms, max : 3770 ms)

179466 operations (143629 CW Inserts (0 errors), 17887 CW Updates (0 errors), 17950 CW Deletions (0 errors))
299.1100 average operations per second

Aggregation:

8 agents

9 Q1 queries (avg : 102 ms, min : 5 ms, max : 424 ms, 0 errors)
11 Q2 queries (avg : 11 ms, min : 4 ms, max : 49 ms, 0 errors)
10 Q3 queries (avg : 36 ms, min : 4 ms, max : 161 ms, 0 errors)
10 Q4 queries (avg : 10 ms, min : 3 ms, max : 29 ms, 0 errors)
10 Q5 queries (avg : 1930 ms, min : 5 ms, max : 12334 ms, 0 errors)
11 Q6 queries (avg : 178 ms, min : 46 ms, max : 699 ms, 0 errors)
9 Q7 queries (avg : 363 ms, min : 89 ms, max : 983 ms, 0 errors)
11 Q8 queries (avg : 7 ms, min : 2 ms, max : 46 ms, 0 errors)
0 Q9 queries (avg : 0 ms, min : 0 ms, max : 0 ms, 0 errors)
12 Q10 queries (avg : 240 ms, min : 4 ms, max : 1191 ms, 0 errors)
10 Q11 queries (avg : 170 ms, min : 78 ms, max : 385 ms, 0 errors)
12 Q12 queries (avg : 33 ms, min : 12 ms, max : 152 ms, 0 errors)

115 total retrieval queries (0 errors)
0.1919 average queries per second

6.4. SCENARIO 2: FORWARD-CHAINING MATERIALIZATION IN MONGODB

In this scenario SBP rich semantic knowledge graph is stored in GraphDB as originally designed by the benchmark, however all SBP assets are stored as documents in MongoDB. Furthermore, we use forward-chaining approach (described in section 2.1 Rule-based inference) to implement inference and distribute it using native MongoDB distribution mechanism. The materialisation is performed on MongoDB document insertions (write) time where the materialisation for each document is executed in two steps:

Step 1. All references to knowledge graph originally found in the asset's metadata are looked up in GraphDB where their inference closer is retrieved from. In this way the inference is originated in GraphDB and provided as pre-calculated output to MongoDB along with the document original content.

Step 2. When writing documents to MongoDB all concepts that are retrieved from GraphDB are inserted into the document, expanding the document, and only then the documents is submitted to MongoDB. In this way the document is enlarged to accommodate already pre-calculated inference and is distributed along with the document on MongoDB's native distribution mechanism.

Our Implementation of this scenario is realised as a modification of the original synthetic data generation tool where data is generated, where only semantic knowledge graph is stored in GraphDB. Extension of this tool retrieves relevant data from GraphDB and imports it along with the documents in MongoDB. In general, it monitors a collection in MongoDB and extends all new documents with inferred statements from GraphDB.

Alternative implementation, foreseen in the future that is easier for administration but more complex as development efforts, is to use GraphDB remote notification mechanism (see section 4.3 GraphDB remote notifications) and perform in-memory inference and extend the documents concept list while submitting documents to MongoDB via GraphDB.

The **advantage** of this scenario is that one can query MongoDB directly having all semantics materialised and will improve query time (see section Evaluation results, where the evaluation setting creates an array cwork:tags which combines cwork:about and cwork:mentions). This will greatly reduce the execution complexity of queries compared to other scenarios.

Main **disadvantage** of this scenario comes from the fact that the knowledge graph data in GraphDB can evolve and this requires proper synchronisation mechanisms between already materialised inference in MongoDB and GraphDB updates. This is potentially possible to be implemented with GraphDB remote notification mechanism (see section 4.3 GraphDB remote notifications) and is subject of future development. Still keeping the documents is MongoDB up-to-date when the reference data in GraphDB is changed may be tricky.

6.5. SCENARIO 2 EVALUATION RESULTS:

Seconds 600
2019-03-22 10:42:14 (completed query mixes: 17)

Editorial:

4 agents

5808 inserts (avg : 363 ms, min : 178 ms, max : 3436 ms)
678 updates (avg : 366 ms, min : 223 ms, max : 1708 ms)
732 deletes (avg : 23 ms, min : 1 ms, max : 467 ms)

7218 operations (5808 CW Inserts (0 errors), 678 CW Updates (0 errors), 732 CW Deletions (0 errors))

12.0300 average operations per second

Aggregation:

8 agents

20 Q1 queries (avg : 50 ms, min : 5 ms, max : 176 ms, 0 errors)
20 Q2 queries (avg : 4 ms, min : 3 ms, max : 8 ms, 0 errors)
20 Q3 queries (avg : 14 ms, min : 3 ms, max : 85 ms, 0 errors)
21 Q4 queries (avg : 6 ms, min : 3 ms, max : 51 ms, 0 errors)
20 Q5 queries (avg : 1042 ms, min : 106 ms, max : 10375 ms, 0 errors)
19 Q6 queries (avg : 95 ms, min : 47 ms, max : 200 ms, 0 errors)
19 Q7 queries (avg : 33 ms, min : 21 ms, max : 107 ms, 0 errors)
20 Q8 queries (avg : 4 ms, min : 2 ms, max : 11 ms, 0 errors)
17 Q9 queries (avg : 176029 ms, min : 166625 ms, max : 185911 ms, 0 errors)
21 Q10 queries (avg : 345 ms, min : 5 ms, max : 1396 ms, 0 errors)
18 Q11 queries (avg : 323 ms, min : 70 ms, max : 1507 ms, 0 errors)
20 Q12 queries (avg : 3 ms, min : 2 ms, max : 6 ms, 0 errors)

6.6. SCENARIO 3: FORWARD-CHAINING MATERIALISATION IN GRAPHDB

In this scenario we again use forward-chaining approach (described in section 2.1 Rule-based inference), which is performed on document insertions (write) time. In contrast to Scenario 2: Forward-chaining materialization in MongoDB where documents are enlarged with materialised inference results, in this second scenario the knowledge graph is enlarged with relations data between assets and the extracted metadata. All reference data – knowledge graph – is stored in GraphDB, all asset data along with its metadata is stored in MongoDB, while some of the metadata is duplicated in GraphDB as well. This is performed in two steps:

For each document the materialisation is executed in two steps:

Step 1. Each asset is modelled as a document and stored in MongoDB and reference to this document is modelled as "creative work" concept and stored in GraphDB as well.

Step 2. All references in each asset's metadata to concepts in the knowledge graph are modelled as relations between the corresponding "creative work" and referred concept.

In this way all references to knowledge graph originally found in the asset's metadata are stored in GraphDB with links to the MongoDB documents by URI. The inference is natively materialised in GraphDB and only the streaming part of the data which constitutes the larger volume of is distributed.

Querying data in this scenario is realised as hybrid queries where document URIs are identified in GraphDB and the actual document are retrieved from MongoDB.

The **advantage** of this scenario addresses the disadvantage of Scenario 1 and changes in the knowledge graph are natively handled in GraphDB. As the links between screaming assets and the corresponding concepts they refer to are stored in GraphDB and the inference is materialised in the same place, changes in the knowledge graph are handled efficiently by GraphDB.

Main **disadvantage** of this approach is that some of the data is de-normalised and duplicated in GraphDB and potentially can hit its scalability limits.

6.7. SCENARIO 3 EVALUATION RESULTS:

Seconds : 600
2019-03-15 16:58:08 (completed query mixes : 37)
Editorial:
4 agents

10609 inserts (avg : 169 ms, min : 45 ms, max : 3949 ms)
1345 updates (avg : 212 ms, min : 87 ms, max : 2019 ms)
1376 deletes (avg : 203 ms, min : 38 ms, max : 3982 ms)

13330 operations (10609 CW Inserts (0 errors), 1345 CW Updates (0 errors), 1376 CW Deletions (0 errors))
22.2167 average operations per second

Aggregation:
8 agents

42 Q1 queries (avg : 113 ms, min : 4 ms, max : 716 ms, 0 errors)
40 Q2 queries (avg : 11 ms, min : 4 ms, max : 29 ms, 0 errors)
42 Q3 queries (avg : 16 ms, min : 3 ms, max : 33 ms, 0 errors)
39 Q4 queries (avg : 43 ms, min : 4 ms, max : 112 ms, 0 errors)
40 Q5 queries (avg : 62931 ms, min : 51161 ms, max : 79797 ms, 0 errors)
40 Q6 queries (avg : 533 ms, min : 42 ms, max : 3732 ms, 0 errors)
40 Q7 queries (avg : 9309 ms, min : 2550 ms, max : 30476 ms, 0 errors)
41 Q8 queries (avg : 3 ms, min : 2 ms, max : 8 ms, 0 errors)
42 Q9 queries (avg : 379 ms, min : 127 ms, max : 873 ms, 0 errors)
39 Q10 queries (avg : 9138 ms, min : 4 ms, max : 36280 ms, 0 errors)
42 Q11 queries (avg : 31 ms, min : 19 ms, max : 96 ms, 0 errors)
40 Q12 queries (avg : 21 ms, min : 9 ms, max : 120 ms, 0 errors)

487 total retrieval queries (0 errors)
0.8122 average queries per second

6.8. EVALUATION RESULTS ANALYSIS

In the evaluation result comparison, we are analysing the performance of each of the three scenarios described above. The baseline for evaluation is served by original design on LDBC SPB for querying a graph database, in our experiment GraphDB. All documents as well as the inferred statements are stored using GraphDB native forward-chaining mechanism.

Our analysis considers both aspects of the benchmark: Insert, Update and Delete; as well as runtime query performance for selected 12 queries described in APPENDIX I - SPB Query Description.

6.9. INSERT, UPDATE AND DELETE

As expected, Scenario 1: Backword-chaining query rewrite (see the table below) performs much better on all basic database interactions as the load of inference is moved on query time. Scenario 2: Forward-chaining materialization in MongoDB performs worse because of the extra queries sent to GraphDB for each reference from a document (to be stored in MongoDB) and corresponding concepts in the knowledge graph.

Evaluation	Baseline	Scenario 1	Scenario 2	Scenario 3
Insertion per 600 sec	11,217 inserts	143,629 inserts	5,808 inserts	10,609 inserts
Insert average	24 ms	6 ms	363 ms	169 ms
Updates per 600 sec	1,408 updates	17,887 updates	678 updates	1,345 updates

Update average	194 ms	34 ms	366 ms	212 ms
Deletes per 600 sec	1,384 deletes	17,950 deletes	6732 deletes	1,376 deletes
Deletes average	174 ms	32 ms	23 ms	203 ms

6.10. QUERY PERFORMANCE

As noted in the table below, those queries that don't imply usage of inference performed better in Scenario 1 and Scenario 2, while heavy inference queries as queries that rely on aggregation functions as Q7 and Q9, perform better in GraphDB.

Evaluation	**Baseline**	**Scenario 1**	**Scenario 2**	**Scenario 3**
Q1 avg in ms	171	102	50	113
Q2 avg in ms	6	11	4	11
Q3 avg in ms	74	36	14	16
Q4 avg in ms	57	10	6	43
Q5 avg in ms	131	1930	1042	62,931
Q6 avg in ms	70	178	95	533
Q7 avg in ms	9	363	33	9,309
Q8 avg in ms	5	7	4	3
Q9 avg in ms	277	0	176029	379
Q10 avg in ms	513	240	345	9,138
Q11 avg in ms	98	170	323	31
Q12 avg in ms	14	33	3	21

7. METHODOLOGY FOR CHOOSING AN APPROACH

We have 3 possible approaches. It is important to identify which cases do each of the approaches cover. This is essential in order to pick an approach for our datasets. The scenarios are evaluated based on the two dimensions:

- Insertion & update performance
- Query time performance

There is an obvious tradeoff between the two dimensions. The backward chaining vs forward chaining approaches differ exactly in those dimensions. Usually the backward chaining approach is fast for insertion and deletes but has poorer performance as compared to the forward chaining where the inference is calculated during ingestion. The difference in performance could be negligible

if short graph traversals are used for the inference. Such a corner case can only be discovered through evaluation. Here we have our very first differentiator which has to be taken into consideration when choosing an approach. If the data you ingest is extremely volatile and your first concern is to have fast updates then you should choose the backpropagation approach. This will have an impact on the query time performance but your database will be kept up to-date and will be able to scale efficiently. Generally speaking this will not be the most used approach. When opting for using triple stores the people usually want to have quick graph traversals thus some kind of inference should be involved.

If you care for query time performance and need non-minimal inference then you should choose between scenario 2 and 3. The differences between the two are more subtle than those between forward and backward chaining thus picking out the right one for your use case is harder.

7.1 INSERT PERFORMANCE

Scenario 2 has performance 2N, while Scenario 3 has performance N. Where N is the time for insert. This would mean that Scenario 3 performs two times better than scenario 2. This can be explained because in scenario 3 the data is present locally within the JVM of the GraphDB. No communication with other services is required in order to infer some statements.

7.2 QUERY PERFORMANCE

In Table X we have analyzed the differences between the performance of the two scenarios. Where N is the time smallest time taken for the two scenarios. The other columns are normalized by the original value divided by N. Where we see the value 1 for a use case it means that it is the faster option. We can see that scenario 2 performs better for 9 out of the 12 queries. The differences between the other 3 (Q8, Q9 & Q11) should be analysed.

- Q8 the difference is negligible, in absolutes it is 1ms.
- Q9 the difference is staggering and the scenario basically fails in this use case. The results are returned in over 3 minutes compared to 379ms for scenario 3. Root cause analysis should be done to further understand the difference in performance.
- Q11 the difference is of significance, but still the result is returned within 323ms.

Evaluation	N	Scenario 2 in times N	Scenario 3 in times N
Q1 avg in ms	50	1	2.26
Q2 avg in ms	4	1	2.75
Q3 avg in ms	14	1	1.14285714285714
Q4 avg in ms	6	1	7.16666666666667
Q5 avg in ms	1042	1	60.39443378119
Q6 avg in ms	95	1	5.61052631578947
Q7 avg in ms	33	1	282.090909090909
Q8 avg in ms	3	1.33333333333333	1

Q9 avg in ms	379	464.456464379947	1
Q10 avg in ms	345	1	26.4869565217391
Q11 avg in ms	31	7.48387096774194	1
Q12 avg in ms	3	1	7

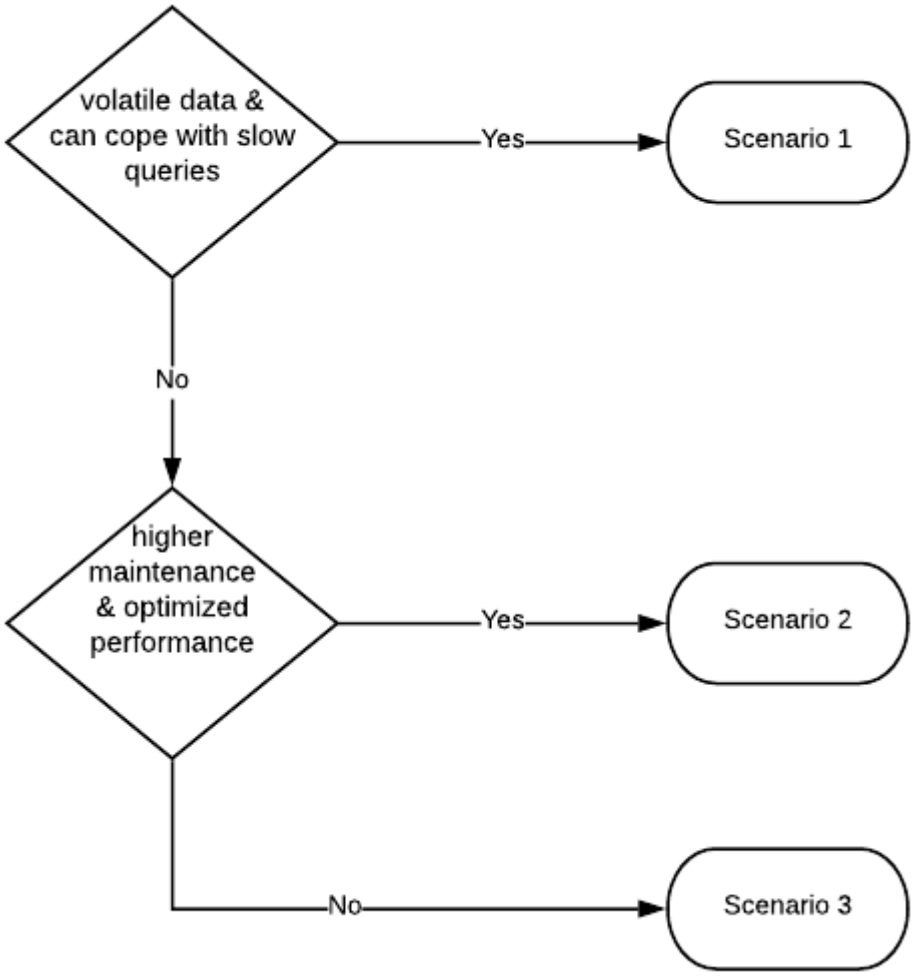
7.3 Q9 QUERY TIME BEHAVIOUR

The query is characterized by a complex aggregation logic. In the SPARQL the query seems simple and optimized by the underlying query engine. In order to accommodate the query via MongoDB we have created the aggregation pipeline. For the purpose of supporting generic queries we have created a single index per field we have used. This is not optimal for this particular type of query. Here we have filtering by multiple fields with different cardinality which results in query time joining of the two indexes. This is not the most efficient way possible for these types of queries. MongoDB supports hierarchical indexing. By using such indexing the query time would be decreased and could possibly become less than a minute.

7.4 CHOOSING THE BEST SCENARIO

Based on the analysis we can easily define the rules and guidelines for choosing the best approach for your use case.

1. Do you have very volatile data and a paramount need to be always up to date with it? Don't you use inference frequently or can you compromise query time performance for it? - If yes, then choose scenario 1.
2. Do you need very fast and optimizable queries? Are you willing to invest in maintenance and higher cost for set up? Are you ok with analysing your need for specific indexes and checking for degrading performance? - If yes, then choose scenario 2.
3. Do you need a generic purpose graph with consistent performance (though not the best one) with low maintenance costs? - If yes, then choose scenario 3 (best for exploratory work).


volatile data & can cope with slow queries
Yes
Scenario 1
No
higher maintenance & optimized performance
Yes
Scenario 2
No
Scenario 3

8. THE USE CASE

We have picked a use case scenario on which we will apply a distributed inference approach discussed in the previous sections. In this use case we have data for estates and two types of observations related to these estates - meteorological data and samples gathered from a field and tested in a lab.

The data sources were transformed into RDF format by using the TARQL tool. TARQL (Tabular SPARQL) is a command-line tool for converting CSV files to RDF using SPARQL 1.1 syntax. It's written in Java and based on Apache ARQ. The CSV file's contents are input into the query as a table of bindings. This allows manipulation of CSV data using the full power of SPARQL 1.1 syntax, and in particular the generation of RDF using CONSTRUCT queries. TARQL is particularly useful as it is lightweight, efficient and works in streaming mode.

8.1 ESTATES DATA

The estates data includes a unique identifier, the estate name and a geographic polygon for its location. The name and geographic boundaries of the estates don't change often thus we can deem this data static or at least close to static.

	estate	name
1	http://data.bigdatagrapes.eu/resource/Symbeeosis/estate/RIRA	"RIRA Vineyards"
2	http://data.bigdatagrapes.eu/resource/Symbeeosis/estate/Skouras	"Skouras winery"
3	http://data.bigdatagrapes.eu/resource/Symbeeosis/estate/Papagiannoulis	"Papagiannoulis Winery"
4	http://data.bigdatagrapes.eu/resource/Symbeeosis/estate/Papagiannakos	"Papagiannakos domaine"
5	http://data.bigdatagrapes.eu/resource/Symbeeosis/estate/Semeli	"Semeli winery"

8.2 METEOROLOGICAL DATA

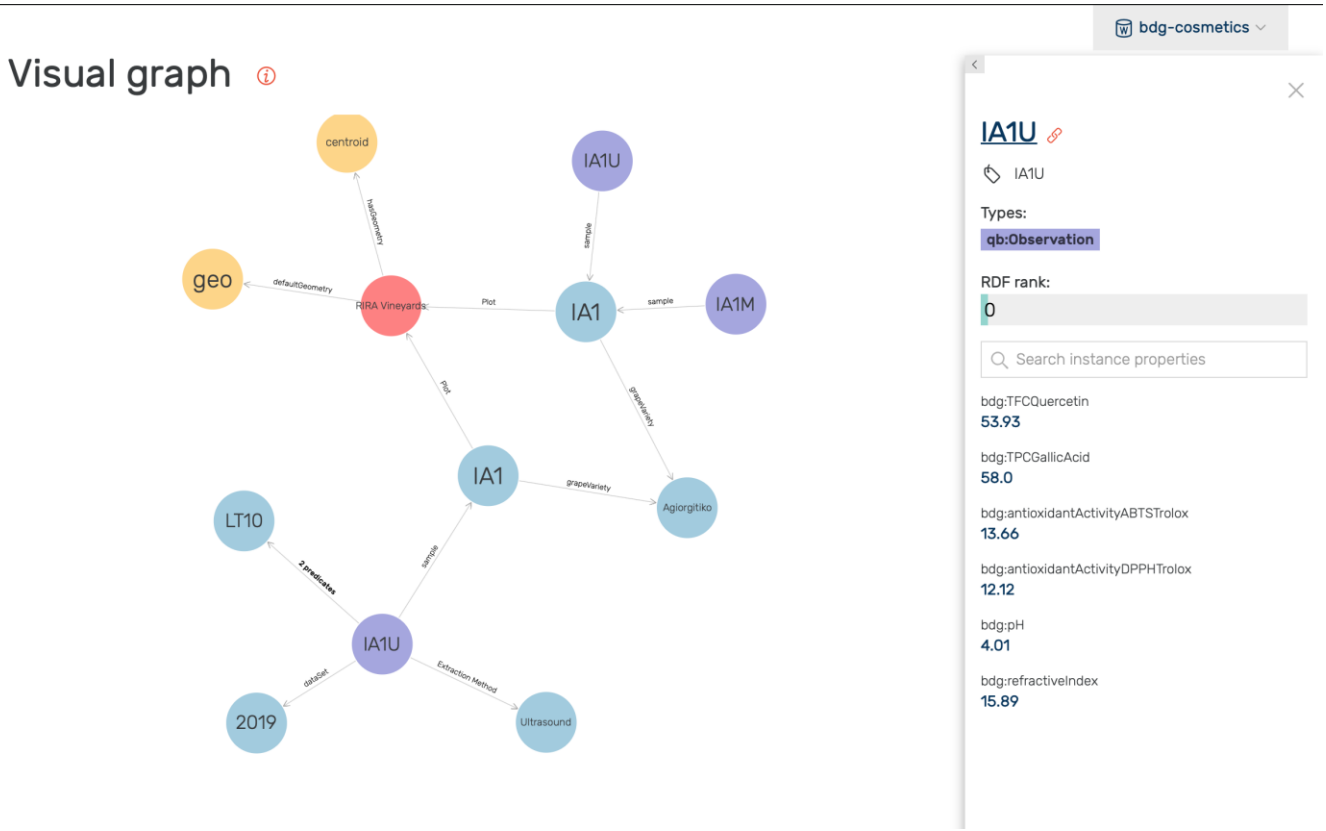
The meteorological data is streamed from an on premise device. It sends frequent updates on the weather conditions. This data is volatile and we could expect that with time it will only grow in size. Each observation includes information for atmospheric pressure, wind direction, air humidity, rainfall, wind speed and the estate for which this information applies. We have transformed the data into a JSON-LD serialization for RDF.


```
{
  "_id":{
    "$oid":"5f2d4999e7b331040b92b497"
  },
  "@context":{
```

```
    "@base":"http://data.bigdatagrapes.eu/",
    "bdg":"http://data.bigdatagrapes.eu/resource/ontology/",
    "crs-ogc":"http://www.opengis.net/def/crs/OGC/1.3/",
    "geo":"http://www.opengis.net/ont/geosparql#",
    "gn":"http://www.geonames.org/ontology#",
    "owl":"http://www.w3.org/2002/07/owl#",
    "qb":"http://purl.org/linked-data/cube#",
    "qb4st":"http://www.w3.org/ns/qb4st/",
    "rdf":"http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "rdfs":"http://www.w3.org/2000/01/rdf-schema#",
    "skos":"http://www.w3.org/2004/02/skos/core#",
    "wgs":"http://www.w3.org/2003/01/geo/wgs84_pos#",
    "xml":"http://www.w3.org/XML/1998/namespace",
    "xsd":"http://www.w3.org/2001/XMLSchema#"
  },
  "@id":"data/cosmetics-weather/2019/Paros/2019-07-26T14:00:00",
  "@type":"qb:Observation",
  "bdg:dateTime":{
    "@type":"xsd:dateTime",
    "@value":"2019-07-26T14:00:00"
  },
  "bdg:direction_wind":{
    "@id":"compass/north"
  },
  "bdg:humidity_air":"73.0",
  "bdg:plot":{
    "@id":"Symbeeosis/estate/Moraitis"
  },
  "bdg:pressure_atmospheric":"1010.8",
  "bdg:rainfall":"0.0",
  "bdg:rainfall_MAX":"0.0",
  "bdg:speed_wind":"24.1",
  "bdg:speed_wind_MAX":"32.2",
  "bdg:temp_air":"26.0"
}
```


8.3 LABORATORY OBSERVATIONS

The laboratory observations data is modelled in a similar manner. The samples for the observations are manually collected once a year from each estate. This type of data is again very static and closer to the estates metadata in terms of velocity. In the figure below we can see that the purple nodes are of type qb:Observation and in fact include the information for the laboratory observations. The extraction method in this case "Ultrasound" is another vertice linked by an edge to the Observation. The observation can be linked to the estate if the sample was taken by two hops. The estate is the vertice in red in this case the "Rira Vineyard".



8.4 Picking an inference scenario

Based on the defined approaches in the previous sections we needed to pick an appropriate inference scenario implementation. We have two datasets which are relatively static (estates and laboratory observations) and one which is volatile (meteorological observations). The meteorological observations will be held in a document store for better scalability. Document stores are great for storing self contained documents. The meteorological observations are self contained and only hold a reference to the estate on which the observations are taken. The other more static and interconnected data will reside in GraphDB.

Based on this setup we can decide on the most appropriate scenario. We will do this by answering the predefined questions:

1. *Do you have very volatile data and a paramount need to be always up to date with it? Don't you use inference frequently or can you compromise query time performance for it? - If yes, then choose scenario 1.*
   We have volatile data but handling some lag in the processing of meteorological data is ok. We will use inference. Scenario 1 does not fit our use case.
2. *Do you need very fast and optimizable queries? Are you willing to invest in maintenance and higher cost for set up? Are you ok with analysing your need for specific indexes and checking for degrading performance? - If yes, then choose scenario 2.*
   We want fast queries but investing in too much maintenance does not seem justified.
3. *Do you need a generic purpose graph with consistent performance (though not the best one) with low maintenance costs? - If yes, then choose scenario 3 (best for exploratory work).*
   We prefer to spend our time analysing the data rather than optimizing it for quick accessing. Scenario 3 is most appropriate for our use case. We will proceed with it.

8.5 PERSISTING INFERRED

After some digging we decided that the meteorological data could be meaningful if aggregated for a window of time. We decided daily aggregates can be stored within the GraphDB repository. Usually such an operation would require a data cleaning stage. It would be responsible for handling fluctuations within the data reported by the sensors. For instance a sensor might be reporting temperature of 25 degrees for an hour and then report a value of 50. Such a value should not be part of any aggregates as it is probably faulty. Identifying such outliers is part of the preloading step of the samples into MongoDB. The following implementation of the inference assumes that we have trustworthy data within the MongoDB collection. Using this approach we achieve decoupling of tasks.

Persisting the inferred statements is implemented with SPARQL queries.

8.5.1 CONNECTOR QUERY

The purpose of the connector query is to authenticate to a remote MongoDB instance. In our case the MongoDB instance is deployed on a separate server. This server is accessed via its DNS name - rolle. The real username and password are replaced with superficial ones for the purpose of the deliverable. After this query is successfully executed the MongoDB collection can be queried through a SPARQL query.

```
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
INSERT DATA {
  inst:bdg :service "mongodb://rolle:30000" ;
    :database "my-database" ;
    :collection "bdg" ;
    :user "root" ;
    :password "secretpassword" ;
        :authDb "admin" .
}
```

8.5.2 TEST CONNECTION QUERY

After the connection has been established we use the below query to test it. If no results are returned then there is an issue.

```
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
select * where {
          bind('{"@id":"data/cosmetics-weather/2018/Markopoulo/2018-05-08T11:50:00"}' as ?q)
  ?search a inst:bdg ;
        :find ?q;
        :entity ?obs .
  GRAPH inst:bdg {
    ?obs ?p ?o
  }
```

```
} limit 100
```

8.5.3 CREATE DAILY AGGREGATED WEATHER DATASET

This query is a bit more complicated than the other ones but it is the one doing the actual work. It iterates over all of the estates. For each estate it queries the MongoDB fetching the measurements and aggregates them by date. A bucket per day per predicate is created. Each bucket stores the measured values. After the collection phase for each bucket an average is calculated. The average values are then written into the GraphDB repository. This approach allows the aggregates to be indexed within GraphDB. The GraphDB indexes allow for various access patterns thus complex queries can be run. In the next section we will give an example of such a query.

With this inference with query approach we ensure an eventual consistency guarantees between the two datastores. MongoDB serves as the source of truth and a master of the data for meteorological measurements. GraphDB stores the derived version of that data. Periodically running the inference query will ensure that MongoDB and GraphDB remain in a consistent state. We introduce some lag between the data stored in Mongo and that within GraphDB. For analytical purposes it should be affordable. When the meteorological data accumulates to a considerable size doing the input asynchronously would be preferable. Synchronous inserts into the system would pose performance issues and a possibility for data loss.

```
BASE <http://data.bigdatagrapes.eu/resource/>
PREFIX bdg: <http://data.bigdatagrapes.eu/resource/ontology/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
PREFIX qb: <http://purl.org/linked-data/cube#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
INSERT {
  graph <graph/cosmetics/daily-weather> {
    ?OBSURI a qb:Observation ;
      ?p ?AVG ;
      bdg:plot ?plot ;
      bdg:date ?date ;
      qb:dataSet <data/cosmetics/daily-weather> ;
    .
  }
}
WHERE {
  bind(strafter(str(?plot),"http://data.bigdatagrapes.eu/resource/") as ?plot_str)
  bind(uri(concat("data/cosmetics/daily-weather/",?plot_str,"/",str(?date))) as ?OBSURI)
  {select ?plot ?date ?p (avg(?val) as ?AVG) where {
      #bind(<Symbeeosis/estate/Skouras> as ?plot)
      ?plot a bdg:Estate .
      bind(strafter(str(?plot),"http://data.bigdatagrapes.eu/resource/") as ?plot_str)
      bind(replace('{"bdg:plot" : {"@id":"<Q>"}}',"<Q>",?plot_str) as ?q)
      ?search a inst:bdg ;
          :find ?q;
```

```
            :entity ?meteo_obs .
        values ?p {
          bdg:humidity_air
          bdg:rainfall
          bdg:speed_wind
          bdg:pressure_atmospheric
          bdg:temp_air
        }
        GRAPH inst:bdg {
          ?meteo_obs ?p ?o ;
          bdg:dateTime ?dateTime .
        }
        bind(if(!contains(?o,"-"),strdt(?o,xsd:float),?null) as ?val)
      bind(strdt(replace(str(?dateTime),"T.*$",""),xsd:date) as ?date)
  } group by ?plot ?date ?p }
}
```

8.5.4 Querying over aggregated daily data for weather

Now we can easily link between estates, meteorological measurements and laboratory measurements. The following query does exactly that.

```
BASE <http://data.bigdatagrapes.eu/resource/>
PREFIX bdg: <http://data.bigdatagrapes.eu/resource/ontology/>
PREFIX qb: <http://purl.org/linked-data/cube#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
select * where {
  #bind(<http://data.bigdatagrapes.eu/resource/data/cosmetics/2018/IA10> as ?sample)
  ?lab_obs a qb:Observation ;
           bdg:sample ?sample ;
         bdg:extractionMethod <extractionMethod/Maceration> ;
         bdg:TFCQuercetin ?tfc ;
  .
  ?sample bdg:plot ?plot ; bdg:date ?date .
  ?meteo_obs a qb:Observation ;
          qb:dataSet <data/cosmetics/daily-weather> ;
          bdg:plot ?plot ;
          bdg:date ?date ;
          bdg:humidity_air ?hum ;
          bdg:temp_air ?temp ;
  .
}
```

9. SUMMARY

Inference distribution is a topic that has been studied previously, however the reported results vary in terms of applicability to a wide range of use cases. Reviewed literature, general methodology and individual tools for both distribution and parallelization provide valuable insights of possible options for applying distributed inference to BigDataGrapes project use-cases.

To provide pragmatic evaluation of the use case scenario similar to BigDataGrapes use-cases in terms of level of complexity and distribution of related datasets and need for particular type of inference, we have performed a number of experiments. This allows us to experiment with state-of-the-art tools and draw the roadmap for future implementations using one of the mechanisms for extending GraphDB build-in inference capabilities.

We have selected three inference scenarios to test while distributing data using state-of-the-art distributed storage – MongoDB. Evaluation results performed on standardised benchmark (LDBC) clearly show that each of the scenarios provides an optimal solution for different use-case.

We applied one of the three scenarios which fits best our use case. We were able to integrate highly volatile data with big volume from streaming measurements and more static interconnected data. The inference is implemented with eventual consistency guarantees. Once the inference processing is completed making queries over the interconnected graph is both fast and easy.

10. REFERENCES

Baader, F., Calvanese, D., McGuinness, D., Nardi, D. and Patel-Scheider, P. (2003). The Description Logic Handbook. Theory, Implementation, Applications, Cambridge University Press.

Brickley, D. and Guha, R. (2004). Resource Description Framework (RDF) Schemas. W3C Recommendation.

Dean M. and G. Schreiber, "OWL Web Ontology Language Reference," W3C Recommendation, 2004.

Grosof, B., Horrocks, I., Volz, R. and Decker, S. (2003).Description Logic Programs: Combining Logic Programs with Description Logic. In WWW2003, Budapest, Hungary.

Hayes, P. (2004). RDF Semantics

Horrocks, I., Patel-Schneider, P., Bechhofer, S., and Tsarkov, D. (2005). OWL Rules: A Proposal and Prototype Implementation. Journal of Web Semantics, pp. 23-40.

Kim J. M. and Park, Y.T. (2015). Scalable OWL-Horst ontology reasoning using SPARK. [Online]. Available: https://ieeexplore.ieee.org/document/7072815/.

Motik, B., Nenov, Y., Piro, R., Horrocks, I., Wu, Z. and Banerjee, J. (2015). RDFox: A Highly-Scalable RDF Store. In ISWC 2005.

Motik, B., Nenov, Y., Piro, R., Horrocks, I. and Olteanu, D. (2014).Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In 28th AAAI Conf. on Artificial Intelligence (AAAI 2014).

Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A. and Lutz, C. (2009). OWL 2 Web Ontology Language Profiles. W3C Candidate Recommendation.

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

Priya, S., Guo, Y., Spear, M. and Heflin J. (2014). Partitioning OWL Knowledge Bases for Parallel Reasoning, IEEE, p. 108–115.

Redaschi N. and the UniProt Consortium (2009). UniProt in RDF: Tackling Data Integration and Distributed Annotation with the Semantic Web. S. I. o. Bioinformatics. [Online]. Available: http://precedings.nature.com/documents/3193/version/1/files/npre20093193-1.pdf.

Shironoshita, P., Zhang, D., Kabuka, M. and Xu, J. (2017).Parallelization of Query Processing over Expressive Ontologies, in CEUR Workshop Proceedings.

Simeonov, B., Alexiev, V., Simov, K. and Kotsev, V. (2016). Final semantic infrastructure and final decision support system. Deliverable D5.4. MULTISENSOR.

Ter Horst, H. (2005). Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity. In ISWC 2005, Galway, Ireland.

Urbani, J., Margara, A., Jacobs, C., van Harmelen, F. and Bal, H. (2013). DynamiTE: Parallel Materialization of Dynamic RDF Data. In: Alani H. et al. (eds) The Semantic Web, Berlin.

Urbani, J. (2009). RDFS/OWL Reasoning Using MapReduce Framework.

Urbani, J., Piro, R., van Harmelen, H. and Bal, H. (2013). Hybrid reasoning on OWL RL, Semantic Web Journal, vol. 5, no. 6, pp. 423-447.

11. APPENDIX I - SPB QUERY DESCRIPTION

Full reference to queries can be found in LDBC Github project.

11.1. QUERY 1 DESCRIPTION

Retrieve creative works about thing t (or that mention t)
reasoning: rdfs:subClassOf, rdf:type
join ordering: cwork:dateModified rdf:type owl:FunctionalProperty
join ordering: cwork:dateCreated rdf:type owl:FunctionalProperty

Choke Points:
- join ordering based on cardinality of functional properties cwork:dateCreated, cwork:dateModified
Optimizer should use an efficient cost evaluation method for choosing the optimal join tree
- A sub-select which aggregates results. Optimizer should recognize it and execute it first
- OPTIONAL and nested OPTIONAL clauses (treated by query optimizer as nested sub-queries)
Optimizer should decide to put optional triples on top of the join tree (i.e. delay their execution to the last possible moment) because OPTIONALs are treated as a left join
- query optimizer has the chance to recognize the triple pattern: ?cWork a ?type . ?type rdfs:subClassOf cwork:CreativeWork and eliminate first triple (?cwork a ?type .) since ?cwork is a cwork:CreativeWork

11.2. QUERY 2 DESCRIPTION

Retrieve properties of a concrete creative work.
reasoning rdfs:subClassOf, rdf:type
join ordering: cwork:dateModified rdf:type owl:FunctionalProperty
join ordering: cwork:dateCreated rdf:type owl:FunctionalProperty

Choke Points:
- join ordering based on cardinality of functional proerties cwork:dateCreated, cwork:dateModified
Optimizer should use an efficient cost evaluation method for choosing the optimal join tree
- OPTIONAL clauses (treated by query optimizer as nested sub-queries)
Optimizer should recognize that FILTER condition contains variables which are part of the OPTINAL clauses and unlike query1 to start execution of OPTIONAL clause as soon as possible thus eliminating the intermediate results.
- query optimizer has the chance to recognize the triple pattern : ?creativeWork a ?type . ?type rdfs:subClassOf cwork:CreativeWork
and eliminate first triple (?creativeWork a ?type .) since ?creativeWork is a cwork:CreativeWork

11.3. QUERY 3 DESCRIPTION

Describes all creative works about a topic with certain fixed properties and order them by creation date. The size of the result-set is limited by a random number between 5 and 20.

Choke Points:
- UNIONS - optimizer should execute the UNIONs in terms or in parallel
- OPTIONAL clauses (treated by query optimizer as nested sub-queries)
Optimizer should recognize that FILTER condition contains variables which are part of the OPTINAL clauses and start execution of OPTIONAL clause as soon as possible thus eliminating the intermediate results.
- Optimizer should be able to split the FILTER conditions into conjunction of conditions and start their execution as soon as possible thus eliminating intermediate results
- Optimizer could consider the possibility to choose a query plan that would facilitate the ordering (ORDER BY) of result

11.4. QUERY 4 DESCRIPTION

Describes all blog posts tagged with a topic and order them by creation date. The size of the result-set is limited by a random number between 5 and 20.

Choke Points:
- Optimizer could consider the possibility to choose a query plan that would facilitate the ordering (ORDER BY) of result

11.5. QUERY 5 DESCRIPTION

Retrieve entities that are most tagged within one-hour interval. Restriction on audience type and Creative Work type further limits result.

Choke Points:
- Full scan query
Optimizer should not consider the ORDER BY as important clause in cases where all results are counted (COUNT(*))
- A sub-select which aggregates results. Optimizer should recognize it and execute it first
- Join ordering based on cardinality of functional property cwork:dateModified
Optimizer should use an efficient cost evaluation method for choosing the optimal join tree
- Optimizer should be able to split the FILTER conditions into conjunction of conditions and execute the as soon as possible, which will limit the amount of intermediate results

11.6. QUERY 6 DESCRIPTION

Retrieve creative works within a certain range defined by geo-coordinates. Retrieves a list of all creative works that are mentioning entities within a geo-spatial range.

Choke Points:
- A geo-spatial query
Allows each RDF engine could use its custom geo-spatial implementations.
- Optimizer should be able to split the FILTER conditions into conjunction of conditions and execute them as soon as possible, which will limit the amount of intermediate results

11.7. QUERY 7 DESCRIPTION

Retrieve creative works that have been created within a defined date-time range of one hour. Additional constraint added the type of creative works created in that time range.

Choke Points:
- Date range query
- Optimizer should be able to split the FILTER conditions into conjunction of conditions and execute them as soon as possible, which will limit the amount of intermediate results

11.8. QUERY 8 DESCRIPTION

Retrieve the N most popular topics creative works that have been modified in a time range of one hour. Restriction on audience type and Creative Work type further limits result. reasoning : owl:ObjectProperty, owl:DataProperty join ordering cwork:dateModified rdf:type owl:FunctionalProperty

11.9. QUERY 9 DESCRIPTION

Retrieve most recent Creative Works related to a particular one, namely such that are tagged with the same concepts. Calculates a score for a particular Creative Work, based on the number of Creative Works that it

shares tags with. The different combinations of cwork:about and cwork:mention count with factors between 0.5 and 2. When calculating the score, multiplication of results due to owl:sameAs equivalence should be suppressed. For instance, if only the following two statements are asserted in the repository
<cw1 cwork:tag e1> and <e1 owl:sameAs e2>
The query SELECT (COUNT(*) AS ?cnt) { cw1 cwork:tag ?e } should return 1, instead of 2
Reasoning : rdfs:subPropertyOf reasoning with respect to cwork:tag; owl:sameAs with respect to tags

Choke Points:
- Optimizer should consider cardinality of star-shaped sub-queries for choosing the optimal join ordering.
- Optimizer should identify the possibility of asynchronous execution of the aggregate sub-queries.
- Optimizer should consider the selectivity of the DISTINCT for choosing the right execution plan. The distinct's state # should be shared between threads or should be merged after the top order sort.
- Engines which support optimized handling owl:sameAs reasoning that allows for control of query results expansion can implement this query in a much simpler and efficient way. The first sub-query may look as follows:
SELECT (COUNT(*) AS ?cnt_2)
WHERE {
?other_cw cwork:about ?oa .
<CreativeWorkUri> cwork:about ?oa .
}

11.10. QUERY 10 DESCRIPTION

Retrieve creative works that mention locations in the same province (A.ADM1) as the specified one. Additional constraint on time interval further limits returned result within one hour.

11.11. QUERY 11 DESCRIPTION

Retrieve a list of the most recent Creative Works that have tagged with entities, related to a specific popular entity from reference dataset # Relations can be (inbound and outbound; explicit or inferred)

11.12. QUERY 12 DESCRIPTION

Retrieve the descriptions of the latest creative works tagged with a specific location. Consider that the description of each specific. Creative Work is stored in dedicated named graph. The result should include only the explicit statements about the creative work, without owl:sameAs equivalence and without statements inferred otherwise.