Computing Explanations for Unlively Queries in Databases

Guillem Rull[1,2] Carles Farré[2] Ernest Teniente[2] Toni Urpí[2]
Universitat Politècnica de Catalunya
Jordi Girona 1-3
08034 - Barcelona
phone: +34934137887
{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 which is independent of the particular method used to determine 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.

1. INTRODUCTION

A query is lively [2,4,8] in a database schema if it returns a non-empty answer for some database satisfying the schema. Clearly, queries which are not lively are meaningless since they will always have an empty extension. Unliveliness of a certain query may be due either to the query definition itself (which may, for instance, contain a contradiction) or to the integrity constraints of the schema that prevent the query to ever have any instance. In a similar way, liveliness applies also 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. Note that the empty set of constraints will also be an explanation when the query definition itself already contains a contradiction.

The explanations are intended to help the database designer to find the problem and to fix it. 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 designer only should focus on the constraints forming the explanation and on the tested query definition itself.

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. I.e., 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 to be more efficient, it provides three levels of explanation search. The first level is aimed at finding just one explanation. This is done by reducing the number of constraints in the schema until only the constraints forming the explanation remain. In the second level, our method finds the maximal set of non-overlapping explanations, which includes the one found in the previous phase. 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. The first two levels require only a linear number of calls to the underlying liveliness method, with respect to the number of constraints in the schema. The third level introduces an exponential number of such 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 lively.

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 error diagnosis. We also study the behavior of the backward approach when varying some parameters like the number and size

[1] *This work was supported in part by Microsoft Research through the European PhD Scholarship Programme.*
[2] *This work was partially supported by the Spanish Ministerio de Educación y Ciencia under project TIN2005-05406.*

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 thorough the paper.

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

$$p(\bar{X}) \leftarrow r_1(\bar{X}_1) \wedge \ldots \wedge r_n(\bar{X}_n) \wedge \neg r_{n+1}(\bar{Y}_1) \wedge \ldots \wedge \neg r_m(\bar{Y}_s) \wedge C_1 \wedge \ldots \wedge C_t$$

A *constraint* (or *condition*) has the denial form:

$$\leftarrow r_1(\bar{X}_1) \wedge \ldots \wedge r_n(\bar{X}_n) \wedge \neg r_{n+1}(\bar{Y}_1) \wedge \ldots \wedge \neg r_m(\bar{Y}_s) \wedge C_1 \wedge \ldots \wedge C_t.$$

The symbols p and r_1, ..., r_m are *predicates*. The tuples $\bar{X}$, $\bar{X}_1$, ..., $\bar{X}_n$, $\bar{Y}_1$, ..., $\bar{Y}_s$ contains *terms*, which are either variables or constants. Each C_i is a *built-in literal* in the form of $t_1\ \theta\ t_2$, where t_1 and t_2 are terms and operator θ is $<$, $\leq$, $>$, $\geq$, $=$ or $\neq$. The atom $p(\bar{X})$ is the *head* of the rule, and $r_1(\bar{X}_1)$, ..., $r_n(\bar{X}_n)$, $\neg r_{n+1}(\bar{Y}_1)$, ..., $\neg r_m(\bar{Y}_s)$ 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 $\bar{X}$, $\bar{Y}_1$, ..., $\bar{Y}_s$, C_1, ..., C_t must also appear in some $\bar{X}_i$. Note that we express constraints stating what may not happen instead of what should happen.

Predicates that appear in the head of a deductive rule are *derived predicates* also called *intensional database* (IDB) predicates. 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 $\leftarrow L_1 \wedge ... \wedge L_k$ if $(L_1 \wedge ... \wedge L_k)\sigma$ is true on $DR(D)$ for some ground substitution σ. 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., $A_Q(D) = \{q(\bar{a}) \mid q(\bar{a}) \in (Q \cup DR)(D)\}$.

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 check constraint defined over column *B* stating that *B* has to be lower than *10*. Table *T* has a check constraint that forces column *G* to be greater than *30*. Finally, there is a foreign key defined in table *R* over column *C* that references column *D* in table *T*. More formally, schema *S* is:

$$\leftarrow R(A,B,C) \wedge B \geq 10$$
$$\leftarrow T(D,G) \wedge G \leq 30$$
$$\leftarrow R(A,B,C) \wedge \neg fkRtoT(C)$$
$$fkRtoT(C) \leftarrow T(C,G)$$

Note the use of an auxiliary derived predicate to express the foreign key constraint. 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*:

$$Q(A) \leftarrow R(A,B,C) \wedge T(C,G) \wedge B > G$$

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, *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*:

$$E = \{\leftarrow R(A,B,C) \wedge B \geq 10, \leftarrow T(D,G) \wedge G \leq 30\}$$

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 foreign key constraint 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, *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 *isLively* with an increasing number of constraints -starting from the empty set of constraints; or *backward*, if *isLively* is

called 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, *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, *isLively*(Q, S' = (DR, $\varnothing$)), returned 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, *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 *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 *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.

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.

```
phase_1(Q: predicate, S = (DR, IC): schema): explanation
    U := IC          // set of "unchecked" constraints
    E := IC          // explanation
    while (∃c ∈ U)
        E := E – {c}
        if (isLively(Q, S' = (DR, E)))
            E := E ∪ {c}
        endif
        U := U – {c}
    endwhile
    return E
```

Figure 1: Phase 1 of the backward approach.

For the sake of example, let us assume that Q is a query defined as follows:

$$Q \leftarrow R(X,Y,Z) \wedge V(Z,A,B) \wedge T(Z,U,V) \wedge Y > 5 \wedge B < X \wedge V = 2$$

Let us also assume that S is a database schema with no deductive rules but containing the following constraints, labeled as *c1*, *c2*, *c3* and *c4*, repectively:

$\leftarrow T(X,Y,Z) \wedge Z = 2$ (*c1*)
$\leftarrow R(X,Y,Z) \wedge Y > X$ (*c2*)
$\leftarrow R(X,Y,Z) \wedge X > 5$ (*c3*)
$\leftarrow V(X,Y,Z) \wedge Z < 10$ (*c4*)

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

E1 = {*c1*}
E2 = {*c2*, *c3*}
E3 = {*c3*, *c4*}

Let us call phase_1(Q, S), with S = {*c1*, *c2*, *c3*, *c4*} to find one of these three explanations. If we assume that the constraints are considered in the order that they were listed above, *c1* is considered first. Since *isLively*(Q, {*c2*, *c3*, *c4*}) returns false, *c1* is discarded. Constraint *c2* is considered next. Since *isLively*(Q, {*c3*, *c4*}) returns false, *c2* is also discarded. Constraint *c3* is considered next. In this case, *isLively*(Q, {*c4*}) returns true. Therefore, *c3* is not discarded. Finally, constraint *c4* is considered. Since *isLively*(Q, {*c3*}) returns true, *c4* may not be discarded either. As a result, phase_1(Q, S) returns {*c3*, *c4*}, that is, explanation *E3*. Note that if the constraints had been considered in reverse order,

for instance, the returned explanation would have been another: {*c1*} = *E1*.

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.

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.

```
phase_2(Q: predicate, S = (DR, IC): schema,
        EP1: explanation): Set(explanation)
    SE := {EP1}   // set of explanations
    R := IC - EP1  // set of "remaining" constraints
    while (not isLively(Q, S' = (DR, R)))
        E := phase_1(Q, S' = (DR, R))
        SE := SE ∪ {E}
        R := R – E
    endwhile
    return SE
```

Figure 2: Phase 2 of the backward approach.

Continuing with the example that we introduced to illustrate Phase 1, recall that we found that {*c3*, *c4*} was an explanation for the fact that *isLively*(*Q*, {*c1*, *c2*, *c3*, *c+4*}) had returned false. According to Phase 2, we start now by calling *isLively*(*Q*, {*c1*, *c2*}). Since this call returns false too, it means that there is another explanation in {*c1*, *c2*}. Therefore, we call phase_1((*Q*, {*c1*, *c2*}), which returns {*c1*} as a new explanation. Next, we call *isLively*(*Q*, {*c2*}), which returns true and, thus, Phase 2 ends. The final output for this phase is {{*c3*, *c4*}, {*c1*}} as a set of disjoint explanations.

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.

```
phase_3(Q: predicate, S = (DR, IC): schema,
        SE: Set(explanation)): Set(explanation)
    AE := SE
    Combo := combinations(AE)
    while (∃C ∈ Combo)
        R := IC – C
        if (not isLively(Q, S' = (DR, R)))
            E := phase_1(Q, S' = (DR, R))
            NE := phase_2(Q, S' = (DR, R), E)
            AE := AE ∪ NE
            Combo := combinations(AE)
        endif
        Combo := Combo – {C}
    endwhile
    return AE

combinations(SE: Set(explanation)): Set(Set(constraint))
// returns all possible sets of constraints that can be obtained
by selecting one constraint from each explanation in SE.
```

Figure 3: Phase 3 of the backward approach.

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:

$\leftarrow R(X,Y,Z) \wedge \neg fkRtoS(Z)$
$\leftarrow R(X,Y,Z) \wedge Z < 5$
$\leftarrow S(X,Y) \wedge X \geq 5$
$\leftarrow T(X,Y,Z) \wedge Y < Z$

$fkRtoS(Z) \leftarrow S(Z,Y)$

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:

$\{\leftarrow R(X,Y,Z) \wedge Z < 5, \ \leftarrow S(X,Y) \wedge X \geq 5\}$

Taking into account that our candidate does not contain the foreign key 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, $\leftarrow S(X,Y) \wedge X \geq 5$, 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. If the candidate was make the test fail, then the same would happen after removing the non-relevant constraint and thus the candidate would be 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 $\leftarrow T(X,Y,Z) \wedge Y < Z$ 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: $\{\leftarrow R(X,Y,Z) \wedge Z < 5, \leftarrow S(X,Y) \wedge X \geq 5\}$. Applying step 2, we would recompute the relevant predicates. Now, as we are not considering the foreign key, the predicate S is not relevant and, thus, neither the constraint $\leftarrow S(X,Y) \wedge X \geq 5$. Applying step 3, we would discard the candidate without performing the liveliness test, and we would move to the next candidate.

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 $\leftarrow T(X,Y,Z) \wedge Y < Z$ 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 foreign key constraint. That would leave two constraints in the schema: $\leftarrow R(X,Y,Z) \wedge Z < 5$ and $\leftarrow S(X,Y) \wedge X \geq 5$, 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: $\leftarrow R(X,Y,Z) \wedge Z < 5$. Because the predicate becomes lively, we should put back the two removed constraints (the foreign key and the one about *S*). The phase 1 would remove then the next constraint: $\leftarrow R(X,Y,Z) \wedge Z < 5$, 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 is reformulated as a rule defining a derived predicate IC_i in such a way that the constraint is violated when its corresponding predicate IC_i has a fact in the database.

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

$$Q(\bar{X}) \leftarrow P^1_1(\bar{X}_1) \wedge ... \wedge P^1_{s1}(\bar{X}_{s1}) \wedge C^1_1 \wedge ... \wedge C^1_{r1} \wedge \neg S^1_1(\bar{X}_1) \wedge ... \wedge \neg S^1_{m1}(\bar{X}_{m1})$$
$$...$$
$$Q(\bar{X}) \leftarrow P^k_1(\bar{X}_1) \wedge ... \wedge P^k_{sk}(\bar{X}_{sk}) \wedge C^k_1 \wedge ... \wedge C^k_{rk} \wedge \neg S^k_1(\bar{X}_1) \wedge ... \wedge \neg S^k_{mk}(\bar{X}_{mk})$$

The symbols $P^1_1, ..., P^1_{s1}, S^1_1, ..., S^1_{m1}, ..., P^k_1, ..., P^k_{sk}, S^k_1, ..., S^k_{mk}$ are predicates and $C^1_1, ..., C^1_{r1}, ..., C^k_1, ..., C^k_{rk}$ are built-in literals. We will define *neg_preds*(Q) as the predicates of those negative literals that appear in the definition of Q, taking into account all possible unfoldings. Formally:

$$neg\_preds(Q) = \{\{S^j_i \mid 1 \le i \le m_j\} \mid 1 \le j \le k\} \cup \{\{neg\_preds(P^j_i) \mid 1 \le i \le s_j\} \mid 1 \le j \le k\}$$
$$neg\_preds(R) = \emptyset \quad \text{if } R \text{ is a base predicate}$$

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-relevancy* and *q-relevancy*. 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^j_i with $1 \le i \le s_j$ and $1 \le j \le k$, are also p-relevant predicates.
- If Q is a derived predicate and $P^j_1, ..., P^j_{sj}$ are p-relevant or q-relevant, for some $1 \le j \le k$, 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 $P^j_1, ..., P^j_{sj}$ are p-relevant or q-relevant predicates, for some $1 \le j \le k$, then $S^j_1, ..., S^j_{mj}$ and the predicates in $neg\_preds(P^j_1) \cup ... \cup neg\_preds(P^j_{sj})$ are p-relevant.
- If $IC_i \leftarrow P_1(\bar{X}_1) \wedge ... \wedge P_s(\bar{X}_s) \wedge C_1 \wedge ... \wedge C_r \wedge \neg S_1(\bar{X}_1) \wedge ... \wedge \neg S_m(\bar{X}_m)$ is an integrity constraint and $P_1, ..., P_s$ are p-relevant or q-relevant predicates, then IC_i is q-relevant and the predicates in $neg\_preds(IC_i)$ are p-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 third point of the definition is applicable.

DEFINITION 3.3. We will say that an *integrity constraint* $IC_i \leftarrow L_1 \wedge ... \wedge L_n$ *is 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:

$V(X,Y) \leftarrow R(X,A,B) \wedge S(B,C,Y) \wedge \neg W(A,C)$
$W(X,Y) \leftarrow P(X,Y) \wedge Y > 100$
$P(X,Y) \leftarrow T(X,Y) \wedge \neg H(X)$
$Q(X) \leftarrow S(X,Y,Z) \wedge Y \ge 5 \wedge Y \le 10$
$IC_1 \leftarrow R(X,Y,Z) \wedge \neg T(Y,Z)$
$IC_2 \leftarrow F(X,Y) \wedge X \le 0$

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 star with *p-relevant* = ∅ and *q-relevant* = ∅
(2) The first point in the definition of predicate's relevancy says us that, as we are testing the liveliness of *V*, *V* is a p-relevant predicate.
(3) Then, *p-relevant* = {**V**} and *q-relevant* = ∅
(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* = ∅
(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 p-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 *H* 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 *V* 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_i be a constraint from the database schema. If IC_i is not relevant for the liveliness test of *P*, then *P* is still non-lively after removing IC_i from the schema.

PROOF. Let us assume that after removing IC_i from the schema *P* becomes lively. 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 lively after removing IC_i, database D should violate IC_i. Our goal now is to show that it follows that IC_i 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 thus 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_i is true in D, we can conclude that IC_i 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 [7]. 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 there is a finite consistent instance satisfying the goal, or when the goal is unsatisfiable.

SVT (Schema Validation Tool) [7] 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], but 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:

$$R^1_1(A^1_1,B^1_1),\ \ ...,\ R^1_N(A^1_N,B^1_N)$$
$$...$$
$$R^K_1(A^K_1,B^K_1),\ ...,\ R^K_N(A^K_N,B^K_N)$$

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. $R^j_i.B^j_i$ references $R^j_{i+1}.A^j_{i+1}$, and a check constraint requiring that the first column must be greater that the second, i.e. $R^j_i.A^j_i > R^j_i.B^j_i$. Additionally, the first table of each chain has a check constraint stating that its first column must not be greater than *5*, i.e. $R^j_1.A^j_1 \le 5$. The last table of each chain has another check constraint stating that its second column must not be lower than *100*, i.e., $R^j_N.B^j_N \ge 100$. 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 N 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

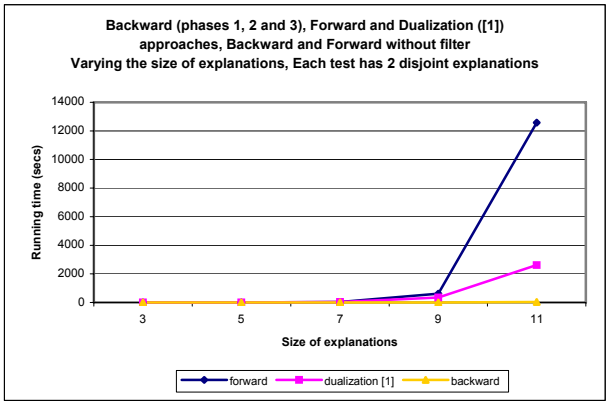

Figure 4: Comparison of the backward, forward and dualization [1] approaches.

Table 1: Number of calls to the CQC Method in Figure 4

Size of explanations	Forward (no filter)	Dualization	Backward (no filter)
3	51	46	19
5	963	161	41
7	16131	400	71
9	261123	734	109
11	4190211	1290	155

Table 2: Running times (secs) in Figure 4

Size of explanations	Forward (no filter)	Dualization	Backward (no filter)
3	0.05	0.28	0.05
5	1.08	4.31	0.22
7	27.89	38.87	1.14
9	611.31	346.11	5.34
11	12587.67	2606.53	29.59

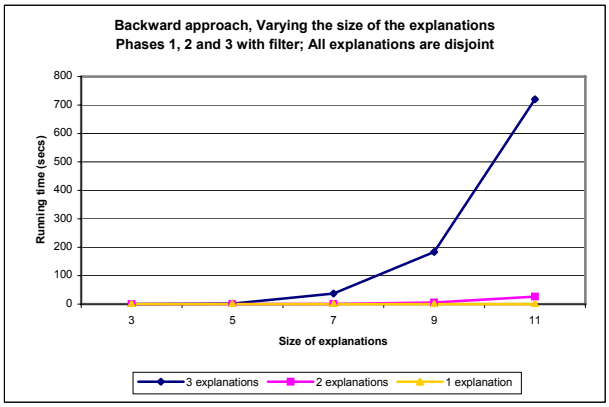

Figure 5: Effect of the number of explanations in the backward approach.

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:

$$P \leftarrow R^1_1(X^1_1,X^1_2) \wedge ... \wedge R^K_1(X^K_1,X^K_2)$$

The symbols X^1_1, X^1_2, ..., 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:

$$R^1_1(A^1_1,B^1_1), ..., R^1_{N-1}(A^1_{N-1},B^1_{N-1}), R^1_N(A^1_N,B^1_N,C^1_N)$$
$$R^2_1(A^2_1,B^2_1), ..., R^2_N(A^2_N,B^2_N)$$

The integrity constraints are also similar than those in the previous schema but with two additions: a check constraint in R^1_N stating $A^1_N \geq C^1_N$, and another check, also in R^1_N, stating $C^1_N \geq 200$. The derived predicate P is now the following:

$$P \leftarrow R^1_1(X,Y) \wedge R^2_1(U,V)$$

In this schema, there will be three explanations for the liveliness test of P. The first chain will provide two of them, which will

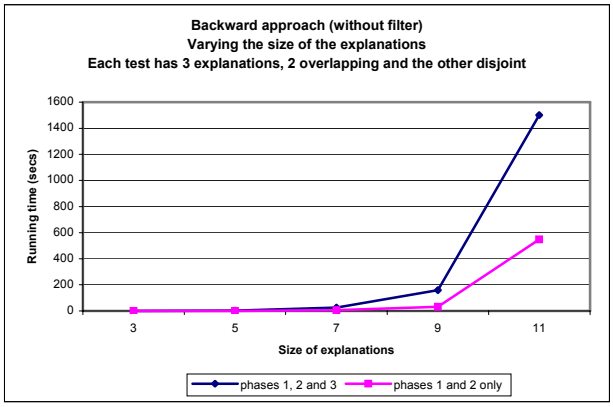

Figure 6: Comparison of the three phases of the backward approach and the first two phases only.

Table 3: Number of calls to the CQC Method in Figure 6 comparing with dualization [1] approach

Size of explanations	Dualization	Backward phases 1,2 and 3 (no filter)	Backward phases 1 and 2 (no filter)
3	98	44	15
5	270	98	21
7	605	176	27
9	1089	278	33
11	1726	404	39

Table 4: Running times (secs) in Figure 6 comparing with dualization [1] approach

Size of explanations	Dualization	Backward phases 1,2 and 3 (no filter)	Backward phases 1 and 2 (no filter)
3	2.05	0.53	0.39
5	37.27	2.86	0.55
7	327.88	26.06	6.72
9	2834.48	158.31	31.73
11	22968.95	1501.47	549.02

overlap. These two explanations will share all its constraints except those in R^1_N; one explanation will have the constraints: $A^1_N \geq B^1_N$ and $B^1_N \geq 100$, and the other explanations the constraints: $A^1_N \geq C^1_N$ and $C^1_N \geq 200$. 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. The exponential cost of the used method (in this case, the CQC Method) cannot be avoided because of the complexity of the liveliness problem.

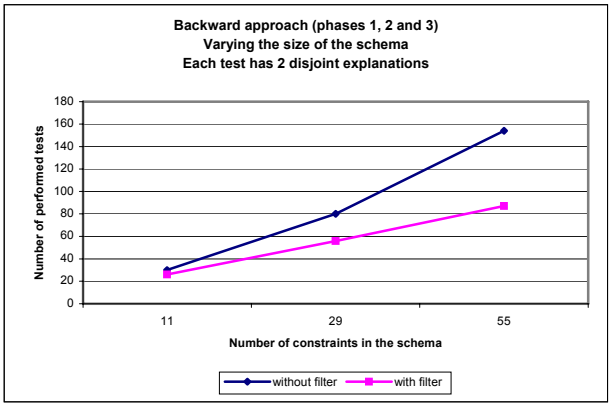

Figure 7: Effect of the filter described in Section 3.3 in the number of test executions.

Table 5: Running times (secs) from Figure 7

Number of constraints	Backward without filter	Backward with filter
11	0.17	0.16
29	2.25	0.81
55	294.98	108.64

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 but with some additions. First, we added a new chain of tables:

$$S_1(A_1,B_1), \ldots, S_N(A_N,B_N)$$

Each one of these tables has a check constraint, $S_i.A_i > S_i.B_i$, and a foreign key to next table in the chain, $S_i.B_i$ references $S_{i+1}.A_{i+1}$. Then, we also added the following chain for each table R^j_i:

$$R^j_{i,1}(A_1,B_1), \ldots, R^j_{i,N}(A_N,B_N)$$

Each one of these new tables has also the following constraints: $R^j_{i,s}.A_s > R^j_{i,s}.B_s$ and $R^j_{i,s}.B_s$ references $R^j_{i,s+1}.B_{s+1}$. We also add an additional foreign key to each table R^j_i that references the first table of its corresponding chain, i.e, $R^j_i.B^j_i$ references $R^j_{i,s}.A_s$. 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 phases 1 and 2 of the backward approach with and without filter, when increasing the number of constraints in the database schema. It can be seen how using the filter reduces considerably the number of executions of the liveliness test. 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 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 for the non-liveliness of a certain query. 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. These non-overlapping explanations seem to be the most relevant ones from a methodological perspective since at least all of them must be fixed in order to make the query lively.

In Description Logics (DL), the axiom pinpointing process described in [6] is similar to our definition of the computation of explanations. Nevertheless, the proposed techniques for pinpointing are strongly related with the DL context, restricted to unfoldable ALC TBoxes, and they rely also on *glass box* techniques. In [5], the authors explore *black box* techniques in order to debug unsatisfiable classes in DL. This work is related to ours in the sense that their techniques are also independent of the concrete underlying satisfiability checking service, but the authors focus mainly on the detection of dependencies between classes. They also present a black box heuristic approach to trace the axioms that lead the unsatisfiable root classes to be a subclass of two incompatible classes. As a difference from our work, both techniques in [5] are strongly related with the DL context and, moreover, the later requires a reasoner that provides the two incompatible classes.

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

[1] James Bailey, Peter J. Stuckey: Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization. *Practical Aspects of Declarative Languages (PADL)* 2005: 174-186

[2] Hendrik Decker, Ernest Teniente, Toni Urpí: How to Tackle Schema Validation by View Updating. *International Conference on Extending Database Technology (EDBT)* 1996: 535-549

[3] Carles Farré, Ernest Teniente, Toni Urpí: Checking query containment with the CQC method. *Data Knowl. Eng.* 53(2): 163-223 (2005)

[4] Alon Y. Halevy, Inderpal Singh Mumick, Yehoshua Sagiv, Oded Shmueli: Static analysis in datalog extensions. *J. ACM* 48(5): 971-1012 (2001)

[5] Aditya Kalyanpur, Bijan Parsia, Evren Sirin: Black Box Techniques for Debugging Unsatisfiable Concepts. *Description Logics* 2005

[6] Stefan Schlobach, Ronald Cornet: Non-Standard Reasoning Services for the Debugging of Description Logic Terminologies. *International Joint Conference on Artificial Intelligence (IJCAI)* 2003: 355-362

[7] Ernest Teniente, Carles Farré, Toni Urpí, Carlos Beltrán, David Gañán: SVT: Schema Validation Tool for Microsoft SQL-Server. *VLDB* 2004: 1349-1352

[8] Xubo Zhang, Z. Meral Özsoyoglu: Implication and Referential Constraints: A New Formal Reasoning. IEEE *Trans. Knowl. Data Eng.* 9(6): 894-910 (1997)