UNIVERSITY OF EDINBURGH

MiniCon : A Scalable Algorithm for Answering Queries Using Views

ATFD Project by Paridhi Mishra s1549106

April 8, 2016

This project studies the problem of answering queries using views with focus on efficient methods using materialized views rather than accessing the database relations. It presents a novel algorithm - MiniCon, compares it with the existing two algorithms (bucket algorithm and inverse-rules algorithm), conducts experiments for the same and describes an extension to the MiniCon algorithm to handle comparison predicates. This project contributes in the implementation of the MiniCon algorithm and implementing the extension which can handle functional dependencies in a mediated schema.)

SUMMARY

(Pottinger & Halevy, 2001) study the problem of answering queries over views for a given database(s). In particular, it studies methods that use previously materialized views instead of database relations to answer these queries. This problem is frequently encountered in database management and has been approached in the literature for (1) query optimization or maintenance of physical data independence where a set of the views equivalent to the original query may be used and (2) in data integration, where the query is directed over a mediated schema [1]. For this second case, the query needs modification before it can be optimized and executed.

This query modification, known as the *query reformulation* problem, suffers from limited coverage of the data sources in a data integration scenario so finding a rewriting that is equivalent to the user query is difficult. A close solution called *maximally-contained rewriting*, which is a union of conjunctive queries [2] without comparison predicates over the views, may be utilized in this case. (Pottinger & Halevy, 2001) considers the problem of answering conjunctive queries which are NP-complete as the search space can have an exponential number of rewritings for a given query. Previous literature provides two algorithms to tackle this problem known as the *bucket* algorithm and the *inverse-rules* algorithm. Both are unable to scale with the number of views. A novel algorithm, called **MiniCon** algorithm is introduced which addresses limitations of above both algorithms and

[1] A mediated schema presents a unified user view for a set of heterogeneous data sources, is designed manually by an integration framework using either GAV and LAV approach

[2] A conjunctive query is a select-project-join query of first-order logic using only conjunction $\wedge$ and existential quantification $\exists$

also scales up to a large number of views. The study in this paper is focused on the problem of answering queries using views for select-project-join queries under set semantics and on scalability issues in the experimental evaluation of these algorithms.

Conjunctive query has the form $q(\bar{X}) : e_1(\bar{X_1}), \cdots, e_n(\bar{X_n})$ where q and $e_1, \cdots, e_n$ are predicate names. The atoms $e_1(\bar{X_1}), \cdots, e_n(\bar{X_n})$ are the subgoals in the body of the query, where $e_1, \cdots, e_n$ refer to database relations. The atom $q(\bar{X})$ is called the head of the query, and refers to the answer relation. The tuples $\bar{X}, \bar{X_1}, \cdots, \bar{X_n}$ have either variables or constants. A *safe* query means $\bar{X} \subseteq \bar{X_1} \cup \cdots \cup \bar{X_n}$. The variables in $\bar{X}$ are called *distinguished* and remaining variables in the body are called *existential* variables. $Vars(Q)(Subgoals(Q))$ is the set of variables (subgoals) in Q.

Containment mappings : $Q_1 \subseteq Q_2$ means query Q_1 is contained in query Q_2 [3]. Both queries are equivalent if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. A containment mapping τ from $Vars(Q_1)$ to $Vars(Q_2)$ maps every subgoal in the body of $Vars(Q_2)$ to a subgoal in the body of $Vars(Q_1)$, and also maps the head of Q_2 to the head of Q_1. The query Q_2 contains Q_1 if and only if there is a containment mapping from Q_2 to Q_1. Given query Q and set of view definitions $V = V_1, \cdots, V_m$ rewriting of this query is another query Q'whose body predicates are either $V_1, \cdots, V_m$ or comparison predicates. Two types of query rewritings are :

- *Equivalent rewritings:* Given a set of view definitions $V = V_1, \cdots, V_n$ and a query Q, its equivalent rewriting , Q'is another query where the result of evaluating Q'over $V_1(D), \cdots, V_n(D)$ is same as $Q(D)$.

- *Maximally-contained rewritings:* Given a conjunctive query Q and a set of conjunctive views $V = V_1, \cdots, V_n$, Q'w.r.t. a query language L if:

  (1) for any database D, and extensions $v_1, v_2 \cdots, v_n$ of the views such that $v_i \subseteq V_i(D)$, for $1 \leq i \leq n$, then $Q'(v_1, v_2, \cdots, v_n) \subseteq Q(D)$ for all i,

  (2) there is no other query Q_1 in the language L, such that for every database D and extensions $v_1, v_2 \cdots, v_n$ as above (1) $Q'(v_1, v_2 \cdots, v_n) \subseteq Q(v_1, v_2 \cdots, v_n)$ and (2) $Q(v_1, v_2 \cdots, v_n) \subseteq Q(D)$ (there must exist at least one database for which (1) is a strict set inclusion)

Citing Papers example: Relation *cites(p1,p2)* stores pairs of publication identifiers where p1 cites p2. The relation *same-Topic* stores pairs of papers that are on the same topic. The unary relations *inSIGMOD* and *inVLDB* store papers published in SIGMOD and VLDB. Query Q gives all pair of papers citing each other and having the same topic whereas Q'is an equivalent rewriting for Q

Q(x,y):- sameTopic(x,y), cites(x,y), cites(y,x)
Q'(x,y):- V1(x,y), V2(x,y)
where
V1(a,b):- cites(a,b), cites(b,a)
V2(c,d) :- sameTopic(c,d), cites(c,c1), cites(d,d1)

[3] if the answer to Q_1 is a subset of the answer to Q_2 for any database instance

Consider two data sources S1 storing SIGMOID papers and S2 storing VLDB papers such that:

S1(a,b):- cites(a,b), cites(b,a), inSIGMOD(a), inSIGMOD(b)

S2(a,b):- cites(a,b), cites(b,a), inVLDB(a), inVLDB(b)

In this data integration scenario, data sources S1 and S2 are managed autonomously, which means that the views defined above may not contain all tuples [4]. Also, sometimes an equivalent rewriting is not possible, say if neither S1 nor S2 have *all data* to calculate Q. Consider the following queries:

Q́(x,y):- S1(x,y), V2(x,y)

Q́(x,y):- S2(x,y), V2(x,y)

Above queries will not contain the pairs when one paper is in S1, and its pair is in S2, and both cite each other.

S3(a,b):- cites(a,b), cites(b,a), inSIGMOD(a), inSIGMOD(b)

Here, S1 and S3 are identical, but may have different tuples because of source incompleteness. So the new rewriting should also include a third query

Q'(x,y):- S3(x,y), V2(x,y)

Together, the above three queries are maximally-contained rewritings.

The search for a maximally-contained rewriting is always restricted to a finite space when there are no comparison predicates in the query: an algorithm needs to consider every possible conjunction of *n* or fewer view atoms, where *n* is the number of subgoals in the query. Both the following algorithms, the bucket algorithm and the inverse-rules algorithm, use this restrictive space to produce rewritings faster.

The Bucket Algorithm : Each subgoal in the query in considered separately and all relevant views are put in that subgoal's bucket

Q1(x) :- cites(x,y),cites(y,x),sameTopic(x,y)

V4(a) :- cites(a,b), cites(b,a)

V5(c,d) :- sameTopic(c,d)

V6(f,h) :- cites(f,g),cites(g,h),sameTopic(f,g)

Step1: Relevant views [5] for each subgoal in the query are put in *buckets* for each subgoal separately as below:

Bucket cites(x,y) = V4(x),V6(x,y)

Bucket cites(y,x) = V4(x),V6(x,y)

Bucket sameTopic(x,y) = V5(x,y),V6(x,y)

Step2: For every possible combination of the *cartesian product* of the buckets, a conjunctive rewriting is done and checked for containment in the query. The result of is a union of these conjunctive rewritings. The combination of V4 from first bucket fails, but when V6 is combined, a contained rewriting is possible. Next, the algorithm combines V6 and V5 and final result is

Q1'(x) :- V6(x,x)

[4] S1 may not have all pairs of SIGMOD papers that cite each other

[5] views which have distinguished variables to which the distinguished variables of the query are mapped

The Inverse-Rules Algorithm : This algorithm creates a set of rules that invert the view definitions [6]. For the above example, the algorithm would construct the following inverse rules:

R1: cites(a, f1(a)) :- V4(a)

R2: cites(f1(a), a) :- V4(a)

R3: sameTopic(c,d) :- V5(c,d)

R4: cites(f, f2(f,h)) :- V6(f,h)

R5: cites(f2(f,h), h) :- V6(f,h)

R6: sameTopic(f, f2(f,h)) :- V6(f,h)

To connect two separate tuples with a relation to these rules, a functional *Skolem term* f1(Z) is used for specifying that there is some unknown value which connects both tuples. Rewriting of queries is done using the set of views and the combination of the inverse rules. These rules are created before the queries are run, so this algorithm is faster, as it takes some of the run time offline. It also becomes independent of the query. Its disadvantages include the possibility of inversion of some useful steps which might have been done to produce the initial views and the chances of accessing some irrelevant views in the process.

The MiniCon Algorithm : In the first phase, this algorithm considers which views contain subgoals that correspond to subgoals in the query. Once it finds a partial mapping from a subgoal g in the query to a subgoal g_1 in a view V, it looks at the variables in the query to find the join predicates (variables occurring multiple times) and finds the minimal additional set of subgoals that need to be mapped to subgoals in V; this set of subgoals and mapping information is **MiniCon Description** (MCD). In the second phase, the algorithm combines these MCDs to produce the rewritings. Note that this algorithm does not require containment checks in this second phase, an advantage over the bucket algorithm. Given a mapping τ from Vars(Q) to Vars(V), a view subgoal g_1 covers a query subgoal g if $\tau(g) = g_1$. An MCD is a mapping from a subset of the variables in the query to variables in one of the views. Every MCD has an associated head homomorphism h; h on a view V is a mapping h from $Vars(V)$ to $Vars(V)$ that is the identity on the existential variables, but may equate distinguished variables. If x is a distinguished variable, then so is $h(x)$. Also $h(x) = h(h(x))$

An MCD C for a query Q over a view V is a tuple of the form $(h_C, V(\bar{Y})_C, \phi_C, G_C)$ where:

- h_C is a head homomorphism on V
- Applying h_C to V gives $V(\bar{Y})_C$ ($\bar{Y} = h_C(\bar{A})$)
- φ_C is a partial mapping from $Vars(Q)$ to $h_C(Vars(V))$
- G_C is a subset of subgoals in Q covered by some subgoal in $h_C(Vars(V))$ using the mapping φ_C

C, created in first step, can only be used in a non-redundant rewriting of Q in the second step if: **Clause 1.** For each head variable x of Q which is in the domain of ϕ_C, $\phi_C(x)$ is a head variable

[6] rules that show how to compute tuples for the database relations from tuples of the views

in $h_C(V)$ **Clause 2.** If $\phi_C(x)$ is an existential variable in $h_C(V)$, then for every g, subgoal of Q, that includes x: (1) all the variables in g are in the domain of ϕ_C; and (2) $\phi_C(g) \in h_C(V)$

procedure formMCDs(Q**,**V**)** /*if Q and V are conjunctive queries*/
$C = \emptyset$
For each subgoal $g \in Q$
For view V in V and every subgoal $v \in V$
Let h be the least restrictive head homomorphism on V such that there exists
a mapping ϕ , where $\phi(g) = h(v)$.
If h and ϕ exist, then add to C any new MCD that can be constructed where:
(a) ϕ_C (resp.h_C) is an extension of ϕ (resp. h),
(b) G_C is the minimal subset of subgoals of Q such that G_C, ϕ_C satisfy Property 1, and
(c) it is not possible to extend ϕ and h to ϕ'and h'such that (b) is satisfied
and G_C', as defined in (b), is a subset of G_C.
Return C

Combining the MCDs : The algorithm now considers all possible combinations of MCDs, and a conjunctive rewriting of the query for each valid combination using the property; Given Q, V, and MCDs ÂČC, the only combinations of MCDs that can result in *non-redundant rewritings* of Q are of the form $C_1, \cdots, C_l$, where:

1. $G_{C_1} \cup \cdots \cup G_{C_l} = Subgoals(Q)$, and
2. for every $i \neq j$, $G_{C_1} \cup \cdots \cup G_{C_j} = \emptyset$

procedure combineMCDs(C**)** /* C are MCDs formed by the first stage of the algorithm.*/
/* Each MCD has the form $((h_C, V(\bar{Y})_C, \phi_C, G_C, EC_C))$ */
Given a set of MCDs, $C_1, \cdots, C_n$, we define the function EC on $Vars(Q)$ as follows:
If for $i \neq j$, $EC_{\phi_i}(x) \neq EC_{\phi_j}(x)$, define ECC(x) to be one of them arbitrarily but consistently
across all y for which $EC_{\phi_i}(y) = EC_{\phi_i}(x)$
Let $Answer = \emptyset$
For every subset $C_1, \cdots, C_n$ of C s.t. $G_{C_1} \cup \cdots \cup G_{C_n} = subgoals(Q)$and for every $i \neq j$, $G_{C_i} \cap G_{C_j} = \emptyset$
Define a mapping ψ_i on the $\bar{Y_i}$'s as follows:
If there exists a variable $x \in Q$ such that $\phi(x) = y$
$\psi_i(y) = x$
Else
ψ_i is a fresh copy of y
Create the conjunctive rewriting $Q'(EC(\bar{X})) : -V_{C_1}(EC(\psi_1(\bar{Y_{C1}}))), \cdots, V_{C_n}(EC(\psi_n(\bar{Y_{C_n}})))$
Add Q'to Answer.
Return Answer.

While combining MCDs C_1 and C_2, both having x in their domain, either both C_1 and C_2 should map x to the same constant or one of them should map x to a constant and the other one should map x to a distinguished variable in the view. Theorem summarizing MiniCon can be stated as

- *Given a conjunctive query Q and conjunctive views V, both without comparison predicates or constants, the MiniCon algorithm produces the union of conjunctive queries that is a maximally-contained rewriting of Q using V*. The running time of MiniCon is $O(n * m * M)^n$, where n is the number of subgoals in the query, m is the maximal number of subgoals in a view, and M is the number of views.

The experimental results proved that the MiniCon algorithm can scale up (of the order thousands of views) and is far more efficient than the other two algorithms. The finding of the experiments is summarized in the table below.

Query type	Distinguished	No of subgoals	No of views
Chain	All	3	45
Chain	All	12	3
Chain	Two	5	9225
Chain	Two	99	115
Star	Non Joined	5	12235
Star	Non Joined	99	35
Star	Joined	10	4520
Star	Joined	99	75

This algorithm scales to thousands of views, which has not been possible with the previous algorithms. The bucket algorithm, in particular performed the worst in all cases. MiniCon algorithm managed to outperform the inverse rules algorithm in all cases. The unpredictability in performance of the inverse-rules algorithm is highlighted.

Extension for Comparison predicates: The MiniCon algorithm has been considered using simple conjunctive queries and views and without comparison predicates. An extension to the MiniCon algorithm is presented to handle the case of comparison predicates. This extension claims to always find only correct rewritings. Also , for most of the common cases, it claims to find the maximally-contained rewriting in many of the common cases (of comparison predicates).

PROJECT CONTRIBUTION

Contribution - Extension of MiniCon algorithm for Schemas with functional dependencies

Algorithms like bucket and Minicon for rewriting queries using views have mostly focused on conjunctive queries. Other algorithms in literature considers conjunctive queries with comparison predicates, queries and views with grouping and aggregation, OQL[7] queries. This problem of answering queries using views has been considered for schemas with functional and inclusion dependencies, languages that query both data and schema and disjunctive views. (Pottinger & Halevy, 2001) hints at a possible extension of the MiniCon algorithm for these cases. This project extends the Minicon algorithm for functional dependencies.

(Hong et al., 2005) states that the presence of functional dependencies has not been considered by existing algorithms for answering queries using views. For example, MiniCon algorithm can-

[7]OQL is a SQL-like query language to query Java heap

not handle the case which involves functional dependencies between its relations and will subsequently miss correct query rewritings. (Hong et al., 2005) presents an extension of the MiniCon algorithm using the concept of joint views. It claims that this extension will retain the main properties of the Minicon, especially its efficiency and scaling properties of other existing algorithms.

As part of this project, the MiniCon algorithm has first been implemented and then this extension for Minicon, as suggested in (Hong et al., 2005) has been implemented.

Functional Dependency : A functional dependency $r : a_1, \cdots, a_n \rightarrow b$ in the mediated schema, where a1, ..., an and b refer to attributes in the relation r, states that for every two tuples t and u in r if $t.a_i = u.a_i$ for $i = 1, ..., n$, then $t.b = u.b$

Consider a mediated schema for a student S who is enrolled in a degree program P in year Y. Department D is related to the program code C for a degree program P. Let us assume that there is a dependency between student and the degree program as well as between the student and the year (i.e. a student can take only one program and in only one year). Also, assume a degree program is uniquely associated with its program code and the department. Following functional dependencies will now exist in the mediated schema as per the assumptions:

$student : S \rightarrow P, S \rightarrow Y \qquad taught : P \rightarrow D \qquad program : P \rightarrow C$

Now consider following three views:

$v1(S, Y, D) : -student(S, P, Y), taught(P, D)$

$v2(S, P) : -student(S, P, Y)$

$v3(P, C) : -program(P, C)$

For a query such as *"Which degree program a student takes and in which year the student is?"* may be expressed as follows:

$q(S, P, Y) : -student(S, P, Y)$

A valid rewriting using above three views will be

$q'(S', P', Y') : -v1(S', Y', D'), v2(S', P')$

because the functional dependencies S → P and S → Y present. MiniCon algorithm is run for the same query; it cannot map query subgoal in q with view v1 since the distinguished variable P cannot be mapped to the distinguished variables in v1. From the FormMCD() procedure mentioned in the last section, while creating the MCD, Clause C1 of Property 1 fails. So v1, even though correctly forms the rewriting, is missed by Minicon. The workaround proposed by (Hong et al., 2005) suggests a ***joint view*** that has all the distinguished variables in either of its views as its distinguished variables, and all the subgoals in either of the views as its subgoals.

Minicon Extension: Forming MCDs over Joint Views : The joint view provides all the distinguished variables that the distinguished variables in the query subgoal can be mapped to. For the above-stated example, (Hong et al., 2005) proposes a solution by creating a ***joint view*** v(1,2) of v1 and v2 which has all the distinguished variables in either v1 or v2 as its distinguished variables. Also all the subgoals of the query in either v1 or v2 as its subgoals. This joint view still satisfies Clause C1 of Property 1, so MiniCon can successfully generate MCD for q over v(1,2) covering the only subgoal in q.

Hence, a non-redundant rewriting of q possible from MCD now
$V_1(S,Y,D) :- student(S,P,Y), taught(P,D)$ $V_2(S,P) :- student(S,P,Y)$
Joint view $V_{1,2}(S,Y,D,P) :- student(S,P,Y), taught(P,D)$

Forming MCDs over Joint Views: Given a set of views $v_1(\bar{X_1}), \cdots, v_n(\bar{X_n})$, a joint view, $v_{1,2,...,n}(\bar{X})$, is defined as a single view having all the distinguished variables and subgoals from all the given views as its distinguished variables and its subgoals respectively.

Conditions: The subgoals in different views with the same predicate are unified to get a single subgoal. As a result, some variables in different views may be mapped to a representative variable in the joint view. This representative variable is preferred to be a distinguished variable where ever possible. To enforce this, a proposition is presented as follows:

Let $v_{1,2,...,n}(\bar{X})$ be the joint view of views $v_1(\bar{X_1}), \cdots, v_n(\bar{X_n})$. Given that there exists a set of variables $X_1, ..., X_m$, where $X_1, \cdots, X_m \in \bar{X_1}$, $X_1, \cdots, X_m \in \bar{X_2}$ and $X_1, \cdots, X_m \in \bar{X_n}$, and for any other variable X', where X'â ĹĹX_i for 1â Ľd'iâ Ľd'n, the functional dependency $X_1, ..., X_m \leftarrow X'$ holds in $v_i(\bar{X_i})$, then $v_{1,2,...,n}(\bar{X})$ is equivalent to the join of $v_1(\bar{X_1}), \cdots, v_n(\bar{X_n})$.

Combining MCDs over Joint Views : Once the correct MCDs are constructed, in the second phase of the extended MiniCon algorithm, valid combinations of MCDs should be formed to create conjunctive rewritings of the query. For creating the rewriting q, the extended MiniCon algorithm works almost the same as the second phase of the MiniCon algorithm, simply treating joint views as single views. In the last step of the extended MiniCon algorithm, it however needs to replace every joint view in q with its correct rewriting.

Please note that as part of algorithm implementation, only the first part of MiniCon (formMCD()) has been implemented (where this new extension is inserted), as the second part combineMCD() is similar to the original algorithm as explained above (except for converting the joint view back to original views as a last step).

Implementation of MiniCon algorithm and its extension

The MiniCon procedure FormMCD() is implemented in a single python class (*minicon.py*). Using *Main()* method, this calls to initialize a set of Queries and Views *(initializeQV(), citingPapersSchema(), studentSchema())*. Once *FormMCD*() is called, it internally calls *formMCDsforView*() to iterate over individual views for a given set of views, which in turn calls other functions of this class to construct MCD.

Extension: Before the FormMCD() procedure gets called, the set of views are iterated looking for functional dependencies in *createJointViews*() function in class (minicon.py). If a pair of views are found to contain functional dependency (this dependency is provided in Main method), *joinView*() method is called which returns a new Conjunctive Query representing the joint view. This will indirectly help later when check for cases for Clause 1 of Property 1 passes, allowing formation of MCD for functional dependency case. Other python class used is *SubGoals.py* or the individual atoms $e_1(\bar{X_1}), \cdots, e_n(\bar{X_n})$ in the body of the query as a python class. To represent a CQ which can encapsulate the SubGoals class, a *ConjQuery.py* class is used. Finally *MCD.py* is used to separately store the MCD information.

Limitations:1. Transitive dependencies are not taken into account in this extended version. The algorithm is only checked for single functional dependency (of form s->p,s->y). 2. This algorithm will be extremely difficult to scale given that the implementation methods have multiple loops and all the views are iterated everytime a functional dependency is to be inspected. 3. This version doesnot take into account merging of multiple views (n>2), it only merges pairs. It is expected that the running time of the algorithm will increase rapidly as number of views grow. 4.The algorithm has been tested on very limited cases, and is prone to multiple errors.

Results:

python minicon.py

Query is : q(S, P, Y) : student(S, P, Y)

Views are:

v1(S1, Y1, D1) : student(S1, P1, Y1), taught(P1, D1)

v2(S1, P1) : student(S1, P1, Y1)

v3(P1, C1) : program(P1, C1)

Joint views formed are:

v2v1(S1, P1, S1, Y1, D1) : taught(P1, D1), student(S1, P1, Y1)

The MCDs constructed are : 3

MCD = V(Y)c = v2v1(S1, P1, S1, Y1, D1) : taught(P1, D1), student(S1, P1, Y1) gc = student(S, P, Y) vsg = student(S1, P1, Y1) phi = S->S1

MCD = V(Y)c = v1(S1, Y1, D1) : student(S1, P1, Y1), taught(P1, D1) gc = student(S, P, Y) vsg = student(S1, P1, Y1) phi = S->S1

MCD = V(Y)c = v2(S1, P1) : student(S1, P1, Y1) gc = student(S, P, Y) vsg = student(S1, P1, Y1) phi = S->S1

Analysis: According to (Hong et al., 2005), the computational complexity of the extended algorithm is not expected to incur large additional runtime since there are no new loops introduced and the number of joint views formed will always be in order of n, n being the total number of subgoals in the query or in the views whichever higher (in the worst case). For the first phase, the running time may be some times of the running time of the MiniCon algorithm, whereas in the second phase, the run time is expected to be almost same because of the minimal changes to this part of MiniCon algorithm. But the algorithm implemented indicates this is not the case. It might be possible that there are better algorithmic ways to implement the same extension in a more efficient manner, but there will always be an upper bound on the run time complexity since for finding the views to be joined has to iterate over the powerset of the set of views.

The results above show that one joint view ($v2v1(S1, P1, S1, Y1, D1)$) is created for the student and taught relation as expected. Also this results in formation of three separate MCDs, with one MCD having the joint view itself. This will help in Minicon detecting the functional dependency and skipping the check for Property1 Clause1, and later allowing the correct rewriting (this part has not been implemented in this project). An additional overhead is to find the original views from the joint view during the combineMCD() procedure.

Implementation in Python: The python files are attached with this report.

REFERENCES

Hong, J., Liu, W., Bell, D., & Bai, Q. (2005). Answering queries using views in the presence of functional dependencies. In *Database: Enterprise, skills and innovation* (pp. 70–81). Springer.

Pottinger, R., & Halevy, A. (2001). Minicon: A scalable algorithm for answering queries using views. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, *10*(2-3), 182–198.