An Algorithm for Answering Queries Efficiently Using Views

Prasenjit Mitra
Department of Electrical Engineering
Stanford University
Stanford, CA, 94305, U.S.A.
mitra@db.stanford.edu

Abstract Algorithms for answering queries using views have been suggested and used to integrate information from multiple sources. Each source exports a view of the data it contains. A query answering algorithm needs to select which sources to consult in order to construct the solution. This is done by reformulating the query in terms of the views. The bucket algorithm has been predominantly used to reformulate queries. It involves generating candidate solutions and testing their validity to reject the irrelevant ones. The validation is performed using an expensive containment test for conjunctive queries. Scalability of such algorithms is largely dependent upon the efficiency with which the irelevant solutions can be pruned out before performing the costly validation. In this paper, an algorithm is highlighted that eliminates the necessity of performing a conjunctive query containment test and significantly reduces the number of irrelevant candidate solutions to be considered. Experimental results prove the superiority of this algorithm over previous solutions.

1 Introduction

Materialized views have been used in traditional databases to answer queries efficiently. Information sources on the world-wide web provide a *view* of their underlying data to the user. Queries posed by end-users can, often, not be answered from a single source but require consulting multiple sources. Integration of information from multiple sources in order to answer such queries is performed using these views.

Views and queries are represented using the logical rule notation [1]. A view is expressed as a rule whose body is the conjunction of *logical global predicates* that represent global concepts across sources. The view, *CarsFS* listing cars that are for sale, in Example 1, consists of tuples that can be obtained as the conjunction of the global predicates *car* and *stocks*.

```
Example 1:
CarsFS( Vehidno, Retailprice, Dealer ) :-
    car( Vehidno, Make, Model, Retailprice)
  & stocks( Vehidno, Dealer )
```

A view is not exhaustive with respect to a predicate that appears in its body, i.e. the view CarFS is not guaranteed to contain information about all cars that are at a dealer's shop but might often contain information about some cars in some selected dealers' shops.

User queries are also expressed as conjunctions of global predicates. Answering queries from the data available in the information sources requires rewriting the queries as the conjunction of the available views. Computing the answer to the query then involves computing a join of the data from the relevant views. Computing such joins is costly and therefore, mandates that the query be rewritten using the *minimal* number of views, i.e., if any view is taken out from the solution rewriting, it will no longer satisfy the query.

¡ formally define the problem ¿

Several algorithms have been developed to solve this problem. Notable among them are the *bucket algorithm* proposed in the Information Manifold system [?] and the *inverse rules algorithm* designed to answer recursive queries as part of the Infomaster system [?].

The Bucket Algorithm

The bucket algorithm attempts to rewrite a query using the materialized views instead of the logical schema predicates. The algorithm proceeds in two stages. Initially, a bucket is created for each query subgoal. A view is put in the bucket corresponding to a subgoal if that subgoal can be unified with a subgoal in the view definition. If the view contains multiple subgoals that can be unified with the subgoal from the query, the view occurs multiple times in the bucket of that subgoal. In the second stage of the algorithm, candidate solutions are generated by picking one conjunct (view) from each bucket. The candidate solutions are then verified using conjunctive query containment tests to see that they are indeed contained in the conjunctive query.

The number of candidate solutions that the bucket algorithm considers is exponential in the number of subgoals in the query and the number of views. Considering an exponential number of candidate solutions is unavoidable because potentially all the candidate solutions might be valid solutions. However, the bucket algorithm overlooks the fact that this number can be efficiently pruned by eliminating irrelevant views before putting them in the bucket. Furthermore, the verification of candidate solutions, in the bucket algorithm, is performed using the conjunctive query containment test which is exponential in the number of subgoals in the query and those in the views. Such a query containment test can be also be avoided by careful construction of the buckets.

¡ talk about the expansion and unique mapping ¿

This paper highlights a Simplified Bucket(SB) algorithm that reduces the number of irrelevant views by performing a few simple checks before putting a view in a bucket. The most significant performance gains are obtained by avoiding the exponential conjunctive query containment test. Today, information integration involves consulting a large number of sources. Therefore, the ability to prune unnecessary solutions easily reduces the response time to (often frustrated) users, improves the scalability and enables us to handle the increasing number of sources.

2 The Simplified Bucket Algorithm

The simplified bucket algorithm has two major steps:

1. Bucket construction by subgoal unification.
2. Candidate solution construction and minimization of the number of views in candidate solutions

2.1 Bucket Construction

A view is put in the bucket of a query subgoal if a view subgoal can be *unified* with it and a number of other conditions, as described below, hold. In the SB algorithm the views have to pass a more stringent test, before being put in a bucket, than in the bucket algorithm. This helps in weeding out irrelevant solutions and speeds up the solution generation and minimization stage.

One of the deficiencies of the bucket algorithm can be understood by considering the following query.

```
Example2:
CarsFS( Vehidno, Retailprice, Dealer ) :-
    Car( Vehidno, Make, Model, Retailprice)
  & Stocks( Vehidno, Dealer )
and the views:
Cars( Make, Model, Retailprice ) :-
    Car( Vehidno, Make, Model, Retailprice )
DealerInfo( Vehidno, Dealer ) :-
    Stocks( Vehidno, Dealer )
```

The bucket algorithm is able to unify the subgoals in the body of the query with those in the views and thus generates candidate solutions involving the views Cars and DealerInfo. It then tries to check that the candidate solutions are contained in the given query. The query CarFS cannot be rewritten using Cars and DealerInfo because the Cars view does not export the vehicle identification numbers, Vehidno which is required for constructing a tuple for CarFS. The candidate solution CarsFS(V, RP, D) :- Cars(M, Mo, RP) DealerInfo(V, D) is not adequate since obviously with this definition we can not ensure that the retail price Cars exports corresponds to the vehicle (identification number) that the dealer carries (and is obtained from the DealerInfo view) in question. The bucket algorithm rejects this obviously erroneous solution only after conducting an expensive conjunctive query containment test. The next section details a technique which will identify the limitations of the view Cars and not generate any candidate solutions. The proposed algorithm, thus, generates a smaller (but complete) set of candidate solutions by eliminating bad solutions up front and hence reduces the effort that is expended in checking the containment of the bad candidate solutions in the query.

In order to understand the efficient pruning technique used in the SB algorithm let us define a few essential terms:

Definition 1 *A distinguished variable is a variable that occurs in the head of a rule. A distinguished variable is universally quantified.*

Definition 2 *A repeated variable is a variable that occurs more than once in the body of the rule.*

The problem with the bucket algorithm is that it does not consider whether the distinguished variables in the predicate of the query map to distinguished variables in the views. If this is not the case, (like Vehidno in Cars) then the view does not export that variable to the outside world. A tuple satisfying the query needs this value which is not possible to obtain using the view. Hence, the view should be in a bucket only if the distinguished variables appearing in the query subgoal it covers map to distinguished variables in the view during the unification of the subgoals (Lemma 1, item 1).

Unification of subgoals for the purpose of rewriting queries using views is performed provided a number of conditions hold. For each subgoal in the query a *single-subgoal bucket* is created. Views that pass the most stringent tests for unification are put in the single-coverage bucket.

2.1.1 Single-Subgoal Buckets

Let the query Q have a subgoal $g = p(Qv_1, Qv_2, ..., Qv_n)$. Let the view V have a subgoal $g' = p'(Vv_1, Vv_2, .., Vv'_n)$.

Proposition 1 *The subgoal g' can be unified with the subgoal g, for the purposes of query rewriting, if the following conditions hold:*

1. $p = p'$ *and* $n = n'$
2. *If* Qv_i *is distinguished,* Vv_i *is also distinguished.*

Let μ be a mapping from an element in the set of query variables Qv to an element from the set of view variables Vv. Let λ be a mapping that is the inverse of μ, i.e., if $\mu(X) = Y$, then $\lambda(Y) = X$.

Proposition 2 *The view* V *is said to* cover *the subgoal* g*, if the following conditions hold:*

1. *g and g' can be unified as defined above.*
2. *Let* $\lambda(Vv) = Qv$*. If* Vv *is undistinguished then* Qv *is not a repeated variable*

A view V is put in the single-subgoal bucket for a subgoal g if it covers g.

2.1.2 Multiple-Subgoal Bucket

Due to repeated variables in the query, certain views cannot be put in the single-subgoal bucket but need to be included in a *multiple-subgoal bucket*. Let us suppose that a query variable, Q_v, occurs in two subgoals g and g' in a query. Let g and g' be covered by views V_1 and V_2 and Q_v maps to variables Vv_1 and Vv_2. A query variable must map to a unique variable in the expansion of a solution. Therefore, it needs to be ensured that Vv_1 and Vv_2 are substituted by the same variable in the expansion of the solution. If Vv_1 or Vv_2 are undistinguished, there is no way in the query rewriting to make them the same if V_1 and V_2 are distinct views. The subgoals g and g' need to be covered by the same view. To handle these cases we have multi-subgoal buckets. If a view is chosen from a multi-subgoal bucket in the solution it will cover all the subgoals in the query whose bucket it is from at the same time. We cannot chosen to cover a few of those multiple subgoals with this view and have other views covering the rest of those subgoals.

A multiple-subgoal bucket represents multiple subgoals.

Proposition 3 *The view V is said to* multiply cover *the subgoals g_i, if the following conditions hold:*

1. *Each g_i can be unified with a view subgoal g'_i.*
2. *Let $\mu(Qv_i) = Vv_i, .., Vv_k$ where Qv_i is a variable occuring in any g_i Then either*
   All Vv_x is distinguished,
   or all Vv_x is undistinguished and $Vv_i = .. = Vv_k$.

If a view multiply covers several subgoals g_i, it is put in the multiple-subgoal bucket corresponding to the subgoals.

2.2 Solution Generation and Minimization

2.2.1 Candidate Solution Generation

A candidate solution is generated by covering all subgoals in the query by exactly one view chosen from a bucket representing it. A view is chosen from either a single-subgoal or multiple-subgoal bucket and this view *covers* the subgoal(s) that the bucket represents. In a candidate solution, each query subgoal is covered by one and exactly one view. All possible candidate solutions are generated.

2.2.2 Minimization of Solutions

A solution might need to be minimized if the same view occurs more than once. The SB algorithm tries to minimize the solution by *coalescing* the views, if possible. The results of such coalescing can be cached and reused since the same views with similar bindings might occur in another candidate solution.

Proposition 4 *Two occurences of the same view $V(v_1, v_2, .., v_n)$ and $V(v'_1, v'_2, .., v'_n)$ can be coalesced if*

1. *$v_i = v'_i$*
2. *If both v_i and v'_i are undistinguished and at least one of them is non-repeated.*

3 Implementation and Performance Analysis

The SB algorithm performs additional checks up front while putting views in the bucket and avoids the costly containment test for conjunctive queries.

Distinguished and repeated variables can be determined by simply sorting the variables in the rules and making a linear pass thereafter on the sorted array of variables. Then the view subgoals can be tested as indicated above and if suitable inserted into the single-subgoal buckets.

In order to construct the multiple-subgoal bucket we need to find the sets of query subgoals that have variables in common among them. This is achieved by constructing a common-variable graph and finding out all connected components in them.

The common-variable graph $CVG = (V, E)$

- has a vertex $v_i \in V$ representing a subgoal g_i in the query if g_i has at least one repeated variable.
- and has an edge $(v_i, v_j) \in E$ if the subgoals g_i and g_j have at least one variable in common.

A bucket is then created for each connected component in the graph. The CVG can be constructed using a single pass over the sorted array of query variables. Its connected components can be determined using a standard algorithm. If a view multiply-covers the subgoals in a component, it is inserted into the multiple-subgoal buckets.

Clearly, the checks that the SB algorithm performs involve polynomial running times with respect to the number of variables in the query and those in the view. This avoids an exponential conjunctive query containment test that needs to be performed in the bucket algorithm.

4 Conclusion

In the current work, a simplified bucket algorithm has been presented that avoids the costly conjunctive query containment test. The construction of buckets has been streamlined generating fewer useless candidate solutions and thus speeds up the algorithm. The SB algorithm considerably speeds up the query reformulation stage of an information integration system. This enables the construction of scalable integration systems that can handle large amounts of information. In today's world, with huge amounts of information obtainable from the World-Wide Web, an efficient query rewriting algorithm is necessary - the SB algorithm fills this void.

References

[1] J. D. Ullman. Computer Science Press, New York, 1988.