Rewriting Queries Using Views

Chen Li
University of California, Irvine, CA 92697
http://www.ics.uci.edu/∼chenli

DEFINITION

Given a query on a database schema and a set of views over the same schema, the problem of query rewriting is to find a way to answer the query using only the answers to the views. Rewriting algorithms aim at finding such rewritings efficiently, dealing with possible limited query-answering capabilities on the views, and producing rewritings that are efficient to execute.

HISTORICAL BACKGROUND

Query rewriting is one of the oldest problems in data management. Earlier studies focused on improving performance of query evaluation [9], since using materialized views can save the execution cost of a query. In 1995, Levy et al. [10] formally studied the problem and developed complexity results. The problem became increasingly more important due to new applications such as data integration, in which views are used widely to describe the semantics of the data at different sources and queries posed on the global schema. Many algorithms have been developed, including the bucket algorithm [11] and the inverse-rules algorithm [15, 7]. See [8] for an excellent survey.

SCIENTIFIC FUNDAMENTALS

Formally, a query Q_1 is *contained* in a query Q_2 if for each instance of their database, the answer to Q_1 is always a subset of that to Q_2. The queries are *equivalent* if they are contained in each other. Let T be a database schema, and $\mathcal{V}$ be a set of views on T. The *expansion* of a query P using the views in $\mathcal{V}$, denoted by P^{exp}, is obtained from P by replacing all the views in P with their corresponding base relations. Given a query Q on T, a query P is called a *contained rewriting* of query Q using $\mathcal{V}$ if P uses only the views in $\mathcal{V}$, and P^{exp} is contained in Q as queries. P is called an *equivalent rewriting* of Q using $\mathcal{V}$ if P^{exp} and Q are equivalent as queries.

Examples: Consider a database with the following three relations about students, courses, and course enrollments:

```
  Student(sid, name, dept);
  Course(cid, title, quarter);
  Take(sid, cid, grade).
```

Consider the following query on the database:

```
Query Q1:  SELECT C.title, T.grade
```

```
          FROM Student S, Take T, Course C
          WHERE S.dept = 'ee' AND S.sid = T.sid AND T.cid = C.cid;
```

The query asks for the titles of the courses taken by EE students and their grades. Queries and views are often written as conjunctive queries [4]. For instance, the above query can be rewritten as:

```
Q1(T, G) :- Student(S, N, ee), Take(S, C, G), Course(C, T, Q).
```

We use lower-case arguments (such as "ee") for constants, upper-case arguments (such as "T") for variables. The right-hand side of the symbol "`:-`" is the *body* of the query. It has three *subgoals*, each of which is an occurrence of a relation in the body. The constant "`ee`" in the first subgoal represents the selection condition. The variable `S` shared by the first two subgoals represents the join between the relations `Student` and `Take` on the student-id attribute. The variables `T` and `G` in the head of the query, which is the left-hand side of the symbol "`:-`", represent the final projected attributes.

Consider the following materialized views defined on the base tables:

```
Views: V1(S, N, D, C, G) :- Student(S, N, D), Take(S, C, G);
       V2(S, C, T)       :- Take(S, C, G), Course(C, T, Q).
```

The SQL statement for the view definition of $V1$ is the following:

```
CREATE VIEW V1 AS
          SELECT S.sid, S.name, S.dept, T.cid, T.grade
          FROM Student S, Take T
          WHERE S.sid = T.sid;
```

This view is the natural join of the relations `Student` and `Take`. Similarly, view $V2$ is the natural join of the relations `Take` and `Course`, except that the attributes about grades and quarters are dropped in the final results. The following is a rewriting of the query $Q1$ using the two views.

```
answer(T, G) :- V1(S, N, ee, C, G), V2(S, C, T).
```

This rewriting takes a natural join of the two views on the attributes of student ids and course ids, then does a projection on the title and grade attributes. This rewriting can always compute the answer to the query on every instance of the base tables. In particular, after replacing each view in the rewriting with the body of its definition, the rewriting becomes the following expansion:

```
answer(T, G) :- Student(S, N, ee), Take(S, C, G),
                Take(S, C, G'), Course(C, T, Q').
```

`G'` and `Q'` are fresh variables introduced during the replacements. This expansion is equivalent to the query, thus the rewriting is an equivalent rewriting of the query.

Now, assume in the definition of $V2$, there is another selection condition on the quarter attribute. The following is the view definition:

```
V2'(S, C, T) :- Take(S, C, G), Course(C, T, fall2006).
```

That is, it only includes the information about the courses offered in the fall quarter of 2006. If only views $V1$ and $V2'$ are given, then the following is a rewriting of the query $Q1$:

```
answer(T, G) :- V1(S, N, ee, C, G), V2'(S, C, T).
```

In particular, its expansion, which is obtained by replacing each view with the body of its definition, is the following:

```
answer(T, G) :- Student(S, N, ee), Take(S, C, G),
                Take(S, C, G'), Course(C, T, fall2006).
```

This expansion is contained in the original query, thus this rewriting is a contained rewriting of the query $Q1$. It is not an equivalent rewriting, since it does not include information about courses offered in other quarters. On the other hand, each fact in the answer to this rewriting is in the answer to the original query.

Suppose the view definition of $V2$ does not have the attribute about course ids. Then using this modified view and $V1$, there is no rewriting of the query, since the modified view does not have the course id to join with view $V1$. As another example, if the view definition of $V1$ does not keep the grade information, the following is the new view:

```
V1'(S, N, D, C) :- Student(S, N, D), Take(S, C, G).
```

Using this new view and the original view $V2$, there is no rewriting to answer the query, since the views do not provide any information about grades, which is requested by the query. All these examples show that, when deciding how to answer a query using views, it is important to consider the conditions in the query and the views, including their selections, joins, and projections.

Algorithms: There are two classes of algorithms for rewriting queries using views: the first one includes the bucket algorithm [11] and its variants, and the second one includes the inverse-rules algorithm [15, 7]. Notice that the number of possible rewritings of a query using views is exponential in the size of the query. Here the main idea of the bucket algorithm is explained using the running example, in which the query $Q1$ needs to be answered using the views $V1$ and $V2$. Its main idea is to reduce the search space of rewritings by considering each subgoal in the query separately, and deciding which views could be relevant to the query subgoal.

The bucket algorithm has two steps. In step 1, for each subgoal in the query, the algorithm considers each view definition, and checks if the body (definition) of the view also includes a subgoal that can be used to answer this query subgoal. For each view, if it includes a subgoal that can be unified with the query subgoal, and the query and the view are compatible after the unification, the corresponding head of the view definition is added to the bucket of this query subgoal. The following shows the buckets for the three query subgoals.

```
Student(S, N, ee):    {V1(S, N, ee, C', G')};
Take(S, C, G):        {V1(S, N', D', C, G)};
Course(C, T, Q):      {V2(S', C, T)}.
```

Each primed variable is a fresh variable introduced in the corresponding unification process. The bucket of the second query subgoal does not include the view $V2$ because the query subgoal requires the grade information be included in the answer, while the corresponding grade information in the view subgoal is not exported in the head of $V2$.

In step 2, the algorithm selects one view from each bucket, and combines the views from these buckets to construct a contained rewriting. The following is a contained rewriting:

```
Q1(T, G) :- V1(S, N, ee, C', G'), V1(S, N', D', C, G), V2(S', C, T).
```

The final output of the algorithm is the union of contained rewritings in order to maximize the set of answers to the query using the views, since these rewritings could produce different pieces of information.

One main advantage of the bucket algorithm is that it can prune those views that do not contribute to a condition in the query, thus it can reduce the number of candidate rewritings to be considered. One limitation of the algorithm is that each query subgoal introduces a view in a rewriting. For instance, in the example above, view $V1$ could be used to answer the first two query subgoals. But the algorithm needs to use three view instances in each candidate rewriting, which requires more postprocessing steps to simplify this rewriting. In addition, the algorithm does not use the fact that if a view can be used to cover a query subgoal using a view variable that is not exported in the head of the view, then the view has to cover all the query subgoals that use the corresponding query variable. Based on these observations, a new algorithm, called MiniCon, was developed to make the rewriting process significantly more efficient [14]. A similar idea was used in the shared-bucket-variable (SVB) algorithm [13].

In some cases, especially in the context of data integration, where a view is a description of the content at a data source, the views could have limited query capabilities. For instance, imagine the case where the view $V1$ above is a materialized table, such that it can be accessed only if a student id is provided to the table, and the table can return its information about that student id. The table does not accept arbitrary queries such as “return all records,” or “retrieve all information about students from the CS department.” These limitations on the views present new challenges for the development of query-rewriting algorithms. The problem in this setting was studied in [16]. It is shown that the the inverse-rules algorithm [7] can handle such restrictions with minor modifications.

Other algorithms have been developed to study variants of the query-rewriting problem. The CoreCover algorithm [1] was developed for the problem of generating an *efficient* equivalent rewriting efficiently. There was also a study [2] for the case where the query and the views can have comparison conditions such as `salary > 30K` and `year <= 2004`. The work in [6] studied how to compute a set of views with a minimal size to compute the answers to a set of queries. In some settings, applications need to find a rewriting called “maximally-contained rewriting,” which can compute the maximal set of answers to the query using the views. The problem is also different depending on whether the closed-world assumption is taken (as in data warehousing, in which each materialized view is assumed to include all the facts satisfying the view definition) or the open-word assumption is taken (as in data integration, in which each view includes a subset of the facts satisfying the view definition). In the literature there is another related problem called “query answering.” See [3] for a comparison between “query rewriting” and “query answering.”

KEY APPLICATIONS

The problem of rewriting queries using views is related to many data-management applications, including information integration [18, 12], data warehousing [17], and query optimization [5].

CROSS REFERENCES

Answering queries using views, Query optimization, Data integration, Data warehousing, Query containment, Global-as-view (GAV), Local-as-views (LAV), Open-world assumption (OWA), Closed-world assumption (CWA).

RECOMMENDED READING

See the references.

Recommended Reading

[1] F. Afrati, C. Li, and J. D. Ullman. Generating efficient plans using views. In *SIGMOD*, pages 319–330, 2001.

[2] F. N. Afrati, C. Li, and P. Mitra. Answering queries using views with arithmetic comparisons. In *PODS*, pages 209–220, 2002.

[3] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. View-based query processing: On the relationship between rewriting, answering and losslessness. In *ICDT*, pages 321–336, 2005.

[4] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[5] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[6] R. Chirkova and C. Li. Materializing views with minimal size to answer queries. In *PODS*, pages 38–48, 2003.

[7] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *PODS*, pages 109–116, 1997.

[8] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[9] P.-Å. Larson and H. Z. Yang. Computing queries from derived relations. In *VLDB*, pages 259–269, 1985.

[10] A. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.

[11] A. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB*, pages 251–262, 1996.

[12] C. Li. Query processing and optimization in information-integration systems. *Ph.D. Thesis, Computer Science Dept., Stanford Univ.*, 2001.

[13] P. Mitra. An algorithm for answering queries efficiently using views. In *The 12th Australasian database conference*, pages 99–106, 2001.

[14] R. Pottinger and A. Levy. A scalable algorithm for answering queries using views. In *Proc. of VLDB*, 2000.

[15] X. Qian. Query folding. In *ICDE*, pages 48–55, 1996.

[16] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, pages 105–112, 1995.

[17] D. Theodoratos and T. Sellis. Data warehouse configuration. In *Proc. of VLDB*, pages 126–135, 1997.

[18] J. D. Ullman. Information integration using logical views. In *ICDT*, pages 19–40, 1997.