Answering Continuous Description Logic Queries: Managing Static and Volatile Knowledge in Ontologies

Carlos Bobed, Fernando Bobillo, Sergio Ilarri, and Eduardo Mena
IIS Department, University of Zaragoza, Spain

During the last years, mobile computing has been the focus of many research efforts, due mainly to the ever-growing use of mobile devices. In this context, there is a need to manage dynamic data, such as location data or other data provided by sensors. As an example, the continuous processing of location-dependent queries has been the subject of thorough research. However, there is still a need of highly expressive ways of formulating queries, augmenting in this way the systems' answer capabilities. Regarding this issue, the modeling power of Description Logics (DLs) and the inferring capabilities of their attached reasoners could fulfill this new requirement. The main problem is that DLs are inherently oriented to model static knowledge, that is, to capture the nature of the modeled objects, but not to handle changes in the property values (which requires a full ontology reclassification), as is common in *mobile* computing environments (e.g., the location is expected to vary continually).
In this paper, we present a novel approach to process continuous queries that combines 1) the DL reasoning capabilities to deal with static knowledge, with 2) the efficient data access provided by a relational database to deal with volatile knowledge. By marking at modeling time the properties that are expected to change during the lifetime of the queries, our system is able to exploit both the results of the classification process provided by a DL reasoner, and the low computational costs of a database when accessing changing data (mobile environments, semantic sensors, etc.), following a two-step continuous query processing that enables us to handle continuous DL queries efficiently. Experimental results show the feasibility of our approach.
Keywords: Stream Reasoning; Continuous Queries; Ontologies; Description Logic Reasoning

1. Introduction

Mobile computing has attracted an intensive research attention and is nowadays everywhere. Indeed, important progress has been performed thanks to the popularity and technological advances of mobile devices and wireless networks. So, we now consider applications where users continuously access data of interest independently of the specific communication technology used (e.g., 3G, WiFi), the type of network (infrastructure-based vs. ad hoc), the access model (push, pull, or hybrid) (Delot, Ilarri, Thilliez, Vargas-Solar, & Lecomte, 2011), the data providers (powerful data servers, peers, or simple sensors with limited capabilities), the volatility of the data of interest (e.g., access to relatively static data stored on databases vs. highly-dynamic data fed by a Data Stream Management System (Golab & Özsu, 2003) or retrieved by sensors on the fly), and even the mobility of the users and the data sources (e.g., classical mobile users with a smartphone vs. drivers in a vehicular ad hoc network (Cenerario, Delot, & Ilarri, 2011)). In general, scenarios with dynamic data (e.g., location data or other data provided by sensors) are quite frequent in this context.

Abstracting users from the heterogeneity of the underlying data and providing them with more accurate and understandable results is crucial, and Semantic Web technologies could facilitate undertaking this challenge. They can help the development of mobile computing applications, as they facilitate understanding the meaning of data and the meaning of the user requests. As an example, the problem of processing continuous location-dependent queries has been extensively studied (Ilarri, Mena, & Illarramendi, 2010). However, most existing systems do not support a flexible way of formulating queries.

Whereas the use of ontologies (Gruber, 1995) and Description Logics (DL) (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2003) could help in this sense, they are not suitable for situations where there are changes in the property values. For instance, the GPS position of a mobile device is subject to constant changes. These changes in the knowledge can require to reclassify the full ontology before reasoning with it. For example, if the definition of the class of available emergency vehicles depends on their location, the constant changes in their positions require to reclassify the ontology continuously. However, this is not acceptable in applications where the changes in the data are a common situation, as this may dramatically decrease the performance of the system.

Apart from mobile scenarios, where the location is potentially continuously changing, there is indeed an important number of applications that require dealing with data streams, such as intrusion detection in computer networks, monitoring applications using sensor data, or online analysis of stock prices (Cugola & Margara, 2012). Indeed, attributes

Email: {cbobed, fbobillo, silarri, emena}@unizar.es

that could change frequently are very common in many environments (e.g., availability of a taxi cab, occupants of a vehicle, etc.).

In this paper, we give a step forward in the direction of performing reasoning with continuously changing data by proposing an efficient approach to process continuous queries over semantic data streams. Specifically, we present *QueryGen*, a Knowledge Management system which is able to deal with data that are continuously changing. QueryGen is not an ontology reasoner, but combines the benefits of DL reasoners and databases to manage the dynamicity of the knowledge, making it possible to retrieve the instances that are relevant to a given query in a continuous way.
The main features of our proposal are:

- It is based on the identification of dynamic data at modeling time, in order to deal with such data in a way that the performance of the reasoning is not compromised. To this end, we propose classifying properties into two classes: *volatile* and *static* ones[1]. For instance, the location of a car could constantly change, so we consider it as volatile, but its color is a static property[2]. In our approach, stream data can be managed as volatile data outside the knowledge model, but it is also possible to use volatile properties as vocabulary to define the ontologies.
- It combines the advantages of DL reasoners and traditional Database Management Systems (DBMS) to process both the static and dynamic data that may be interesting for a query. Each of these properties are treated in a slightly different way. Our approach is able to support the full expressivity of OWL 2 if there are only static properties, but volatile properties impose some restrictions, some of them to guarantee soundness of the answers. As noted by other authors, most of the times soundness or completeness can be sacrificed for a significant speed-up of reasoning (Dentler, Cornet, ten Teij, & de Keizer, 2011).
- It performs very well. Experimental results show the feasibility of our approach and its clear advantages over a traditional approach just based on the use of a DL reasoner.

The structure of the rest of this paper is as follows. In Section 2, we introduce the notions of static and dynamic knowledge, and motivate our work with the help of an illustrating use case where answering continuous queries is needed. Then, in Section 3, we summarize the architecture of our system and the decisions that have to be made at modeling time. In Section 4 and Section 5, we describe the query processing and the updating of the volatile information, respectively. Then, Section 6 details the syntax and semantics of our supported language, discussing some modeling constraints imposed in the current approach. Next, in Section 7, we further explain the functioning of QueryGen via some examples. In Section 8, we present an extensive set of experimental results that show the feasibility and interest of our approach, as well as its advantages regarding pure DL-based approaches. In Section 9, we present some related works. Finally, in Section 10, we present our conclusions and some ideas for future research. An appendix recalling the syntax of the ontology language OWL 2 is included at the end of the paper.

2. Motivation

This section starts with a reflection about the nature of the modeled properties of objects regarding the amount of time their values are expected to keep constant. Next, we present an illustrating example of a mobile scenario.

2.1. Static versus Volatile Knowledge

Ontologies provide a powerful tool to model different domains or scenarios using their three main elements, namely concepts, properties and instances. DLs are the most extended logical formalism for representing ontologies, and the existence of DL reasoners allows to extend the modeling capabilities with inferring ones, which makes them more appealing. However, ontologies are oriented to model intensional knowledge as opposed to databases, more focused on extensional data. This makes ontologies more useful when dealing with static knowledge, i.e., when capturing the nature of the objects they are modeling.

There exist scenarios where ontologies could be very useful to enhance the expressivity of the models but, unfortunately, they are not suitable due to the ever changing nature of their extensional knowledge. In particular, when dealing with mobile computing environments, there may be properties of moving objects whose values are inherently volatile, in contrast to the static properties, which have constant values. For example, the location of a vehicle and the availability of a taxi cab are examples of values that change frequently. While our view is novel, the distinction between static and dynamic knowledge has already been study in ontologies (Flouris, Manakanatas, Kondylakis, Plexousakis, & Antoniou, 2008) and, more generally, in knowledge representation (Nissen, 2013).

In general, after some information is asserted in an ontology, it is usually assumed that it will not change and, therefore, we are used to working with the newly inferred information assuming it as correct. However, this information might not be valid if some of the initial assumptions change. Nevertheless, this conflict does not mean that volatile properties should not be modeled in an ontology, as it can be interesting to represent them at the intensional level.

These considerations lead us to differentiate between *volatile* and *static* properties. As an example, stock exchange values are constantly changing, whereas the types of operations in the stock market are not. Since volatile properties are always changing, one cannot assume that the value at a certain time t is going to be the same at $t + \Delta t$. This behavior comes directly into conflict with the nature of the knowledge stored in ontologies, which could actually evolve along time

[1] We use the term *volatile* as the C programming language uses it to avoid assuming that the value does not change; the term *dynamic* applied to properties is often used in existing modeling literature as properties/methods that can be added to an object dynamically, but without making any assumption about the evolution of their values.

[2] Note that the value of this property could change. However, due to the extremely low chances of change, the performance would not be affected and we would model it as a static property.

but is usually assumed to remain unchanged or, at least, it is assumed to be static enough to be used to answer queries.

Definition 1. *A* volatile property *is a property whose values are expected to change anytime and frequently.*

Definition 2. *A* static property *is a property whose value either does not change or remains unchanged long enough to consider it static.*

Ideally, when all the properties in an ontology are static, to maintain the knowledge stored up-to-date, we would only have to reclassify the instances when:

1. A change in the intensional knowledge (i.e., in the TBox) happens, such as the insertion/retraction of definitions, properties, etc.

2. A change in the extensional knowledge (i.e., in the ABox) happens. Thus, even if there are no changes in the TBox, the insertion or retraction of an instance might affect the classification of other instances. For instance, if *CrowdedVehicle* is defined as a vehicle with at least 3 occupants, saying that a person is not an occupant of a vehicle anymore can imply that it is not a *CrowdedVehicle* anymore. Moreover, when dealing with volatile properties, this situation is even more problematic, as every change in the value of a property of an instance might affect the rest of instances and the inferences made up to that point.

The first situation can be foreseen, as we have the control over what to assert to the ontologies used by our system, and we can evaluate the price to pay for each classification process. However, in mobile and dynamic environments the second situation can frequently happen; for example, if we want to model the availability of mobile devices, we would need to take into account that this can be based on features that may change along time: the availability of battery, the existence of wireless coverage, etc. So, changes in the ABox can frequently happen due to modifications in the volatile properties of the objects.
Let us consider the following definitions[3]:

$C_1 := R_{v1}$ only C_2
$C_2 := R_{v2}$ some C_1

being C_i concept definitions and R_{vi} properties tagged as volatile. With a classical reasoner, if values of R_{v1} and R_{v2} are changing continuously, it might be unable to finish the classification process and, even if it manages to classify the ABox, the result would be immediately outdated, as we cannot assure that the starting assumptions (the values of R_{v1} and R_{v2}) still hold.

In particular, in mobile computing scenarios, we have to provide systems that are flexible enough to cope with these temporal inconsistencies due to the continuous and costly reclassification process, as they are not only possible but a reality. So, our goal is to exploit both static and volatile knowledge when answering the queries while keeping the computation costs low enough to provide an efficient way of processing the queries continuously.

To do this, we will distinguish between the static and the volatile part of an ontology. Before defining these notions, we will introduce some notation that we will use in the rest of this paper to shorten the most common axioms (OWL 2 syntax is recapped in the appendix):

- "a Types C" is denoted as C(a).
- "a Facts R b" is denoted as R(a, b).
- "C1 SubClassOf C2" is denoted as C1 => C2.
- "C1 EquivalentTo C2" is denoted as C1 := C2.

Definition 3. *The* static part of an ontology *is composed by:*

- *the TBox,*
- *axioms of the form C(a), and*
- *axioms of the form R_s(a, b), for a static property R_s.*

Definition 4. *The* volatile part of an ontology *is composed by axioms of the form R_v(a, b), for a volatile property R_v.*

Our queries will consist of DL concept expressions. Such expressions can be divided into static and volatile parts.

Definition 5. *A* volatile concept *is a DL concept that verifies one of the following conditions:*

- *C is of the form R_v only D,*
- *C is of the form R_v some D,*
- *C is of the form R_v self,*
- *C is a cardinality restriction on a volatile property R_v,*
- *C is a conjunction or disjunction of concepts such that at least one of them is a volatile concept,*
- *C is the negation of a volatile concept.*

Definition 6. *A DL concept is a* static concept *iff it is not a volatile concept.*

Definition 7. *A* volatile part of a query *contains a volatile concept appearing in the query. A* static part of a query *contains a static concept appearing in the query.*

For instance, in the query concept *C or (F and R_v some D)*, *C* and *F* are static parts and (R_v some *D*) is a volatile part.

2.2. Motivating Example

Here, we present the motivating use case that we will use along the paper. The ontology proposed in Figure 1 models an example scenario of moving vehicles. The ontology includes information about the different vehicles in the system and about their passengers. The location of the moving vehicles is continuously changing.

LOQOMOTION (LOcation-dependent Queries On Moving ObjecTs In mObile Networks) (Ilarri, Mena, & Illarramendi, 2006; Ilarri, Bobed, & Mena, 2011) is a system that performs a distributed processing of continuous location-dependent queries issued by mobile users. The answer to a location-dependent query depends on the locations of the objects involved; for example, a user with a smartphone may want to locate available taxi cabs that are near him/her while walking home in a rainy day. These queries require a continuous monitoring of the locations of moving objects. The answer to this query must be continuously refreshed because

[3] In this paper we will use the subscripts $_s$ and $_v$ to denote a static and a volatile entity, respectively.

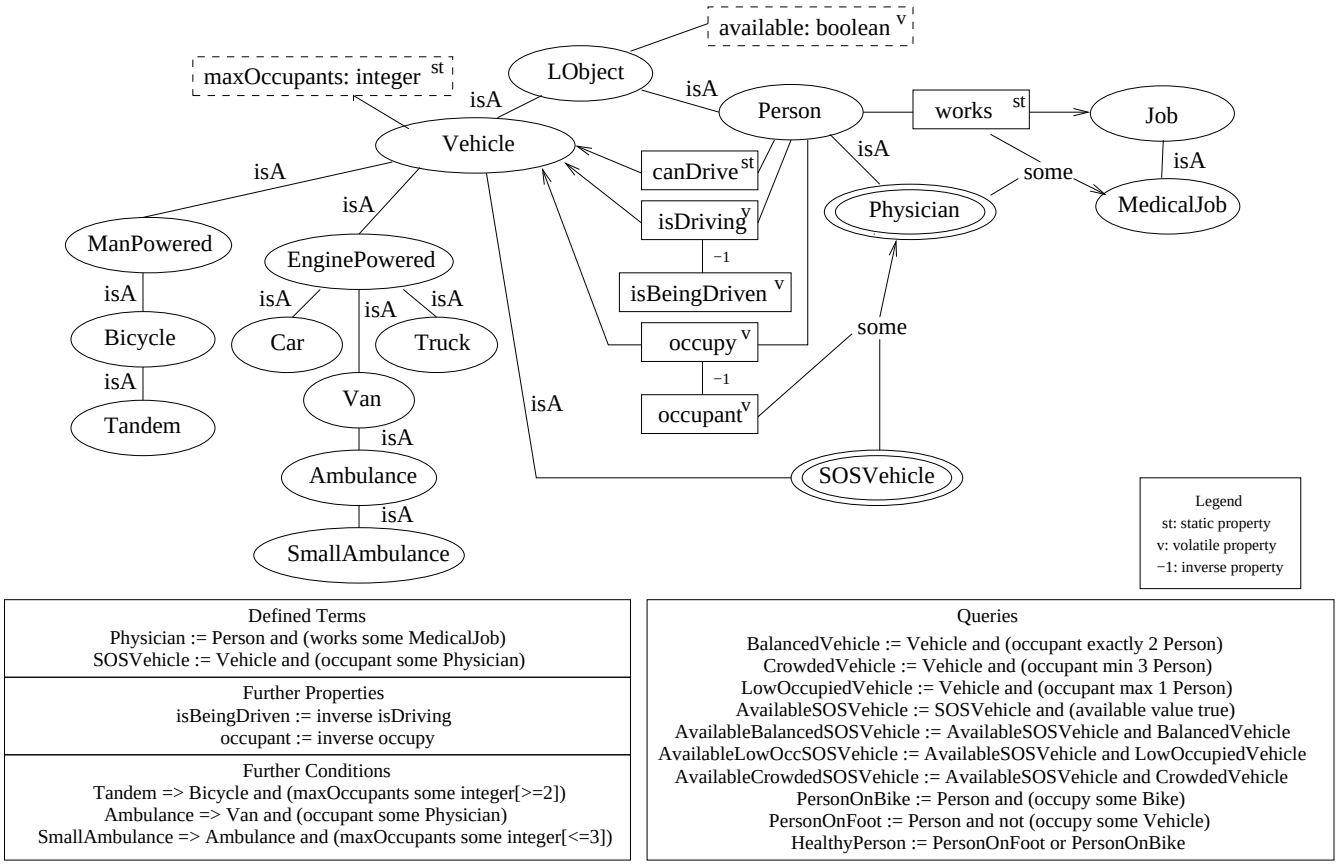


Figure 1. Ontology that models the motivating use case.

it can change immediately due to the movements of people and taxi cabs. Moreover, even if the set of taxis satisfying the query condition does not change, their locations and distances to the user do change continuously, and therefore the answer to the query must be updated with the new location data.

While *LOQOMOTION* is able to handle efficiently location-dependent queries, it fails when we want to express semantic constraints over the objects to be retrieved. In this example, we want to retrieve vehicles that could help in an emergency situation, that is, SOS vehicles that would be available and near a certain place. Having only the taxonomy of types of moving objects would only allow us to retrieve the objects that are registered in the system as SOS vehicles. However, we want to alert any vehicle that is a potential helper. For example, a physician in a motorbike could come very handy if he/she could be the first to arrive.

Such knowledge could be automatically inferred if the system was able to use the ontology in Figure 1. In this case, the previous task can be formulated as continually retrieving the instances of the concept *AvailableSOSVehicle*, equivalent to the query *SOSVehicle* and (*available* value *true*). The system would retrieve the ambulances that would be available, but also all the available vehicles that are carrying people who have medical jobs and therefore would be able to help. Having all the instances of the ontology correctly classified depends directly on several properties that can change at any time: *available*, *occupy*, *occupant*, etc. Consequently, due to the existence of these volatile properties, the facts inferred at one moment can become obsolete quickly.

However, note that maintaining the instances classified and updated is not feasible, as the reclassification process is too expensive to be performed every time the value of a volatile property changes. Moreover, it could be potentially never-ending, as changes could happen so often that the inferred consequences could be already obsolete when the reasoning finishes, as the property values might have changed again, and so it must be recomputed, and so on.

This example could be solved using a traditional DL reasoner only if the updates to the ontology are infrequent. However, there are some properties that can have continuous changes, such as the position of a vehicle. In such cases, new solutions are needed.

We use the scenario of LOQOMOTION only as a motivating example, as we think that the use of ontologies would be a great help in this kind of contexts. However, we will focus on describing a centralized deployment of our approach, as a distributed deployment would address issues that we consider that are out of the scope of this paper.

It is also worth noting that the ontology in our running

example is relatively expressive –it uses the expressivity of the Description Logic $\mathcal{ALCIQ}(\mathbf{D})$– and it cannot be supported by other approaches similar to ours (see Section 9 for a longer discussion).

3. Overview of our System

In this section, we present a general overview of our system, hereinafter referred to as QueryGen, describing its aim and general architecture. We would like to clarify from the beginning that it is not a DL reasoner, but a system for the management of dynamic data, which combines the advantages of DL reasoners and traditional Database Management Systems (DBMS) to efficiently process the updates of the knowledge base and answer queries by retrieving the instances that are relevant to a given DL concept expression (handling both static and dynamic data). Our approach trades off completeness of the reasoning for a reduction of the response time, as suggested in (Dentler et al., 2011).

To this end, we firstly preprocess the static knowledge by classifying the ontology (i.e., computing the class and property hierarchies) and by computing an instance retrieval test over all the static concepts (i.e., retrieving all the instances of the static concepts). Since static data cannot change, this is more efficient than computing the instances of the static concepts before answering every query. The next step is to use a database to store the instances of the static concepts[4] and the object property assertions involving volatile properties. Then, whenever there are changes in the property values, we update the database keeping the semantics of the data. Finally, continuous query answering can be performed by using the DL reasoner if the query does not involve volatile properties, or by using the database otherwise.

Since data streams are inevitably noisy (Stuckenschmidt, Ceri, Della Valle, & van Harmelen, 2010), stream reasoning requires some mechanism to deal with spurious data. In our approach, whenever there is a contradiction between the static classified knowledge and the dynamic updates, we give preference to the static knowledge, which is not expected to change. The intensional knowledge (the TBox) is expected to be stable and unchangeable as it represents a modelization of the domain, while the extensional knowledge (the ABox) stores the particular state of the objects of the domain; so, the TBox is static and the ABox can be dynamic. Hence, in case of contradiction we give preference to the inferences obtained by using the stable knowledge schema rather than the possibly noisy assertions.

We present now the architecture of QueryGen, describing its four main components, depicted in Figure 2, and then discussing their interactions with a running example:

- *Continuous Query Processor* (CQP): It is the main module of the architecture and orchestrates the rest of the modules to obtain answers to the different continuous queries.
- *Knowledge Storage*: It is the module that stores the ontology properly classified according to the static knowledge, as well as some information about the volatile properties, namely their definitions and the concepts that are affected by them. It also provides reasoning services (as a normal DL reasoner does) to help the *Continuous Query Processor* to build the execution plan for each query and to populate the initial tables. As we will see, this component uses a DL reasoner to compute an initial classification of the static part of the ontology and to solve several subsumption tests in order to simplify the query representation.
Volatile properties must not be populated by asserting their values in the *Knowledge Storage* (i.e., the ABox must not contain any assertion about their values). Otherwise, the volatility of the asserted data could lead to temporal inconsistencies when reasoning with the DL reasoner, which would imply an invalidation of all the knowledge inferred up to that point; or even worse, it could lead to a never-ending reclassification process, as already discussed in Section 2.2.
- *Data Storage*: It stores the volatile knowledge using different tables needed to carry out the execution plan. Intuitively, these tables can be thought of as the concepts participating in the query; however, there is not a one-to-one correspondence between the tables and the concepts in the ontology, as one table can store the instances of a static expression. There are two sources of information to populate them: the static knowledge obtained from the *Knowledge Storage* (i.e., the extension of the ABox), and the volatile data updated via the *Update Wrapper* (i.e., the actual values that are asserted for the different volatile properties during the query lifetime).
- *Update Wrapper*: This module is in charge of managing updates to the different volatile properties and, with the help of the *Knowledge Storage*, updates properly the tables in the *Data Storage* that correspond to volatile properties.

The workflow of our architecture is as follows:

1. An initial step receives a reference ontology and information about the volatile parts provided by the ontology developer, and performs some preprocessing (Figure 2, step 1).
2. The *Continuous Query Processor* analyzes the query and obtains an execution plan which separates the static part from the volatile parts of the query (Figure 2, step 2) .
3. The *Data Storage* creates and updates the tables that are needed to solve the query plan (Figure 2, step 3).
4. The *Knowledge Storage* populates the tables using the instances that are statically classified by the reasoner (Figure 2, step 4).
5. The *Continuous Query Processor* solves an SQL query to retrieve freshly updated data (Figure 2, step 5).
6. The *Update Wrapper* manages update requests for the values of the volatile properties (Figure 2, step 6).

The intuitive idea behind this architecture is to defer as much as possible the processing of volatile knowledge, that is, the processing of the extension of volatile properties (their definitions are part of the static knowledge of the ontology).

We will explain now how QueryGen exploits both the static and volatile knowledge to be able to retrieve correct answers in a continuous way. First of all, we present the preprocessing step (Section 3.1) that has to be performed before both the query processing (Section 4) and the volatile information updating (Section 5).

[4] We create one table per static concept.

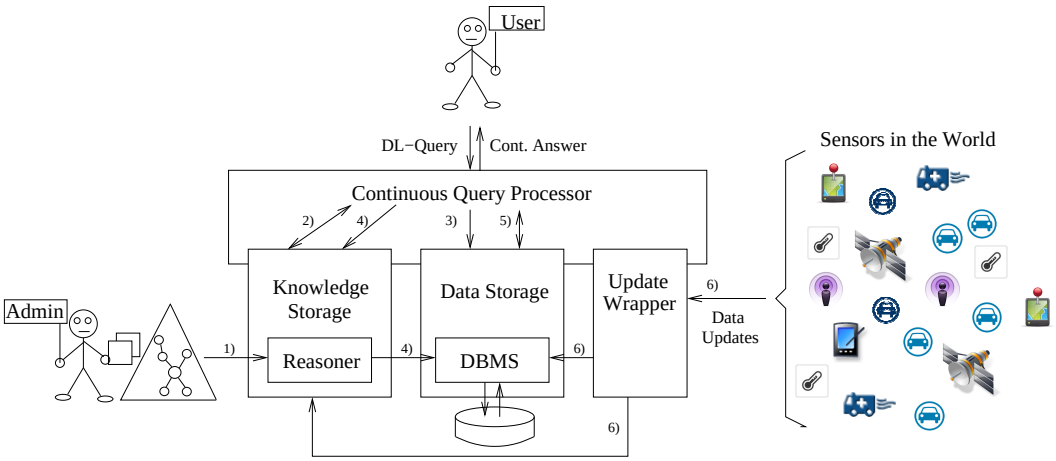


Figure 2. Overview of the main modules of the query processor.

Preprocessing

Previously to the query processing, QueryGen has to be provided with the ontology to be used (Figure 2, step 1). This ontology must have their volatile properties annotated, and this has to be done manually by the ontology developer as the volatility of a property depends directly on the domain that is being modelled. This assumption would be justified in detail in Section 6.3.

After this step, the *Knowledge Storage* analyzes this information off-line to annotate the rest of the entities (i.e., concepts and properties) of the ontology as volatile or static. The *Knowledge Storage* computes the propagation of the volatility among the properties and defined concepts. Intuitively, a defined concept whose definition has a constraint on a volatile property becomes volatile as well. However, as we will see, the subclass of a volatile class is not treated as volatile. Hence, the volatility of a class is propagated upwards but not downwards.

In the example shown in Figure 1, *available*, *isDriving*, *occupy*, *isBeingDriven* (the inverse property of *isDriving*) and *occupant* (the inverse property of *occupy*) are volatile properties. Hence, the following concepts and queries (defined concepts using a volatile property) are marked as volatile: *SOSVehicle*, *CrowdedVehicle*, *BalancedVehicle*, *LowOccupiedVehicle*, *AvailableSOSVehicle*, etc.

Next, we preprocess the static knowledge by classifying the ontology and by computing an instance retrieval over all the static concepts, in order to process the static knowledge more efficiently. To consider only the static knowledge, this initial classification does not take into account the values of the volatile properties (recall that volatile properties must not be populated in the ontology itself).

In the following sections, we detail how this information is processed and updated to answer the queries presented to QueryGen.

4. Query Processing

Once the user has posed a query to QueryGen, the *Continuous Query Processor* (CQP) analyzes the query and obtains an execution plan which separates the static parts from the volatile parts of the query.

As we have seen in the previous section, there are two concurrent tasks that QueryGen has to perform: query processing and data updating. In this section, we focus on the former one, detailing the steps that the *CQP* has to perform to obtain a proper answer for a continuous query, while the latter part will be discussed in Section 5. In brief, the query processing is composed by four steps:

1. *Query analysis*: the *CQP* analyzes the query to obtain an executable plan. To do so, the *CQP* follows a four-step process: 1) *volatility propagation*, 2) *query expansion*, 3) *semantic reduction of the query*, and 4) *executable plan obtention*.
2. *Creation and population of data tables*: the *Data Storage* creates and maintains the necessary tables for this query.
3. *Translation into relational algebra*: the *CQP* retrieves the names of the tables that are participating in the query, and translates the query tree into a relational algebra expression that will lead to an executable SQL query.
4. *Query execution*: QueryGen continuously executes the SQL query to obtain the answers to the query.

In the following subsections, we give a thorough description of these steps.

4.1. Query Analysis

In the following, we detail each of the above-mentioned four steps that the query analysis comprises.

4.1.1 Volatility Propagation. QueryGen uses a tree-based representation of the queries (originally proposed in (Bobed, Trillo, Mena, & Bernad, 2008)) where, in addition, the branches are tagged as static or volatile parts of the query. The volatility of the properties is propagated upwards to determine which branches have to be processed in a continuous

way (i.e., the volatile branches). This process requires imposing a restriction on the TBox: we assume that there are no cycles in the definition of volatile concepts.

In Figure 3, the query that serves as a running example is expressed in this way. For the sake of clarity, in the tree we transform Manchester syntax into a prefix notation. Note that the volatile expression *SOSVehicle* is expanded to its definition *Vehicle* and (*occupant* some *Physician*).

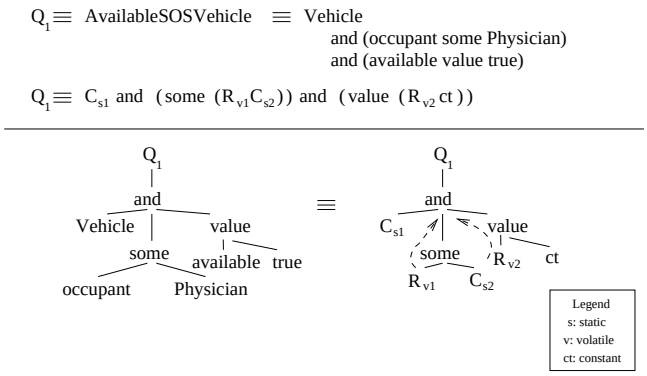


Figure 3. Our running example represented as a query tree: the volatility is propagated upwards to mark the branches to be processed in a continuous way.

4.1.2 Query Expansion. The analysis of the query is oriented to obtain appropriate expressions that describe the sets of objects that are affected by the evaluation of constraints on volatile properties. In Figure 4, the representation of a generic evaluation is shown. For a generic volatile property R_v, the set of objects that could fulfill the constraint on it is given by the following expression:

$$Domain(R_v) \cap DomModifier$$

with *DomModifier* being an expression that might affect the actual domain of the property in the query. For brevity, we will sometimes abbreviate $Domain(R_v)$ as $D(R_v)$.

Similarly, the set of objects the property can take values from is given by the following expression:

$$Range(R_v) \cap RangeModifier$$

with *RangeModifier* being an expression that might affect the actual range of the property in the query.

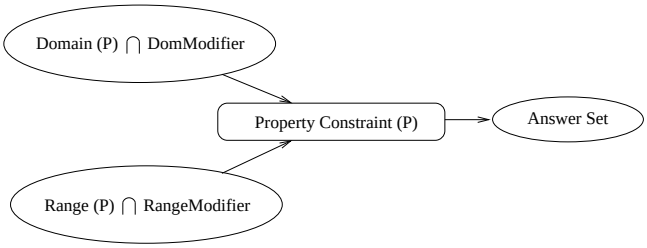


Figure 4. Set-based representation of a generic property constraint.

To obtain such expressions, the *CQP* expands the query tree (this requires some restrictions on the TBox, that will be discussed in Section 6.1):

- Those defined concepts such that a volatile property participates in their definition (*volatile concepts*) are substituted in the tree by their definitions. The system keeps track of which defined concept was substituted to create a *branch* in the query plan tree. This is done because a volatile concept is composed by the union of two sets: the instances statically classified as its members compose its *static extension*, while the instances satisfying the expanded definition (considering the volatile data) compose its *volatile extension*. This information will be used later when building the final query.
In the motivating example, a query involving *SOSVehicle* should retrieve both the instances statically classified as SOS vehicles and those vehicles having a physicist as one of their occupants.
The volatile superclasses of a static class cannot be treated in the same way. For instance, *Ambulance* is static even if it is a subclass of the volatile class *SOSVehicle*; instances that become SOS vehicles do not necessarily become ambulances, as this is not a necessary and sufficient condition.
- The constraints on volatile properties are substituted by the conjunction of their domains and the constraints themselves. This way, we can obtain the actual set of objects that have to be checked against the constraint.

In Figure 5, the actual expanded query is shown. In the first step, the *CQP* unfolds the *SOSVehicle* definition and marks the node as defined (defined nodes in Figure 5 are marked with an asterisk). As commented before, this is needed to keep track of both the static and the dynamic extension of the defined concept. In the second step, the domains of the volatile properties are added. Recall that the domain of *occupant*, denoted by $D(R_{v1})$, is *Vehicle*, and the domain of *available*, denoted by $D(R_{v2})$, is *LObject* (*Locatable Object*), both static and primitive concepts.

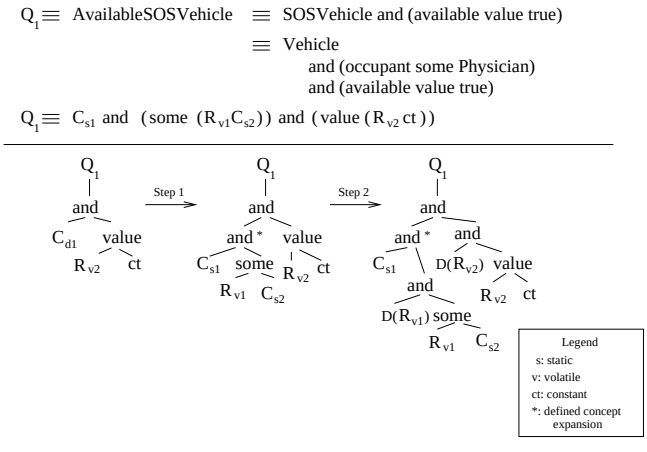


Figure 5. Example of the query expansion process.

4.1.3 Semantic Reduction. Once the query has been completely expanded, QueryGen compacts the tree according to

the properties of the operators. This process is called *semantic reduction* and enables QueryGen to reduce the number of subexpressions in the query, resulting in a reduction in the number of tables to be maintained and avoiding redundancies between equivalent expressions. Symmetry and associativity of the operators are taken into account to compact the query tree (Figure 6, step 1). This step is not applied to the nodes that are marked as defined due to the aforementioned reasons: the query plan must compute the union of its *static extension* and its *volatile extension*. Otherwise, we would be missing this information in the reduction.

Then, QueryGen considers the *restrictiveness* and *inclusiveness* of the operators (Bobed, Trillo, Mena, & Ilarri, 2010) (e.g., the *and* is restrictive and the *or* is inclusive) to further reduce the query tree. Using a classical DL reasoner to solve some subsumption tests, we can prune our query tree as follows:

- If a restrictive operator has two child nodes C and D in the query tree such that C subsumes D, then C is removed, i.e., the most specific expression is kept.
- If an inclusive operator has two child nodes C and D in the query tree such that C subsumes D, then D is removed, i.e., the most general expression is kept.

In Figure 6, step 2, note how the two nodes *Vehicle* (C_{s1} and $D(R_{v1})$) have been merged as they are equivalent and are under a *restrictive* operator, and how the node *Vehicle* ($D(R_{v2})$ has been reduced as it is subsumed by the node marked with an asterisk ($SOSVehicle$).

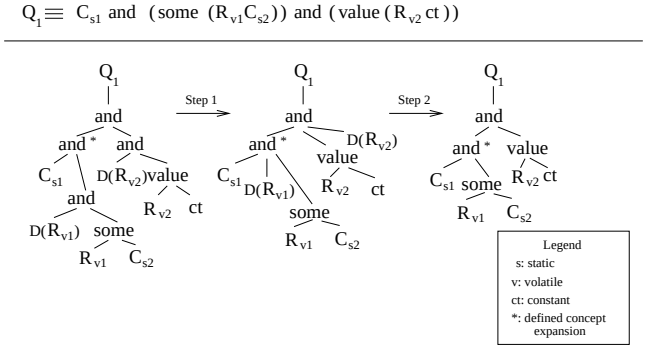


Figure 6. Example of the semantic reduction process.

4.1.4 Executable Plan Obtention. Finally, QueryGen obtains an executable plan. For efficiency, part of the semantic reduction described in the previous section is performed concurrently with the obtention of the query plan. The executable plan contains the tables that have to be created and populated, and the order in which the different parts of the query have to be processed. The query is partitioned in levels that correspond to the levels in the final relational algebra expression that QueryGen uses to calculate continuously the volatile dimension of the query. So, QueryGen is able to solve a full query reusing the tables that participate in different levels.

To reduce semantically the query and obtain this plan, QueryGen applies Algorithms 1 and 2.

Algorithm 1 analyzeQuery(q, reasoner)

Require: q is the query that must be analyzed, *reasoner* is an object that encapsulates the reasoning service offered by the *Knowledge Storage*.
Ensure: returns an executable plan for the query q.
1: *plan* ← ∅
2: *analyzeQTNode*(*q.root*, *reasoner*, *plan*, 0); // see Algorithm 2
3: *return plan*;

Algorithm 2 analyzeQTNode(node, reasoner, plan, level)

Require: *node* is a node of the query tree, *reasoner* is an object that encapsulates the reasoning services offered, *plan* is the plan being built, and *level* is the plan level that corresponds to the current node.
Ensure: Analyzes the input node and builds recursively the plan.
1: **if** plan.levels == level **then**
2: // if they are equal, a new level has to be created in the plan
3: plan.createLevel();
4: **end if**
5: **if** node.isNonTerminal() **then**
6: // it is an intermediate node
7: // it has to check whether any of the subexpressions has to be reduced
8: **if** node.getOperator().isRestrictive() **then**
9: reduceRestricted(node, reasoner);
10: **else if** node.getOperator().isInclusive() **then**
11: reduceIncluded(node, reasoner);
12: **end if**
13: // we now separate static from volatile children
14: separateChildren(node, volatileChildren, staticChildren);
15: plan.put(node, staticChildren, level);
16: **for all** *volatileChild* ∈ *volatileChildren* **do**
17: analyzeQTNode(volatileChild, reasoner, plan, level+1);
18: **end for**
19: **else**
20: // it is a leaf node
21: **if** node.isVolatile() **then**
22: plan.put(node, level);
23: **end if**
24: **end if**

Algorithm 2 creates a new level when a call has increased the level needed (lines 1–4). It visits recursively the nodes in the query tree in a depth-first way. If the node is an operator which is restrictive or inclusive (Bobed et al., 2010) then a semantic reduction is applied (lines 8–12): neither the nodes restricted nor the nodes included by their siblings are needed to obtain the answer. As anticipated, given a restrictive operator, if there is a child node that is subsumed by one sibling, it has to be removed as it does not add any description to the query. Then, the children are separated as volatile and static ones (line 14), to assign the static ones to the current level, and to open a new level for the volatile ones (lines 15–18). Finally, when a leaf node is treated, if it is volatile, it is assigned to the current level (lines 21–23), as the static ones have been already assigned to a previous level.

An example of the resulting plan for our running example is shown in Figure 7. In level 0 we have the top operator,

in this case *and*. Level 1 contains the static concept *Vehicle*, the value *true* appears, as well as two operators, namely *value* and the node indicating that a defined concept was expanded. In level 2, the volatile property *available*, and the *some* restriction link to the previous level. Finally, level 3 just contains the volatile property *occupant*. Of course, before obtaining the answer set (ASet) of a level, the answer sets of its sub-levels must have already been obtained.

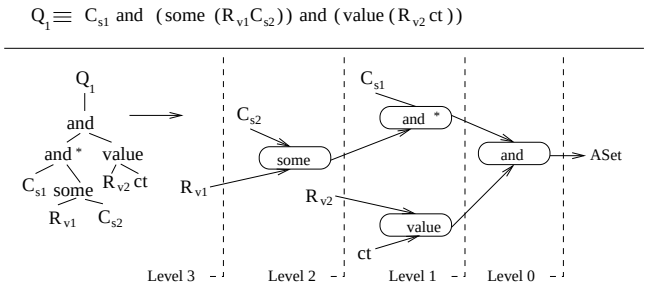


Figure 7. Example of obtention of the execution plan.

We can state the following result showing the validity of the expansion of the query.

Proposition 1. *Given an ontology O and a DL concept (or query) Q, the expanded query tree Q′ verifies that every instance of Q′ is an instance of Q.*

It follows directly from the following facts:

- Defined concepts are substituted by their definitions, which are stated to be equivalent.
- Constraints with volatile properties are substituted by the expression $Dom(R_v)$ and $constraint(R_v)$, which is semantically equivalent.
- The semantic reduction is performed using the properties of the operators (discussed in (Bobed et al., 2010)) and with the help of the reasoner by eliminating only redundant concepts under sufficient conditions.

4.2. Creation and Population of Data Tables

Once the query has been analyzed and we have an execution plan, the *CQP* knows which tables have to be created and maintained in the *Data Storage* for this query.

In particular, it creates a table *Thing*, containing all the instances of the ontology, a table for each volatile property, and a table for each static expression in the execution plan. The tables have the following format:

- For static expressions or concepts, the tables are named as their corresponding concept/expression and the only field they have is the identifier of its instances $C = (idString)$.
- The object and datatype properties are represented in the database just as tables named as their corresponding property, with two fields: the subject and the target of each instance of the relationship.

$$R_{objProperty} = (sbjString, tgtString)$$
$$R_{dataProperty} = (sbjString, value\ dataType)$$

The *Data Storage* stores these tables, keeping track of the expressions that are stored in each table to maximize reusing them among different queries. Then, the *Knowledge Storage* populates the tables using the instances that are statically classified by the ontology.

In our example, the properties *available* and *occupant* will have their own table because the former one appears in the query and the latter one in the expansion of a concept in the query. These tables will be properly updated by the *Update Wrapper*. Then, three tables are also created for each of the static knowledge expressions *Vehicle* and *Physician*, and the static extension (i.e., the statically classified instances) of *SOSVehicle*. In this example every table corresponds to exactly one ontology concept, but in general we may have a more complex static expression appearing in the query execution plan. Then, QueryGen asks the DL reasoner to retrieve all the instances of the concepts corresponding to the complex static expressions and populates the tables by creating one tuple for each instance.

4.3. Translation into Relational Algebra

The translation of the query execution plan is performed by applying recursively the translations shown in Table 1, where C is a static concept expression, R_v is a volatile object property, T_v is a volatile datatype property, $\tau(C)$ denotes the translation of C into a table, and $Domain(R_v)$ and $Range(R_v)$ are functions that return the tables corresponding to the domain and range of R_v, respectively. The reader not familiar with relational algebra operations is referred to (Codd, 1990). The nodes in the plan marked as defined ones are translated into the union of the concept table and the translation of their definition to retrieve both the static and the volatile dimensions.

In our running example, for *AvailableSOSVehicle*, the final SQL query obtained by QueryGen is shown in Figure 8:

```
SELECT subQuery4.subject    // extension
FROM (
    SELECT SOSVehicle.subject
    FROM SOSVehicle
// static part of SOSVehicle
    UNION DISTINCT
        SELECT subQuery5.subject
        FROM Vehicle
// dynamic part of SOSVehicle
        AS subQuery5
        WHERE subQuery5.subject IN( // gets the Vehicles
            SELECT occupant.subject // with an occupant
            FROM occupant           // classified
            INNER JOIN Physician    // as Physician
            AS subQuery6
            ON occupant.object =
               subQuery6.subject
        )
)
AS subQuery4
WHERE subQuery4.subject IN (    // gets the SOSVehicles
    SELECT available.subject    // (dynamic + static)
    FROM available              // that are available
    WHERE available.value = 'true'
)
```

Figure 8. Final SQL query obtained by QueryGen for *AvailableSOSVehicle*.

We are aware of the subtle change of semantics that this translation implies, as it moves from the world of the DL

Table 1
Translation of DL class expressions.

τ(A)	A
$\tau(C_1 \text{ or } C_2)$	$\tau(C_1) \cup \tau(C_2)$
$\tau(C_1 \text{ and } C_2)$	$\tau(C_1) \cap \tau(C_2)$
τ(not C)	$Thing - \tau(C)$
τ(Thing)	$Thing$
τ(Nothing)	τ(not Thing)
$\tau(\{i_1, \ldots, i_n\})$	$\sigma_{id=\{i_1,\ldots,i_n\}}(Thing)$
$\tau(R_v \text{ only C})$	$(Domain(R_v) - \Pi_{R_v.src}) \cup (\sigma_{count(R_v,R_v.sbj)=count(R_v \bowtie_{(R_v.tgt=\tau(C).id)} \tau(C),R_v.sbj)}(R_v))$
$\tau(R_v \text{ some C})$	$\Pi_{R_v.sbj}(R_v \bowtie_{R_v.tgt=\tau(C).id} \tau(C))$
$\tau(R_v \text{ self})$	$\Pi_{R_v.sbj}(R_v \bowtie_{R_v.sbj=R_v.tgt} R_v)$
$\tau(R_v \text{ exactly n})$	$\Pi_{R_v.sbj}(\sigma_{count(R_v,R_v.sbj)=n}(R_v))$
$\tau(R_v \text{ exactly n C})$	$\Pi_{R_v.sbj}(\sigma_{count(R_v,R_v.sbj)=n}(R_v \bowtie_{R_v.tgt=\tau(C).id} \tau(C)))$
$\tau(R_v \text{ max n})$	$\Pi_{R_v.sbj}(Domain(R_v) - \sigma_{count(R_v,R_v.sbj)>n}(R_v))$
$\tau(R_v \text{ max n C})$	$\Pi_{R_v.sbj}(Domain(R_v) - \sigma_{count(R_v,R_v.sbj)>n}(R_v \bowtie_{R_v.tgt=\tau(C).id} \tau(C)))$
$\tau(R_v \text{ min n})$	$\Pi_{R_v.sbj}(\sigma_{count(R_v,R_v.sbj)\geq n}(R_v))$
$\tau(R_v \text{ min n C})$	$\Pi_{R_v.sbj}(\sigma_{count(R_v,R_v.sbj)\geq n}(R_v \bowtie_{R_v.tgt=\tau(C).id} \tau(C)))$
$\tau(T_v \text{ only D})$	$(Domain(T_v) - \Pi_{T_v.src}) \cup (\sigma_{count(T_v,T_v.sbj)=count(\sigma_{T_v.value\in D}(T_v),T_v.sbj)}(T_v))$
$\tau(T_v \text{ some D})$	$\Pi_{T_v.sbj}(\sigma_{T_v.value\in D}(T_v))$
$\tau(T_v \text{ exactly n})$	$\Pi_{T_v.sbj}(\sigma_{count(T_v,T_v.sbj)=n}(T_v))$
$\tau(T_v \text{ exactly n D})$	$\Pi_{T_v.sbj}(\sigma_{count(T_v,T_v.sbj)=n}(\sigma_{T_v.value\in D}(T_v))$
$\tau(T_v \text{ max n})$	$\Pi_{T_v.sbj}(Domain(T_v) - \sigma_{count(T_v,T_v.sbj)>n}(T_v))$
$\tau(T_v \text{ max n D})$	$\Pi_{T_v.sbj}(Domain(T_v) - \sigma_{count(T_v,T_v.sbj)>n}(\sigma_{T_v.value\in D}(T_v)))$
$\tau(T_v \text{ min n})$	$\Pi_{T_v.sbj}(\sigma_{count(T_v,T_v.sbj)\geq n}(T_v))$
$\tau(T_v \text{ min n D})$	$\Pi_{T_v.sbj}(\sigma_{count(T_v,T_v.sbj)\geq n}(\sigma_{T_v.value\in D}(T_v)))$

reasoner (the static part of the ontology, classified once by the DL reasoner), where the Open World Assumption is imposed, to the world of the database manager, that implies the Closed World Assumption (CWA). This issue will be discussed in more detail in Section 6.3. However, we are not concerned about this change of semantics because our goal is to provide a correct answer as fast as possible rather than keeping the classification of the instances up-to-date and, as we have seen before, this is impossible due to the continuous changes of the data.

The following results show the correctness of the reduction into relational algebra. We will firstly consider the static part of the query and then the volatile part of the query.

Proposition 2. *Given an ontology O, a static concept C, and an individual i, there is a tuple $t = [i]$ in the database such that $t \in \tau(C)$ if and only if the ontology entails the axiom C(i) under OWA.*

This proposition is trivial since QueryGen manages static concepts by populating the database exactly with the results returned by the DL reasoner .

Proposition 3. *Given an ontology O, a volatile concept C, and an individual i, if there is a tuple $t = [i]$ in the database such that $t \in \tau(C)$ then the ontology entails the axiom C(i) under CWA.*

This proposition can be shown by induction on the structure of concepts. The case base (atomic concepts) is trivial since we create a new table for each atomic concept and we populate it exactly with its instances. After the induction step, the other cases can easily be shown by noting that the relational algebra translation preserves the semantics of the corresponding concept. Let us discuss some cases:

- The first 6 cases are trivial for the reader familiar with relational algebra.
- In the case of an enumeration, the translation produces a selection of the tuples that contain the values $i_1, \ldots, i_n$.
- In local reflexivity concepts, the translation retrieves tuples denoting individuals that are related to themselves by computing tuples that are related to some individual via R_v such that the subject and the target of the relation are the same individual.
- The idea of the translation of $\tau(R_v \text{ some C})$ is similar: it retrieves the tuples that are related to some individual via R_v such that it also belongs to the table $\tau(C)$.
- The translation of $\tau(R_v \text{ only C})$ is more involved. It retrieves those tuples that are not related to any individual via R_v ($Domain(R_v) - \Pi_{R_v.src}$) plus ($\cup$) those tuples that are related via R_v but only to instances of the table $\tau(C)$. This latter part is computed by checking that the number of relations via R_v is the same as the number of relations via R_v with members of $(\sigma_{count(R_v,R_v.sbj)=count(R_v \bowtie_{(R_v.tgt=\tau(C).id)} \tau(C),R_v.sbj)}(R_v))$.
- The translations of the cardinality restrictions are similar but also taking into account the number of tuples that satisfy the conditions. In fact, the translation of τ(R min n) extends that of $\tau(R_v \text{ some C})$ by checking that the number of relations via R_v is at least n ($\sigma_{count(R_v,R_v.sbj)} \geq n$). The translation of τ(R exactly n) extends that of $\tau(R_v \text{ some C})$, and the translation of τ(R max n) extends that of $\tau(R_v \text{ only C})$. The case of unqualified restrictions and restrictions involving data properties are trivial variations of these ideas.

The converse of Proposition 3 is not true in general, as it will be illustrated in Example 5.

4.4. Query Execution

Finally, QueryGen continuously executes the SQL queries computed by the previous step to retrieve the relevant data that constitute the answers to the queries. We adopt a solution based on the use of appropriate refreshment rates, as proposed in (Ilarri et al., 2006; Ilarri, Mena, & Illarramendi, 2008).

Thus, as the involved data are constantly changing, the processing tasks must be performed with a certain *task frequency* to limit the cost of the query processing. In (Ilarri et al., 2008), several approaches are studied to deal with situations where the delays involved do not allow achieving the desired frequency, including (among others) a proposal that automatically adjusts QueryGen to the maximum supported frequency.

5. Volatile Data Updating

Concurrently to the query answering process, once the tables have been created and populated, the *Update Wrapper* begins accepting update requests for the volatile properties that are involved in the plans, and, using the intensional knowledge stored in the *Knowledge Storage*, it updates the values stored in the *Data Storage*.

As it has been stated above, the extension of the volatile properties is stored in the *Data Storage*. To ensure the consistency of the data, QueryGen relies on a wrapper that treats the different data updating requests.

In general, the addition of new volatile knowledge is treated as insertions in a table, the modification of volatile knowledge is treated as an update of the table, and the retraction of volatile knowledge is treated as a deletion from the table. Furthermore, this wrapper uses the knowledge asserted in the *Knowledge Storage* to check the validity of the possible updates and to maintain the data well updated according to the properties of the volatile properties. In particular, the *Update Wrapper* deals with:

- *Inverse object properties*: inserting/deleting (a, b) in a table T_R implies inserting/deleting (b, a) in T_S for every object property S which is inverse of the object property R; similarly, updating (a, b) with (a, c) in R implies updating (b, a) with (c, a) in S.
- *Functional object/datatype properties*: inserting (a, b) into a table T_R implies updating if there exists a previous value of the form (a, x).
- *Disjoint object/datatype properties*: it has to check that, when inserting new values, the tuple (a, b) is not participating in the other disjoint properties.
- *Reflexive object properties*: the instances that belong statically to the domain of the volatile properties have to be inserted into the relationship.
- *Irreflexive object properties*: when inserting or updating, the wrapper has to check that the new value for the subject of the relationship is not equal to the target object.
- *Symmetric object properties*: inserting/deleting (a, b) in/from R also implies inserting/deleting (b, a) in/from R. The same is applied to updates.
- *Asymmetric object properties*: when inserting (a, b) in R, the wrapper has to check that (b, a) is not in R. The same is applied to updates.

It should be noted that, since the DBMS does not have a reasoning mechanism, we need a module responsible for keeping the semantics of the data. The existence of an external updater allows us to process the updates according to the semantics that are stored in the *Knowledge Storage*. These updates should not be encapsulated in database triggers, as several of them would need to be re-entrant (i.e., they would affect the triggering table itself) and re-entrant triggers are usually heavily constrained in existing DBMS. As an example of re-entrant trigger, maintaining symmetric object properties with a trigger would require inserting in the same table that fired the trigger.

Moreover, this architecture enables to easily attach several data sources (sensors, location readers, etc.) to the query processor, as the *Update Wrapper* can export its services in several ways (software libraries, Web Services, etc.).

Another important observation is that DBMS are optimized to access data and usually know how to reuse the results of previous queries. Hence, the DBMS is able to reuse the unchanged part of the volatile part of the ontology without requiring an extra effort from the system proposed in this paper.

Once we have presented the different steps of our system, we can state now the following results about our approach.

Proposition 4. *Given an ontology O, a volatile property R, its associated table T_R, and two individuals i_1, i_2, if there exists a tuple $t = [i_1, i_2]$ in the database such that $t \in T_R$ then $R(i_1, i_2)$ holds.*

Notice that the converse is not true in general. For example, as we will discuss in Section 6.1, QueryGen does not currently handle transitive roles.

Proposition 5. *Given an ontology O and a query concept Q, QueryGen only retrieves instances of Q.*

This latter result can be shown by noting that our query expansion process builds a correct plan (Proposition 1) and that our translation into the relational algebra preserves the semantics of the query concept (Propositions 2–4).

6. Syntax and Semantics

In this section, we detail the syntax and the semantics of the language supported by QueryGen. Then, we summarize all the assumptions imposed so far.

6.1. Syntax

If there are only static properties, the syntax and semantics are exactly that of OWL 2 (see the appendix for details).

QueryGen behaves exactly as a DL reasoner, and what it actually does is just call it. This is however a very unlikely scenario because QueryGen is built for scenarios where volatile properties appear.

When there are volatile properties, we have the following restrictions:

(R1) Properties are annotated so the system can be aware of their volatile/static nature. In particular, we use the annotation property "dc:type" with values "volatile" or "static", respectively.

(R2) Every property appearing in a subproperty axiom must be static.

(R3) Every property appearing in a subproperty chain axiom must be static.

(R4) Every property appearing in a property definition must be static.

(R5) We cannot assert that a volatile property is transitive.

(R6) The domain of a volatile property cannot be a volatile concept.

(R7) The range of a volatile property cannot be a volatile concept.

(R8) If the TBox has an axiom of the form $C := D$ for a volatile concept D, C is an atomic concept.

(R9) If the TBox has an axiom of the form $C := D$ for a volatile concept D, there is no other axiom of the form $C := D'$ for a volatile concept D'[5].

(R10) The class definitions of volatile concepts cannot contain cycles.

Restriction (R1) is needed because our system cannot automatically discover volatile properties, as discussed in Section 6.3. Restrictions (R2)–(R5) are necessary because QueryGen is not currently able to propagate the changes according to their semantics. We also need restrictions (R6)–(R10) to expand the query tree in a correct way. For instance, (R10) is needed to guarantee the termination of the expansion process.

Hence, when volatile properties are used, the supported expressivity is a subset of OWL 2. The supported fragment is an extension of the DL $\mathcal{ALCOIQ}(\mathbf{D})$ with limitations on the ABox but with additional role constructors, namely universal roles, negated role assertions and disjointness, reflexivity, irreflexivity, symmetry, and asymmetry of roles. For an explanation of the names of the different DLs, we refer the reader to (Baader et al., 2003).

6.2. Semantics

As explained in Section 6.1, if there are only static properties, the syntax and semantics are exactly that of OWL 2. When volatile properties are involved, we require the Unique Name Assumption and the Closed World Assumption in order to take advantage of databases to solve queries, which is one of the key features of QueryGen. The reader is referred to Section 6.3 for a discussion about these two paradigms and a justification of our choice.

It is worth discussing how this can change the semantics of the ontological volatile concepts. In particular, we will focus on their volatile dimension[6]. Let us consider the cases that make a difference in detail:

- For QueryGen, R_v some C does not contain an individual a if there is not a known property value R(a,b) such that C(b) holds. This is not the case in OWA, where this could also be true for the unknown values of R.
- For QueryGen, R_v only C is composed by every individual a such that for every known property value R(a,b), C(b) holds. This is not enough in OWA, where C(b) might not hold for unknown values of $R - fillers$ b.
- For QueryGen, R_v min n C does not contain an individual a if there are not n known property values R(a,b) such that C(b) holds. This is not the case in OWA, where this could also be true for the unknown values of R.
- For QueryGen, R_v max n C is composed by every individual a such that there are at most n property values R(a,b) such that C(b) holds. This is not enough in OWA, where C(b) might not hold for unknown values of $R - fillers$ b.
- For QueryGen, R_v exactly n C is composed by every individual a such that there are exactly n property values R(a,b) such that C(b) holds. This is not enough in OWA, where C(b) might not hold for unknown values of $R - fillers$ b.

6.3. Assumptions

Firstly, we will enumerate the assumptions of our approach and then we will explain them in detail:

- Only volatile property assertions can change.
- Volatile properties must be manually identified by the ontology developer.
- Only instance retrieval is supported.
- Volatile extensions impose some restrictions on the expressivity of the ontology language: some concept constructors and axioms cannot be used with volatile properties.
- Volatile extensions assume the Unique Name Assumption (UNA) and the Close World Assumption (CWA).
- Completeness is not guaranteed in exchange for the efficiency of the reasoning.

Only Volatile Property Assertions Can Change. In our approach, only the ABox (in particular, the volatile property values) can change, but the TBox remains constant. In fact, after the development of the ontology is finished, it seems reasonable to keep the TBox constant because the intensional knowledge is much more stable and unchangeable than the extensional knowledge.

Volatility Marked at Modeling Time. We assume that the ontology developer marks every volatile property. Right now,

[5] This is not an important limitation, as both axioms $C := D$ and $C := D'$ could be replaced by either one axiom of the form $C := (D \text{ or } D')$, or one axiom of the form $C := (D \text{ and } D')$. However, the ontology developer must decide which transformation is more appropriate: in the former case the change on one volatile concept is enough to change the extension of C, whereas the latter case requires a change on each of the involved volatile concepts.

[6] Recall that their static dimension is treated using OWA.

QueryGen is not able to provide any help at this step. For instance, if an object property is volatile, then the ontology developer is responsible of marking as volatile its inverse property as well.

It is worth noting the work in (Ren, Pan, & Zhao, 2010), which is able to automatically detect changed knowledge. In our setting, the separation between static and volatile properties is included in the ontology itself. We consider that the volatility of a property depends heavily on the specific domain that is being modeled (e.g., the locations of buildings in an ontology about buildings are static, whereas the locations of objects in an ontology about moving objects are not), and thus it is subject to a modeling decision. Furthermore, an automatic detection of the volatile properties is challenging since the fact that some property value has not changed for some time does not imply that it will never change in the future (e.g., the fact that a car has been stopped for a long time does not imply that its position is static, as in fact the car may be parked and it could start moving at any time).

Only Instance Retrieval is Supported. The only supported reasoning task for the moment is query answering, i.e., instance retrieval of a DL concept expression. In our view, other reasoning tasks such as consistency checking, concept satisfiability, or concept subsumption, make more sense in a static scenario.

Restrictions on the Expressivity when Dealing with Volatile Properties. As stated in Section 6.1, there are no restrictions on static properties but there are some syntactic restrictions involving volatile properties.

UNA and CWA Assumed when Dealing with Volatile Data. Under *Unique Name Assumption (UNA)*, if two entities have a different name they are considered to be different. This is usually assumed in databases but not in ontology languages, where two individuals only represent different entities if this is logically implied by the ontology. The OWL 2 axiom *DifferentFrom* makes it possible to state that two individuals are indeed different. We adopt the UNA assumption because each entity in our approach will have a unique identifier.

With the *Open World Assumption (OWA)*, the unknown information is not considered to be necessarily false. This assumption is the usual one in ontology languages, because in most cases we only have partial information about the world. For instance, a vehicle in the ontology could be an ambulance, even if it is not asserted to be an ambulance in the ontology. This way, reasoning is monotonic, since the new information cannot contradict previous assumptions.

With the *Closed World Assumption (CWA)*, the unknown information is considered to be false. For example, if there is a list of physicians of a hospital, it is reasonable to think that people not included in the list are not doctors of the hospital. This assumption, which is the usual one in databases, can be done when the knowledge is complete.

A perfect query rewriting[7] requires OWA, but CWA is a reasonable choice in query evaluation (see Section 9). Consequently, when volatile properties are involved, we also require the Closed World Assumption in order to take advantage of databases to solve queries.

Sound, but not Complete. A query system is *sound* if all the retrieved results are correct, i.e., relevant to the original query, and *complete* if all the correct results are retrieved. When considering only static properties, QueryGen is sound and complete. This is very easy to show because we invoke a classical DL reasoner and the knowledge base will not change over time because the properties are static. However, if we want to use volatile properties, the situation is different, as we explain in the following.

QueryGen is sound if the knowledge is consistent. If not, there is a contradiction between the static classification and the dynamic values. In our approach, whenever there is a contradiction between the static classified knowledge and the dynamic updates, we give preference to the static knowledge. The reason is that the static knowledge is not expected to change. On the contrary, data streams are inevitably noisy and could contain spurious data. Hence, QueryGen assumes that the inconsistency is produced by an error in the data and gives preference to the static classification by assuming that it is valid.

Regarding completeness, it is not guaranteed if there are volatile properties, as Example 5 in the next section will show. However, the completeness of the reasoning can be sacrificed in exchange of speeding the query answering process (Dentler et al., 2011).

7. Examples of Query Answering

In this section we will illustrate now the differences between our approach and a classical ontology reasoning approach using six examples. Our objective now is not to motivate our approach as in Section 2.2, but to help the reader understand how QueryGen works.

Example 1. *Let* occupant *be a volatile property, and assume an ontology* O_1 = { SOSVehicle *:=* (occupant *some* Physician)*,* Physician(person1)*,* Vehicle(vehicle1) }.

Firstly, QueryGen create four tables: $T_{Physician}$ represents the concept *Physician*, $T_{Vehicle}$ represents the concept *Vehicle* which is the domain of *occupant*, $T_{SOSVehicle}$ represents the concept *SOSVehicle*, and $T_{occupant}$ represents the relation *occupant*. Furthermore, the table $T_{Physician}$ is initialized to contain a tuple representing the individual *person1*, and $T_{Vehicle}$ is initialized to contain the individual *vehicle1*. At this point, if we submit a query to retrieve the individuals of *SOSVehicle*, the answer would be the empty set. QueryGen and a traditional DL reasoner would provide the same answer.

Example 2. *Consider again Example 1 but assume that, after some time,* occupant(vehicle1*,* person1) *holds, which is possible because* occupant *is volatile.*

[7] *Query rewriting* means to rewrite an ontological query to answer it with the help of another formalism. It is considered *perfect* when it is completely equivalent.

Let $\mathcal{O}'_1 = \mathcal{O}_1 \cup \{occupant(vehicle1, person1)\}$. The system inserts a tuple representing the pair (*vehicle1*, *person1*) into $T_{occupant}$. If we submit the previous query again, now both systems would retrieve individual *vehicle1*. However, the difference is that the classical ontology reasoning needs to classify again before solving the query, which may require an important computational effort, whereas QueryGen simply queries a database, which is faster. It is interesting to emphasize that we are not performing reasoning on the database extension; on the contrary, we take advantage of the database information to answer queries in a faster way.

Example 3. *Let* occupant *be a volatile property, and assume an ontology* $\mathcal{O}_2$ = { Ambulance *:= (*occupant *only* Physician*),* NonPhysician*(*person1*),* Ambulance*(*vehicle1*),* Physician *disjointWith* NonPhysician }.

QueryGen creates tables $T_{Ambulance}$, $T_{Physician}$, $T_{NonPhysician}$, and $T_{occupant}$, inserts a tuple *person1* into $T_{NonPhysician}$, and inserts a tuple *vehicle1* only to $T_{Ambulance}$. Now, if we query for the individuals of *Ambulance*, the answer would be *vehicle1*, both in QueryGen and in a classical DL reasoner.

Example 4. *Consider again Example 3. Now let us see what happens if, after some time,* occupant*(*vehicle1*,* person1*) is asserted in the ontology.*

Let $\mathcal{O}'_2 = \mathcal{O}_2 \cup \{occupant(vehicle1, person1)\}$. After inserting a tuple representing the pair (*vehicle1*, *person1*) into $T_{occupant}$, there is a contradiction regarding *person1*. On the one hand, $T_{NonPhysician}$ contains *person1*, so *person1* is an instance of *NonPhysician*. On the other hand, every instance of *Ambulance*, is also an instance of *(*occupant *only* Physician), so *person1* is an instance of *Physician* as well. However, *Physician* and *NonPhysician* are disjoint.

Let us see what happens when we submit a query to retrieve the individuals of *Physician*. Both approaches would obtain a different result:

- A classical DL reasoner would answer that there is a contradiction, as *person1* is an instance of two disjoint classes.
- Our approach works differently. To avoid the erroneous results that may be obtained when there are volatile properties, and so the potential contradictions, we give more preference to the static classification of primitive concepts (e.g., *Physician* and *NonPhysician* in the example) and use the volatile properties only if they are not inconsistent with the previous information. In this particular case, QueryGen assumes that *person1* is an instance of *NonPhysician* (as stated in the ontology), so the answer would be the empty set. Rather than being counter-intuitive, this result is the effect of giving more preference to the static classification than to dynamic updates, in order to avoid the effect of spurious property values.

As we will see in Section 9, some approaches in the literature behave differently, performing only database reasoning, and being limited to the use of less expressive ontologies.

Finally, we will show a situation where QueryGen answers are not complete.

Example 5. *Let* occupant *be a volatile property, and assume an ontology* $\mathcal{O}_3$ = { Ambulance => *(*occupant *some* Physician*),* Ambulance*(*vehicle1*) }.*

If we query for the individuals of *Ambulance*, the answer would be *vehicle1*, both in QueryGen and in a classical DL reasoner. However, if we query for the individuals of (*occupant* some *Physician*), QueryGen would not retrieve any individual, so it is not complete. The reason is that, since *occupant* is volatile, QueryGen would just retrieve from the DBMS those instances being related to some physician through the occupant relation, which is an empty set.

This situation would not happen if the ontology would contain a definition instead of a subclass axiom, as shown in the following example.

Example 6. *Let* occupant *be a volatile property, and assume an ontology* $\mathcal{O}_4$ = { Ambulance := *(*occupant *some* Physician*),* Ambulance*(*vehicle1*) }.*

Now, when querying for the individuals of (*occupant* some *Physician*), QueryGen would indeed retrieve individual *vehicle1*, as the defined concept would be expanded and its static extension would be taken into account to calculate the answer.

8. Experiments

We have performed an extensive experimental evaluation to validate the interest of our proposal. In this section, basic details of the implemented prototype are explained and some experimental results are shown.

8.1. Experimental Design

The objective of the experiments is to prove empirically three *hypotheses*:

1. QueryGen outperforms classical DL reasoners in the presence of volatile properties.
2. QueryGen is scalable even for very large ABoxes, thanks to the use of databases.
3. QueryGen can support a change rate suitable for applications that continuously need up-to-date data.

Our aim is by no means to compare QueryGen with a specific DL reasoner, but to compare two different paradigms to solve the problem of efficient query answering in highly-dynamic scenarios. Classical DL reasoners are not designed to support frequent changes, whereas QueryGen is able to take advantage of the ability of databases to deal efficiently with changes in big amounts of data. In such scenarios, it makes sense to use specific solutions like ours, but classical DL reasoners are of course more appropriate for other purposes and applications.

Our objective is to support an ontological language as expressive as possible. As we will discuss in detail in Section 9, state-of-the-art ontology reasoners for continuous answering cannot support the expressivity supported by QueryGen. Hence, we will compare our system with OWL 2 reasoners, that can support our expressivity, but not with other less expressive continuous DL answering systems.

Technical Details. Our prototype has been developed using Java 1.6 as the programming language, MySQL 5.6.10 (64 bits) as the DBMS, Pellet 2.2.2 (Sirin, Parsia, Cuenca-Grau, Kalyanpur, & Katz, 2007) as the background DL reasoner, and OWLAPIv3.2 (Horridge & Bechhofer, 2011) as an interface to access Pellet. All the experiments were run on a computer with an Intel Core i5-2320 processor running at 3.00 GHz and 16 GB of RAM memory.

The choice of the DL reasoner can be justified as follows. Besides Pellet, we also performed experiments using HermiT (Rob Shearer & Horrocks, 2008) as the DL reasoner, as it implements the most recent classifying algorithms (Glimm, Horrocks, Motik, Shearer, & Stoilos, 2012). HermiT really worked very well with the TBox, but the processing times obtained to handle our ABox were prohibitive even for testing purposes, which led us to use Pellet[8]. Moreover, Pellet allows using the Unique Name Assumption (UNA) by default, instead of having to relate all the instances using *DifferentFrom* axioms.

We also considered using Fact++ (Tsarkov & Horrocks, 2006) and RacerPro (Haarslev, Hidde, Möller, & Wessel, 2012). However, since QueryGen uses the DL reasoner to compute several subsumption tests, we chose DL reasoners that are fast in ontology classification. Hence, we chose Pellet and HermiT because they are usually faster than Fact++ in ontology classification, as shown in (Rob Shearer & Horrocks, 2008). RacerPro was discarded because it does not completely support OWL 2, since some axioms are approximated or ignored for reasoning.

Used Ontology. The reference ontology used is the one previously presented in Section 2.2, being its expressivity $\mathcal{ALCIQ}(\mathbf{D})$. Different details on the size of the ontology and the distribution of the instances are shown in Table 2. The mean number of datatype property assertions is quite different from the number of object properties, as there is only one datatype property in the ontology and it is volatile; moreover, we do not allow asserting any value for the volatile properties.

Note that the number of axioms is different from the number of RDF triples, as each of the asserted axioms might result in several underlying RDF statements.

Methodology. The comparison with a DL reasoner has been performed by asserting one query at a time, as the inference time with the traditional reasoner-based approach grows quickly with the number of defined concepts and with the expressivity of the definitions. However, our approach did not notice the addition of more definitions using volatile properties: they have no values asserted at query initialization time, so they just have a slight impact on the time needed to classify the static knowledge. This also makes our approach more scalable in terms of the number of concurrent queries that can be processed.

The conducted tests consisted of the execution of 6 refreshments of the queries shown in Table 3. They were repeated for different scenarios with an increasing number of instances. In the general case, we were not able to continue incrementing the number of instances further than 5000 because Pellet was not able to support a higher number. Nevertheless, just in order to test the scalability of our approach, we considered up to 100000 instances.

In our initial experiments, the amount of volatile data that changed between answer refreshments was about 35%. However, in order to evaluate the approaches in a more challenging scenario, we repeated the experiments changing all the volatile data from step to step. So, the experiments reported in this paper consider the challenging situation of a 100% update rate. This will be justified in Section 8.3.

A 100% update rate does not imply processing just one message but 3 multiplied by the number of instances, since every instance has 3 property assertions. Although it is more efficient to group several messages in order to process them in batches (the system could keep a queue of messages and update the system in batches every second), we have also performed some additional tests to measure the change rate when we forced the system to update after receiving every single message.

Table 2
Statistics of the dataset used.

Mean size of the ABox (#Axioms)			
#instances	#Class Axioms	#ObjProp Axioms	#DataProp Axioms
1250	1250	1753	436
2500	2500	3507	795
3750	3750	5254	1339
5000	5000	6998	1737
Distribution of instances			
Vehicle (and subclasses)		64%	
Person (and subclasses)		30%	
Job (and subclasses)		6%	

Evaluated Policies. We performed the tests using 3 different approaches:

1. *Using a DL reasoner to load and classify each of the snapshot ontologies*. By *snapshot ontologies* we mean ontologies with the volatile properties populated with the right values at the moment of the refreshment.

2. *Using a DL reasoner to load the base ontology* (just the TBox and the static part of the ABox) and *updating the volatile properties through its API*. This is theoretically the way the reasoner should be used, as there are volatile values that could make the ontology inconsistent. To avoid this, the reasoner has to retract the facts that no longer hold. For example, the *available* property is functional and its range is boolean; so, when its value changes for an instance, the reasoner has to retract the previous value due to two reasons: it can only have one value, and it is not consistent to have both a true and a false value for the availability. We will refer to this strategy as the *classical approach*.

[8] For the same number of instances in the ABox, the times obtained by using HermiT were about 2.5 times higher than the times obtained by using Pellet.

Table 3
Query set used in the experiments.

Query Set	
Queries where CWA and OWA produce the same answer (CWA = OWA)	
SOSVehicle	Vehicle and (occupant some Physician)
AvailableSOSVehicle	SOSVehicle and (available value true)
CrowdedVehicle	Vehicle and (occupant min 3 Person)
AvailableCrowdedSOSVehicle	AvailableSOSVehicle and CrowdedVehicle
PersonOnBike	Person and (occupy some Bicycle)
Queries where CWA and OWA produce different answers (CWA <> OWA)	
BalancedVehicle	Vehicle and (occupant exactly 2 Person)
LowOccupiedVehicle	Vehicle and (occupant max 1 Person)
AvailableBalancedSOSVehicle	AvailableSOSVehicle and BalancedSOSVehicle
AvailableLowOccSOSVehicle	AvailableSOSVehicle and LowOccupiedVehicle
PersonOnFoot	Person and not (occupy some Vehicle)
HealthyPerson	PersonOnFoot or PersonOnBike

This classical approach can be combined with 2 alternative *update policies*: a buffered policy and an unbuffered policy. With the *buffered* policy, all the updates are accepted by the reasoner and the reclassification process is performed at query time. With the *unbuffered* policy, the updates are processed as they arrive, and so the reclassification process is performed at update time.

3. *Using the approach presented in this paper*. In this case, the *Knowledge Storage* is in charge of the static knowledge, and the management of volatile data is performed by the *Data Storage* component. All the updates on the volatile properties are performed through the *Updater Wrapper*.
This approach can be combined with 4 alternative *update policies* regarding the use of database *transactions* and *indexes*. On the one hand, to manage the update requests we can either consider individual transactions, each of them involving all the required operations according to the semantics of the updated property, or consider a unique complex transaction for all the updates. Clearly, the latter approach is faster, but in case of a general failure during the update the database could finish in an incoherent state (for example, the DBMS may insert the tuple corresponding to an object assertion involving a volatile property but not the tuple corresponding to its inverse property). This is not an acceptable policy in practice, but we want to test the cost of keeping the database in a coherent state.
On the other hand, it is possible to create extra indexes to cover separately each of the different columns of the tables involved in the query, or we can have no additional indexes and use just the ones created for the primary keys. In the first case, updating is harder but answering is faster. In particular, we created indexes for the columns that are used in the join operations. One could think of using additional indexes, but this is not easy without imposing assumptions on the data (e.g., about the distribution of values). Since the database schema obtained as a query plan is not very complicated, these indexes seem a reasonable choice.
Hence, there are 4 possibilities: indexes and transactions, indexes but not transactions, transactions but not indexes, and neither indexes nor transactions.

The first approach was considered just to have a ground set of answers to compare the correctness of the results obtained using the other two approaches. In the remaining we will discuss the results of the two other approaches.

We also wanted to test the *incremental consistency check* that Pellet provides[9], but unfortunately it only works with ontologies whose expressivity is $\mathcal{SHIQ}$ or $\mathcal{SHOQ}$, that is, it does not support datatypes for incremental reasoning, which is crucial in mobile and dynamic scenarios.

8.2. Results

A complete description of the results, including the detailed results for every query in our dataset, can be found at `http://sid.cps.unizar.es/ContQueries/index.html`[10].

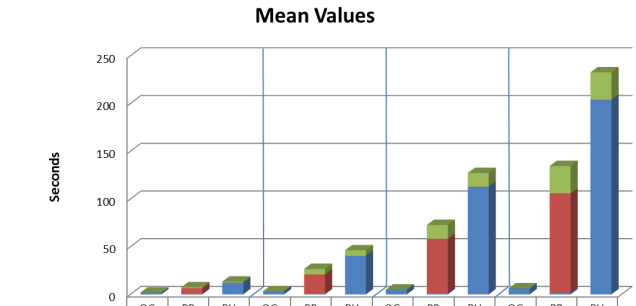
The results are shown in Figures 9–11, illustrating the mean values of the answering times when using QueryGen (QG) and when using the Pellet reasoner with both update policies (buffered reasoner –PB– and unbuffered reasoner –PU–). Recall that, as explained in Section 8.1 when discussing the methodology, we only consider up to 5000 instances because the DL reasoner Pellet was not able to support more. In these experiments, QG uses transactions but not indexes because this is the best configuration, as we will show later. In particular:

- Figure 9 presents the global mean values considering all the queries.
- Figure 10 presents the mean values of those queries for which CWA and OWA lead to the same answer.
- Figure 11 presents the mean values of those queries for which CWA and OWA produce a different answer.

In each case, we show three parameters:

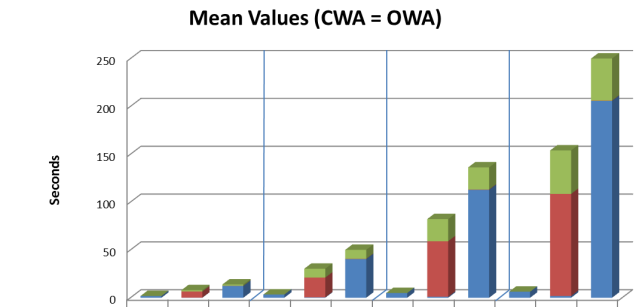
[9] According to (Dentler et al., 2011), Pellet is the only reasoner that provides incremental reasoning and consistency checking allowing both the assertion and the retraction of existing axioms.

[10] This document will be especially useful for a reader interested in the results that do not contribute significantly to the total time, such as the answering time for the different configurations of QueryGen.



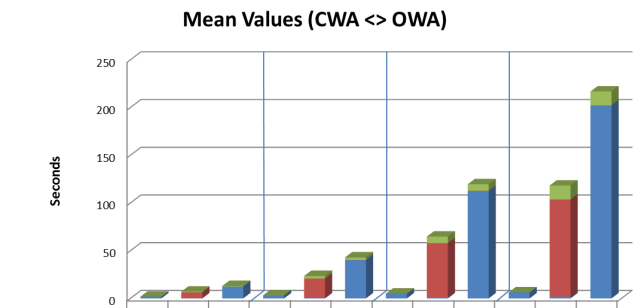
	1250 inst.			2500 inst.			3750 inst.			5000 inst.		
	QG	PB	PU	QG	PB	PU	QG	PB	PU	QG	PB	PU
Answer	0.01	0.97	0.97	0.01	5.71	5.74	0.02	14.30	14.25	0.02	28.57	28.60
Reclassify	0.00	6.34	0.12	0.00	20.50	0.19	0.00	57.41	0.29	0.00	104.09	0.27
Update Volatile Data	1.78	0.14	12.28	3.33	0.39	40.24	4.96	1.00	112.61	6.30	1.78	203.71

Figure 9. Performance taking all the queries into account.



	1250 inst.			2500 inst.			3750 inst.			5000 inst.		
	QG	PB	PU	QG	PB	PU	QG	PB	PU	QG	PB	PU
Answer	0.00	1.35	1.35	0.01	9.30	9.38	0.01	23.11	23.08	0.02	45.42	45.58
Reclassify	0.00	6.41	0.16	0.00	20.66	0.24	0.00	58.09	0.36	0.00	106.70	0.29
Update Volatile Data	1.76	0.14	12.32	3.35	0.39	40.44	4.95	0.99	112.79	6.31	1.80	205.73

Figure 10. Performance for queries where CWA = OWA.



	1250 inst.			2500 inst.			3750 inst.			5000 inst.		
	QG	PB	PU	QG	PB	PU	QG	PB	PU	QG	PB	PU
Answer	0.01	0.66	0.65	0.02	2.71	2.70	0.02	6.97	6.90	0.03	14.54	14.45
Reclassify	0.00	6.29	0.09	0.00	20.37	0.15	0.00	56.84	0.23	0.00	101.92	0.26
Update Volatile Data	1.80	0.14	12.25	3.31	0.39	40.07	4.97	1.00	112.46	6.29	1.76	202.03

Figure 11. Performance for queries where CWA <> OWA.

- Update volatile data time: seconds needed to update the knowledge base.
- Reclassify time: seconds needed to reclassify the ontology after the changes in the volatile data.
- Answer time: seconds needed to return an answer after the two previous steps.

The initialization times are not described in this section as they are not significant in comparison with the other times.

We distinguish the queries that did not return the same answer with CWA and OWA (Figure 11) because their results were quite tricky. The DL reasoner knows when it cannot include an individual in an answer due to the OWA and sometimes it answers really quickly by simply returning an empty set. However, as QueryGen considers CWA for the volatile properties, it gives a correct answer with the data that are actually collected. In this way, questions such as *HealthyPerson* or *AvailableLowOccupiedSOSVehicle* can be answered by QueryGen, while this is not possible with a classical approach. The interesting result is that the difference in the answering time between QueryGen and the DL reasoner is even bigger when the reasoner with CWA and OWA produces the same answer.

Next, Figure 12 compares the 4 update policies in QueryGen to discover the best one: indexes and transactions (*Idx. ON, Trans. ON*), indexes but not transactions (*Idx. ON, Trans. OFF*), transactions but not indexes (*Idx. OFF, Trans. ON*), and neither indexes nor transactions (*Idx. OFF, Trans. OFF*). For each of the cases we compute the update volatile data time and the answer time[11].

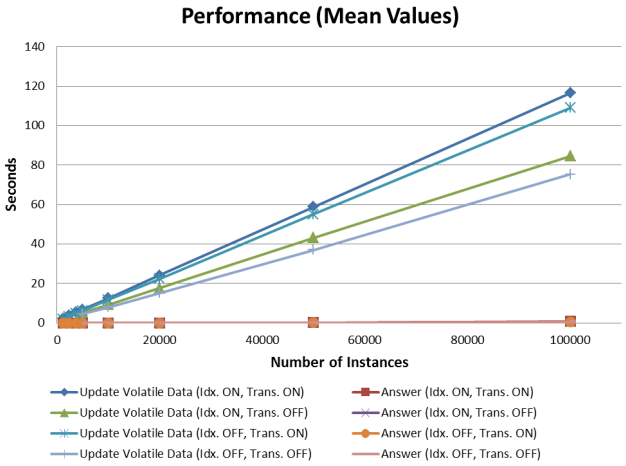


Figure 12. Comparison of the 4 update policies in QueryGen.

Figure 13 shows the scalability of QueryGen (for the two *Idx. OFF* policies) compared with the classical reasoner (for the buffered policy).

Finally, Figure 14 shows the results of the performance of our system regarding the *change rate* that it can support. The change rate is defined as the inverse of the time needed to process one single change in the data. In particular, Figure 14 shows the change rate in QueryGen for two *Trans. ON policies* (*Idx. ON, Trans. ON*, and *Idx. OFF, Trans. ON*). In this case, since the system is answering after every message, it is mandatory to use *Trans. ON* to keep the consistency of the databases. As shown in the figure, our system supports a high change rate (between 2100 and 2500 messages per second) independently of the total number of instances.

[11] Recall that QueryGen does not need to reclassify.

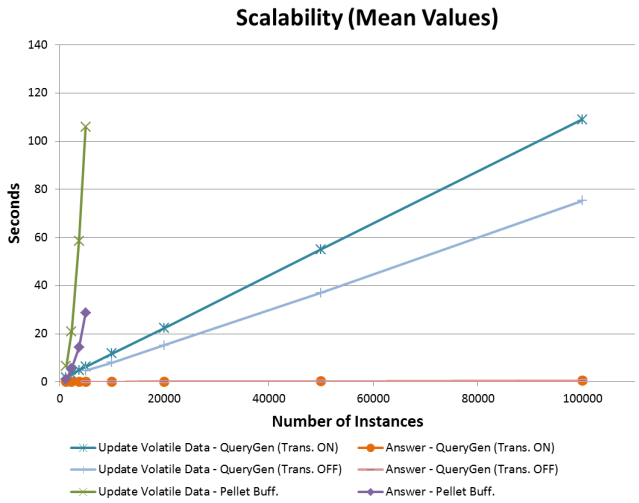


Figure 13. Scalability of the approaches.

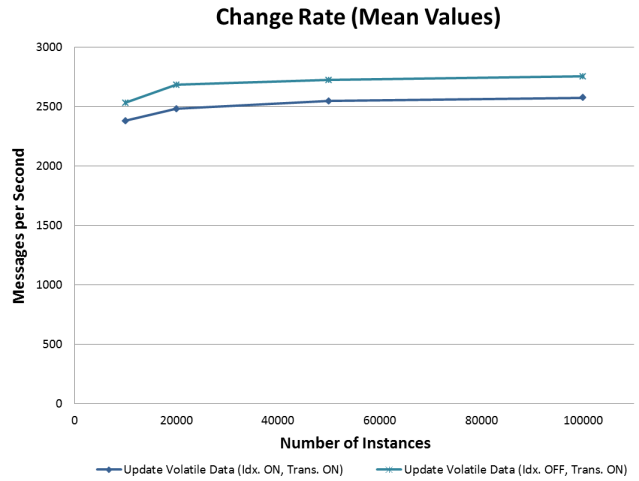


Figure 14. Change rate in QueryGen.

8.3. Discussion

Regarding the First Hypothesis. Independently of the update policy chosen, our approach outperforms the classic approach. It can be easily seen that the most important contribution to QueryGen's query processing time is the updating time. Answering is very fast thanks to the use of a DBMS and, moreover, it does not change in a significant way with the number of instances. Recall that the figures show the results of updating the whole dataset and not just one single message; this latter case will be discussed later.

This fact justifies our decision of using a 100% update rate, i.e., a complete update of the volatile properties. This is the most challenging scenario for QueryGen as it requires a lot of updates, which is the most costly part and does not add further difficulties to the DL reasoner since its results are similar independently of the data update rate, as the reclassification process has to be performed independently of the amount of data change. In scenarios where the update rate is lower, the processing times of our system will strongly decrease.

Regarding the Second Hypothesis. We can also see that QueryGen scales very well, incrementing the answering time linearly with the size of the ABox, due to the use of a DBMS, oriented to data handling. Notice that QueryGen could support 100000 instances in a case where the classical approach was not able to support more than 5000 instances.

Regarding the Third Hypothesis. Our experiments also show that QueryGen really supports a change rate suitable for applications that continuously need up-to-date data. In every case, it could process more than 2000 messages per second.

The change rate is better when we can group several messages to update them together, as the database connections can be kept open, the DBMS can optimize the updates, etc. This can be seen in Figure 14, where, in spite of processing just one message each time, the higher the number of messages to be processed, the higher the impact of the different optimization policies of the DBMS (the change rate stabilizes at about 2500 messages per second).

Analyzing the Different Policies in Detail. In the classical approach, the buffered policy produces better results than the unbuffered one. The reason is that, in the former case, almost all the time is dedicated to the update task as it also involves the reclassification of the instances, whereas in the latter case the update cost is really low but the answer cost rises.

In our approach, it is not surprising that when additional indexes are created the updating times slightly increase due to the need of updating these extra indexes as well. The answering times slightly decrease, but it is almost not appreciable. Hence, having additional indexes is not worth it. The updating of the properties needs the transaction mechanism (*Trans. ON*) to ensure that all the affected tables are consistently updated. Anyway, we observe that it is an affordable overhead. For example, for 5000 instances[12], the updating time using transactions is just 6.3 seconds vs. 4.65 seconds with transactions turned off (both using *Idx. OFF* configuration). The policy *Trans. OFF* produces a reduction in the total answering time, but it is more risky. In our opinion, *Trans. ON, Idx. OFF* is the best configuration for QueryGen, as the risk of having the database in an incoherent state is unacceptable in a practical application.

Final Remarks. It is worth noting that we have presented the mean values because in our approach all the queries behave in the same way. The reason is that, even though the different queries produce different SQL queries, the time needed by the DBMS to solve them is similar. In general, different SQL queries can have different complexities, but the types of SQL queries that QueryGen produces have a

[12] 5000 instances is the limit that Pellet could handle in our previous tests.

controlled complexity due to our translation of DL into relational algebra.

Even if QueryGen outperforms the classical approach, it should be kept in mind that it imposes several limitations (see Section 6), that may not be feasible in several applications. Notably, completeness is sacrificed to have faster answering times.

To conclude this section, it is convenient to explain that we have not compared our approach with any other similar stream reasoning approaches because, to the best of our knowledge, none of them is similar enough to our approach to produce a significant experimental comparison. In the next section we will provide an example that other related systems cannot currently support.

9. Related Work

In this section, we review some related and complementary approaches. Firstly, Section 9.1 analyzes other approaches sharing our objective of answering continuous queries in ontologies. Then, Sections 9.2, 9.3, 9.4, and 9.5 overview existing work on the use of databases in ontology reasoning, semantic sensor data, CWA in ontologies, and ontology dynamics, respectively.

9.1. Stream Reasoning

Several works have contributed to bridging the existing gap between *Data Stream Management Systems* (*DSMS*) (Golab & Özsu, 2003), which are specialized in the processing of highly-dynamic and continuously arriving data, and reasoners (Mishra & Kumar, 2011), which provide inferencing and reasoning.

The so-called *Stream Reasoning* (Della Valle, Ceri, van Harmelen, & Fensel, 2009) tries to unify the positive aspects of data streams, reasoners, and the Semantic Web (Shadbolt, Berners-Lee, & Hall, 2006). Enhancing data stream technology with these capabilities is interesting, as the importance of data stream technologies is expected to grow (Ilarri et al., 2010). In relation to stream reasoning, we could highlight the following works:

- In (Barbieri, Braga, Ceri, Della Valle, & Grossniklaus, 2010b), which extends an algorithm proposed in (Volz, Staab, & Motik, 2005), a technique to efficiently maintain a materialized view of RDF triples in the presence of continuous updates is proposed. The approach is based on the processing of incoming streaming data according to sliding windows and on the computation of expiration time information for the RDF triples (explicitly inserted or inferred). As opposed to this approach, our proposal does not process the data in sliding windows (alternatively, we could say that we assume *now windows* (Krämer & Seeger, 2009)), it does not rely on expiration times, and it is driven by the active continuous queries (Terry, Goldberg, Nichols, & Oki, 1992).
- Similarly, the works presented in (Ren, Pan, & Zhao, 2010) and (Ren & Pan, 2011) rely on syntactic approximations of the ontologies and on truth maintenance systems, respectively, to update the materialized knowledge as the facts change. Although these approaches are able to automatically detect changed knowledge, they do not separate static and volatile knowledge, and restrict the expressivity of the whole model to OWL 2 EL. On the contrary, in our proposal we advocate the use of different strategies to manage the static and the volatile knowledge, and we also support a higher expressivity (see Section 6 for more details). Moreover, the experiments conducted in (Ren & Pan, 2011) are with relatively low-changing data (10% of data updates), which in scenarios such as mobile environments is not enough because there are properties (e.g., the locations of moving objects) that are continuously changing. As opposed to this, in our experiments we considered a worst-case scenario with 100% of data updates, and the experimental results obtained show that our approach can deal with such highly-dynamic scenarios. Furthermore, these works are not able to deal with datatypes, which in our opinion are one of the main sources of dynamic changes.
- ETALIS (Anicic, Rudolph, Fodor, & Stojanovic, 2012) copes with stream reasoning from the point of view of Complex Event Processing (CEP). However, it focuses on the detection and processing of events over the streams by converting them into *event streams* (which provide higher semantic events). So, this is in fact complementary to our work as, through the use of our *Update Wrapper* and without losing any expressivity, their event streams can be attached seamlessly to QueryGen as another source of dynamic data that feed the volatile properties of QueryGen.

A notable difference with our approach is that we support having an OWL 2 ontology as background knowledge. Consider a modification of our example in Figure 1 where *CrowdedVehicle* is not a query but a defined term in the ontology, i.e., it belongs to the TBox. To the best of our knowledge, none of the other existing systems can directly support qualified cardinality restrictions.

It is also worth mentioning that several *languages* have been proposed to query RDF streaming data, such as *Continuous SPARQL* (*C-SPARQL*) (Barbieri, Braga, Ceri, Della Valle, & Grossniklaus, 2010a; Stuckenschmidt et al., 2010), *StreamingSPARQL* (Bolles, Grawunder, & Jacobi, 2008), *Time-Annotated SPARQL* (Rodríguez, McGrath, Liu, & Myers, 2009), *Event Processing SPARQL* (*EP-SPARQL*) (Anicic, Fodor, Rudolph, & Stojanovic, 2011), and *Spatio-Temporal SPARQL* (*stSPARQL*) (Koubarakis & Kyzirakos, 2010). These are complementary works to ours, as we do not focus on the definition of a query language but on the development of an approach that enables an efficient query processing on both static and highly-dynamic data.

Leaving aside semantics, there is a considerable amount of effort in the processing of flows of information without taking into account semantic data. We refer the interested reader to the good overview provided in (Cugola & Margara, 2012).

9.2. Using Databases for Ontology Reasoning

Several works have also proposed using at the same time databases and reasoners. For example:

- In (Al-Jadir, Parent, & Spaccapietra, 2010), the authors

propose the use of databases to store large ontologies and develop the appropriate reasoning capabilities as PL/SQL stored procedures that assert and remove axioms as needed. The goal is to ensure a good scalability, which cannot be offered by existing reasoners. Their prototype, called *OntoMinD* (*Ontology Management in Databases*), stores an ontology in multiple tables (one table per class and one binary table per property), materializes the inferred information, and performs reasoning when the updates are performed instead of at query time, in order to favor query-intensive applications. However, the main concern of that work is how to map ontologies to databases, and it does not focus on contexts where the data are continuously changing. Besides, it only supports OWL 2 QL (DL-Lite).

- Other approaches also propose the linked use of databases and reasoners (such as the *OWL Instance Store* (Bechhofer, Horrocks, & Turi, 2005), *Minerva* (Zhou et al., 2006), *SOR* (Ma et al., 2007), or *OntoDB* (Hondjack, Pierra, & Bellatreche, 2007)), the use of deductive databases as a supporting environment for rules obtained from ontology axioms (such as (Grosof, Horrocks, Volz, & Decker, 2003; Volz et al., 2005; Weithöner, Liebig, & Specht, 2004)), or extending existing database engines with inferencing capabilities (e.g., semantic technologies in Oracle (Oracle, 2009)). In contrast with these approaches, our work aims at supporting an efficient processing of continuous queries in environments with highly-dynamic data. We aim at providing the highest expressivity possible at reasonable computational costs, crucial for mobile environments but also profitable in other scenarios which share the challenges of managing data that change very frequently.
- A third possibility is called *query rewriting*, where most of the work has focused on answering conjunctive queries over databases (Poggi et al., 2008; Calì, Gottlob, & Pieris, 2012; Gottlob, Orsi, & Pieris, 2011). However, these works do not consider the volatility of the data. Actually, they are complementary to our work: works on query rewriting could use our approach to deal with volatile properties, as they need to compute instance retrievals as a base service to build up their answers.

9.3. Semantic Sensor Web

A complementary effort in the field that should also be mentioned is the so-called *Semantic Sensor Web* (Corcho & Garcia-Castro, 2010), which aims at integrating seamlessly all the sensors that are becoming available with the current Semantic Web, more oriented to the field of *Linked Data* (Bizer, Heath, & Berners-Lee, 2009; Sequeda & Óscar Corcho, 2009). The use of data produced by sensors in real-time will lead to a new generation of web applications. To do so, several researchers propose to use and adapt the techniques developed for the Semantic Web: modeling the sensors[13] and the data they provide (Koubarakis & Kyzirakos, 2010), integrating them through the use of ontologies (Calbimonte, Óscar Corcho, & Gray, 2010; Calbimonte, Jeung, Óscar Corcho, & Aberer, 2012), facilitating their discovery and usage via semantic techniques (Gray et al., 2011), etc. These initiatives are complementary to ours, as QueryGen can benefit from their results to obtain more data sources to work with. In this field, *CQELS* (Le-Phuoc, Dao-Tran, Parreira, & Hauswirth, 2011) has recently been proposed to efficiently process continuous SPARQL queries over Linked Data and Linked Streams, that is, integrating static and volatile sources. However, it focuses on optimizing low-level aspects of the query processing (in particular, on how to process efficiently SPARQL queries over streams of triples, leaving aside the reasoning capabilities of the system), while our proposal is oriented to support highly expressive continuous queries and takes the semantics into account along all the process. Again, CQELS can benefit from our semantic processing to enhance their expressivity.

9.4. Close World Assumption in Ontologies

The already cited works that use databases for ontology reasoning must somehow face the difference between OWA and CWA. In OWL 2 QL, it is possible to obtain rewritings where these different semantics do not pose a problem (Poggi et al., 2008), but in general this is not the case.

There are also some previous efforts in adding some mechanism to close the world in ontologies. It is actually natural to consider ontology languages combining both OWA and CWA, which is referred as *Local Closed World Assumption* (LCWA), and hence there are several approaches in the literature, summarized in the rest of the section.

Usually, DLs are extended with non-monotonic rules. An important line of work in this direction are *hybrid MKNF knowledge bases* (Donini, Nardi, & Rosati, 2002). The logic of Minimal Knowledge and Negation as Failure (MKNF) is an extension of first-order logic with two epistemic operators K and A. Hybrid MKNF knowledge bases are essentially restricted MKNF formulae; they can be seen as a DL knowledge base and a finite set of modal rules (Knorr, Alferes, & Hitzler, 2011). Similarly, DLs have been combined with other types of rules. The combination with the so-called Description Logic Rules produces Description Logic Programs (Eiter, Ianni, Lukasiewicz, Schindlauer, & Tompitsa, 2008).

Another approach is to use *circumscription* in DLs, which makes it possible to model defeasible inheritance (Bonatti, Lutz, & Wolter, 2009; Sengupta, Krisnadhi, & Hitzler, 2011), or a relaxed form of CWA called *Generalized Closed World Assumption* (GCWA) in DLs (Gries, 2009). A simpler approach is to extend the DL with an *NBox* indicating to which assertions CWA should be applied and consider negation as failure (Ren, Pan, & Yuting, 2010).

However, all these approaches aim at performing a complete DL reasoning, while we only use the CWA to take advantage of the data management capabilities of databases in the query answering.

[13] `http://www.w3.org/2005/Incubator/ssn/wiki/Semantic_Sensor_Network_Ontology`

9.5. Ontology Change

Ontology change can be defined in a very general way as the problem of modifying an ontology in response to some needs and managing the effects of the change. This general view can result in more concrete situations, such as ontology evolution (modifying an ontology) (Haase & Stojanovic, 2005), ontology versioning (managing different versions of an ontology) (Noy & Musen, 2004), or ontology debugging (modifying an ontology because of logical contradictions) (Stuckenschmidt, 2008). Temporal Description Logics provide the logical formalism for managing temporal knowledge in DLs (Artale & Franconi, 2000). For a good survey on the field of ontology change we refer the reader to (Flouris et al., 2008), whereas recent work in the area can be found in (Hartung, Terwilliger, & Rahm, 2011).

These approaches are different to ours. While these approaches are also focused on dynamic knowledge in ontologies, they are more concerned with managing evolving schemas. On the contrary, we consider evolving dynamic environments where ontology schemas are constant and only the values of properties are always changing. Our aim is to answer highly expressive queries, not to maintain the knowledge base continuously classified. This enables QueryGen to be more robust against spurious values (which could lead to inconsistencies) and to give a correct answer in a faster way.

10. Conclusions

In this paper, we have presented a system that is able to process continuous DL queries efficiently and correctly in the presence of highly-dynamic data. To do so, our approach proposes to distinguish between static and volatile knowledge to benefit from DL reasoning capabilities, and, at the same time, supports data that continuously change their values by exploiting the capabilities of databases. The main advantages of our approach are:

- It exploits both static and volatile knowledge to obtain a correct answer.
- It supports an expressive knowledge representation. For static properties, it does not restrict the expressivity of the ontologies used to represent the knowledge, and for volatile properties it supports an important fragment of OWL 2. In particular, we support datatypes, which is a crucial aspect in mobile and dynamic scenarios.
- It uses the extensional knowledge (the TBox) to process the queries efficiently, reusing and sharing intermediate results (i.e., tables in the *Data Storage*) as much as possible.
- It provides a translation of DL expressions into relational algebra, which allows QueryGen to handle dynamic data efficiently by using an underlying DBMS.
- It performs very well. The tests carried out show that our approach can be used in highly-dynamic scenarios, since it scales up well with the number of objects and supports a change rate suitable for applications that need up-to-date data.

As future work, we would like to adapt our approach to distributed location-dependent query processing systems, such as *LOQOMOTION* (Ilarri et al., 2006, 2011). We would also like to study the theoretical complexity of our approach and whether some of the limitations can be removed while maintaining a good performance. In particular, we have envisioned some strategies to support role hierarchies and role transitivity. It would be interesting as well to extend QueryGen to support alternative reasoning tasks, such as conjunctive query answering. Finally, even if none of the related approaches is similar enough to ours to produce a fair experimental comparison, and even if the existing approaches have different goals and limitations compared to our work, we would like to identify common intersections of the approaches to perform a joint comparison of the systems.

Acknowledgments

This research work has been supported by the CICYT projects TIN2010-21387-C02-02, TIN2013-46238-C4-4-R, and DGA-FSE.

References

Al-Jadir, L., Parent, C., & Spaccapietra, S. (2010). Reasoning with large ontologies stored in relational databases: The OntoMinD approach. *Data & Knowledge Engineering*, *69*(11), 1158–1180.

Anicic, D., Fodor, P., Rudolph, S., & Stojanovic, N. (2011). EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th International Conference on the World Wide Web (WWW 2011)* (pp. 635–644). ACM Press.

Anicic, D., Rudolph, S., Fodor, P., & Stojanovic, N. (2012). Stream reasoning and complex event processing in ETALIS. *Semantic Web Journal*, *3*(4), 397–407.

Artale, A., & Franconi, E. (2000). A survey of temporal extensions of description logics. *Annals of Mathematics and Artificial Intelligence*, *30*(1–4), 171-210.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., & Patel-Schneider, P. F. (Eds.). (2003). *The description logic handbook: Theory, implementation, and applications*. Cambridge University Press, UK.

Barbieri, D. F., Braga, D., Ceri, S., Della Valle, E., & Grossniklaus, M. (2010a). C-SPARQL: a continuous query language for RDF data streams. *International Journal of Semantic Computing*, *4*(1), 3–25.

Barbieri, D. F., Braga, D., Ceri, S., Della Valle, E., & Grossniklaus, M. (2010b). Incremental reasoning on streams and rich background knowledge. In *Proceedings of the 7th Extended Semantic Web Conference (ESWC 2010)* (Vol. 6088, pp. 1–15). Springer.

Bechhofer, S., Horrocks, I., & Turi, D. (2005). The OWL Instance Store: System description. In *Proceedings of the 20th International Conference on Automated Deduction (CADE 2005)* (Vol. 3632, pp. 177–181). Springer.

Bizer, C., Heath, T., & Berners-Lee, T. (2009). Linked data – the story so far. *International Journal on Semantic Web and Information Systems*, *5*(3), 1–22.

Bobed, C., Trillo, R., Mena, E., & Bernad, J. (2008). Semantic discovery of the user intended query in a selectable target query language. In *Proceedings of the 7th International Conference on Web Intelligence (WI 2008)* (pp. 579–582). IEEE Computer Society Press.

Bobed, C., Trillo, R., Mena, E., & Ilarri, S. (2010). From keywords to queries: Discovering the user's intended meaning. In *Proceedings of the 11th International Conference on Web Informa-*

tion System Engineering (WISE 2010) (Vol. 6488, pp. 190–203). Springer.

Bolles, A., Grawunder, M., & Jacobi, J. (2008). Streaming SPARQL extending SPARQL to process data streams. In *Proceedings of the 5th European Semantic Web Conference on the Semantic Web: Research and Applications (ESWC 2008)* (Vol. 5021, pp. 448–462). Springer.

Bonatti, P. A., Lutz, C., & Wolter, F. (2009). The complexity of circumscription in description logic. *Journal of Artificial Intelligence Research*, *35*, 717–773.

Calbimonte, J.-P., Jeung, H., Óscar Corcho, & Aberer, K. (2012). Enabling query technologies for the semantic sensor web. *International Journal on Semantic Web and Information Systems*, *8*, 43–63.

Calbimonte, J.-P., Óscar Corcho, & Gray, A. (2010). Enabling ontology-based access to streaming data sources. In *Proceedings of the 9th International Semantic Web Conference (ISWC 2010)* (Vol. 6496, p. 96-111). Springer.

Calì, A., Gottlob, G., & Pieris, A. (2012). Ontological query answering under expressive entity-relationship schemata. *Information Systems*, *37*(4), 320-335.

Cenerario, N., Delot, T., & Ilarri, S. (2011). A content-based dissemination protocol for VANETs: Exploiting the encounter probability. *IEEE Transactions on Intelligent Transportation Systems*, *12*(3), 771–782.

Codd, E. F. (1990). *The relational model for database management, version 2*. Addison-Wesley.

Corcho, Ó., & Garcia-Castro, R. (2010). Five challenges for the semantic sensor web. *Semantic Web*, *1*(1-2), 121–125.

Cugola, G., & Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, *44*(3), 15.

Della Valle, E., Ceri, S., van Harmelen, F., & Fensel, D. (2009). It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, *24*(6), 83–89.

Delot, T., Ilarri, S., Thilliez, M., Vargas-Solar, G., & Lecomte, S. (2011). Multi-scale query processing in vehicular networks. *Journal of Ambient Intelligence and Humanized Computing*, *2*(3), 213-226.

Dentler, K., Cornet, R., ten Teij, A., & de Keizer, N. (2011). Comparison of reasoners for large ontologies in the OWL 2 EL Profile. *Semantic Web*, *2*, 71-8-7.

Donini, F. M., Nardi, D., & Rosati, R. (2002). Description logics of minimal knowledge and negation as failure. *ACM Trans. on Computational Logic*, *3*(2), 177–225.

Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., & Tompitsa, H. (2008). Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, *172*(12–13), 1495–1539.

Flouris, G., Manakanatas, D., Kondylakis, H., Plexousakis, D., & Antoniou, G. (2008). Ontology change: Classification and survey. *The Knowledge Engineering Review*, *23*(2), 117–152.

Glimm, B., Horrocks, I., Motik, B., Shearer, R., & Stoilos, G. (2012). A novel approach to ontology classification. *Journal of Web Semantics*, *14*, 84–101.

Golab, L., & Özsu, M. T. (2003). Issues in data stream management. *SIGMOD Record*, *32*, 5–14.

Gottlob, G., Orsi, G., & Pieris, A. (2011). Ontological query answering via rewriting. In *Proceedings of the 15th International Conference on Advances in Databases and Information Systems (ADBIS 2011)* (Vol. 6909, pp. 1–18). Springer.

Gray, A. J. G., Garcia-Castro, R., Kyzirakos, K., Karpathiotakis, M., Calbimonte, J.-P., Page, K. R., ... Gómez-Pérez, A. (2011). A semantically enabled service architecture for mashups over streaming and stored data. In *Proceedings of the 9th extended semantic web conference (eswc 2011)* (Vol. 6644, pp. 300–314). Springer.

Gries, O. (2009). Generalized closed world reasoning in description logics with extended domain closure. In *Proceedings of the 22nd international workshop on description logics (dl 2009)* (Vol. 477).

Grosof, B. N., Horrocks, I., Volz, R., & Decker, S. (2003). Description logic programs: combining logic programs with description logic. In *Proceedings of the 12th International World Wide Web Conference (WWW 2003)* (pp. 48–57). ACM Press.

Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, *43*(5–6), 907–928.

Haarslev, V., Hidde, K., Möller, R., & Wessel, M. (2012). The racerpro knowledge representation and reasoning system. *Semantic Web Journal*, *3*(3), 267–277.

Haase, P., & Stojanovic, L. (2005). Consistent evolution of OWL ontologies. In *Proceedings of the 2nd european semantic web conference (eswc 2005)* (Vol. 3532, pp. 182–197). Springer.

Hartung, M., Terwilliger, J. F., & Rahm, E. (2011). Recent advances in schema and ontology evolution. In *Schema matching and mapping* (pp. 149–190). Springer.

Hondjack, D., Pierra, G., & Bellatreche, L. (2007). OntoDB: An ontology-based database for data intensive applications. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA 2007)* (Vol. 4443, pp. 497–508). Springer.

Horridge, M., & Bechhofer, S. (2011). The OWL API: A Java API for OWL ontologies. *Semantic Web*, *2*(1), 11–21.

Horridge, M., & Patel-Schneider, P. F. (2009). *OWL 2: Web Ontology Language Manchester syntax.* (`http://www.w3.org/TR/owl2-manchester-syntax`, accessed April 25, 2013)

Horrocks, I., Kutz, O., & Sattler, U. (2006). The even more irresistible $\mathcal{SROIQ}$. In *Proceedings of the 10th International Conference of Knowledge Representation and Reasoning (KR 2006)* (pp. 452–457).

Ilarri, S., Bobed, C., & Mena, E. (2011). An approach to process continuous location-dependent queries on moving objects with support for location granules. *Journal of Systems and Software*, *84*(8), 1327–1350.

Ilarri, S., Mena, E., & Illarramendi, A. (2006). Location-dependent queries in mobile contexts: Distributed processing using mobile agents. *IEEE Transactions on Mobile Computing*, *5*(8), 1029–1043.

Ilarri, S., Mena, E., & Illarramendi, A. (2008). Using cooperative mobile agents to monitor distributed and dynamic environments. *Information Sciences*, *178*(9), 2105–2127.

Ilarri, S., Mena, E., & Illarramendi, A. (2010). Location-dependent query processing: Where we are and where we are heading. *ACM Computing Surveys*, *42*(3), 1–73.

Knorr, M., Alferes, J. J., & Hitzler, P. (2011). Local closed world reasoning with description logics under the well-founded semantics. *Artificial Intelligence*, *175*(9–10), 1528–1554.

Koubarakis, M., & Kyzirakos, K. (2010). Modeling and querying metadata in the Semantic Sensor Web: The model stRDF and the query language stSPARQL. In *Proceedings of the 7th Extended Semantic Web Conference (ESWC 2010)* (Vol. 6088, p. 425-439). Springer.

Krämer, J., & Seeger, B. (2009, April). Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems*, *34*(1), 4:1–4:49.

Le-Phuoc, D., Dao-Tran, M., Parreira, J. X., & Hauswirth, M. (2011). A native and adaptive approach for unified processing of linked streams and linked data. In *Proceedings of the 10th International Semantic Web Conference (ISWC 2011)* (Vol. 7031, pp. 370–388). Springer.

Ma, L., Zhang, L., Brunner, J.-S., Wang, C., Pan, Y., & Yu, Y. (2007). SOR: A practical system for ontology storage, reasoning and search. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007)* (pp. 1402–1405). VLDB Endowment.

Mishra, R. B., & Kumar, S. (2011). Semantic web reasoners and languages. *Artificial Intelligence Review*, *35*(4), 339–368.

Nissen, M. E. (2013). Harnessing dynamic knowledge principles in the technology-driven world. IGI Global.

Noy, N. F., & Musen, M. A. (2004). Ontology versioning in an ontology management framework. *IEEE Intelligent Systems*, *19*(4), 6–13.

Oracle. (2009). *Oracle database semantic technologies developer's guide, 11g release 1 (11.1).* (`http://docs.oracle.com/cd/B28359_01/appdev.111/b28397.pdf`, accessed April 25, 2013)

Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., & Rosati, R. (2008). Linking data to ontologies. *Journal of Data Semantics*, *X*, 133–173.

Ren, Y., & Pan, J. Z. (2011). Optimising ontology stream reasoning with truth maintenance system. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management (CIKM 2011)* (pp. 831–836). ACM Press.

Ren, Y., Pan, J. Z., & Yuting, Z. (2010). Closed world reasoning for OWL2 with NBox. *Journal of Tsinghua Science and Technology*, *15*(6), 692-701.

Ren, Y., Pan, J. Z., & Zhao, Y. (2010). Ontological stream reasoning via syntactic approximation. In *Proceedings of the 4th International Workshop on Ontology Dynamics (IWOD 2010)* (Vol. 651). CEUR Workshop Proceedings.

Rob Shearer, B. M., & Horrocks, I. (2008). HermiT: A highly-efficient OWL reasoner. In *Proceedings of the 5th workshop on owl: Experiences and directions (owled 2008)* (Vol. 432).

Rodríguez, A., McGrath, R., Liu, Y., & Myers, J. (2009). Semantic management of streaming data. In *Proceedings of the 2nd International Workshop on Semantic Sensor Networks (SSN 2009).*

Sengupta, K., Krisnadhi, A. A., & Hitzler, P. (2011). Local closed world semantics: Grounded circumscription for OWL. In *Proceedings of the 10th international semantic web conference (iswc 2011) - part i* (Vol. 7031, pp. 617–632). Springer.

Sequeda, J., & Óscar Corcho. (2009). Linked stream data: A position paper. In *Proceedings of the 2nd International Workshop on Semantic Sensor Networks (SSN09)* (Vol. 522, pp. 148–157). CEUR Workshop Proceedings.

Shadbolt, N., Berners-Lee, T., & Hall, W. (2006). The semantic web revisited. *IEEE Intelligent Systems*, *21*(3), 96–101.

Sirin, E., Parsia, B., Cuenca-Grau, B., Kalyanpur, A., & Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, *5*(2), 51–53.

Stuckenschmidt, H. (2008). Debugging owl ontologies - a reality check. In *Proceedings of the 6th international workshop on evaluation of ontology-based tools and the semantic web service challenge (eon-swsc-2008)* (Vol. 359).

Stuckenschmidt, H., Ceri, S., Della Valle, E., & van Harmelen, F. (2010). Towards expressive stream reasoning. In *Proceedings of the Dagstuhl Seminar on Semantic Aspects of Sensor Networks.*

Terry, D. B., Goldberg, D., Nichols, D. A., & Oki, B. M. (1992). Continuous queries over append-only databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'92)* (pp. 321–330). ACM Press.

Tsarkov, D., & Horrocks, I. (2006, 8). FaCT++ description logic reasoner: System description. In *Proceedings of the 3rd international joint conference on automated reasoning (ijcar 2006).*

Volz, R., Staab, S., & Motik, B. (2005). Incrementally maintaining materializations of ontologies stored in logic databases. *Journal of Data Semantics*, *3360*(2), 1–34.

Weithöner, T., Liebig, T., & Specht, G. (2004). Efficient processing of huge ontologies in logic and relational databases. In *On the Move to Meaningful Internet Systems (OTM) Workshops* (Vol. 3292, pp. 28–29). Springer.

Zhou, J., Ma, L., Liu, Q., Zhang, L., Yu, Y., & Pan, Y. (2006). Minerva: A scalable OWL ontology storage and inference system. In *Proceedings of the 1st Asian Semantic Web Conference (ASWC 2006)* (Vol. 4185, p. 429-443). Springer.

Appendix: The Ontology Language OWL 2

The language OWL 2 is the current W3C recommendation for ontology specification. OWL 2 is based on the DL $\mathcal{SROIQ}(\mathbf{D})$. In this appendix we will only describe the syntax of the language, but more details about the underlying logic, including the semantics, can be found in (Horrocks, Kutz, & Sattler, 2006).

OWL 2 provides several syntaxes. Among them, we will use Manchester syntax (Horridge & Patel-Schneider, 2009), specifically designed to be easily understood by humans.

OWL 2 ontologies have 5 elements: individuals, concepts (or classes), datatypes (or concrete domains), object properties, and data properties. Essentially, concepts are sets of individuals, datatypes are sets of values defined over a concrete domain (such as integers or dates), object properties are binary relations between individuals, and datatype properties relate individuals and datatypes.

In ontologies, complex concepts can be inductively built from simpler ones. Table 4 shows the supported concepts in OWL 2, where C is a concept, R is an object property, n is a natural number, i is an individual, T is a datatype property, v is a datatype value, and D is a datatype.

The knowledge of an ontology is structured in two parts: an *ABox* (Assertional Box), with the extensive knowledge, and a *TBox* (Terminological Box), with the intensive knowledge[14]. Table 5 summarizes the main axioms in OWL 2. The top part of the table (with 7 axioms) is the ABox, and the rest is the TBox. Some of the axioms are just syntactic sugar and can be represented using class definitions such as disjoint classes, disjoint union of classes, and domain and range axioms.

In order to guarantee the decidability of the logic, there are some restrictions in the role hierarchies axioms and some roles are required to be simple. The interested reader may find the formal specification at (Horrocks et al., 2006).

Some typical reasoning tasks over ontologies include:

[14] Sometimes the TBox is split into two parts: the TBox itself, with knowledge about concepts, and an *RBox* (Role Box), with knowledge about the properties.

Table 4
OWL 2 concepts.

A	Atomic/primitive concept
C1 or C2	Disjunction
C1 and C2	Conjunction
not C	Negation
Thing	Universal concept
Nothing	Empty concept
$\{i_1, \ldots, i_n\}$	Enumeration/nominals
R only C	Universal restriction
R some C	Existential restriction
R value i	Value restriction
R self	Local reflexivity concept
R exactly n	Exact cardinality restriction
R exactly n C	Qualified exact cardinality restriction
R max n	Maximal cardinality restriction
R max n C	Qualified maximal cardinality restriction
R min n	Minimal cardinality restriction
R min n C	Qualified minimal cardinality restriction
T only D	Universal restriction
T some D	Existential restriction
T value v	Value restriction
T exactly n	Exact cardinality restriction
T exactly n D	Qualified exact cardinality restriction
T max n	Maximal cardinality restriction
T max n D	Qualified maximal cardinality restriction
T min n	Minimal cardinality restriction
T min n D	Qualified minimal cardinality restriction

- *Consistency checking*: Checks if there exists a logical model satisfying all the axioms in the ontology.
- *Instance retrieval*: Gets all the instances of a concept.
- *Concept satisfiability*: Checks if a concept can have instances i.e., if it does not necessarily denotes the empty set.
- *Subsumption*: Checks if a concept/property C can be considered more general than (i.e., it subsumes) a concept D.
- *Classification*: Computes a concept/property hierarchy based on the relations of concept/property subsumption.

Some of these reasoning tasks can be reduced to others. The interested reader is referred to (Baader et al., 2003).

Table 5
OWL 2 axioms.

i Types C	Class assertion
i_1 Facts R i_2	Property assertion
i_1 Facts not R i_2	Negated property assertion
i Facts T v	Property assertion
i Facts not T v	Negated property assertion
SameIndividual i_1 i_2	Equality assertion
DifferentIndividuals i_1 i_2	Inequality assertion
C_1 SubClass Of C_2	Subclass axiom
C_1 EquivalentTo C_2	Class definition
C_1 DisjointWith C_2	Disjoint classes
C_1 DisjointUnionOf $C_2 \ldots C_n$	Disjoint union of classes
R_1 SubClass Of R_2	Subproperty axiom
R_0 SubPropertyChain $R_1 \ldots R_n$	Subproperty chain axiom
R_1 EquivalentTo R_2	Property definition
T_1 SubClass Of T_2	Subproperty axiom
T_1 EquivalentTo T_2	Property definition
R_1 DisjointWith R_2	Disjoint properties
T_1 DisjointWith T_2	Disjoint properties
R Domain C	Domain of a property
T Domain D	Domain of a property
R Range C	Range of a property
T Range D	Range of a property
R_1 InverseOf R_2	Inverse roles
R Functional	Functional property
T Functional	Functional property
R InverseFunctional	Inverse functional property
R Transitive	Transitive property
R Reflexive	Reflexive property
R Irreflexive	Irreflexive property
R Symmetric	Symmetric property
R Asymmetric	Asymmetric property