Endowing Semantic Query Languages with Advanced Relaxation Capabilities

Géraud Fokou, Stéphane Jean, Allel Hadjali

LIAS/ENSMA-University of Poitiers
1, Avenue Clement Ader, 86960 Futuroscope Cedex, France
(geraud.fokou, stephane.jean, allel.hadjali)@ensma.fr

Abstract. The problem of relaxing Semantic Web Database ($\mathcal{SWDB}$) queries that return an empty/unsatisfactory set of answers, has been addressed by several works in the last years. Most of these studies have focused on developing new relaxation techniques or on optimizing the top-k query processing. However, only few works have been conducted to provide a fine and declarative control of query relaxation to end users using an $\mathcal{SWDB}$ query language. This paper is a first step towards this direction. We first define a set of requirements for an $\mathcal{SWDB}$ cooperative query language. Then, based on these requirements, we propose an extension of $\mathcal{SWDB}$ query languages with a new clause to use and combine the relaxation operators we introduce. A similarity function is associated with these operators to rank-order the approximate answers retrieved. Finally, the implementation as well as the set of experiments we have conducted to evaluate the performance of the proposed relaxation operators, are explicitly described.

Keywords: Semantic Web Databases, Query relaxation, Empty answer, Ontology, Fuzzy set, Similarity.

1 Introduction

With the widespread adoption of RDF, the need to store and manage huge amounts of semantic data has appeared. As a consequence, specialized databases called Semantic Web Databases ($\mathcal{SWDB}$) have been developed during the last decade (e.g, RDF-3X [12] or OntoDB [6]). Unlike relational databases where the schema is fixed, $\mathcal{SWDB}$s use a generic schema (a triple table or one of its variants) that can be used to store a diverse set of data, ranging from structured data to unstructured data [3]. This flexibility makes that schema more difficult for users to formulate queries in a correct and complete way. This can often lead to the problem of empty/unsatisfactory answers. To address such problem, query relaxation techniques have been proposed to weaken unsuccessful user queries and retrieve alternative approximate answers.

Several works have been proposed to relax queries in the $\mathcal{SWDB}$s context [2, 5, 4, 7, 9]. They mainly focus either on the proposition of new relaxation operators or on the efficient processing of the top-k approximate answers. But none of them has addressed the need of defining a cooperative query language ($\mathcal{CQL}$) for $\mathcal{SWDB}$s that is capable of expressing most meaningful relaxation operators in a convenient and declarative manner. This paper is a first step towards this direction. It takes as a starting point our paper [13] where three relaxation operators have been proposed: *GEN* (super classes/properties), *SIB* (sibling classes/properties) and *PRED* (predicates) which are associated with a ranking function. The following novel main contributions are made in this paper.

- A set of requirements for $\mathcal{SWDB}$'s $\mathcal{CQL}$ are defined and a critical review of existing works according to these requirements is then provided;
- A clear and complete formalization of the proposed relaxation operators, their formal properties and their combination are discussed explicitly;
- Extensive experiments of the performance of our proposal are conducted on a real-life application.

The paper is structured as follows. First, Section 2 presents the main requirements we have defined for a $\mathcal{SWDB}$'s $\mathcal{CQL}$ and the limitations of existing works. Section 3 describes our approach to extend $\mathcal{SWDB}$ languages with relaxation operators on the one hand, and discusses their mixed use on the other hand. The implementation part as well as a set of experiments done on the $\mathcal{SWDB}$ OntoDB are presented in Section 4. Finally, Section 5 concludes the paper and outlines some future work.

2 SWDB's CQL Requirements and Limitations of Previous Works

As a first step for designing an $\mathcal{CQL}$ for $\mathcal{SWDB}$s, we define below a set of requirements which a $\mathcal{CQL}$ must fulfill.

R_1: **Diversified set of relaxation operators.** The $\mathcal{CQL}$ should offer diversified relaxation operators. It should include generic relaxation techniques (e.g, predicate relaxation) as well as specific techniques stemmed from ontologies (i.e., based on entailment rules).

R_2: **Relaxation operators combination.** It is highly desirable for an end-user to be able to call for a query relaxation that specifies the combination of different relaxation operators.

R_3: **Ranking function.** The $\mathcal{CQL}$ should be able to provide the user with a discriminated set of answers of a relaxed query. This rank-ordering should leverage the similarity between the original and relaxed queries and the satisfaction scores of the retrieved answers.

R_4: **Guide and control of the relaxation process.** The $\mathcal{CQL}$ should support a fine tuning and controlling of the relaxation process. Thus, according to the user needs, the relaxation process could range from being completely automatic (top-k query) to being completely specified and tuned by the user using several parameters.

R_5: **Implementation and performance optimization.** The $\mathcal{CQL}$ language should be implemented in a $\mathcal{SWDB}$. Due to the hugeness of RDF data managed by $\mathcal{SWDB}$s, optimization techniques should be developed to avoid a time consuming relaxation process.

Currently, works on the empty answers problem have focused on developing relaxation techniques to find alternative answers and on the optimization of such techniques w.r.t to time and quality criteria. Dolog et al. [2] propose a method based on user's preferences. The idea is to operate a query rewriting by substituting concepts/values in the original query by concepts/values preferred by the user. Other techniques which don't require user's preferences have also been developed. In the spirit of such methods, Hurtado et al.[10] propose a relaxation approach based on the inferences rules of RDFS. This approach leverages the *subClassOf*, *subPropertyOf*, *domain* and *range* relationships of RDFS to relax a SPARQL query. The end-user can choose the triples that must be relaxed in a query by using the *RELAX* clause. Huang et al. [8] use the same relaxation techniques based on RDFS entailment. To ensure quality of alternatives answers, they leverage a semantic similarity measure based on concept statistics. Several optimization techniques are also proposed to obtain the top-k approximate answers efficiently. Elbassuoni et al.[4] show a process for finding similar values of a precise value needed for a query relaxation.

In Table 1, we provide a synthetic summary of existing works w.r.t. the previous requirements. One can observe that there is no work that fulfill all the five requirements for designing an ($\mathcal{CQL}$) in the $\mathcal{SWDB}$ context. In particular, the integration of relaxation operators in a $\mathcal{CQL}$ as well as the precise control of the relaxation process has not been addressed by most of these work. This paper is a first step to fill this gap.

Table 1: Characterization of existing works w.r.t. the five requirements

	Requirements				
	R_1	R_2	R_3	R_4	R_5
Dolog et al.[2]	Substitution Rules	n/a	Based on users preferences	trigger rules (no tuning)	Sesame
Hurtado et al.[11]	Based on RDFS rules	RELAX clause	Distance-based	Top-k (no tuning)	n/a
Huang et al.[8]	Based on RDFS rules	n/a	Distance-based and content-based	Top-k (no tuning)	Jena (LUBM)
Elbassuoni et al.[4]	statistical language models	n/a	content-based	Domain target approach	Tests on real data set
Hogan et al.[5]	matchers	n/a	content-based	Domain target approach	Generic framework for EADS

3 Extending of SWDB query languages

In this section, we present an extension of the language OntoQL with operators and clauses of relaxation. Precisely, we introduce and define the following clauses and operator: *Fuzzy* and *Approx* clauses, *PRED*, *GEN* and *SIB* operators.

3.1 Fuzzy Clause

The *Fuzzy* clause is an extension of fuzzy predicates proposed in SQLf [1]. This clause allows users to obtain alternative answers with a satisfiability degree thanks to the membership function paradigm [1]. For relaxing a predicate with *Fuzzy* clause, the user must precise the tolerance value authorized and the satisfiability degree (s)he desires. The syntax of the fuzzy clause in OntoQL/OntoDB[2] is : $Fuzzy(Prop, [bornInf, bornSup, pasInf, pasSup]) \geq degree.$

Example 1. Let's consider the following example with the query Q:
select $Name$, $Price$ from $Motel$ where $Price \geq 113$ and $Price \leq 114$.
A relaxed query Q' of Q with a tolerance value 2, writes as follows :
select $Name$, $Price$ from $Motel$ where $Fuzzy(Price, [113, 114, 2, 2]) \geq 0$.
It is the default use case of the *Fuzzy* clause. We can have another relaxed query Q" defined such as:
select $Name$, $Price$ from $Motel$ where $Fuzzy(Price, [113, 114, 2, 2]) \geq 0.5$.
Here, only the prices between 112 and 115 will be selected (see Figure 1b).

3.2 Approx Clause

The clause *APPROX* shows the parameter of explicit relaxation. In this clause, we give operators of relaxation and how we use them or combine them. The main operators of relaxation we use are: *PRED*, *GEN* and *SIB* which will be presented in the next section. The syntax of *APPROX* clause is defined as follows:

$$\begin{aligned}
\prec \textit{approx clause} \succ &::= \text{APPROX } \prec \textit{approx expression} \succ \\
&\quad \prec [\text{TOP } \prec \textit{integer} \succ] \\
\prec \textit{approx expression} \succ &::= \prec \textit{relax operator} \succ \\
&\quad \mid \ \prec \textit{approx expression} \succ \text{ AND } \prec \textit{relax operator} \succ \\
\prec \textit{relax operator} \succ &::= \prec \textit{pred operator} \succ \mid \prec \textit{gen operator} \succ \mid \prec \textit{sib operator} \succ
\end{aligned}$$

Let us note that this clause is removed when executing the original query. If the query execution results in an empty answer set, the clause APPROX triggers the process of the relaxation of the query.

[1] A fuzzy set F [1] on the universe X is described by a membership function $\mu_F : X \rightarrow [0, 1]$, where $\mu_F(x)$ represents the membership degree of x in F.

[2] We use the OntoQL language to present our proposition. However the proposed extension could also be applied to other $\mathcal{SWDB}$ query languages such as SPARQL.

3.3 Relaxation operators

Now, we discuss a set of primitives operators for query relaxation. Each operator has a precise action on the query to relax. This action will be performed according to some given parameters.

PRED: The operator *PRED* is a variant of the *Fuzzy* clause where the satisfiability degree is set to 0. Moreover, the *PRED* operator can be repeated many times until a top-k answers is obtained or the relaxation amount does not lie in a validity interval V. This interval can be set by the user or by default to the value $V = \big[(\sqrt{5}-1)/2, (\sqrt{5}+1)/2\big]$ for numerical predicate [13] (in this case the tolerance value should be in $\big[0, (3-\sqrt{5})/2\big]$.). The *PRED* operator computes all the approximate answers as well as their satisfiability degrees and then rank-orders the answers.

The syntax of *PRED* operator is defined as follows:

$$\prec \textit{pred operator} \succ ::= \textit{PRED}\ (\prec \textit{var} \succ [, \textit{tol}, \textit{interval}])$$

var is the property to relax, the constant *tol* is the tolerance value and *interval* is the validity interval. So, the signature of PRED is: $\mathbb{Q} \times \mathbb{P} \times \textit{Real} \times \textit{Interval} \to \mathbb{Q}$ where $\mathbb{Q}$ is the set of users' queries, $\mathbb{P}$ the set of properties.

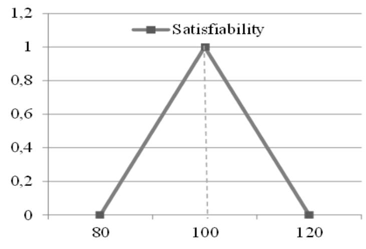
Example 2. Let us come back to the example 1, the relaxed query Q' of Q can be obtained using the *PRED* as follows: $Q' = \textit{PRED}(Q, \textit{Price}, 2, [10, 10])$. In OntoQL syntax, Q' writes:
select $Name$, $Price$ from $Motel$ where $Price \geq 113$ and $Price \leq 114$
$\textit{Approx}(\textit{Pred}(\textit{Price}, 2, [10, 10]), \textit{top} - k)$.

Note that the *PRED* operator is still valid in case where the property has a single value or a closed/opened interval value (see Figure 1).

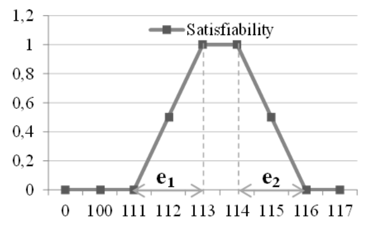
GEN: Generalization is a particular substitution of concept. This operator uses entailment of ontology $\mathbb{O}$ to guide the relaxation. Let C be an ontology's class. With the explicit definition of classes and relationship between them in the ontology, one can know all the superclasses of C. With the operator *GEN* all these super-classes can be used for relaxing C.
The syntax of the operator *GEN* is defined as follows:

$$\prec \textit{gen operator} \succ ::= \textit{GEN}\ (\prec \textit{var} \succ [, \textit{Integer}])$$

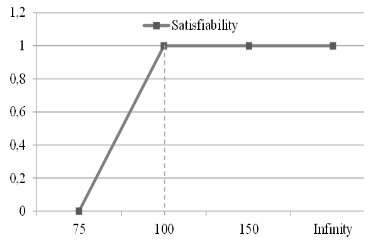
var is the property to relax and the constant *Integer* is the maximal level of generalization authorized, its default value is 1.
Let C' be one of superclasses of C. Now, for a query Q on class C, the *GEN* operator will relax Q by replacing C by C'. As for the properties selected by the user, called *Projection of* Q and noted $Proj(Q)$, we apply a particular treatment. All the properties of C present in $Proj(Q)$ and in C' are conserved in $Proj(Q')$



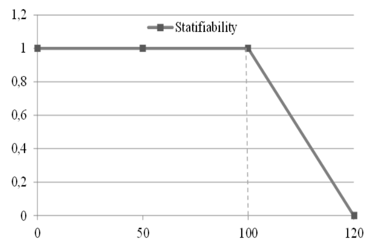
(a) Predicate on a Single Value



(b) Predicate on an Closed Interval



(c) Predicate on an Open Interval



(d) Predicate on an Open Interval

Fig. 1: Different Predicate relaxations

and for the other we assign them a *null* value. The signature of *GEN* writes then: $\mathbb{Q} \times \mathbb{C} \times Integer \rightarrow \mathbb{Q}$ where $\mathbb{C}$ is the set of classes. In $GEN(Q, C, i)$, Q is the query to relax, C is the class to generalize, it must be in the set of classes on which the selection is operated (noted $Dom(Q)$), and i is the maximal level allowed for the superclass C' which will generalize C. The reference level 0 refers to the class C itself.

Proposition 1. *Let Q and Q', be two elements of the set of queries $\mathbb{Q}$ and $dom(C)$ the set of properties of the class C, we have $Q' = GEN(Q, C, i)$ if and only if (where SC means Sub-Class relation):*

$$\exists\ C_0, ..., C_i \in \mathbb{O} \diagup\ \forall\ k \in \{0..i-1\}, (C_k, \mathrm{SC}, C_{k+1});$$
$$C_0 = C \text{ and } C_k = C'$$
$$C \in Dom(Q) \text{ and } C' \in Dom(Q')$$
$$Proj(Q') = (Proj(Q) - dom(C)) \cup (dom(C) \cap dom(C')$$

Let $Sim(Q, Q')$ denotes the similarity between Q and Q'. It is easy to check that:

Corollary 1. $\forall\ C_0, ..., C_i \in \mathbb{O} \diagup\ \forall\ k \in \{0..i-1\}, (C_k, \mathrm{SC}, C_{k+1})$*;*
we have $\forall\ k \in \{0..i-1\}, Sim(Q, Q_k) \geq Sim(Q, Q_{k+1})$
with $C_0 \in Dom(Q)$ *and* $\forall\ k \in \{0..i\}, C_k \in Dom(Q_k)$.

One can observe that to obtain high quality of approximate answers, it is better to apply this operator in an incremental way (w.r.t to the parameter i).

Example 3. Considering the ontology *HotelBase* (figure 4 in annex) and the query Q of the example 1:
select $Name$, $Price$ from $Motel$ where $Price \geq 113$ and $Price \leq 114$.
If we have $Q' = GEN(Q, Motel, 2)$ then Q' writes:
select $Name$, $Price$ from $Hotel$ where $Price \geq 113$ and $Price \leq 114$
union select $Name$, $Price$ from $Lodging$ where $Price \geq 113$ and $Price \leq 114$.
where $Hotel$ is the superclass of $Motel$ at *level 1* and *Lodging at level 2* .

SIB: Sibling is another form of substitution, it uses entailment but in a different way than *GEN* operator. Let C be a class, sibling operator replaces the class C by the class C_i which has the same direct superclass than C. Assume that the direct superclass of C is C_s. Then, the relation (C, SC, C_s) holds in the ontology $\mathbb{O}$. The syntax of the operator *SIB* is defines as follows:

$$\prec \textit{gen operator} \succ ::= SIB\ (\prec var \succ [, var_1[(, var_2)^*]])$$

where the first variable is the class to relax, the second is the sibling class used for the relaxation and the optional other variables denote other sibling classes, in the case where the relaxation process is applied more than one time.
This operator can be seen as a particular restriction of the generalization operator. Indeed, when we generalize a class C with its direct superclass C_s, we add all the sibling classes of C because they are also subclasses of C_s. With this operator, one can choose specifically one sibling for the substitution. The usage of this operator requires knowledge of the data model, since the user must know which classes are sibling of the class at hand.
The signature of *SIB* is: $\mathbb{Q} \times \mathbb{C} \times 2^{\mathbb{C}} \rightarrow \mathbb{Q}$. In $SIB(Q, C, [C_1, C_2, ..., C_n])$, Q is the query to relax, C is the class in $Dom(q)$ to replace and $C_1, C_2, ..., C_n$ a list of sibling classes of C.

Proposition 2. *Let Q and Q', be two elements of the set of queries $\mathbb{Q}$ and $dom(C)$ the set of properties of the class C, we have $Q' = SIB(Q, C, [C_1, C_2, ..., C_n])$ if and only if:*

$$\exists\ C_{Sp} \in \mathbb{O}\ /\ (C, \mathrm{SC}, C_{Sp}) \text{ and } \forall i \in \{1...n\}, (C_i, \mathrm{SC}, C_{Sp});$$
$$C \in Dom(Q) \text{ and } \forall i \in \{1...n\}, C_i \in Dom(Q')$$
$$Proj(Q') = (Proj(Q) - dom(C)) \cup (\bigcup_{i=1..n} (dom(C) \cap dom(C_i)))$$

Corollary 2. *From Proposition 2, we have $\forall_{i \in 1..n}, Sim(Q, Q_i) \leq Sim(Q, Q_{Sp})$ with $C \in Dom(Q)$ and $\forall i \in \{1...n\}\, C_i \in Dom(Q_i)$ and $C_{Sp} \in Dom(Q_{Sp})$.*

One can check that $GEN(Q, C, 1) \supseteq SIB(Q, C, [C_1, ..., C_n])$ where $C, C_1, ..., C_n$ are all subclasses of a same direct superclass C_{Sp}.

Example 4. Considering the ontology of $HotelBase$ (figure 4) and the query Q of the example 1:
select $Name$, $Price$ from $Motel$ where $Price \geq 113$ and $Price \leq 114$.
If we have $Q' = SIB(Q, Motel, [Inn, Resort, Retreat])$ then Q' writes:
select $Name$, $Price$ from Inn where $Price \geq 113$ and $Price \leq 114$
union select $Name$, $Price$ from $Resort$ where $Price \geq 113$ and $Price \leq 114$
union select $Name$, $Price$ from $Retreat$ where $Price \geq 113$ and $Price \leq 114$.
As mentioned above, $Q" = GEN(Q, Motel, 1) \supseteq Q' = SIB(Q, Motel, [Inn, Resort, Retreat, Hostel, BedAndBreakfast])$

3.4 Combining relaxation with AND logic Operator

The operator *GEN*, *SIB* and *PRED* can be associated with a logical connector *AND* for extending the relaxation operation. But for using this logic operator one needs to define its syntax and semantics. If *PRED* extends the value selected and *GEN* or *SIB* changes the domain, these two kind of operators can be then handled separately.

Conjunction of PRED: Let Q be a query with condition clauses on properties p_1 and p_2, we have only the following case:

1. $Q' = PRED(Q, p_1, \epsilon_1, I_1)$ AND $PRED(Q, p_2, \epsilon_2, I_2)$ since the two operators act on two different properties, each property can be relaxed independently of the other before execution of Q', $Q' = (Q_1, Q_2)$ where $Q_1 = PRED(Q, p_1, \epsilon_1, I_1)$ and $Q_2 = PRED(Q, p_2, \epsilon_1, I_2)$ and (Q_1, Q_2) means simultaneous relaxation.

Conjunction of GEN and/or SIB: For the conjunction of the same operator on different attribute classes:

1. $Q' = GEN(Q, c_1, level_1)$ AND $GEN(Q, c_2, level_2)$ the relaxation is applied independently on C_1 and C_2. The relaxation is also simultaneous, which means if $Q_1 = GEN(Q, c_1, level_1)$ and $Q_2 = GEN(Q, c_2, level_2)$ we will have $Q' = (Q_1, Q_2)$.
2. $Q' = SIB(Q, c_1, [c_1, .., c_n])$ AND $SIB(Q, c_2, [c'_1, ...; c'_m])$ can be written under the form $Q' = \bigcup_{c_i}\bigcup_{c'_j}(Q_{c_i}, Q_{c'_j})$, with $Q_{c_i} = SIB(Q, c_1, [c_i])$ and $Q_{c'_j} = SIB(Q, c_2, [c'_j])$.

We can also have conjunction of *GEN* and *SIB*, on the same classes or on different classes:

1. $Q' = GEN(Q, c, level_1)$ AND $SIB(Q, c, [c_1, .., c_n])$, since $level_1 \geq 1$ the generalization of c will use a superclass of c, as $Q_1 = GEN(Q, c, level_1)$ will already include all the sibling of c, so $Q_2 = (Q, c, [c_1, .., c_n]) \subset Q_1$. Hence $Q' = GEN(Q, c, level_1)$.
2. $Q' = GEN(Q, c, level_1)$ AND $SIB(Q, c', [c'_1, .., c'_n])$ the class c is substituted in all the sub-queries of the *Union* clause of *SIB*. So, $Q' = \bigcup_{i \in \{1...level_1\}} (SIB(GEN(Q, c, i), c', [c'_1, .., c'_n]))$.

As it can be seen, all these operators allow users to control precisely and easily the relaxation process using a $\mathcal{SWDB}$ query language.

Computation of similarity and satisfiability: Jean et al.[13] propose the following similarity measure.

Proposition 3. *The similarity between an original class C and its relax class C' is :*

$$Sim(c, c') = \frac{IC(msca(c, c'))}{IC(c) + IC(c') - IC(msca(c, c'))} \quad (1)$$

Where $IC(c) = -log(Pr(c))$ corresponds to the information content of the class c which is defined according to the probability $(Pr(c))$ of getting an instance of the class c in the ontology $\mathbb{O}$. $msca(c, c')$ corresponds to the first concept which subsumes the two concepts c and c'.

Proposition 4. *The satisfiability of an approximate answer h_i of the original query Q is given by:*

$$SatQ(h_i) = \min(Sim(Q', Q), SatQ'(h_i)) \quad (2)$$

with

$$Sim(Q', Q) = \min_{i=1..n} (Sim(c_i, c'_i)) \quad (3)$$

where c_i are all the classes relaxed in Q and c'_i the corresponding relaxed classes; and

$$SatQ'(h_i) = min(\max_{t \in directType(h_i)} Sim(t, c'), \mu_p(h_i.propRelax)) \quad (4)$$

where c' is the relaxed class which gives answer h_i and propRelax is the property which have been relaxed if it is the case.

This similarity is used for a single relaxation operation. For a conjunction of relaxation operators, we propose an extension of these similarities. Since the logical connector *AND* is associative, we can use this property to extend the combination for n relaxation operators. In case of a conjunction of relaxation operators, the satisfiability degree of an approximate answer w.r.t. the original query Q is given as follows:

Proposition 5. *Let Q and Q' be two elements of the set of queries $\mathbb{Q}$ with $Q' = \underset{i}{AND}\ Op_i(Q)$, $\forall i, Op_i \in (GEN, SIB, PRE)$ and h_r an answer of Q', we have the following:*

$$SatQ(h_r) = \min(Sim(Q', Q), SatQ'(h_k))$$
$$Sim(Q, Q') = \min_i(Sim(Q, Op_i(Q)))$$
$$SatQ'(h_r) = min((\max_{c \in directType(h_r)} Sim(c, c')), (\min_j \mu_{P_j}(h_r.P_j))$$

where c' is the relaxed class in the case of GEN or SIB operator, and μ_{P_j} the membership function of the j^{th} property P_j relaxed by the PRED operator, Op_i is the i^{th} operation (GEN, SIB or PRE) on Q, h_r is the r^{th} answer.

4 Implementation and Experiments

4.1 Implementation

The relaxation operators discussed have been implemented in the $\mathcal{SWDB}$ OntoDB/OntoQL. In this $\mathcal{SWDB}$, ontology and instances are stored using a table per class layout. In this layout, a table is created for each class with a column for each single-valued property of this class. Multi-valued properties are represented as two columns tables. At the query engine of OntoQL, we add a module for relaxation. Figure 2 shows the previous query engine (Analyzer and Interpreter) enriched with a module for generating the relaxed queries (Generate Next Step Query). Another module for storing the top-k answers (storing Answer) is also added with a module for ranking alternative answers (Ranking) following the similarity measure defined and the result is given to the users.

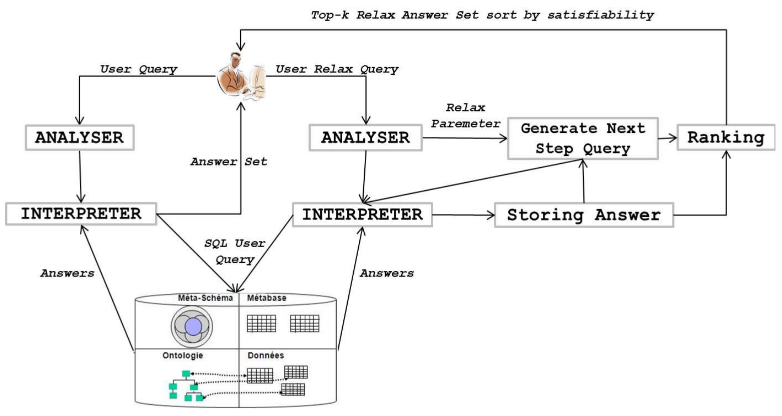


Fig. 2: Relaxation Query Engine

4.2 Experimentation

The experimentation has been done on real data with usual queries. We use model data HotelBase (Figure 4 in the annex) where the repartition of data is given in Table2 and execute 11 queries also given in the annex.

Table 2: Number of Instances by class

Class	Lodging	Hotel	Vacation Rental	Other	Apartment	Motel	Inn	B&B	Hostel	Retreat	Resort
#Instances	473165	426357	2863	17630	26315	8853	6560	18452	7186	539	14410

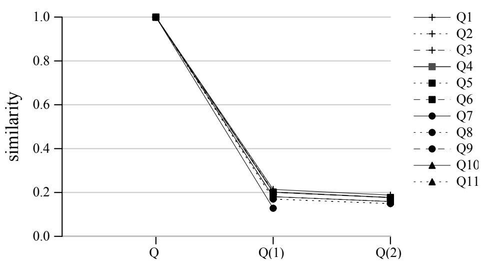
For each query, we get the cost time of the original query execution and the cost time of each operation of relaxation applied on this query. We compute also for each query the variation of the similarity at each the step of the relaxation. We measure also the size of instances on which the relaxed queries will be executed (Such measures are not presented here due to the limitation pages). The results of this experiment are given and analyzed in the next section.

4.3 Experiments Results and Analysis

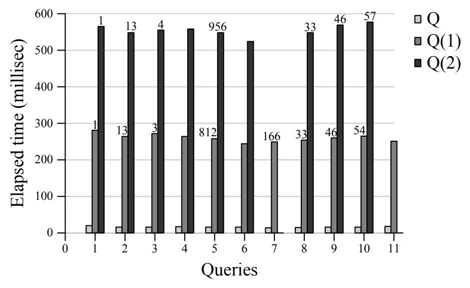
Cost Time: The figure 3 gives the cost time for each kind of relaxation operator applied to the 11 queries considered. It shows that the execution time of *GEN* and *SIB* at each step of relaxation depends also on the size of data provided by *From* clause. Except for *PRED* operator where the size of data is the same at each step (since *PRED* does not modify the domain of the query at hand). Note that the cost time for *PRED* at the step i includes the cost time of each previous step j ($j = 1..(i-1)$). In practice, this operator leads to a best cost time than SIB and *GEN* since the same set of instances is used to evaluate the original query and its relaxed variants. The performance of *PRED* can be improved using the cache memory to store this set of instances.

For *GEN*, the histogram of the figure 3b shows the great difference of execution times between successive relaxation steps is proportional to the difference between the size of instances (which strongly increase according to the level of relaxation). This great difference explains the heterogeneous of the dataset of HotelBase. The figures 3b and 3d confirm that *SIB* is better than *GEN*, which is intuitive since *SIB* takes sibling classes which have small instance of data than the superclasses.

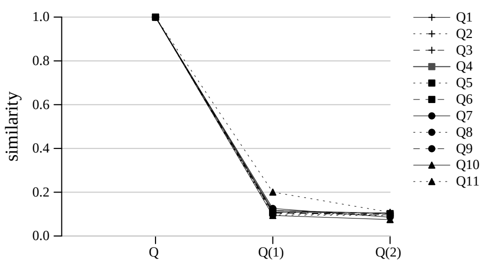
Answer Set: Figure 3 gives also the similarity at each step of relaxation and the size of the result for each operator. Figures 3a, 3c and 3e show that the similarity decreases when the relaxation step is increasing. In case of *PRED*, this similarity



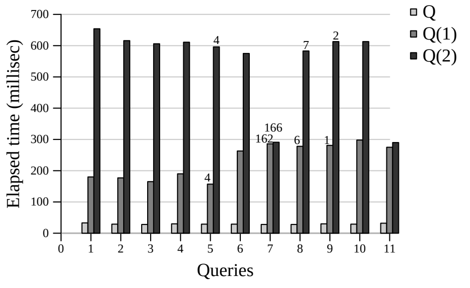
(a) GEN Similarity variation



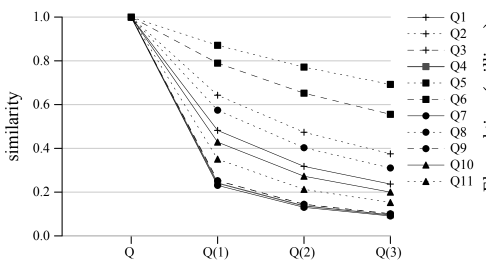
(b) GEN Cost Time and Answers set Size



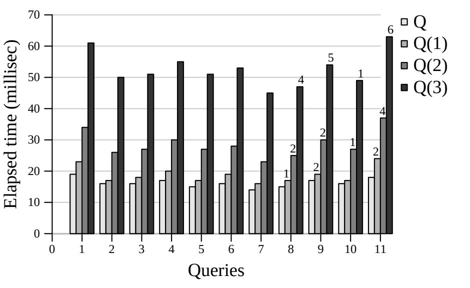
(c) SIB Similarity variation



(d) SIB Cost Time and Answers set Size



(e) PRED Similarity variation Similarity



(f) PRED Cost Time and Answers set Size

Fig. 3: Cost Time, size of Answers set and Similarity variation for each operator

does not decrease as fast as in case of *GEN* and *SIB*. Additionally to its time performance, *PRED* is the operator that conserves better the similarity in the relaxation process. Now, since the similarity measure depends on information content, the rapid fall of similarity of *GEN* and *SIB* can be explained by the heterogeneous repartition of instances between classes. Figures 3b, 3d and 3f give the number of answers for each operator (numbers on the top of each histogram). These numbers show that *PRED* gives less answers than SIB and *GEN*, this is due to the heterogeneous repartition of values between instances of classes. While this heterogeneity does not affect *GEN* and *SIB* since they retrieve answers in other classes.

5 Conclusion

In this paper, we have addressed the issue of query relaxation in the SWDB context. We have proposed a set of primitive relaxation operators and also shown how these operators can be integrated in a query language. A set of experiments has been conducted to demonstrate the feasibility of the approach and study both the time performance and the quality of alternative answers for each operator. The analysis of experiment results reveals that the structure of the data model (i.e., ontology) and the data repartition in classes impact the performance and quality for each operator.

For instance,statistics on data could be used as indicator for choosing the best relaxation operator to apply. Another line of research is to implement a relaxation advisor, for advising user to the inconvenient and advantage of using different operators or their combination.

Finally, one can also leverages user feedback to evaluate the results provided by the relaxation process.

References

1. P. Bosc and O. Pivert. Sqlf: A relational database language for fuzzy querying. *Trans. Fuz Sys.*, 3(1):1–17, February 1995.
2. Peter Dolog, Heiner Stuckenschmidt, Holger Wache, and Jörg Diederich. Relaxing rdf queries based on user and domain preferences. *IJIIS*, 33(3):239–260, 2009.
3. Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: A comparison of rdf benchmarks and real rdf datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 145–156, New York, NY, USA, 2011. ACM.
4. Shady Elbassuoni, Maya Ramanath, and Gerhard Weikum. Query relaxation for entity-relationship search. In *ESWC'11*, pages 62–76, 2011.
5. Aidan Hogan, Marc Mellotte, Gavin Powell, and Dafni Stampouli. Towards fuzzy query-relaxation for rdf. In *ESWC'12*, pages 687–702, 2012.
6. Dehainsala Hondjack, Guy Pierra, and Ladjel Bellatreche. Ontodb: An ontology-based database for data intensive applications. In *DASFAA*, pages 497–508, 2007.
7. Hai Huang and Chengfei Liu. Query relaxation for star queries on rdf. In *Proceedings of the 11th international conference on Web information systems engineering*, WISE'10, pages 376–389, Berlin, Heidelberg, 2010. Springer-Verlag.

8. Hai Huang, Chengfei Liu, and Xiaofang Zhou. Approximating query answering on rdf databases. *World Wide Web*, 15(1):89–114, January 2012.
9. Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. A relaxed approach to rdf querying. In *Proceedings of the 5th international conference on The Semantic Web*, ISWC'06, pages 314–328, Berlin, Heidelberg, 2006. Springer-Verlag.
10. Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. Journal on data semantics X. chapter Query relaxation in RDF, pages 31–61. 2008.
11. Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. Ranking approximate answers to semantic web queries. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC 2009 Heraklion, pages 263–277, Berlin, Heidelberg, 2009. Springer-Verlag.
12. Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB Journal*, 19(1):91–113, 2010.
13. Jean Stéphane, Hadjali Allel, and Mars Ammar. Towards a cooperative query language for semantic web database queries. In *ODBASE 2013*, pages 519–526.

Annex

1.
```
 SELECT name, nbstart, price, cityname FROM BedAndBreakfast B
WHERE B.price<= 29 AND B.nbstart=5 AND B.cityname='Belgrade'
```

2.
```
 SELECT name, nbstart, price, cityname FROM Inn I
WHERE I.price<= 15 AND I.nbstart=5 AND I.cityname='Shanghai'
```

3.
```
 SELECT name, nbstart, price, cityname FROM Motel M
WHERE M.price<= 80 AND M.nbstart=5 AND M.cityname='Istanbul'
```

4.
```
 SELECT name, nbstart, price, statename FROM Resort R
WHERE R.price<= 85 AND R.nbstart=5 AND R.statename='California'
```

5.
```
 SELECT name, nbstart, price, cityname FROM Motel M
WHERE M.nbstart<=4 AND M.cityname='Istanbul'
```

6.
```
 SELECT name, nbstart, price, cityname FROM Resort R
WHERE R.nbstart>=8 AND R.cityname='Istanbul'
```

7.
```
 SELECT name, nbstart, price, cityname FROM VacationRental R
WHERE R.nbstart=4  AND R.cityname='Istanbul' AND price<90
```

8.
```
 SELECT name, nbstart, price, cityname FROM Inn I
WHERE I.price<= 20 AND I.nbstart=3 AND I.cityname='Shanghai'
```

9.
```
 SELECT name, nbstart, price, statename FROM Resort R
WHERE R.price< 80 AND R.nbstart=3 AND R.statename='Pennsylvania'
```

10.
```
 SELECT name, nbstart, price, cityname FROM Motel M
WHERE M.price >= 40 AND M.nbstart=3 AND M.cityname='Nanjing'
```

11.
```
 SELECT name, nbstart, price, cityname FROM Apartment A
WHERE A.nbstart = 4  AND A.cityname='Kolobrzeg' AND A.price< 50  AND A.price >= 45
```

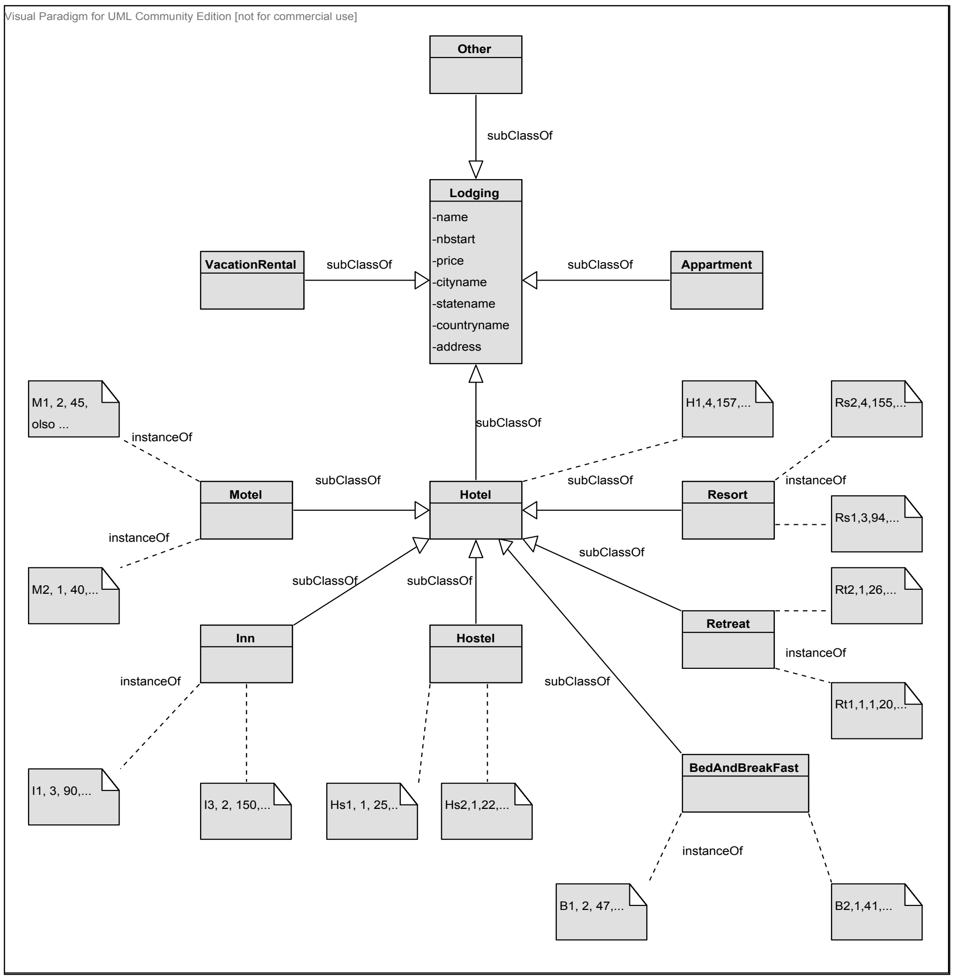


Fig. 4: Model and Instance of HotelBase