Query Relaxation across Heterogeneous Data Sources

Verena Kantere
University of Geneva
Switzerland
verena.kantere@unige.ch

George Orfanoudakis
National Technical University
of Athens, Greece
gorfas@ece.ntua.gr

Anastasios
Kementsietsidis
Google Inc.
akement@google.com

Timos Sellis
RMIT University, Australia
timos.sellis@rmit.edu.au

ABSTRACT

The fundamental assumption for query rewriting in heterogeneous environments is that the mappings used for the rewriting are *complete*, i.e., every relation and attribute mentioned in the query is associated, through mappings, to relations and attributes in the schema of the source that the query is rewritten. In reality, it is rarely the case that such complete sets of mappings exist between sources, and the presence of partial mappings is the norm rather than the exception. So, practically, existing query answering algorithms fail to generate any rewriting in the majority of cases. The question is then whether we can somehow relax queries that cannot be rewritten as such (due to insufficient mappings), and whether we can identify the interesting query relaxations, given the mappings at hand.

In this paper, we propose a technique to compute query relaxations of an input query that can be rewritten and evaluated in an environment of collaborating autonomous and heterogeneous data sources. We extend traditional techniques for query rewriting, and we propose both an exhaustive and an optimized heuristic algorithm to compute and evaluate these relaxations. Our technique works with input of any query similarity measure. The experimental study proves the effectiveness and efficiency of our technique.

Categories and Subject Descriptors

H.2.5 [Database Management]: Heterogeneous Databases

Keywords

big heterogeneous data; approximate querying; query relaxation

1. INTRODUCTION

In typical data exchange [9], integration [13], and sharing [24] settings, schema mappings (often in the form of Local-As-View (LAV) mappings [13]) are commonly used to alleviate the heterogeneity across *pairs* of autonomous sources (or sets of sources in general). In the case of data exchange, such mappings are used for instance generation, i.e., to generate instances of a *target* schema based on those of the *source* schema. In data integration and shar-

VacPackage (vacid, fno, hid, from, until)
Flights (fno, depart, return, carrier, origin, dest, price)
Hotels (hid, chkin, chkout, name, location, served, rate)

(a) The iTravel.com schema

SELECT *

FROM VacPackage V, Flights F, Hotels H

WHERE V.fno = F.fno AND V.from = F.depart AND V.until = F.return AND
F.depart = "11/01" AND F.return = "11/09" AND F.origin = "NYC" AND
F.depart = H.chkin AND F.return = H.chkout AND F.dest = H.served AND
V.hid = H.hid AND H.location = "Hana"

(b) Planning a vacation to Hana, Hawaii

iAirline.com: Segment (fno, depart, return, origin, dest, price)
iHotel.com: Locations (location, served, rate)
ilnn.com: Inn (name, chkin, chkout, served, rate)

(c) Related travel sources

Segment (fno, dep, ret, org, dst, pr) :- Flights (fno, dep, ret, car, org, dst, pr)
Locations (loc, srv, pr) :- Hotels (hid, chkin, chkout, nm, loc, srv, pr)
Inn (nm, ckin, ckout, srv, pr) :- Hotels (hid, ckin, ckout, nm, loc, srv, pr)

(d) Local-as-View Mappings

Figure 1: Querying the iTravel.com

ing, the mappings are used during query evaluation, i.e., to *translate* or *rewrite* queries that are posed over a source schema to new queries that can be evaluated over the constructs (schemas and attributes) of a target schema.

Here, we focus on the query rewriting problem and on the techniques that have been developed to this end. Classical rewriting algorithms, like Inverse Rules [8], Bucket [15] and MiniCon [23] have already addressed some of the fundamental issues in this area, both in terms of computing complete or maximally-contained rewritings (given an input query and a set of mappings), and in terms of doing so efficiently. Yet, a common assumption in all these algorithms is that the set of available mappings should provide at least one combination that *fully* rewrites the query. If this is not the case, that is, if the input query refers to a relation or to an attribute that does not participate in any of the input mappings, then the rewriting *fails* (no output is produced). In what follows, we argue that in many practical settings, the mappings are inherently incomplete and that the assumption of existing algorithms is rather restrictive.

Consider a travel web-site iTravel.com offering services similar to the ones found in web-sites like Expedia or Travelocity. Like these web-sites, a user can go to iTravel.com and enter (a) the origin; (b) the destination; and (c) the dates of the vacation; and the system can generate candidate vacation packages that include the flight(s) and hotel room(s) for the duration of the vacation. Conceptually, one can think of this as a single query over iTravel.com database, as illustrated in Figure 1(a). The database includes three relations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

CIKM'15, October 19–23, 2015, Melbourne, Australia.

© 2015 ACM. ISBN 978-1-4503-3794-6/15/10 ...\$15.00.

DOI: <http://dx.doi.org/10.1145/2806416.2806529>.

(with key attributes underlined in each). Relation *VacPackage* stores for each vacation package its id (*vacid*), the associated flight (*fno*) and hotel (*hid*), as well as the dates the package is available (from and until). Relation *Flights* includes a flight number (*fno*), the depart and return dates, as well as the flight carrier, origin, destination and ticket price. Relation *Hotels* includes the identifier (*hid*), hotel name, location and closest airport (*served*), as well as the check-in and check-out dates (*chkin* and *chkout*) and room rate.

Consider a user looking for vacation packages between the 1st and 9th of November, flying out of New York to the city of Hana, Hawaii. Figure 1(b) shows the equivalent SQL query over the schema of *iTravel.com*. The query joins the three relations on the *fno* and *hid* attributes, as well as on the corresponding date attributes (to ensure that the right flight and hotel are selected for the vacation period). Notice that the flight and hotel relations are also joined (term *F.dest = H.served*) to ensure that the flight lands into an airport that serves the hotel. Web-sites like *iTravel.com* usually act as brokers. They do not store the latest information for flight and hotel prices (or availability). Instead, they often access the airline and hotel databases (at query-time) to get the latest data. Figures 1(c) and (d) show three such databases along with their corresponding mappings to the *iTravel.com* schema (expressed as LAV mappings).

Notice that the user query in Figure 1(b) *cannot* be rewritten using the above mappings. For one thing, there is no mapping for *VacPackage* since none of the individual sources stores information for vacation packages. Also, identifiers like *vacid* and *hid* are internal to the *iTravel.com* database and cannot be mapped to any of the attributes of the individual sources. As well, the join on these identifiers is also not *covered* by any combination of mappings. Any of the classical algorithms on query rewriting will fail to provide a rewriting. Yet, it is not hard to see that the individual sources provide us with enough information to *reconstruct* vacation packages since the *important* attributes, like the dates, origin and destination of flights, and the check-in and check-out dates of hotels, are present in the sources. Indeed, it is not hard to see that the following two queries can be computed by *relaxing* (removing the join with *VacPackage*) the query in Figure 1(b):

```
SELECT *
FROM Segment S, Locations L
WHERE S.depart = "11/01" AND S.return = "11/09" AND S.origin = "NYC"
AND S.dest = L.served AND L.location = "Hana"

SELECT *
FROM Segment S, Inn I
WHERE S.depart = "11/01" AND S.return = "11/09" AND S.origin = "NYC"
AND S.depart = I.chkin AND S.return = I.chkout
AND S.dest = I.served
```

The first query identifies flight/hotel combinations where the user gets to stay in a hotel, while in the second the user gets to stay in an inn. Given an input query and a set of mappings there are *multiple* possible relaxations and not all relaxations are equally *good*.

- How can we efficiently compute the set of possible relaxations of an input query, given a set of mappings?
- How can we examine the *best* query relaxations first?

Detailed contributions. In response to the above challenges, we propose a technique for approximate query answering in an environment of autonomous data sources. To the best of our knowledge, this is the first work towards this direction.

- We build a new technique that performs query relaxation and approximate rewriting of the originally posed query. This technique complements and reuses existing traditional query answering algorithms using LAV mappings.
- The search space considered by the approximate rewriting technique is represented such that: (i) it is smaller than the number of

possible approximate rewritings (ii) it increases at a slow rate with respect to the query size, and (iii) it accepts heuristics.

- An exhaustive algorithm is presented for approximate query rewriting that searches the space in an efficient way, such that, the *best* possible approximations are considered first. It is proved that the algorithm is complete, i.e. it finds the optimal approximation. Furthermore, an optimized heuristic algorithm is proposed, that trades a much higher efficiency for a very low approximation error. The algorithms take as input a user-defined query similarity function.
- The algorithms are enhanced with a preprocess that unifies LAV mappings in more generalized ones that can rewrite query parts which could not be rewritten otherwise.
- The experimental evaluation proves the substantial efficiency and scalability of the proposed algorithms compared to a straightforward solution that employs traditional query rewriting and their effectiveness on the solution of the query relaxation and, in general, the approximate query answering problem.

We note here that the problem of defining the notion of query similarity and defining query similarity measures is orthogonal to the problem of finding approximate query versions with respect to such measures, and is out of the scope of this work. Query similarity and respective measures are investigated and proposed in [3, 6, 11, 12] and other works. The algorithms proposed here work with input of any similarity measure as a ‘black box’.

In the rest of the paper, Section 2 summarizes related work and Section 3 gives an overview and introduces preliminary notions. Section 4 discusses the cases of query approximation and Section 5 describes the approximate query answering algorithms. Section 6 presents our experimental study and Section 7 concludes the paper.

2. RELATED WORK

Approximation in query answering has been extensively studied from several aspects. A big trend has been the effort to reduce response time by seeking approximate query answers. Some techniques compute approximate answers based on data summaries, statistics or histograms [22]. A noticeable work is [21], which discusses summarization of XML data. The AQAX system [1, 25] aims to fast generation of approximate results, in order to improve efficiency. Towards the same end, the work in [7] introduces approximate data exchange. The aim is to produce fast approximate answers and exact answers later. The motivation is on trees that are semi-consistent. For this kind of problems it offers a solution that relaxes in a restricted way the data request.

Another trend is query answering in inconsistent databases. The work in [10] is a solution that provides consistent query answers on inconsistent databases with respect to some integrity constraints. Many other works focus on query answering in incomplete databases. The work in [27] presents the QPIAD framework, which focuses on the retrieval of tuples that miss values on constrained attributes. Their approach is based on attribute correlations. The work in [20] deals with answering imprecise queries, with respect to the schema on which they are posed. The focus is on the retrieval of not only the precise answers, but also imprecise ones, with respect to some similarity of the imprecise answers to the precise query. The technique reasons on the data and relaxes the query according to estimation of the similarity between values binding categorical attributes. In the same spirit, but focusing on vocabulary and semantics, the work in [4] deals with the problem of approximate query mapping. The interest is on Boolean query constraints in heterogeneous sources and there is a discussion on closeness and its quantification with respect to precision and recall.

Furthermore, there is some interest on the problem of query relaxation. The work in [5] studies the problem of approximation of

Figure 2: Approximate query rewriting

Datalog queries with a union of conjunctive ones. It calls the similarity of approximations "easiness" or "goodness" and proposes the notion of upper and lower envelopes that characterize the containment of a conjunctive query in a Datalog one. The work in [19] focuses on queries that failed to get an answer and discusses the notion of similar queries. It proposes a query relaxation method for producing query versions that succeed in retrieving similar tuples. This approach is based on training the system beforehand on part of the data. The interest is on queries with conjunctions or disjunctions of atoms and the relaxation procedure is based on bayesian networks, targeting the approximation of domain knowledge.

In spite of the numerous works in approximate query answering, the problem of producing query approximations that can be rewritten on autonomous and heterogeneous databases with respect to a set of mappings has not been tackled. Our work aims to an efficient and practical solution of this problem, that can be employed in several application settings of autonomous databases.

3. OVERVIEW AND PRELIMINARIES

A classical query rewriting algorithm takes as input a query and a set of views, and outputs a rewritten query version (usually the maximally contained version [13]). If the available views are not adequate to form a combination that rewrites the query completely, then no output is produced. Our aim is to offer such a rewriting technique that outputs a rewritten query employing in the *best* way the available views, even if a complete rewriting is not possible.

This work presents a query rewriting technique that produces approximate rewritings or equivalently, rewritings of approximations, of an originally posed query employing a set of Local-As-View (LAV [13, 14]) views, $\mathcal{V}$. Hereafter we refer to the query input to the approximate rewriting algorithm as Q_{orig} , to an approximation of it as Q_{approx} and to a rewritten version of the approximation as Q_{approx_R} . We are interested in *SQL* queries of the *SPJ* form (plus comparisons and value conditions), which can be written in conjunctive form. Whenever not clear from the context we note as Q^{SQL} and as Q^{conj} the *SQL* and conjunctive form, respectively, of Q . Roughly, a user poses Q_{orig} in the local database which should be approximated to Q_{approx} so that it can be rewritten as Q_{approx_R} and answered on a remote database through a set of mappings (in general, chain of mappings [17]).

We define the search space of possible approximations through the removal of distinct query parts. This is a smaller search space than the one that considers arbitrary approximate versions of a query. Furthermore, the size of the selected space not only increases slower than the size of the space of arbitrary approximations, but it also lends itself nicely to efficient pruning heuristics.

VacPackage (vacid, fno, hid, from, until)
 Flights (fno, depart, return, carrier, origin, dest, price)
 HotelsCheck (hid, chkin, chkout, name, location, rate)
 Hotels (name, room#, rate, lowestprice)
 Rooms (room#, floor#)
 HotelLocation (name, lowestprice, location)
 Airport (aid, location, distance)
 SeasonPrices (season, lowestprice)

Figure 3: The extended iTravel.com schema

We develop an exhaustive algorithm that operates on this search space. The input of the algorithm is Q_{orig} , a set of available views $\mathcal{V}$ and a query similarity function M_{sim} , used as a black box. The latter is formed by the user and quantifies the quality of the possible approximate rewritings. Naturally, the exhaustive algorithm finds the optimal approximation with respect to the similarity function. Furthermore, we propose a heuristic that proves to prune the space efficiently, with a small trade of approximation error.

Users are assumed to express their queries and expect answers in *SQL* form, since it is more intuitive and user-friendly. Existing query rewriting algorithms, however, take as input queries and output rewritten versions of them in conjunctive form. A *SQL* query is considered to be a set of *elements*, one for each 'select' attribute and one for each 'where' condition. The transformation of Q_{orig}^{SQL} to Q_{orig}^{conj} is unique; moreover, the transformation of $Q_{approx_R}^{conj}$ to $Q_{approx_R}^{SQL}$ is unique with respect to the elements of Q_{orig}^{SQL} .

A conjunctive query is of the form: $Q(X) : -R_1(X_1) \dots R_n(X_n)$, where Q and R_i , $i = 1, \dots, n$ are names of predicates (i.e. relations) and X_i , $i = 1, \dots, n$ are tuples (i.e. each is an ordered set of attribute variables or constants). $Q(X)$ is the *head* and $\cup_{i=1, \dots, n} R_i$ the *body* of the query. The R_i s, $i = 1, \dots, n$ are called the *subgoals* of the query Q . The variables in X are called *distinguished* and the variables in R_i s that do not appear in the query head are called *existential*. Conjunctive queries can have comparisons as subgoals; if $R_i(X_i)$ is a comparison subgoal, any variables in X_i must also appear in X . A query is *safe* if all variables in X appear in $\cup_{i=1, \dots, n} X_i$. A mapping or as called a *homomorphism* of a query $Q(X) : -R_1(X_1) \dots R_n(X_n)$ on a view $V(Y) : -S_1(Y_1) \dots S_m(Y_m)$ is the mapping of query predicates $R_1, \dots, R_n$ on the view predicates $S_1, \dots, S_m$ and variables $X, X_1, \dots, X_n$ on variables $Y, Y_1, \dots, Y_m$.

The proposed approximate rewriting technique takes advantage of the existing literature on classical query rewriting using LAV views [8, 15, 16, 18, 23] in order to perform the actual rewriting. Specifically, when the algorithm finds an approximation, Q_{approx} , of the original query, which can be thoroughly rewritten using the set of views in hand $\mathcal{V}$, it rewrites it to Q_{approx_R} in the classical manner. Figure 2 is a coarse summary of the technique. Classical query rewriting algorithms [8, 15, 16, 18, 23] rewrite the query to a query in which the subgoals are no more R_i s, but the head of views $V_i \in \mathcal{V}$ or they are new comparison subgoals. In general, the algorithms start from mapping some of the head variables X of Q on some of the head variables Y of V , and proceed by mapping a subset of subgoals R_i s to a subset of subgoals S_i s progressively, such that the whole query body is finally mapped on (or as said *covered by*) one or more views. If there are subgoals that are covered by more than one views, then the check for query containment is performed in order to see if the rewritten query can be reduced to a version the answers of which are contained in the answers of the original query. Figure 3 shows an extended version of the iTravel.com schema, used in the examples throughout the rest of the paper.

Assume that Q_{orig}^{SQL} is:

```

 $Q_{orig}^{SQL}$ : SELECT H. name, H. room#
FROM   Hotels H, HotelLocation HL
WHERE  H.rate = "300" AND H.name = HL.name
AND HL.location = "sea"

```

The conjunctive query form Q_{orig}^{conj} is:

```

 $Q_{orig}^{conj}$ (name, room#) :- Hotels(name, room#, '300', lowestprice1) Hotel-
Location(name, lowestprice2, 'sea')

```

Assume a view:

```

V(name, rate) :- Hotels(name, room#, rate, lowestprice)

```

A rewriting algorithm creates the following partial query mapping on the view (starting from the head and moving on to the body):

```

 $Q_{orig}^{conj}$ .name  $\rightarrow$  V.name
 $Q_{orig}^{conj}$ .Hotels  $\rightarrow$  V.Hotels
 $Q_{orig}^{conj}$ .Hotels.room#  $\rightarrow$  V.Hotels.room#
 $Q_{orig}^{conj}$ .Hotels.'300'  $\rightarrow$  V.Hotels.rate
 $Q_{orig}^{conj}$ .Hotels.lowestprice1  $\rightarrow$  V.Hotels.lowestprice

```

Namely, three very well know classical rewriting algorithms, Inverse Rules [8], Bucket [15] and MiniCon [23] work as follows: Inverse Rules inverts the views such that the subgoals of the views become the heads of rules the subgoal of which is the view head; the inversed rules are matched with the query subgoals to find out a set of rules that covers all of them. The Bucket Algorithm [15] creates a bucket for each subgoal and puts in each bucket the views that can cover this subgoal. It then tries to find a minimal combination of views (one from each bucket). The MiniCon algorithm [23] works similarly to the Bucket algorithm, but instead of creating a bucket for each subgoal and then trying to combine views from within the buckets, it matches each view with as many as possible query subgoals following join chains. When no more subgoals can be matched, the algorithm tries to match another view with as many as possible from the rest such that the two views can be joined in the rewritten query body. This procedure is performed until all subgoals are covered by a set of views.

Overall, all three algorithms have two coarse steps:

1. Map the query on the view in the least restrictive manner (in terms of homomorphism).
2. Combine view mappings produced in the previous step so that all query subgoals are covered in a non-redundant manner.

In the following we refer to query rewriting employing an algorithm based on query containment, such as the above, as *classical* or *complete* query rewriting. Also, we use interchangeably: the terms *rewriting* and *answering* (since answering is performed through rewriting); the terms of query element *mapping*, *covering* and *satisfaction*; query *conditions* and *constraints* (usually the first term is used for SQL and the second for conjunctive forms), *relaxation* and *approximation* (the latter is a broader term than the first and covers cases of query versions that may not only be relaxed but have additional elements - the proposed technique works in case of such query approximation). A complete mapping of query variables on view variables is denoted with $\phi(\cdot)$.

4. APPROXIMATE QUERY ANSWERING

Our approximate rewriting technique searches the optimal approximation based on the removed query elements from Q_{orig} . The algorithm is based on a structure called F (from 'failure') that records the query rewriting procedure at the point of failure, indicating which query part should be removed in order to create an approximation of the Q_{orig} , Q_{approx} , and try to rewrite the latter with possible success. Since the traditional query rewriting algorithms proceed in two coarse steps (described informally in Section

3), failure of rewriting, and, therefore, creation of a new instance of F , can occur in any one of the two steps.

4.1 Creation of F s

With respect to a view $V \in \mathcal{V}$ and the query Q , an instance of F records the failure of mapping V on Q . The latter can either be the originally posed query Q_{orig} , or an approximate version of it. The definition of F is:

Definition 1 A failure F for a view V and a query Q is the set $F(Q, V) = (h_P, \phi_P, G_P, D)$ where:

- h_P is a head homomorphism on V
- ϕ_P is a partial mapping from the subgoal variables of Q to the subgoal variables of $h_P(V)$
- G_P is the subset of the subgoals in Q which are covered by some subgoal in $h_P(V)$ using the mapping ϕ_P .
- $D = \{x\}$, where x is a (a) join, (b) comparison, (c) constant, (d) variable or (e) subgoal of Q .

An instance of F for a query Q and a view V is created when the mapping of the view on the query fails. The D part of F consists of the part of Q that constitutes the point of failure of the extension of h_P, ϕ_P or G_P . Naturally, the creation of a F can be terminated prematurely because of several mismatches in the mapping process of V on Q . Moreover, even if there is no failure in the mapping of a view V on a query Q , the creation of F is possible in the case that the mapping of V cannot be combined with the mapping of other views in order to rewrite Q as a whole. A F is created in the following situations, either during the mapping of V on Q or the combination of the mapping of one view V and another view V' .

4.1.1 Creation of a single F

Failure of mapping V on a query Q and, therefore, the creation of $F(Q, V)$, can occur in the following cases:

Case 1: Query head variables are mapped to variables in the view, but the latter are not distinguished, thus they cannot be exposed to the rewriting algorithm. Formally: If there is a query head variable x which is in the domain of the view mapping ϕ_P of V on Q but is not in h_P , then: $D = \{x\}$. F indicates the elimination of x .

Example 1 Assume the query $Q_{orig}(\text{room\#, floor\#})$: -Rooms (room#, floor#) and the view $V(\text{room\#})$: -Rooms (room#, floor#). Then the head variable floor# should be eliminated in order to avoid view mapping failure, thus $D = \{\text{floor\#}\}$

Case 2: A query subgoal is mapped on the view but a join between variables within the subgoal is neither mapped, nor the respective mapped variables in the view are distinguished. Formally: If a homomorphism cannot be created or extended on V , in order to satisfy a join in a query subgoal $g.x = g.y$, then: $D = \{g.x = g.y\}$. The homomorphism on V is needed to satisfy a query join and F indicates the elimination of this join.

Example 2 Assume the query $Q_{orig}(\text{name, room\#})$: - Hotels (name, room#, price, price) and the view $V(\text{name, room\#})$: - Hotels (name, room#, rate, lowestprice). Then the join Hotels.rate = Hotels.lowestprice should be eliminated in order to avoid the failure; thus $D = \{\text{Hotels.rate} = \text{Hotels.lowestprice}\}$.

Case 3: A variable present in a set of query subgoals is mapped on V as an existential variable, but not all of these subgoals are mapped on V . Formally: If x is a variable in a query subgoal g , and ϕ_P is extended to contain the mapping of x but cannot be extended to contain g , then: $D = \{g\}$. The requirement for the mapping extension is due to a query join. F indicates the elimination of this join in case that the mapping ϕ_P cannot be extended or g cannot be satisfied by V .

Example 3 Assume the query that asks for the location with the lowest price for a specific period: $Q_{orig}(location):- HotelsCheck(hid, '03/11', '04/11', name, location, rate1) Hotels(name, room\#, rate2, lowestprice) HotelLocation(name, lowestprice, location)$ and the view $V(location):- HotelsCheck(hid, chkIn, chkout, name, location, rate1) Hotels(name, room\#, rate2, lowestprice)$. The subgoal $HotelLocation$ should be eliminated to avoid failure; thus $D = \{HotelLocation\}$.

Case 4: Query subgoals are mapped on V but joins between these subgoals are neither mapped in V nor respective variables in V are distinguished, so that the join can be enforced there. Formally: If a set of query subgoals $\mathcal{G}$ is satisfied by V , but there is a query variable $x \in g, \forall g \in \mathcal{G}$ such that ϕ_P cannot be extended to contain $\phi_P(x)$; then for any two subgoals $g_1, g_2 \in \mathcal{G}$: $D = \{g_1.x = g_2.x\}$. The reason for this failure is the incapability of V to rewrite a query join. F indicates the elimination of this join. (Checks for rewriting head variables as well as constant and comparison constraints are treated separately).

Example 4 Assume a query that asks for hotel names and lowest prices by the sea, but double checks the validity of information in both $HotelLocation$ and $Hotels$, $Q_{orig}(name, lowestprice):- HotelLocation(name, lowestprice, 'sea') Hotels(name, room\#, rate, lowestprice)$ and the view $V(name, rate):- HotelLocation(name, rate, 'sea') Hotels(name, room\#, rate, lowestprice)$. Then there is no appropriate homomorphism for the view and the join $HotelLocation.lowestprice = Hotels.lowestprice$ should be eliminated in order to avoid failure; thus $D = \{HotelLocation.lowestprice = Hotels.lowestprice\}$

Case 5: Query variables involved in comparison subgoals are not mapped on V . Formally: If x, y are a query variables, $y \in g$ and there is a subgoal $x.y$, where $.$ is a comparison predicate, $\phi_P(x)$ is an existential variable, but ϕ_P cannot be extended to contain $\phi_P(y)$, then: $D = \{x.y\}$. F indicates the elimination of the comparison atom in case that ϕ_P cannot be extended or a subgoal g cannot be satisfied by V . (Equivalent to case 3).

Example 5 Assume the query $Q_{orig}(name, lowestprice1):- Hotels(name, room\#, rate, lowestprice1) SeasonPrices(season, lowestprice2) rate \leq lowestprice2$ and the view $V(name, lowestprice):- Hotels(name, room\#, rate, lowestprice)$. Since V does not include the subgoal $SeasonPrices$ the comparison cannot be satisfied and $D = \{rate \leq lowestprice2\}$.

Case 6: Query variables in a comparison subgoal are mapped on variables of V that are not distinguished and the comparison subgoal is not mapped in V . Formally: If x, y are a query variables, $y \in g$ and there is a comparison subgoal $x.y$, where $.$ is a comparison predicate, $\phi_P(x)$ and/or $\phi_P(y)$ are existential variables, but ϕ_P cannot be extended to contain $\phi_P(x.y)$, then: $D = \{x.y\}$. V is incapable to satisfy the query comparison, thus, F indicates the elimination of the respective subgoal. (Equivalent to case 4).

Example 6 Assume the query $Q_{orig}(name1, lowestprice1):- Hotels(name1, room\#1, rate1, lowestprice1) Hotels(name2, room\#2, rate2, lowestprice2) lowestprice1 \leq lowestprice2$ and the view $V(name):- Hotels(name, room\#, rate, lowestprice)$. The view does not export sufficient information (distinguished variables) in order for the query comparison to be produced.

Case 7: A query constant is either mapped in a variable of V that is not distinguished, or it is not mapped at all. Formally: If a is a query constant, but $\phi_P(a)$ is neither the constant a nor it is in h_P , then: $D = \{a\}$. F indicates the elimination of a .

Example 7 Assume the query $Q_{orig}(name):- HotelLocation(name, lowestprice, 'sea')$ and the view $V(name):- HotelLocation(name, lowestprice, location)$. A rewriting failure occurs because the view does not make the specialization $location = 'sea'$. To avoid failure the constant constraint should be removed: $D = \{'sea'\}$.

4.1.2 Creation of multiple F s

Even if $F(Q, V)$ is not created while the rewriting algorithm is mapping V on Q , it can be created later on, while the algorithm tries to combine the mapping $\phi(V)$ of V , with the mapping $\phi(V')$ of another view V' (in which case the mappings are complete)¹.

Case 8: Views are covering overlapping query subgoals. Formally: If $\mathcal{V}$ is the set of views for which there is no $F(Q, V), \forall V \in \mathcal{V}$, then, if $\forall \mathcal{V}' \subseteq \mathcal{V}$ there is a set of overlapping query subgoals $\mathcal{G}$ mapped on $\mathcal{V}'$, then $\forall g \in \mathcal{G}$ and $\forall V \in \mathcal{V}'$ such that g is satisfied in all $V \in \mathcal{V}'$, an $F(V)$ is created, such that: $D = \{g\}$. The extension of G satisfied by a view is caused by query joins and variable comparisons. $\mathcal{F}$ is created, such that $\forall F \in \mathcal{F}$, F indicates the elimination of one join or one variable comparison.

Example 8 Assume the query $Q_{orig}(lowestprice, distance):- Hotels(name, room\#, rate, lowestprice) HotelLocation(name, lowestprice, location) Airport(aid, location, distance)$ and the views $V_1(lowestprice, location):- Hotels(name, room\#, rate, lowestprice) HotelLocation(name, lowestprice, location)$ and $V_2(name, distance):- HotelLocation(name, lowestprice, location) Airport(aid, location, distance)$. V_1 and V_2 cannot be combined due to overlapping subgoals.

Case 9: Views have contradictory constant constraints. Formally: If for $\mathcal{V}$ there is no $F(Q, V) \forall V \in \mathcal{V}$, then if there are $V_1, V_2 \in \mathcal{V}$ with contradictory constant constraints, i.e. $\phi(V_1.x) = a \neq b = \phi(V_2.x)$, then $\mathcal{F}_1 \equiv \mathcal{F}(Q, V_1)$ and $\mathcal{F}_2 \equiv \mathcal{F}(Q, V_2)$ are created such that each $F \in \mathcal{F}_1$ and $F \in \mathcal{F}_2$ eliminates a query join such that the mapping of x is not in $\phi_P(V_1), \phi_P(V_2)$.

Example 9 Assume the query $Q_{orig}(name, location):- Hotels(name, room\#, rate, lowestprice) HotelLocation(name, lowestprice, location)$ and the views $V_1(name):- Hotels(name, room\#, rate, '500')$ and $V_2(name, location):- HotelLocation(name, '300', location)$. V_1 and V_2 cannot be combined due to contradictory constants.

Case 10: Views have contradictory comparison constraints. Formally: If $\mathcal{V}$ is the set of views for which there is no $F(Q, V) \forall V \in \mathcal{V}$, then if there are $V_1, V_2 \in \mathcal{V}$ with contradictory comparison constraints, i.e. $\phi(V_1.(x.y)) \neq \phi(V_2.(x.y))$, then $\mathcal{F}_1 \equiv \mathcal{F}(Q, V_1)$ and $\mathcal{F}_2 \equiv \mathcal{F}(Q, V_2)$ are created such that each $F \in \mathcal{F}_1$ and $F \in \mathcal{F}_2$ eliminates a query join or a comparison so that the mapping of $x.y$ is not in $\phi_P(V_1), \phi_P(V_2)$.

Example 10 Assume $Q_{orig}(fno, carrier):- Flights(fno, depart, return, carrier, org, dest, price)$ and the views $V_1(fno):- Flights(fno, depart, return, carrier, org, dest, price), price < '400'$, $V_2(fno, carrier):- Flights(fno, depart, return, carrier, org, dest, price), price > '400'$. V_1 and V_2 cannot be combined due to contradictory comparisons.

Figures 4 and 5 show the procedures that produce the F s.

4.2 Unification of views

The unification of LAV mappings is an optimization integrated in the procedure that preprocesses and rewrites the query. The classical query rewriting algorithms that employ LAV mappings require that each query subgoal has to be translated by a single mapping. Therefore, if this requirement cannot be satisfied, then the subgoal has to be eliminated from the query; however, it may be that a combination of LAV mappings can rewrite this subgoal.

We have developed a procedure that overcomes this kind of situations by combining LAV mappings that can, as a set, rewrite the

¹Although ϕ and ϕ_P are formally defined for query subgoal variables, for simplicity of presentation, we use it loosely with other arguments too, such as a subgoal g and view V , meaning the overall mapping of g and V , respectively

```

Input: set of views  $\mathcal{V}$ , query  $Q$ 
Output: set of  $F$ s
1 begin
2    $\mathcal{F} = \emptyset$ ;
3   foreach subgoal  $g \in Q$  do
4     foreach view  $V \in \mathcal{V}$  do
5       foreach subgoal  $u \in V$  do
6         Let  $h$  be the least restrictive head homomorphism on  $V$ 
          s.t.  $\exists \phi | \phi(g) = h(u)$ ;
7         if  $h = \text{null}$  then
8           foreach join in which  $g$  participates and is not
            mapped in  $V$  do
9             Create  $F$  (case 1);  $\mathcal{F} \cup F$ ;
10          else Let  $G = \{g\}$ ;
11             Let the mapping until this point be  $\phi$ ;
12             repeat
13               if necessary conditions hold, (Sec. 4.1.1) then
14                 Create  $F$ s in the order (w.r.t. Sec. 4.1.1);;
15                 case 6; case 7; case 5; case 4; case 3; case 2;
16                 Create  $h_P$ ; Create  $\phi_P$ ; Create  $G_P$ ;
17                 Create  $D$ ;  $\mathcal{F} \cup F$ ;
18             until No extensions can be made;
19 return  $\mathcal{F}$ ;

```

Figure 4: Create F s for failed view mappings on the query

```

Input: set of views  $\mathcal{V}$ , query  $Q$ 
Output: set of  $F$ s  $\mathcal{F}$ 
1 begin
2    $\mathcal{F} = \emptyset$ ;
3   foreach  $\mathcal{V}' \subset \mathcal{V}$  s.t. views in  $\mathcal{V}'$  cover all  $Q$ -subgoals do
4     foreach  $V_i, V_j \in \mathcal{V}'$ ,  $i \neq j$  do
5       if  $V_i, V_j$  do not have overlapping subgoals then
6         if  $V_i, V_j$  have compatible constants then
7           if  $V_i, V_j$  have compatible comparisonsthen
8             Consider  $V_i, V_j$  compatible;
9           else Create  $\mathcal{F}' = F_1, \dots, F_n$  (case 10);  $\mathcal{F} \cup \mathcal{F}'$ ;
10          else Create  $\mathcal{F}' = F_1, \dots, F_n$  (case 9);  $\mathcal{F} \cup \mathcal{F}'$ ;
11          else Create  $\mathcal{F}' = F_1, \dots, F_n$  (case 8);  $\mathcal{F} \cup \mathcal{F}'$ ;
12 return  $\mathcal{F}$ ;

```

Figure 5: Create F s for failed combinations of complete view mappings on the query

query subgoal. In detail, this procedure searches for LAV mappings, the head of which encompasses the key attributes of the subgoal. In this case, a simple join between the found LAV mappings can produce a new mapping that rewrites the complete subgoal. Figure 6 delineates the steps of this procedure. The unification of LAV mappings not only enables approximate query rewriting, but also *extends* classical query rewriting for the particular case in which a set of views can be completely mapped on the posed query, but the views cover overlapping subgoals, which can be joined on their key. Multiple occurrences of these subgoals are eliminated, as shown in the following example.

Example 11 Assume a SQL query with a star select clause, written as follows in conjunctive form:

$Q_{orig}^{conj}(\text{vacid}, \text{fno}, \text{hid}, \text{from}, \text{until})$:- VacPackage(vacid, fno, hid, from, until) and the available views are:

$V_1(\text{vacid}, \text{fno}, \text{hid})$:- VacPackage(vacid, fno, hid, from, until)

$V_2(\text{vacid}, \text{from}, \text{until})$:- VacPackage(vacid, fno, hid, from, until)

The classical rewriting algorithms do not rewrite the above query, since the query subgoal is covered by both views. However, with view unification, the following view is produced:

$V_{1,2}(\text{vacid}, \text{fno}, \text{hid}, \text{from}, \text{until})$:- $V_1(\text{vacid}, \text{fno}, \text{hid}) V_2(\text{vacid}, \text{from}, \text{until})$

The above view can rewrite the query. The expansion of the view has a repeated subgoal self-joined on the key. Thus, the rewritten query is a classical rewriting.

```

Input: set of views  $\mathcal{V}$ , query  $Q_{orig}$ 
Output: set of views  $\mathcal{V}$ 
1 begin
2   foreach subgoal  $g$  of  $Q_{orig}$  do
3      $V_g = \emptyset$ ;
4      $X_g = \emptyset$ ;
5     foreach  $V$  in  $\mathcal{V}$  do
6       if the primary key of  $g$  is distinguished in  $V$  then
7         Add  $V$  to  $V_g$ ;
8       foreach attribute  $x$  of  $g$  in  $Q_{orig}^{SQL}$  do
9         if  $x$  is distinguished in  $V$  then
10           Add  $x$  to  $X_g$ ;
11       Create subsets of  $V_g, V_{g_i}$ , for some  $i$ , such that:
12       - All  $V \in V_{g_i}$  have compatible constraints.;
13       - Distinguished attributes of all  $V \in V_{g_i}$  cover as many as
        possible attributes in  $X_g$ ;
14       foreach  $V_{g_i}$  do
15         if  $|V_{g_i}| > 1$  then
16           Create new view  $V_{new_i} = \text{null}$ ;
17           Add to the distinguished variables of  $V_{new_i}$  all the
            distinguished variables of all  $V \in V_{g_i}$ ;
18           Add to the body of  $V_{new_i}$  the bodies of all  $V \in V_{g_i}$ ;
19           Merge (w.r.t. constraints) subgoals with the same
            predicate;
20           Remove duplicate subgoals from  $V_{new_i}$ ;
21       Add  $V_{new_i}$  to  $\mathcal{V}$ ;
22 return  $\mathcal{V}$ ;

```

Figure 6: Unification of views

4.3 Approximate rewriting for comparisons

Solving the containment, and therefore, the rewriting of queries with comparison subgoals has been a challenging issue [2]. As stated in [23], there are some cases of queries with comparison predicates that can be rewritten using the available views, but are not, since the respective containment check fails. Specifically, these are the cases that even though the comparison cannot be rewritten by the available views (i.e. no appropriate distinguished variables or comparison predicates in the views), the comparison constraint is satisfied; Actually, the view entails a specific case of the comparison, i.e. a contained rewriting. These situations occur if the comparison includes an equation. Intuitively, an equation reduces the comparison to a join, which can be satisfied by the view. Classical query rewriting cannot make this reduction. However, the proposed approximate query answering technique, achieves to include this reduction, by relaxing the requirement for the comparison satisfaction. Even with this relaxation, the joins in the considered view may be able to rewrite the equation.

Example 12 A query requests vacation packages the information of which is correct in the database, meaning that the 'from' date is before or on the 'until' date: $Q_{orig}(\text{vacid})$:- VacPackage(vacid, fno, hid, from, until), from $\leq$ until (let us assume that dates can be compared). Assume the following view that gives packages for one day is available to rewrite the query: $V(\text{vacid})$:- VacPackage(vacid, fno, hid, from, until) VacPackage(vacid, fno, hid, until, from).

Classical algorithms cannot rewrite Q_{orig} using V because the comparison constraint cannot be satisfied. The approximate rewriting technique, however, relaxes the query by removing the comparison, and achieves to produce the classically contained rewriting: $Q_{orig}(\text{vacid})$:- $V(\text{vacid})$

5. THE ALGORITHMS

In the following the algorithms that produce and rewrite the best Q_{approx} of Q_{orig} with respect to a set of views $\mathcal{V}$ are presented.

```

Input: set of views  $\mathcal{V}$ , query  $Q_{orig}$ 
Output: best approx. rewriting  $Q_{approx}^{best}$  and its  $M_{sim}$  value
1 begin
2    $Q_{approx}^{best} = null$ ;
3    $MaxSim = -\infty$ ;
4    $queue = \emptyset$ ;
5   Add  $Q_{orig}$  to  $queue$ ;
6   while  $queue \neq \emptyset$  do
7      $minQ_{approx} = Q_{approx}$  with min uncovered subgoals;
8     Dequeue  $minQ_{approx}$ ;
9     Run classical query rewriting for  $minQ_{approx}$ ;
10    if  $minQ_{approx}$  can be rewritten then
11      if  $M_{sim}(Q_{orig}, minQ_{approx}) > MaxSim$  then
12         $Q_{approx}^{best} = minQ_{approx}$ ;
13         $MaxSim = M_{sim}(Q_{orig}, minQ_{approx})$ ;
14      else
15        foreach  $F$  created do
16          Create the corresponding query  $Q_{approx_{approx}}$ ;
17          Add  $Q_{approx_{approx}}$  to  $queue$ ;
18  return  $Q_{approx}^{best}, MaxSim$ ;

```

Figure 7: The F Exhaustive algorithm

5.1 Exploring approximations

In order to produce an approximate rewriting of Q_{orig} it is necessary to navigate through the possible approximate versions of it. Imagine a tree of approximate versions of Q_{orig} where each node has one query element less than its father. Specifically, the root of the tree is Q_{orig} ; the children of the root correspond to approximations of Q_{orig} where a single query element is missing. The number of the nodes of the 1st level is equal to the number of query elements in Q_{orig} . Each one corresponds to a Q_{approx} which contains all the elements of Q_{orig} minus 1. Similarly, the children of the k^{th} level contain $n - k - 1$ elements of Q_{orig} , which are all the elements of their father minus 1. The total number of nodes on the k^{th} level is $n!/(n - k)!$. The total number of nodes is quite big with respect to n : $e\Gamma(n + 1, 1)$. Leaf nodes correspond to empty Q_{approx} . It is obvious that more than one tree nodes correspond to the same Q_{approx} . Consequently, there are subtrees that are repeated in the tree. Pruning the repeated subtrees results in a (non-balanced) tree with 2^n nodes for a Q_{orig} with n elements.

A straightforward approach in order to explore query approximations and their rewritings, is to create the tree of approximations and run a classical rewriting algorithm for each of the distinct nodes of this tree. Then, we can compare the rewritings produced by each such execution and find the best approximation Q_{approx} . In detail, for a Q_{orig} of n subgoals, the employment of a classical rewriting algorithm in the above manner has to check $O(2^n)$ Q_{approx} versions, as follows: (1) produce all possible Q_{approx} , (2) order them in descending manner with respect to their similarity to Q_{orig} , and (3) check them iteratively until an output, i.e. a complete rewriting, can be produced. We call this the *Simple Exhaustive* algorithm and we use it as a reference for comparison to the algorithms that are proposed in the following and that are guided by the creation of F structures, as shown in Figure 2. *Simple Exhaustive* is quite expensive, mainly because of step (1), which has to produce all the nodes of the search tree described above and prune repeated subtrees. The complexity of *Simple Exhaustive* is $O(e \cdot \Gamma(n + 1, 1))$.

5.2 The F Exhaustive algorithm

Employing the structures F s, we can explore the tree of possible query approximations, such that far fewer nodes are visited than those that are visited in a straightforward manner with the *Simple Exhaustive* algorithm. Essentially, F s can guide the ex-

ploration to the subtrees that contain approximations which are (a) possible with respect to the available views as well as (b) necessary with respect to the failures of mapping the available views on the query. Exploration is cheaper (i.e. it visits fewer tree nodes) because guided by F s we can prune the subtrees such that approximated queries that are missing elements which can be rewritten by combinations of complete view mappings are not visited. Figure 7 presents the F Exhaustive algorithm that explores the possible approximations guided by F s. Briefly, the algorithm proceeds as follows: First the algorithm computes the initial set of complete view mappings; if this set of views is not able to thoroughly rewrite Q_{orig} , it creates the tree of F s (and not the tree of possible approximations). Each edge of this tree corresponds to a F and each node to the Q_{approx} produced by the path of F s that leads to it from the root. The tree is created as follows: Initially, the approximation similarity $MaxSim$ is set to minimum and all the necessary F s are created and form the set $\mathcal{F}$. For each $F \in \mathcal{F}$ the respective Q_{approx} is created and if its similarity is higher than the current value of $MaxSim$, search explores more this possible approximation. The algorithm iterates on this F by resetting the set $\mathcal{V}$ and running a classical rewriting algorithm on Q_{approx} (note that steps 6-9 of Figure 2 are suitably merged for efficiency). In case of failure, the procedure starts from the beginning by computing the new F s and adding them to $\mathcal{F}$. Since F Exhaustive searches guided by the F tree and not by the tree of approximations, its complexity is $O(|\mathcal{F}|)$, where $\mathcal{F}$ is the set of possible F s. The latter are far fewer than the number of possible approximations and in general it holds that their number is proportional to the number of available views $|\mathcal{F}| \propto |\mathcal{V}|$. In practice the number of views is far smaller than the size of the approximation tree, thus, $O(|\mathcal{V}|) < O(e \cdot \Gamma(n + 1, 1))$, or the number of possible approximations.

Theorem 1 *The search space of F Exhaustive is a subset of the search space of Simple Exhaustive.*

Sketch of proof: Assume a Q_{approx} that corresponds to a F with some $D = \{x\}$. Then Q_{approx} is missing a query element corresponding to x . x represents the point of failure during the mapping of a view $V \in \mathcal{V}$ and to a query Q that is either Q_{orig} or an approximation of it Q'_{approx} . Q'_{approx} has a subset of the query elements of Q_{orig} . Thus, the mapping of V on Q'_{approx} can fail in a subset of the possible points of failure of the mapping of V on Q_{orig} . The possible points of failure of the mapping of V created for Q_{orig} correspond to a subset of all the query elements; let us call it $\mathcal{E}_V$. The union of the $\mathcal{E}_V$ for all $V \in \mathcal{V}$, called $\mathcal{E}_\mathcal{V}$ is a subset of all the query elements of Q_{orig} . Therefore, if a Q_{approx} is missing an element that is not in $\mathcal{E}_\mathcal{V}$, it is not in the search space of F Exhaustive. On the opposite, the search space of *Simple Exhaustive* contains all the possible Q_{approx} .

Intuitively, the search space of F Exhaustive is much smaller than that of *Simple Exhaustive* because the first contains only the Q_{approx} that are missing query elements that cannot be rewritten employing the available views, whereas the second contains all the possible Q_{approx} . The following is an intuitive example.

Example 13 *Assume the query:*

```

 $Q_{orig}^{SQL}$  : SELECT H. name
FROM Hotels H, HotelLocation HL, SeasonPrices SP
WHERE H.name = HL.name AND HL.lowestprice = SP.lowestprice
AND SP.season = "spring" AND HL.location = "sea"

```

written in conjunctive form as:

```

 $Q_{orig}^{conj}$  (name):- Hotels (name, room#, rate, lowestprice1) HotelLocation
(name, lowestprice, 'sea') SeasonPrices('spring', lowestprice).
 $Q_{orig}^{SQL}$  has 5 elements and therefore the total number of distinct

```



```

Input: set of views  $\mathcal{V}$ , query  $Q_{orig}$ 
Output: best approx. rewriting  $Q_{approx}^{best}$  and its  $M_{sim}$  value
1 begin
2    $Q_{approx}^{best} = null$ ;
3    $MaxSim = -\infty$ ;
4    $queue = \emptyset$ ;
5    $queue_{children} = \emptyset$ ;
6   Add  $Q_{orig}$  to  $queue$ ;
7   while  $queue \neq \emptyset$  do
8      $Q_{approx} = \text{all } Q_{approx} \text{ with min uncovered subgoals}$ ;
9     while  $Q_{approx} \neq \emptyset$  do
10      Select  $minQ_{approx}$  with max  $M_{sim}(Q_{orig}, Q_{approx})$ ;
11      Remove  $minQ_{approx}$  from  $Q_{approx}$ ;
12      Run classical query rewriting for  $minQ_{approx}$ ;
13      if  $M_{sim}(Q_{orig}, minQ_{approx}) > MaxSim$  then
14        if  $minQ_{approx}$  can be rewritten by MiniCon then
15           $Q_{approx}^{best} = minQ_{approx}$ ;
16           $MaxSim = M_{sim}(Q_{orig}, minQ_{approx})$ ;
17          return  $Q_{approx}^{best}, MaxSim$ ;
18        else foreach  $F$  created do
19          Create the corresponding query  $Q_{approx_{approx}}$ ;
20          Add  $Q_{approx_{approx}}$  to  $queue_{children}$ ;
21       $queue = queue_{children}$ ;
22       $queue_{children} = \emptyset$ ;
23 return  $Q_{approx}^{best}, MaxSim$ ;

```

Figure 8: The F B&B algorithm

approximations is 32 and the size of the approximation tree is 205. Assume that the available views are:

V_1 (name, lowestprice) :- Hotels (name, room#, rate, lowestprice) Hotel-Location (name, lowestprice, 'sea') and

V_2 (lowestprice) :- SeasonPrices (season, lowestprice).

V_1 has a complete mapping on Q and therefore we do not need to examine approximations that miss elements that can be satisfied by V_1 , i.e. the 'select' attribute, the condition $HL.location = \text{"sea"}$ and the join $H.name = HL.name$. V_2 has a partial mapping that creates an F with $D = \{\text{"spring"}\}$. This is the only F created, and examined for produced approximations, since the combination of the complete mapping of V_1 and the partial mapping of V_2 cover the remaining element $HL.lowestprice = SP.lowestprice$.

Theorem 2 F Exhaustive is complete, i.e. it finds the Q_{approx} with the maximal $MaxSim$.

Sketch of proof: We have to prove that the Q_{approx} with the maximal $MaxSim$ is in the search space of F Exhaustive. Assume that Q_{approx_1} and Q_{approx_2} are the approximate queries with the maximal $MaxSim$ in the search space of F Exhaustive and Simple Exhaustive, respectively. If $Q_{approx_1} \not\equiv Q_{approx_2}$, then Q_{approx_1} is missing a query element $E \in \mathcal{E}_{orig}$ and therefore E corresponds to a $D = \{x\}$ of a $F \in \mathcal{F}$. Thus, F has been considered by F Exhaustive and the best approximate query following F has been compared to Q_{approx_1} and rejected. Therefore if E is in Q_{approx_2} then E is in Q_{approx_1} . The opposite holds because Q_{approx_2} is by default the best possible approximation. Therefore $Q_{approx_1} \equiv Q_{approx_2}$.

Intuitively, F Exhaustive is complete because the best Q_{approx} is missing only elements of Q_{orig} that constitute points of failure of mapping views on Q_{orig} ; all of these points are explored by the produced F s. Thus, the best Q_{approx} is in the search space of the algorithm. Since the latter is exhaustive, it finds the best Q_{approx} .

The set of $\mathcal{F}$ in F Exhaustive is augmented in each iteration and one $F \in \mathcal{F}$ is selected for examination. To improve efficiency, we reorder the $\mathcal{F}$ before each iteration with respect to a heuristic criterion that is based on the notion of *uncovered subgoals*:

Definition 2 With respect to a set of views $\mathcal{V}$ and their complete mappings $\phi(\mathcal{V})$ on the query Q_{orig} , the uncovered subgoals of Q_{orig} is the set of the subgoals that cannot be covered by any mapping in $\phi(\mathcal{V})$.

The heuristic criterion Cr_1 is defined as follows:

Definition 3 Create an ascending order of all $F \in \mathcal{F}$ with respect to the number of uncovered subgoals of Q_{orig} that cannot be covered by each F .

Criterion Cr_1 dictates that the more *uncovered subgoals* can be covered by the considered F , the greater the possibility that the view mapping to be created after relaxing the query according to F will achieve the rewriting of currently not covered subgoals of Q_{orig} . The F that does not cover few *uncovered subgoals* leads to a view mapping on the respective Q_{approx} that covers many of the query elements that cannot be covered by all existing mappings $\phi(\mathcal{V})$ for Q_{orig} . The more these query elements are, the fewer are left that could not be covered by the available views and demand the exploration of approximations. Using Cr_1 F Exhaustive achieves to explore the Q_{approx} with a high $MaxSim$ first, such that a big part of the F tree is not visited, making the algorithm very efficient as shown in Section 6.

5.3 The F B&B algorithm

Even though the search space of the F Exhaustive is much smaller than that of the Simple Exhaustive, the exhaustive traversal of the F tree can be very expensive. Thus, we propose a method that prunes the tree such that a partial traversal of it is still eligible to produce a Q_{approx} close, in terms of query similarity, to the best possible one. The following heuristic criterion, Cr_2 , is used complementary to criterion Cr_1 so that such a pruning is achieved.

Definition 4 Create a descending order of all $F \in \mathcal{F}$ with respect to the similarity of Q_{approx} to Q_{orig} , where Q_{approx} is the approximation produced employing F .

Orthogonally to Cr_1 , Cr_2 denotes that the better the approximation of Q_{orig} produced according to the considered F , the bigger the possibility that F leads to the production of the best approximation of Q_{orig} . Oppositely to Cr_1 , Cr_2 is optimistic and judges the possibility of achieving the approximate query rewriting based on the query elements that are covered by the considered F .

The combination of Cr_1 and Cr_2 forms the branch-and-bound version of the F Exhaustive, that we call the F B&B algorithm. This algorithm is presented in Figure 8 and, briefly, it proceeds as follows: At each iteration that a view mapping fails, the children nodes corresponding to appropriate F s are created and are ordered based on the values of Cr_1 ; The F s that lead to query approximations that are of different similarity to Q_{orig} but are in the same partial order according to Cr_1 , are ordered based on Cr_2 . The combination of Cr_1 and Cr_2 that gives the lead to the first and refines the search with the second is adequate to guarantee a near-optimal query rewriting approximation, as shown in Section 6.

6. EXPERIMENTAL STUDY

In what follows, we present the experimental evaluation of producing approximate versions of Q_{orig} using a set of views $\mathcal{V}$.

6.1 Experimental Setup

Methodology. The proposed algorithms, F Exhaustive and F B&B, are compared against Simple Exhaustive with respect to their execution time. For F B&B the similarity difference (denoted as "approximation error") of the computed approximation against the optimal approximation, (computed by both F Exhaustive and Simple

Figure 9: Results for execution time of *F B&B* and *F Exhaustive* and approximation error of *F B&B*

Exhaustive) is measured. In the graphs, the "*F B&B* error" lines refer to the scale on the right and the rest lines refer to the scale on the left (if there are two vertical axes). Among all classical query rewriting algorithms we have used for the implementation of the algorithms and the experimentation with the proposed approximate query rewriting technique the MiniCon algorithm [23], which is the latest and most efficient of the well-known ones. As stated before, defining a query similarity measure is an orthogonal problem to finding query approximations. In the experiments we employ one of the query similarity measures presented in [11] as follows:

$$M_{sim_1}(Q_{orig}, Q_{appr}) = \sum w_i \cdot |sat(E_i, E'_i)| / \sum w_i \quad (1)$$

This measure confines the similarity of two queries by the similarity of their elements. E_i 's are elements ('select' values or 'where' conditions) of Q_{orig} , E'_i 's are the respective elements of Q_{appr} , and $\sum w_i = 1$. The function $sat(E_i, E'_i)$ captures the notion of E_i represented by E'_i and for this work is either 0 or 1.

Datasets. The experiments are performed on synthetic data, following the example of works in query rewriting using views [16, 23, 26]. Synthetic data give the opportunity to control the form/size of Q_{orig} and the set of views $\mathcal{V}$; therefore, it enables the case-study of a big variety of query-views combinations, that cannot be found in real datasets. The parameters of the algorithms are plenty. For both the query and the views: number of subgoals, number of distinguished variables, number of repeated subgoals, shape (random, star and chain queries for repeated subgoals), presence of arithmetic comparisons; More parameters are: the number of views, query-views shape and size combinations, as well as different similarity functions. The number of parameters prohibits the presentation of the thorough experimental study due to lack of space. In the following the most important and interesting experiments are described. If not fluctuated, the following parameter values are used. Number of: relations = 10, views = 10, view subgoals = 4, query subgoals = 5, repeated query subgoals = 2, query constants = 1, view distinguished variables = 2, view joins = 3, view constants = 3, view comparisons = 3; query form = "random", query distinguished variables = 20%, .

6.2 Experimental Results

Figure 9 shows graphically the experimental results, which are commented in the following.

6.2.1 Number of views

Figure 9(a) shows the results for random queries and views as the number of views increases. *F Exhaustive* starts with being 100 times faster from *Simple Exhaustive*, and this difference shrinks but remains substantial for bigger number of views. *F Exhaustive* performs exceptionally for a low number of views compared to *Simple Exhaustive*, because the search in *F Exhaustive* is guided by the views, (i.e. only query parts that cannot be rewritten based on the (few) views are examined for elimination), whereas *Simple Exhaustive*, is guided by approximate query versions; even though it checks approximation with increasing approximation cost, the small number of views results in the consideration of many query elements for elimination, and therefore, of an exponential number of approximate query versions. Since the rewriting of a query based only on a few (less than 5) views is expected to be a common case in real situations, using *F Exhaustive* gives a big advantage. Both algorithms perform much better for a very big (> 90) number of views. The reason is that in this case, most frequently, the query can be rewritten with very few eliminations. The relative performance of *F Exhaustive* increases because it easily finds the appropriate eliminations. *F B&B* is steadily 1000 faster than *Simple Exhaustive* achieving a very low approximations error ($< 4\%$), which makes it very competitive to *F Exhaustive*.

6.2.2 Number of query elements

Figures 9(b), (c) and (d) shows the results varying the number of query elements, i.e. the distinguished variables (the 'select' attributes) and the constraints (i.e. where conditions: joins and variable comparisons, comparisons with constants). Overall, the efficiency of the algorithms steadily increases as the query size increases. *Simple Exhaustive* is affected exponentially by the query size, since each additional element adds combinations with all the rest to the set of possible approximations. However, since the creation of F 's is not based on query elements, but on query - view combinations, the number of F 's is not affected so radically by the increase in the number of query elements. Both algorithms become more efficient for larger queries.

Number of subgoals. Figure 9(b) shows the results for random queries and views as the number of query subgoals increases. The efficiency of both the F algorithms increases with the number of subgoals. The approximation error of *F B&B* increases with an ex-

tremely low rate with the number of subgoals. Naturally, the bigger the number of subgoals the bigger the number of joins, which slightly but consistently affects the refined pruning of *B&B*.

Number of constants. Figure 9(c) shows the results while increasing the number of constants in the query. The efficiency of the algorithms increases with a slightly bigger rate than the respective rate for increasing number of subgoals. The satisfaction of constant constraints is uncorrelated to the satisfaction or not of the rest of the constraints (joins between subgoals), because they do not participate in join pools. *F B&B* shows a very small approximation error that is reduced as the number of constants increases.

Number of distinguished variables Similarly, the results for increasing number of distinguished variables, (Figure 9(d)), show an even bigger rate of efficiency increment: the satisfaction of a distinguished variable is independent from the satisfaction of the rest query elements. Thus, the algorithms insist on specific search paths and examine a restricted number of *F*s. The behavior of the approximation cost is between the one for increasing number of constants and the one for subgoals. *F B&B* is really benefitted by the independence of the satisfaction of distinguished variables.

6.2.3 Number of repeated subgoals

Figures 9(e),(f) present the results for queries with a constant number of elements, but with increasing repeated subgoals (i.e. self-joins). These experiments focus on the role of the query shape: random, chain and star [23,26]. Overall, repeated subgoals increase the dependency of query constraints: because of the self-joins, a view may be able to rewrite a big number of query elements if it satisfies one of them, or the opposite.

Random queries. The results for random queries show that a percentage of 40% of repeated subgoals affects negatively the efficiency of the algorithms: they have to satisfy a big number of tightly dependent subgoals and a similar number of non-related to them subgoals. On the opposite, a small or big percentage of repeated subgoals either weakens or strengthens the effect of their dependance, leading the algorithm either to shortly leave a bad or stick to a good search path. *F B&B* achieves a zero approximation error for queries with more than 25% repeated subgoals.

Star queries. The results for star queries are harsher than in the case of random queries. Star queries indicate intensively to *F*s which constraints should be eliminated. The presence of repeated subgoals reinforces the possible choice of a subset of approximations, reducing the occurrences of backtracking in search space.

Chain queries. Oppositely to star, chain queries may create more approximations, because the satisfaction of constraints is primarily led by the join chain, which gradually indicates the options for approximation in each algorithm iteration. Efficiency results are smoother than those for random or star queries. Repeated subgoals, reinforce some of the approximation options and reduce the error.

6.2.4 Overall experimental conclusions

Overall, *F Exhaustive* is considerably faster than *Simple Exhaustive* and their speed difference is substantial for the important situations of few views or big queries (in terms of number of elements). *F B&B* is much faster than *F Exhaustive* and keeps a steady or even increasing rate of speed up with respect to *Simple Exhaustive*. The heuristic of *F B&B* is very efficient and the approximation error is generally much lower than 10%. Chain queries are the hardest to handle with *F B&B*.

7. CONCLUSIONS

We tackle the problem of approximate query answering in environments of autonomous and heterogeneous sources and we pro-

pose a query relaxation technique that extends traditional query answering. The technique comprises algorithms that search for the best approximation and employ as a black box a query similarity function. The experiments prove the efficiency of the algorithms.

Acknowledgments

This research has funding from the EU Seventh Framework Programme (FP7/2007-2013), Grant Agreement n° 619706 ASAP.

8. REFERENCES

- [1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *SIGMOD*, 1999.
- [2] F. Afrati, C. Li, and P. Mitra. Answering Queries Using Views with Arithmetic Comparisons. In *21th ACM PODS*, 2002.
- [3] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated Ranking of Database Query Results. In *CIDR*, 2003.
- [4] K. C.-C. Chang and H. Garcia-Molina. Approximate query mapping: Accounting for translation closeness. *VLDB J.*, 10(2-3), 2001.
- [5] S. Chaudhuri and P. G. Kolaitis. Can datalog be approximated? In *PODS*, 1994.
- [6] W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. In *SIGMOD*, 1998.
- [7] M. de Rougemont and A. Vieilleribière. Approximate data exchange. In *ICDT*, pages 44–58, 2007.
- [8] O. M. Duschka and M. R. Genesereth. Answering Recursive Queries Using Views. In *16th ACM PODS*, 1997.
- [9] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1), 2005.
- [10] A. Fuxman and R. J. Miller. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci.*, 73(4):610–635, 2007.
- [11] V. Kantere, D. Tsoumakos, T. Sellis, and N. Roussopoulos. Groupeer: Dynamic clustering of p2p databases. *Inf. Syst.*, 34(1):62–86, 2009.
- [12] W. Kieling and G. Kostner. Preference SQL - Design, Implementation, Experiences. In *VLDB*, 2002.
- [13] M. Lenzerini. Data Integration: A Theoretical Perspective. In *21th ACM PODS*, 2002.
- [14] A. Y. Levy. Answering Queries Using Views: A Survey. In *VLDB Journal*, 2001.
- [15] A. Y. Levy, A. Rajaraman, and J. O. Ordille. Query-answering Algorithms for Information Agents. In *13th International Conference on Artificial Intelligence*, 1996.
- [16] V. Lin, V. Vassalos, and P. Malakasiotis. Minicount: Efficient rewriting of count-queries using views. In *ICDE*, page 1, 2006.
- [17] J. Madhavan and A. Y. Halevy. Composing mappings among data sources. In *VLDB*, pages 572–583, 2003.
- [18] P. Mitra. An Algorithm for Answering Queries Efficiently Using Views. In *Australasian Database Conference*, 2001.
- [19] I. Muslea and T. J. Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, pages 831–836, 2005.
- [20] U. Nambiar and S. Kambhampati. Answering imprecise queries over autonomous web databases. In *ICDE*, page 45, 2006.
- [21] N. Polyzotis, M. N. Garofalakis, and Y. E. Ioannidis. Approximate xml query answers. In *SIGMOD Conference*, pages 263–274, 2004.
- [22] V. Poosala and V. Ganti. Fast approximate query answering using precomputed statistics. In *ICDE*, page 252, 1999.
- [23] R. Pottinger and A. Levy. A Scalable Algorithm for Answering Queries Using Views. In *VLDB*, 2000.
- [24] P. Rodríguez-Gianolli, A. Kementsietsidis, M. Garzetti, I. Kiringa, L. Jiang, M. Masud, R. J. Miller, and J. Mylopoulos. Data sharing in the hyperion peer database system. In *VLDB*, 2005.
- [25] J. Spiegel, E. D. Pontikakis, S. Budalakoti, and N. Polyzotis. Aqax: A system for approximate xml query answers. In *VLDB*, pages 1159–1162, 2006.
- [26] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB J.*, 6(3):191–208, 1997.
- [27] G. Wolf, H. Khatri, B. Chokshi, J. Fan, Y. Chen, and S. Kambhampati. Query processing over incomplete autonomous databases. In *VLDB*, pages 651–662, 2007.