Computing Explanations for Unlively Queries in Databases

Guillem Rull\textsuperscript{1,2} Carles Farré\textsuperscript{2} Ernest Teniente\textsuperscript{2} Toni Urpí\textsuperscript{2}
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
1-3 Jordi Girona
Barcelona 08034, Catalonia, Spain
\{grull, farre, teniente, urpi\}@lsi.upc.edu

ABSTRACT

A query is lively in a database schema if it returns a non-empty answer for some database satisfying the schema. Debugging a database schema requires not only determining queries (as well as views or tables) that are not lively, but also fixing them. To make that task easier it is required to provide the designer with some explanation of why a query is not lively. To the best of our knowledge, the existing methods for liveliness checking in databases do not provide such explanations. An explanation is understood as the minimal set of constraints that are responsible for the non-liveliness of the tested query. In this paper, we propose a method for computing such explanations that is independent of the particular method used to determine the liveliness of a given query. Our method provides three levels of search: one explanation, a maximal set of non-overlapping explanations, and all explanations. The first two levels require only a linear number of callings to the underlying method. In addition, we propose a filter to reduce the number of callings to the underlying liveliness method. We also experimentally compare our method with a more naive method for query liveliness and with the best known method for finding unsatisfiable subsets of constraints.

Categories and Subject Descriptors:
H.2.1 [Database Management]: Logical Design - schema and subschema.

General Terms: Verification.

Keywords: Database schema validation, query liveliness.

1. INTRODUCTION

A query is lively [2,4,9] in a database schema if it returns a non-empty answer for some database satisfying the schema. Clearly, queries that are not lively are meaningless since they will always have an empty extension. Liveliness applies also, in a similar way, to views and tables of the schema.

Debugging a database schema requires not only determining queries, views or tables that are not lively, but also fixing them. To make that task easier it is required to provide the designer with some explanation of why a query is not lively. However, as far as we know, existing methods for liveliness checking in databases do not provide such explanations.

An explanation is understood as the minimal set of constraints that are responsible for the non-liveliness of the tested query. Minimality ensures that no proper subset of an explanation will also be an explanation. In general, there may be more than one explanation for a certain liveliness test.

The explanations are intended to help the database designer to find the problem and to fix it. A query may be non-lively due either to (1) its own definition (which may, for instance, contain a contradiction) or to (2) the integrity constraints of the schema that prevent the query to ever have any instance. Assuming that all the tables and views mentioned in the definition of the tested query have already been checked and, thus, they are known to be lively, the first cause of unliveliness (its explanation would be the empty set) is easy to fix since the designer only has to focus on the query’s definition. Instead, the second cause is more difficult to deal with, since the schema may be big and containing many interrelated constraints, in such a way that finding the exact constraints that make the query unlively may be a hard work. Thus, highlighting the problematic constraints would be indeed very helpful when debugging a database schema.

However, finding all explanations is a challenging problem [1]. If it is done naively, it may require checking all possible subsets of constraints. Moreover, in the worst case there are an exponential number of possible explanations.

The main goal of this paper is to propose a method that, given any known method for liveliness checking in databases, allows this method to compute explanations when it determines that a given query is not lively. That is, our method will allow computing explanations independently of the particular method used to determine liveliness.

We analyze two different approaches to compute such explanations. The forward approach follows the straightforward way, which consists on checking all subsets of constraints as a candidate explanation, starting with the empty set and moving to subsets with higher cardinality. In contrast, the backward approach is intended to find a first explanation quickly and then to use the knowledge from that explanation to find the remaining ones.

The method we propose is based on the backward approach and, in addition of being more efficient, it provides three levels of explanation search. The idea is that each one of these levels provides more information than the previous, although also requires more time to be computed. The first level is aimed at

\textsuperscript{1}This work was supported in part by Microsoft Research through the European PhD Scholarship Programme.
\textsuperscript{2}This work was supported in part by the Spanish Ministerio de Educación y Ciencia under project TIN2005-05406.
finding just one explanation. This is done by reducing the number
of constraints in the schema until only the constraints forming the
explanation remain. Thus, it requires only a linear number of calls
to the underlying liveliness method, with respect to the number of
constraints in the schema. In the second level, our method finds
the maximal set of non-overlapping explanations that includes the
one found in the previous phase. This is interesting because we
can provide more than just one explanation while keeping the
number of calls linear and, in addition, given that all remaining
explanations overlap with the ones already found, the designer has
a clue of where the rest of errors might be. Finally, in the third
level, we compute all explanations by taking advantage of the fact
that the remaining explanations must overlap with the ones found
in the previous phases. However, this third level already
introduces an exponential number of calls.

We also propose a filter that can be used in both approaches to
reduce the number of times that the underlying method for
liveliness checking is called to compute the explanations. The
filter is based on discarding those candidate subsets that contain
constraints that are not relevant for the liveliness test. The non-
relevant constraints are those that refer to tables or views that are
not required to contain tuples to make the tested query livel y.

We provide an experimental evaluation to compare the
backward approach with respect to the forward approach and also
with respect to the best known method for finding minimal
unsatisfiable subsets of constraints, the hitting set dualization
approach, proposed in [1] for the context of type error and circuit
e rror diagnosis. We also study the behavior of the backward
approach when varying some parameters like the number and size
of the explanations. These experiments have been performed
using our CQC Method [3] as liveliness method. The CQC
Method is able to handle a broader class of database schemas and,
thus, we are able to consider schemas with a high degree of
expressiveness.

The rest of the paper is organized as follows. Section 2
introduces background concepts. Section 3 describes the two
approaches for computing explanations for non-lively queries in
databases, the backward and forward approaches, and the filter.
Section 4 introduces the experiments we performed and comment
on the results. Section 5 exposes the related work and, finally,
section 6 presents the conclusions and future work.

2. BACKGROUND

In this section, we introduce the basic concepts and the logic
notation used throughout the paper.

A database schema \( S = (DR, IC) \) where \( DR \) is a finite
set of deductive rules and \( IC \) a finite set of constraints. A
deductive rule has the form:

\[
p(\vec{X}) \leftarrow r_1(\vec{X}) \land \ldots \land r_n(\vec{X}) \land
\neg r_{n+1}(\vec{Y}) \land \ldots \land \neg r_m(\vec{Z}) \land C_1 \land \ldots \land C_t
\]

A constraint (or condition) has the denial form:

\[
\leftarrow r_1(\vec{X}) \land \ldots \land r_n(\vec{X}) \land
\neg r_{n+1}(\vec{Y}) \land \ldots \land \neg r_m(\vec{Y}) \land C_1 \land \ldots \land C_t
\]

The symbols \( p \) and \( r_1, \ldots, r_n \) are predicates. The tuples \( \vec{X}, \vec{X}_1, \ldots, \vec{Y}, \ldots, \vec{Z} \) contain terms, which are either variables or
constants. Each \( C_i \) is a built-in literal in the form of \( t \theta \), where \( t_1 \) and \( t_2 \) are terms and operator \( \theta \) is \( \leq, >, \geq, = \) or \( \neq \). The atom
\( p(\vec{X}) \) is the head of the rule, and \( r_1(\vec{X}), \ldots, r_n(\vec{X}), \neg r_{n+1}(\vec{Y}), \ldots, \neg r_m(\vec{Z}) \) are positive and negative ordinary literals (those that are
not built-in). We require every rule and constraint be safe, that is,
every variable occurring in \( \vec{X}, \vec{Y}_1, \ldots, \vec{Y}_t, C_1, \ldots, C_t \) must also appear
in some \( \vec{X} \).

Predicates that appear in the head of a deduc tive rule are
derived predicates also called intensional database (IDB) pred icates. They correspond to views or queries. The rest are base
predicates also called extensional database (EDB) predicates.
They correspond to tables.

For a database schema \( S = (DR, IC) \), a database \( D \) is an EDB,
that is, a set of ground facts about the base predicates of \( S \) (the
tuples stored in the database). We denote by \( DR(D) \) the whole set
of ground facts about base and derived predicates that are inferred
from an instance \( D \), i.e., the fix-point model of \( DR \cup D \).

A database \( D \) violates a constraint \( \leftarrow L_1 \land \ldots \land L_k \)
if \( (L_1 \land \ldots \land L_k) \sigma \) is true on \( DR(D) \) for some ground substitution \( \sigma \). A database
\( D \) is consistent with the schema \( S \) if it violates no constraint in \( IC \).

A query over a database schema is a finite set of deductive
rules that define the same n-ary predicate. Given a database
schema \( S = (DR, IC) \) and a database \( D \), the answer to a query \( Q \),
defining the predicate \( q \), over \( S \) on \( D \), written \( A_q(D) \), is the set of
all ground facts about \( q \) obtained evaluating the deductive rules
from both \( Q \) and \( DR \) on \( D \), i.e., \( A_q(D) = \{ q(\vec{a}) \mid q(\vec{a}) \in (Q \cup DR(D)) \} \).

3. COMPUTING EXPLANATIONS

We assume that we have a procedure \( isLively \) to perform
liveliness tests of query predicates. Therefore, a liveliness test is a
call to \( isLively(Q, S) \), which will return true if \( Q \) is lively in \( S \) and
false otherwise. We say that an explanation for a liveliness test is
a subset of integrity constraints appearing in the database schema
such that they prevent the test to return true. In other words, the
predicate we are testing is still not lively when we remove all
integrity constraints that are not in the explanation.

**Definition 3.1.** An explanation \( E \) for a liveliness test
\( isLively(Q, S = (DR, IC)) \) that returns false is a minimal subset of
constraints from \( S \) such that considering only these constraints the
tested predicate \( Q \) is still not lively, i.e., \( isLively(Q, S' = (DR, E)) \) returns false too.

Note that, because \( E \) is minimal, \( isLively(Q, S' = (DR, E')) \) will return true for any \( E' \subset E \), i.e. the query \( Q \) is lively for any
proper subset of \( E \).

For the sake of example, let us consider a schema \( S \) with two
tables, \( R(A,B,C) \) and \( T(D,G) \). Table \( R \) has a constraint defined
over column \( B \) stating that \( B \) has to be lower than \( 10 \). Table \( T \) has
a constraint that forces column \( G \) to be greater than \( 30 \). Finally,
there is an inclusion dependency defined in table \( R \) over column \( C \)
that references column \( D \) in table \( T \), i.e., \( R.C \subseteq T.D \). More
formally, schema \( S \) is:

\[
\leftarrow R(A,B,C) \land B \geq 10
\]

\[
\leftarrow T(D,G) \land G \leq 30
\]

\[
\leftarrow R(A,B,C) \land \neg TT(C)
\]

\[TT(C) \leftarrow T(C,G).\]

Note the use of an auxiliary derived predicate, \( TT \), to express
the inclusion dependency. That is necessary, as we require
negation to be safe.

Let us assume now that we also have a query \( Q \) that we want to
check if it is lively in \( S \):
As we can see, the query \( Q \) performs a join with the two tables on \( R.C \) and \( T.D \) with the additional condition of selecting only those pairs of tuples from \( R \) and \( T \) such that \( R.B > T.G \).

Since, in this example, \( \text{isLively}(Q, S) \) returns false, we want to figure out the reason for that, i.e., the set of possible explanations. Here, there is only one explanation \( E \):

\[
E = \{ \leftarrow R(A,B,C) \land B \geq 10, \leftarrow T(D,G) \land G \leq 30 \}.
\]

Explanation \( E \) can be interpreted as follows. The requisite that all the values in column \( B \) of \( R \) must be lower than 10 together with the one that forces the values in column \( G \) of \( T \) to be greater than 30 makes impossible to find any pair of tuples from \( R \) and \( T \) where \( R.B > T.G \).

It is worth to note that the inclusion dependency is not included in the explanation. The reason is that the join performed by \( Q \) on \( R.C \) and \( T.D \) is not only compatible with that constraint but, indeed, it selects those pairs of tuples of \( R \) and \( T \) bound together to fulfill the constraint.

In this paper, we address the problem of finding all possible explanations in a way that is independent of the particular liveliness checking method that we use to perform the liveliness tests. Therefore, \( \text{isLively} \) is a black-box procedure that we call several times, modifying the (sub)set of integrity constraints considered in each call. This can be done forward, that is, by calling \( \text{isLively} \) with an increasing number of constraints -starting from the empty set of constraints; or backward, if \( \text{isLively} \) is calling decreasing each time the number of constraints that are considered.

### 3.1. Computing Explanations Forward

This is the most straightforward approach to compute the explanations. It consists in trying all the subsets of integrity constraints of the schema.

We start with the empty subset, and then we check the subsets with only one integrity constraint, then the subsets with two integrity constraints, and so on.

If the liveliness test for a given subset \( E \), \( \text{isLively}(Q, S' = (DR, E)) \), returns false then \( E \) is an explanation and there is no point to check any superset of it. Recall that, by the definition 3.1, supersets of explanations are not explanations because they are not minimal.

This process ends when all the subsets that still have not been checked are supersets of the explanations already founded.

In the case that the liveliness test for the initial empty subset, \( \text{isLively}(Q, S' = (DR, \emptyset)) \), returns false, this would mean that the non-liveliness of the query \( Q \) is not related to any integrity constraint. Therefore, the reason should be found in a contradiction in the definition itself of \( Q \), in which may be involved not only the deductive rules defining \( Q \) but also those rules in \( DR \) that define the derived predicates mentioned directly or indirectly by \( Q \).

The major drawback of this forward approach is the possibly huge number of subsets to be checked. In order to discard some of them in advance to avoid unnecessary executions of the liveliness test, we can apply some filters. Section 3.3 below discusses this point.

### 3.2. Computing Explanations Backward

In summary, the backward approach obtains explanations by discarding successively those constraints that are not included in any explanation. Therefore, in contrast with the forward approach, liveliness tests are performed with a decreasing number of constraints starting from the initial full set.

The backward approach consists of three phases. In the first one, we obtain just one explanation. In the second phase, we obtain those explanations that do not overlap with the one obtained in the first phase nor with other explanations found in this phase. We say that two explanations overlap if they share at least one constraint. Finally, in the third phase, we obtain the remaining explanations, that is, those that overlap with the explanations obtained in the two previous phases. The interesting point of the backward approach is that the two first phases require a linear number of repetitions of the test, independently of the size of the explanations.

#### 3.2.1. Phase 1

Let us assume that a given predicate \( Q \) is not lively on a certain database schema \( S \), so, \( \text{isLively}(Q, S) \) returns false. Phase 1 starts with performing the liveliness test of \( Q \) on a new schema containing all the integrity constraints from the former schema except one, \( c \). If \( \text{isLively}(Q, S - \{c\}) \) returns false, this means that there is at least one explanation that does not contain \( c \). Therefore, we can discard \( c \) definitely and repeat the liveliness test removing another constraint. Note that this does not mean that \( c \) does not belong to any explanation but only that \( c \) will not be included in the single explanation that we will obtain at the end of this phase. In contrast, if \( \text{isLively}(Q, S - \{c\}) \), this means that there are one (at least) or more explanations, each of them including \( c \). Therefore, \( c \) cannot be discarded and, thus, it is re-introduced in the schema and we repeat the liveliness test removing another constraint. We continue this process of removing a constraint, testing liveliness, discard or reintroduce, removing another constraint and so on until all the constraints in the schema have been considered.

```plaintext
phase_1(Q, predicate, S = (DR, IC): schema): explanation
U := IC // set of “unchecked” constraints
E := IC // explanation
while (3c ∈ U)
    E := E - \{c\}  
    if (isLively(Q, S' = (DR, E)))
        E := E \cup \{c\}
    endif
U := U - \{c\}
endwhile
return E
```

**Figure 1: Phase 1 of the backward approach.**

If at the end of this process all the constraints have been removed from the schema, we obtain an empty explanation, meaning that the predicate \( Q \) is not lively even without constraints. Otherwise, we have obtained just one explanation consisting of all those constraints that remain in the schema, that is, the ones that have been considered but not discarded during the process described above. The algorithm in Figure 1 formalizes such a process.

For the sake of example, let us assume that \( Q \) is a query defined as follows:
$Q \leftarrow R(X,Y,Z) \land V(Z,A,B) \land T(Z,U,V) \land Y > 5 \land B < X \land V = 2.$

Let us also assume that $S$ is a database schema with no deductive rules but containing the following constraints, labeled as $c_1$, $c_2$, $c_3$ and $c_4$, respectively:

- $\leftarrow T(X,Y,Z) \land Z = 2 \quad (c_1)$
- $\leftarrow R(X,Y,Z) \land Y > X \quad (c_2)$
- $\leftarrow R(X,Y,Z) \land X > 5 \quad (c_3)$
- $\leftarrow V(X,Y,Z) \land Z < 10. \quad (c_4)$

In this case predicate $Q$ is not lively in $S$. Concretely, there exist three explanations:

$E_1 = \{c_1\}$
$E_2 = \{c_2, c_3\}$
$E_3 = \{c_3, c_4\}$.

Let us call phase_1$(Q, S)$, with $S = \{c_1, c_2, c_3, c_4\}$ to find one of these three explanations. If we assume that the constraints are considered in the order that they were listed above, $c_1$ is considered first. Since $\text{isLively}(Q, \{c_2, c_3, c_4\})$ returns false, $c_1$ is discarded. Constraint $c_2$ is considered next. Since $\text{isLively}(Q, \{c_3, c_4\})$ returns false, $c_2$ is also discarded. Constraint $c_3$ is considered next. In this case, $\text{isLively}(Q, \{c_4\})$ returns true. Therefore, $c_3$ is not discarded. Finally, constraint $c_4$ is considered. Since $\text{isLively}(Q, \{c_3\})$ returns true, $c_4$ may not be discarded either. As a result, phase_1$(Q, S)$ returns $\{c_3, c_4\}$, that is, explanation $E_3$. Note that if the constraints had been considered in reverse order, for instance, the returned explanation would have been another: $\{c_1\} = E_1$.

### 3.2.2. Phase 2

The second phase of this backward approach assumes that we already found a non-empty explanation in the previous phase. The goal now is to obtain, at the end of the phase, a maximal set of explanations such that all the explanations in the set are disjoint, i.e., there is no constraint belonging to more than one explanation. One of these explanations will be the one we already found in phase 1.

```plaintext
phase_2(Q: predicate, S = (DR, IC): schema, E1: explanation): Set(explanation)
SE := \{E1\} // set of explanations
R := IC \setminus E1 // set of “remaining” constraints
while (not \text{isLively}(Q, S' = (DR, R)))
E := phase_1(Q, S' = (DR, R))
SE := SE \cup \{E\}
R := R \setminus E
endwhile
return SE
```

**Figure 2: Phase 2 of the backward approach.**

This phase proceeds as follows. We take the original schema and remove all the constraints included in the first explanation we found. In this way, we “disable” that explanation, in order to discover the other explanations, if any, that in phase 1 were “hidden” by it. Next, we perform the liveliness test with the remaining constraints. If the test returns false that means there is still, at least, another explanation non-overlapping with the one we have. To find out such a new explanation, we apply the first phase over the remaining explanations. On the contrary, if after removing the constraints from the former explanation, the liveliness test returns true that means that all the remaining explanations, if any, overlap with the first we had.

We repeat the process, removing the constraints from all the explanations we have found (the one from the first phase and the new ones we already found in this phase), until there are no more explanations that do not overlap with the ones we already have. The algorithm in Figure 2 formalizes such a process.

### 3.2.3. Phase 3

The third phase assumes that we already obtained a set of disjoint explanations by performing the previous phases. The goal now is to find all the remaining explanations, that is, those that overlap with some of the explanations that we already have. To do this, we must remove one constraint from each known explanation to “disable” them, and then apply the first and second phases over the remaining constraints. The drawback here is that there could be many constraints in each explanation and, thus, many constraints to be the one that will be removed to disable each explanation. Nevertheless, we should try all combinations to ensure we find all the remaining explanations.
Once we have removed one constraint for each explanation and executed the previous two phases over the remaining constraints, we get some new explanations that we will add to the set of explanations we already have. Next, we should repeat this third phase, taking into account these added explanations, until no new explanations are found. The algorithm in Figure 3 formalizes such a process.

Following the example of the previous sections, we already had found two explanations: \{c3, c4\} and \{c1\}. Now, if there is still some other explanation it will overlap with these. Thus, to avoid these explanations to hide the remaining ones, we select one constraint from each explanation and remove them from the original schema. In this example, there are two possibilities:

1) remove \{c1, c3\}
2) remove \{c1, c4\}.

Let us consider the first option. In this case, isLively(Q, \{c2, c4\}) returns true and, thus, no further explanation can be found.

In contrast, if we consider the second option, we get that isLively(Q, \{c3, c2\}) returns false. Therefore, we can still find further explanations. Next, we call phase_1(Q, \{c3, c2\}), which returns a new explanation: \{c3, c2\}. Clearly, phase_2(Q, \{c3, c2\}, \{c3, c2\}) will return \{c3, c2\} as a new set of explanations.

As we have found new explanations, we must repeat the process taking now into account all the explanations discovered so far. This time, there are four possible ways of removing one constraint from each explanation:

1) remove \{c1, c2, c3\}
2) remove \{c1, c2, c4\}
3) remove \{c1, c3\}
4) remove \{c1, c3, c4\}.

It is worth to note the option 3. As one of the constraints of the new explanation is shared with other explanation, by removing it we are “disabling” two of the three explanations. This is the case of constraint c3. Thus, the option 3 requires removing only two constraints as a difference from the other options that require three.

After trying the four possibilities, we reach the conclusion that there are no further explanations and, thus, the phase 3 is ended.

The outcome of this phase and of the entire approach is the set formed by the three explanations: \{\{c3, c4\}, \{c1\}, \{c3, c2\}\}.

3.3. Filtering Candidates with Non-Relevant Constraints

As we have seen, both the forward approach and the backward approach require performing several calls to isLively. In the case of the forward approach to check if the current subset of constraints is really an explanation and in the case of the backward approach to check if it is, indeed, a superset of an explanation. The filter described in this section consists in detecting those candidates that contain some constraint that we can ensure it is not relevant for the liveliness test. We can say that a constraint is not relevant for the test when to get a fact about the tested predicate it is not required to have also a fact about all the positive ordinary predicates in the constraint. The idea is that we do not need to perform the liveliness test for these candidates.

For example, let us assume that we have the following database schema:

\[ \leftarrow R(X,Y,Z) \land X \geq 5 \]
\[ \leftarrow T(X,Y,Z) \land Y < Z \]
\[ SS(Z) \leftarrow S(Z,Y) \]

Let us also assume that we are testing if the query \( Q \) is lively, being \( Q \) defined by the following rule:

\[ Q(X,Y) \leftarrow R(X,Y,Z) \]

Let us suppose that we are using the forward approach. During the process, we will reach the following candidate for being an explanation:

\[ \{\leftarrow R(X,Y,Z) \land Z < 5, \leftarrow S(X,Y) \land X \geq 5\} \]

Taking into account that our candidate does not contain the inclusion dependency from \( R \) to \( S \), \( Q \) is lively if we consider only these two constraints and, thus, this candidate is not an explanation. Applying the filter, we can see that the second constraint, \( \leftarrow S(X,Y) \land X \geq 5 \), is not relevant for the liveliness test of \( Q \) when it is performed over the schema containing only these two constraints. The constraint is not relevant because there is no need to have a fact about \( S \) in order to get a fact about \( Q \), e.g. the database \{\{R(0,0,5)\}\}. Therefore, we could avoid performing the liveliness test for this candidate and go directly to the next one.

The filter can be applied in the two approaches for computing explanations. Next, we are going to explain how apply the filter in each case.

To apply the filter in the forward approach, we can follow the next steps:

1. First, before starting the process, we could remove the integrity constraints that are already not relevant for the test when this is performed over the original database schema.
2. Then, during the process, for each candidate, we could compute the constraints that are relevant for the test not over the original schema but over the schema containing only the constraints in the candidate. The key point is that constraints that were relevant in the original schema may be not relevant now.
3. If at least one of the constraints in the candidate is not relevant, then we can directly discard the candidate. The reason is that if the candidate had made the test fail, then the candidate without the non-relevant constraint would have also made the test fail and, thus, the candidate would have been not minimal.
4. If all the constraints in the candidate are relevant, then we should perform the liveliness test to check whether it is an explanation or not.

Let us take again the previous example. In step 1, we would find that the constraint \( \leftarrow T(X,Y,Z) \land Y < Z \) is not relevant for the liveliness test of \( V \) over the original schema. Thus, we would remove it and perform the backward approach considering the schema only with the three remaining constraints. Let us suppose now that we have reached the candidate we mentioned earlier: \{\leftarrow R(X,Y,Z) \land Z < 5, \leftarrow S(X,Y) \land X \geq 5\}. Applying step 2, we would recompute the relevant predicates. Now, as we are not considering the inclusion dependency, the predicate \( S \) is not relevant and, thus, neither the constraint \( \leftarrow S(X,Y) \land X \geq 5 \) is relevant.
In the backward approach, the filter can be applied along the phase 1 (which is called also from phases 2 and 3). The steps are the following:

1. Before starting the phase 1, we could remove the constraints that are already non-relevant for the test over the original schema (as we did with the forward approach).
2. During the phase 1, when we remove one integrity constraint \( IC_i \) from the schema, we could recompute what predicates are relevant for the test when it is performed over the schema containing only the remaining constraints.
3. If some of the remaining constraints are not relevant, we can remove them before performing the test.
4. If then the test says that the predicate is still not lively we will have removed more than just one constraint and thus reduced the number of test execution we will have to do.
5. Otherwise, if the test says that the predicate is now lively, we will have to put back the constraint \( IC_i \) we initially removed together with the non-relevant ones.
6. If all the constraints are relevant, we can do nothing but continue the normal execution of the phase 1.

Let us consider again the same example as before. As in the case of the forward approach, in step 1 we would detect that the constraint \( \neg T(X,Y,Z) \land Y < Z \) is not relevant and, thus, we could eliminate it and perform the phase 1 over the remaining three constraints. Let us suppose that we follow the order in which the constraints were listed before. Then, we first would eliminate the inclusion dependency. That would leave two constraints in the schema: \( R(X,Y,Z) \land Z < 5 \) and \( S(X,Y) \land X \geq 5 \), this is, the same candidate we mentioned before. As we said, the later constraint is non-relevant for the liveliness test when the schema contains only these two constraints. Thus, we could remove it and perform the test with only one constraint: \( \neg R(X,Y,Z) \land Z < 5 \). Because the predicate becomes lively, we should put back the two removed constraints (the inclusion and the one about \( S \)). The phase 1 would remove then the next constraint: \( \neg R(X,Y,Z) \land Z < 5 \), and it would continue its execution in a similar way.

To characterize formally the constraints that are relevant for a certain liveliness test, we are going to assume that each constraint \( IC_i \) is p-relevant for some predicate \( P_i \) or q-relevant for some predicate \( Q_i \).

Let \( Q \) be a generic derived predicate defined by the following rules:

\[
Q(\bar{x}) \leftarrow P^{d}_i(\bar{X}) \land \ldots \land P^{d}_j(\bar{X}_j) \land C^1_1 \land \ldots \land C^1_{r_1} \\
\quad \land \neg S^1_j(\bar{X}_j) \land \ldots \land \neg S^1_{m_1}(\bar{X}_{m_1}) \\
\quad \ldots \\
Q(\bar{x}) \leftarrow P^{d}_k(\bar{X}) \land \ldots \land P^{d}_u(\bar{X}_u) \land C^k_1 \land \ldots \land C^k_{r_k} \\
\quad \land \neg S^k_j(\bar{X}_j) \land \ldots \land \neg S^k_{m_k}(\bar{X}_{m_k}).
\]

The symbols \( P^{d}_j, \ldots, P^{d}_u, S^{d}_j, \ldots, S^{d}_u, P^{d}_j, \ldots, P^{d}_u, S^{d}_j, \ldots, S^{d}_u \) are predicates and \( C^1_1, \ldots, C^1_{r_1}, \ldots, C^k_1, \ldots, C^k_{r_k} \) are built-in literals.

We will define \( \text{neg_preds}(Q) \) as the predicates of those negative literals that appear in the definition of \( Q \), taking into account all possible unfoldings. Formally:

\[
\text{neg_preds}(Q) = \{ \{ S \mid 1 \leq i \leq m_j \} \mid 1 \leq j \leq k \} \cup \\
\{ \{ \text{neg_preds}(P) \mid 1 \leq i \leq s_j \} \mid 1 \leq j \leq k \}
\]

\[
\text{neg_preds}(R) = \emptyset \quad \text{if } R \text{ is a base predicate.}
\]

We are going to define what predicates are relevant for the liveliness test of a certain predicate \( P \). There will be two types of relevancy: \( p\)-relevant and \( q\)-relevant. The \( p\)-relevant predicates will be those that in order to build a database where \( P \) is intended to be lively, it may be required to insert some fact about them in that database. The \( q\)-relevant predicates will be the derived predicates such that although it is not explicitly required for them to be lively in order to make \( P \) lively, they may become lively as a result of the facts inserted in the database.

**Definition 3.2.** Assuming that we are testing the liveliness of a certain predicate \( P \), we can say the following:

- **Predicate** \( P \) is \( p\)-relevant.
- If \( Q \) is a derived predicate and it is \( p\)-relevant, then \( P_i \) with \( 1 \leq i \leq s_j \) and \( 1 \leq j \leq k \), are also \( p\)-relevant predicates.
- If \( Q \) is a derived predicate and \( P_r, \ldots, P_s \) are \( p\)-relevant or \( q\)-relevant, for some \( 1 \leq j \leq k \), then \( Q \) is \( q\)-relevant.
- If \( Q \) is a derived predicate and there is a negated literal about \( Q \) in the body of a rule of some \( p\)-relevant derived predicate, and \( P_r, \ldots, P_s \) are \( p\)-relevant or \( q\)-relevant predicates, for some \( 1 \leq j \leq k \), then \( P_r, \ldots, P_s \) and the predicates in \( \text{neg_preds}(P_r) \cup \ldots \cup \text{neg_preds}(P_s) \) are \( q\)-relevant.
- If \( IC_i \leftarrow P^{d}_i(\bar{X}) \land \ldots \land P^{d}_j(\bar{X}_j) \land C^{d}_1 \land \ldots \land C^{d}_r \land \neg S^{d}_j(\bar{X}_j) \land \ldots \land \neg S^{d}_{m_d}(\bar{X}_{m_d}) \) is an integrity constraint and \( P_r, \ldots, P_s \) are \( p\)-relevant or \( q\)-relevant predicates, then \( IC_i \) is \( q\)-relevant and the predicates in \( \text{neg_preds}(IC_i) \) are \( q\)-relevant.

It is worth to note that a predicate defined by an integrity constraint cannot be \( p\)-relevant, as it is not mentioned anywhere but in the head of the constraint and, thus, only the last point of the definition is applicable.

**Definition 3.3.** We will say that an integrity constraint \( IC_i \leftarrow L_1 \land \ldots \land L_n \) is \( q\)-relevant for the liveliness test of \( P \) if and only if the derived predicate \( IC_i \) is \( q\)-relevant for that test.

As an example, let us assume that we have the following database schema:

\[
\begin{align*}
V(X,Y) & \leftarrow R(X,A,B) \land S(B,C,Y) \land \neg W(A,C) \\
W(X,Y) & \leftarrow P(X,Y) \land Y > 100 \\
P(X,Y) & \leftarrow T(X,Y) \land \neg F(X) \\
Q(X) & \leftarrow S(X,Y) \land Y \geq 5 \land Y \leq 10 \\
IC_1 & \leftarrow R(X,Y,Z) \land \neg T(Y,Z) \\
IC_2 & \leftarrow F(X,Y) \land X \leq 0
\end{align*}
\]

Derived predicates \( IC_1 \) and \( IC_2 \) correspond to two constraints. Let us also assume that we want to test if \( V \) is lively in this schema. Let us now compute what predicates are relevant for this liveliness test:

1. We start with \( p\)-relevant = \( \emptyset \) and \( q\)-relevant = \( \emptyset \).
2. The first point in the definition of predicate’s relevancy says us that, as we are testing the liveliness of \( V \), \( V \) is \( p\)-relevant predicate.
3. Then, \( p\)-relevant = \( \{ V \} \) and \( q\)-relevant = \( \emptyset \).
4. Now that we know \( V \) is \( p\)-relevant, by the second point of the definition we can infer that \( R \) and \( S \) are also \( p\)-relevant.
5. \( p\)-relevant = \( \{ V, R, S \} \) and \( q\)-relevant = \( \emptyset \).
6. As long as \( S \) is \( p\)-relevant, by the third point of the definition we can say that \( Q \) is \( q\)-relevant,
7. \( p\)-relevant = \( \{ V, R, S \} \) and \( q\)-relevant = \( \{ Q \} \).
(8) By the fifth point, as $R$ is $p$-relevant we can say that $IC_1$ is $q$-relevant and that $T$ is $q$-relevant.

(9) $p$-relevant = \{V, R, S, T\} and $q$-relevant = \{Q, IC_1\}

(10) Once we know that $T$ is $p$-relevant, by the third point again we can conclude that $P$ is $q$-relevant.

(11) $p$-relevant = \{V, R, S, T\} and $q$-relevant = \{Q, IC_1, P\}

(12) We can apply now the fourth point of the definition. The derived predicate $W$ appears negated in the rule of $V$ and $V$ is $p$-relevant. The predicates appearing positively in $W$, that is, $P$, are also relevant. Thus, we can infer that the predicates appearing negated in $W$ or some of its unfoldings are $p$-relevant. That means $U$ is $p$-relevant.

(13) $p$-relevant = \{V, R, S, T, H\} and $q$-relevant = \{Q, IC_1, P\}

(14) We still can apply the third point and say that as $P$ is $q$-relevant then $W$ is $q$-relevant also.

(15) $p$-relevant = \{V, R, S, T, H\} and $q$-relevant = \{Q, IC_1, P, W\}

(16) We cannot infer anything new and, thus, there are no other relevant predicates.

Finally, we can say that $IC_1$ is a relevant constraint for the liveliness test of $P$ if and that $IC_2$ is not relevant. It is easy to see intuitively that $IC_2$ is not relevant because predicate $F$ is not mentioned anywhere else (it is also non-relevant).

**Proposition 3.4.** Let $P$ be a non-lively predicate and let $IC_1$ be a constraint from the database schema. If $IC_1$ is not relevant for the liveliness test of $P$, then $P$ is still non-lively after removing $IC_1$ from the schema.

**Proof.** Let us assume that after removing $IC_1$, $P$ becomes livelively. It follows that exists some minimal database $D$ such that $D$ is consistent and some fact about $P$ is true in $D$. Database $D$ is minimal in the sense that there is no database $D'$ with less tuples than $D$, such that $D'$ is also consistent and contains some fact about $P$.

As long as $P$ becomes livelively after removing $IC_1$, database $D$ should violate $IC_1$. Our goal now is to show that it follows that $IC_1$ is $q$-relevant for the liveliness test of $P$. To reach that, we will do induction over the unfolding level of the predicates. A base predicate has an unfolding level of $0$. A derived predicate such that the maximum unfolding level of the predicates appearing positively in its rules is $n$, has an unfolding level of $n+1$. The base case will be when the predicate is a base predicate. Let $T$ be this predicate. We assume that there is at least one fact about $T$ in $D$. Given that $D$ is minimal, there are only two possibilities. The first is that a fact about $T$ may be required to satisfy the definition of $P$, i.e., a positive literal about $T$ appears in the definition of $P$ (taking into account all possible unfoldings). The second possibility is that the satisfaction of $P$ leads to the violation of some integrity constraint that can be repaired by means of the addition of a fact about $T$, i.e., there is some constraint with a negative literal about $T$ and such that all its positive literals are true in $D$. In both cases, the conclusion is that predicate $T$ is $p$-relevant for the liveliness test of $P$. The induction case will be when $T$ is a derived predicate. As long as some fact about $T$ is true in $D$, some rule defining $T$ should have all its literals true in $D$. By an induction, we can conclude that all the predicates from the positive literals in that rule are $p$-relevant or $q$-relevant and then that $T$ is $q$-relevant itself.

Finally, as $IC_1$ is true in $D$, we can conclude that $IC_1$ is $q$-relevant and, thus, we reach a contradiction.

### 4. EXPERIMENTAL EVALUATION

We have performed some experiments to compare the efficiency of the backward approach with respect to one of the forward approach and also with the best known method for finding minimal unsatisfiable subsets of constraints, the hitting set dualization approach [1]. We have also evaluated the behavior of the backward approach when varying some parameters: the size of the explanations, the number of explanations for each test, and the number of constraints in the schema. We executed the experiments on an Intel Core 2 Duo, 2.16 GHz machine with Windows XP (SP2) and 2 GB RAM.

To perform the liveliness tests in the experiments, we used our Constructive Query Containment (CQC) Method [3] and precisely the version implemented as a core of SVT (Schema Validation Tool) tool [8]. Remind anyway that our approach is independent of the method used. We have used here the CQC Method since it may handle a broader class of database schemas and, thus, we are able to consider schemas with a high degree of expressiveness.

Next, we do a brief overview of the CQC Method and SVT, and then we describe the experiments and comment on the results.

### 4.1. CQC Method and SVT

The CQC (Constructive Query Containment) Method [3], originally defined for query containment, performs a validation test by trying to build a consistent instance for a database schema in order to satisfy a given goal (a conjunction of literals). It is able to deal with database schemas having integrity constraints, safe-negated EDB and IDB literals, and comparisons.

The method starts with the empty instance and uses different Variable Instantiation Patterns (VIPs), according to the syntactic properties of the views/queries and constraints in the schema, to generate only the relevant facts to be added to the instance under construction. If the method is able to build an instance that satisfies all literals in the goal and does not violate any constraint, then that instance is a solution and it shows that the goal is satisfiable. The key point is that the VIPs guarantee that if instantiating the variables in the goal using the constants they provide the method does not find any solution, then no solution exists.

As proved in [3], the CQC Method always terminates when deductive rules have no recursion and either the goal is unsatisfiable or there is a finite consistent instance satisfying the goal.

SVT (Schema Validation Tool) [8] is a prototype tool designed to perform some validation tests on database schemas, in particular the liveliness test in which we are interested here. It accepts the following subset of the SQL language:

- Primary key, foreign key, boolean check constraints.
- SPJ views, negation, subselects (exists, in), union.
- Data types: integer, real, string.

The current implementation of SVT assumes a set semantics of views and queries and it does not allow null values neither aggregate nor arithmetic functions.

SVT implements the CQC Method as a backtracking algorithm. It adds facts to the EDB under construction in order to make true the literals in the goal. After adding a new fact, it checks if the EDB violates some constraint. When it detects that some constraint is violated or some literal in the goal is evaluated to false (e.g., a comparison), it backtracks and reconsiders the last decision. Some constraints, like foreign keys, can be repaired by adding new literals to the goal and thus no backtracking is required in these cases.
Using the CQC Method, and thus SVT, for checking the liveliness of a given predicate requires just providing the database schema and the goal. The goal will only be one literal corresponding to the predicate we want to check if it is lively.

4.2. Experiments

The first experiment, shown in Figure 4, is aimed at comparing the two approaches for computing explanations that we proposed on Section 3, the backward and the forward approaches, with the hitting set dualization approach proposed in [1]. We have used an implementation of the dualization approach that uses incremental hitting set calculation, as described in [1], and replacing the calls to the satisfiability method by calls to the CQC Method. We performed the experiment using a database schema formed by K chains of tables, each one with length N:

\[ R^1(B'_1), \ldots, R^K(B'_K) \]

Each table has two columns and two constraints: a foreign key from its second column to the first column of the next table, i.e. \( R_i, B'_i \) references \( R_{i+1}, A'_{i+1} \), and a check constraint requiring that the first column must be greater than the second, i.e. \( R_i, A'_i > R_i, B'_i \). Additionally, the first table of each chain has a check constraint stating that its first column must not be greater than 5, i.e. \( R_1, A'_1 \leq 5 \). The last table of each chain has another check constraint stating that its second column must not be lower than 100, i.e., \( R_K, B'_K \geq 100 \). This schema is designed to allow us to study the effect of varying the number and size of explanations. Note that the value of N determines the size of the explanations and that the value of K determines their number. When K is set to 1 we found explanations of size 3 and each increment in the value of N results in 2 additional constraints in each explanation. For the case of K, its value is exactly the number of explanations we found.

Note also that in this experiment all the explanations are disjoint. Each chain of tables in the schema provides one explanation, and all the chains are disjoint. That means, when we execute the phase 3 of the backward approach it will not provide any new explanation with respect to the first two phases.

We computed the explanations for the liveliness test of the following derived predicate P:

\[ P \leftarrow R^1(X^1_1, X^1_2) \land \ldots \land R^K(X^K_1, X^K_2). \]

The symbols \( X^i_1, X^i_2, \ldots, X^K_1, X^K_2 \) are different fresh variables. Due to the previous database schema definition, the liveliness test of P does not reach any solution, i.e., P is not lively in the previous described schema.

Figure 4 shows the running times for different values of N. More precisely, ranging N from 1 to 5. The value of K was set to 2. We executed both, the backward and forward approaches without using the filter described in Section 3.3. For the backward approach, we performed the three phases described in Section 3.2.

As seen in the graphic, the forward approach is considerably slower than the other two. This is an expected result since the forward approach is a naive approach and we executed it without the filter. We can also see that the dualization approach is quite much slower than our backward approach. It is worth noting, however, that the dualization approach [1] was proposed for the context of type error and circuit error diagnosis and that we are applying it now in a different context. While in [1] the authors use an incremental satisfiability method for Herbrand equations, in query liveliness there are no incremental methods to check it. Moreover, the dualization approach computes the explanations by means of the relationship existing between the minimal unsatisfiable sets (the explanations) of constraints and the maximal satisfiable sets of constraints. Thus, it finds first a maximal unsatisfiable set, makes its complement, accumulates this complement in a set, and then computes the hitting sets for this set of complements. The resulting hitting sets are the
candidates for being explanations. In a different way, the backward approach finds first a maximal set of disjoint explanations with a linear number of test executions and then focuses on finding other explanations taking into account that they must overlap with the ones already found. In this way, it can significantly reduce the number of candidates to attempt. Table 1 shows the number of calls to the CQC Method performed by each approach. Table 2 shows the detail of the running times.

Figure 5 focuses on the backward approach. It shows the behavior of this approach when there are 1, 2 and 3 disjoint explanations, and the size of each explanation increases. We used the same database schema than in the previous experiment and the same derived predicate \( P \). The graphic shows an increasing of running time when the number of explanations increases, which is higher when going from 2 to 3 explanations. This is expected since although phases 1 and 2 imply a linear number of test executions, phase 3 still requires an exponential number of them.

In Figure 6, we compare the backward approach with its three phases against the first two phases only. This time, we used a database schema similar to the one we used in the previous experiments but formed now by the following two chains:

\[
R^i(A^1,B^1), \ldots, R^i_N(A^1_N,B^1_N), R^i(A^1_B^1,B^1,C^1) \\
R^i(A^2,B^2), \ldots, R^i_N(A^2_N,B^2_N). 
\]

The integrity constraints are also similar than those in the previous schema but with two additions: a check constraint in \( R^i_N \) stating \( A^i_N \geq C^i_N \), and another check, also in \( R^i_N \), stating \( C^i_N \geq 200 \). The derived predicate \( P \) is now the following:

\[
P \leftarrow R^i(A,B) \land R^i(U,V). 
\]

In this schema, there will be three explanations for the liveliness test of \( P \). The first chain will provide two of them, which will overlap. These two explanations will share all its constraints except those in \( R^i_N \); one explanation will have the constraints: \( A^i_N \geq B^i_N \) and \( B^i_N \geq 100 \), and the other explanations the constraints: \( A^i_N \geq C^i_N \) and \( C^i_N \geq 200 \). The second chain will provide the third explanation. Therefore, phase 1 will find one of these three explanations, phase 2 will find an explanation disjoint with the previous, and finally the third phase will find the remaining one. This way, as long as each phase provides one explanation, we will be able to compare them.

The graphics in Figure 6 show a big increment of running time when we introduce the third phase. This is expected since, as we explained, the third phase requires selecting one constraint from each already found explanation and trying all possible combinations. It can also be seen that the graphic for the case of phases 1 and 2 only has also an exponential shape although they require just a linear number of test executions. This result is clearly due to the cost of each one of these test executions, as the exponential cost of the used method (in this case the CQC Method) cannot be avoided because of the complexity of the liveliness problem.

Tables 3 and 4 show the number of calls to the CQC Method and the running times, respectively, from Figure 6, and compare them with the dualization approach [1].
In Figure 7, we study the effect of the filter described in Section 3.3 in reducing the number of tests executions when applied in the backward approach. This time we used a database schema similar to the one from the first experiment (with K = 2), but with some additions. First, for each table \( R_i \), we added the following chain: \( R_i(A_1,B_1), \ldots, R_i(A_n,B_n) \). Each one of these new tables has the following constraints: \( R_i(A_1) > R_i(B_1) \) and \( R_i(A_n) > R_i(B_n) \). We also added an additional foreign key to each table \( R_i \) that references the first table of its corresponding chain, i.e., \( R_i(B_1) \) references \( R_i(A_1) \). Finally, we added \( M \) tables from the relational schema of the Mondial [6] database (out of 28 tables), and connected them with the tables \( R_i \) by means of foreign keys referencing \( R_i(A_1) \). We considered the Mondial database schema with its primary key, unique and foreign key constraints. As a difference from the previous schemas, this one allows us to study the effect of the filter in a scenario containing not only the constraints that form the explanations, but also containing additional constraints that do not affect the liveliness of the tested predicate, as usually happens with the major schemas.

The graphics in Figure 7 show the behavior of the backward approach, with and without filter, when increasing the number of constraints in the database schema. We used schemas with 30, 58, 95 and 139 constraints, respectively, which we got by changing the value of \( N \) and \( M \). It can be seen how using the filter reduces considerably the number of calls to the liveliness method. Table 5 shows the corresponding running times of this experiment.

5. RELATED WORK

The approach of our backward method, with its three phases, presents several similarities with the hitting set dualization approach [1], which was proposed for type error and circuit error diagnosis. As far as efficiency is concerned, we have shown in Section 4.2 that our backward method is more efficient than hitting set dualization in the context of query liveliness checking, by means of two experiments: one with two disjoint explanations and the other with three explanations (two of them overlapping), when varying the size of the explanations. Another significant difference is that our backward method provides three levels of explanation search: one explanation, a maximal set of non-overlapping explanations, and all explanations. It is worth noting the second level, which requires only a linear number of callings to the underlying method, and although it does not provide all explanations, the designer knows that the remaining explanations overlap with those already found.

In Description Logics (DL), the axiom pinpointing process described in [7] is similar to our definition of computing explanations. Nevertheless, it is restricted to unfoldable ALC TBoxes and relies on glass-box techniques. In [5], the authors do follow a black-box heuristic approach to trace the axioms that led the unsatisfiable root classes of a given ontology to be a subclass of two incompatible classes; however, as in [7], the approach presented in [5] is still too strongly related with the DL context to be applicable to the more expressive setting that we consider.

6. CONCLUSIONS AND FURTHER WORK

We have proposed a new method for computing explanations for unlively queries in databases, which is independent of the particular liveliness checking method used to perform the liveliness tests. We have shown that the backward method we proposed is more efficient than related approaches by means of an experimental comparison, which used our CQC Method [3] as liveliness checking method. In particular, we have compared our backward method with a forward method also proposed in this paper and with the best known approach for finding minimal unsatisfiable subsets of constraints, the hitting set dualization approach [1].

Moreover, we have shown that the backward approach provides three levels of search: find an explanation, find a maximal set of disjoint explanations, and find all explanations; and that we can find the maximal set of disjoint explanations with a linear number of calls to the underlying liveliness method. We have also proposed a filter to reduce the number of calls to the liveliness method by discarding those candidates containing constraints that are non-relevant for the liveliness test.

As future work, it would be very interesting to combine the backward approach with glass box techniques, that is, to use a liveliness method able to provide at least one explanation by its own. In this way, the phase 1 of the backward approach could be replaced by just returning the explanation provided by the underlying method (remind that phase 1 is also performed again in phases 2 and 3). Although existing liveliness methods do not provide explanations, we believe that the CQC Method [3] can be successfully improved in order to provide one explanation in such a way that we could also take advantage of these improvements to increase CQC Method’s performance.

7. REFERENCES