Database Reformulation

Rada Chirkova and Michael R. Genesereth

Abstract

We introduce an approach to view materialization which is based on relational reformulation. In this context, we present a method for automatically discovering views of the stored relations that are not already defined yet present valuable opportunities for materialization. The proposed reformulation algorithm works off-line and consists of two stages. At the first stage, Datalog rules relevant to the queries are rewritten in a way that yields predicates of smaller arity, or merges several predicates into one. This stage produces a set of candidates for materialization. At the second stage, decisions are made on which candidates to materialize; the criterion here is storage space available for the data. This method makes use of additional knowledge about relations, namely of the information about functional dependencies in relations. Reformulations produced by this method preserve answers to all queries and, under certain conditions, improve query-processing performance. Moreover, the reformulation algorithm is polynomial in time.

Introduction

Abstraction and reformulation are broad categories of techniques applied in many domains. One of the goals of abstraction and reformulation is to reduce the computational complexity of problems to solve.

We present an application of abstraction and reformulation in the database domain, to the problem of reducing query processing time. Although this problem is formulated in the database context, it is easy to generalize, since broad classes of problems can be viewed and solved as database problems.

We consider the problem of computing answers to queries using materialization of views. In modern databases, it is common to define views of stored relations and to write queries in terms of these defined views. There are many advantages to defining views, including ease of specification, ease of maintenance, and security. The downside of views is that they must be computed during processing of every query involving views. One common solution to this problem is to materialize (in other words, precompute and store) intermediate views and discard the stored relations, in cases where they are no longer needed.

Recent research in materialization in the database community aimed at query optimiziation, e.g. (Gupta 97), (Harinarayan et al 96), (Labio et al 97), has been concentrated on materializing queries or already defined views using quantitative information about a specific database.

In contrast, our objective is to choose relations to materialize from among views which are not already defined by the user.

We introduce a notion of "useful" reformulation; such a reformulation reduces the complexity of at least one query in the set while preserving the set of answers to each query in the set, at the expense of at most linear increase in the storage space taken by the data.

We introduce an approach to view materialization which is based on relational reformulation. In this context, we present a method for automatically identifying views of the stored relations that are not already defined yet present valuable opportunities for materialization. The reformulations produced by this method don't degrade the runtime of any query in the given set of queries. Moreover, under certain circumstances the method produces reformulations which improve processing times of queries in the set. The reformulation algorithm works off-line and is polynomial in time.

In the remainder of the paper, we investigate the problem of relational reformulation of Datalog databases. First, we present an example and discuss how our approach enables us to compute faster the view in this example. We proceed to list some background definitions. Then we give a definition of reformulation, of correct reformulation, as well as a definition of sound reformulation which improves asymptotic efficiency of a set of queries (i.e. "useful" reformulation). Next, we introduce an algorithm that applies a particular type of relational reformulation; this algorithm uses in part the technique of thread variable elimination. We also discuss an extended example and formulate some theorems which provide a theoretical justification for the algorithm. Later on, we discuss related work and, finally, summarize our results and outline future directions of research.

Reformulation Example

Let's consider an airline database example. Suppose one of the views is all airlines which service pairs of directly connected (possibly by other airlines) airports; the bases of all the airlines of interest are also included into the view.

Let's assume the body of this view consists only of stored relations:

view (Airline, Base) :-
    services (Airline, AirportOne),
    services (Airline, AirportTwo),
    directConnection (AirportOne, AirportTwo),
    base (Airline, Base).

This means that airline Airline is included in the view relation if there are two airports, AirportOne and AirportTwo, which are destinations of Airline and, at the same time, there is a direct connection between Airport One and AirportTwo; other information in the view relation consists of all airports (Base) which are bases of the airline Airline.

Imagine a large database where the size of the airport domain is the same as the size of the airline domain; suppose that the size of each domain is N, where N is a large number. It is reasonable to assume that the size of all the stored binary relations is of the order of N^2; therefore, the size of the database itself is some multiple of N^2.

The size of the view relation must be of the order of N^2, too, since the head of the view is a binary relation. The complexity of calculating the view relation is, however, of the order of N^3. This complexity has been computed as follows: one of the ways of constructing the view relation is to look through the whole services relation, whose size is O(N^2), to obtain all airports AirportOne; for each value of AirportOne taken from services, we must look through up to N values of AirportTwo in the relation directConnection; then we look again through the services relation to see which of the values of AirportTwo are there together with Airline; finally, for each value of Airline we must also look through up to N values of Base in the relation base.

However, using the proposed method, we reformulate the body of the view, so that it now looks as follows:

view (Airline, Base) :-
    sds (Airline),
    base (Airline, Base).

Here, the new relation sds (Airline) includes all the airlines which have destinations at a pair of directly connected airports (the identities of the airports are irrelevant for the view) and is calculated using the rule:

sds (Airline) :-
    services (Airline, AirportOne),
    services (AirportOne, AirportTwo),
    directConnection (AirportOne, AirportTwo).

Since sds is a unary relation, its materialization reduces the complexity of computing the view from N^3 to N^2 (see above for an explanation for the original view), where N is the size of the Airline domain and of the Airport domain. This is the maximal possible complexity reduction, since the size of the view itself is also of the order of N^2. If the body of sds occurs in the definitions of multiple frequent queries to the database, all these queries will be computed faster if sds is materialized.

On the other hand, the size of sds, if it is materialized, is of the order of N, which means that the increase in database storage space, for this particular database (whose size is already of the order of N^2), caused by the materialization of sds, is insignificant.

Note that we can now abstract from specific numbers and estimate the size of all relations, as well as the complexity of computing the view, in terms of the parameter N which we choose to be the total number of objects (i.e. the size of the UOD) in the database. Then, for each instantiation of the given database scheme, the asymptotic estimate of the view processing time, as well as the estimate of the size of the materialized relation, will still be true if expressed using this parameter. This is the major point of our approach because our method allow us to predict changes in runtime complexity as well as in the asymptotic size of the database after the materialization.

Background Definitions

The definitions in this section are adapted from (Ullman 88).

We consider descriptions of (parts of) the world which are built from relations. Members of a relation are called tuples. The (formal) arity of a relation is the number of attributes in the relation.

A relation may have one or more functional dependencies between its attributes; there is a functional dependency from arguments A_{i1}, A_{i2}, ..., A_{ik} to argument A_j if the values of A_{i1}, A_{i2}, ..., A_{ik} uniquely determines the value of A_j.

The set of attribute names for a relation is called the relation scheme. The collection of relation schemes used to represent information is called a (relational) database scheme, and the current values of the corresponding relations form the (relational) database.

The universe of discourse (UOD) of a database is the union of the sets of values of attributes, among all the stored relations in the database.

The storage space of a database is the amount of space needed to store all the tuples of the database. We will express the database storage space in terms of the total number of values of attributes stored in the database.

Datalog is a language for writing logical statements about the world, i.e. rules, whose underlying mathematical model of data is the relational model. Predicate symbols in Datalog denote relations. There are two ways relations can be defined in Datalog. A predicate whose relation is stored in the database is called an extensional database (EDB) predicate, while one defined by logical rules is called an intensional database (IDB) predicate. There is a fixed set of predicates which are called queries. A collection of predicates is termed a logic program.

IDB predicates can be computed in two ways. The first way uses operations of relational algebra (select, join, project etc). The second way is Prolog-style; it evaluates predicates in the top-down fashion, starting from IDB relations. Both methods are described at length in (Ullman 88).

A materialization of a predicate is a process of calculating all the tuples of a predicate relation and of storing these tuples in the database.

Definition of Reformulation

Definition (this is a slight modification of a definition introduced in (Levy et al 95)). A *rewriting of a view* V using predicates P $= \{P_1, ...P_m\}$ is a view V' if:

1. V and V' are equivalent (i.e. their relations are the same in any given database);
2. all the predicates in the body of V' are in P.

Definition A *reformulation of a theory*, which consists of a set of EDB predicates (database scheme) and of a logic program which contains rules for the remaining predicates, is a set of new EDB predicates (which are not already stored before the reformulation) and of rewritings of the views using the new predicates.

We define out cost model based on a parameter which we choose to be the size of the UOD in an instantiation of the given database scheme. The size N of the UOD is an asymptotic measure of both database storage space and query processing time. We assume that the domain of each attribute of each relation is comparable to N and that each relations is indexed on each of its attributes. Under these assumptions, we

1. define the *asymptotic storage space of the database* in terms of N (e.g. a database which consists of a single binary relation without functional dependencies between its attributes takes space O(N^2)); the storage space of the database is the sum of the storage space for all its relations.
2. define *asymptotic query processing time* in terms of N; e.g. it takes O(N^3) time to calculate (if there are no functional dependencies in either p or r) the following query:

   q(X,Y) :- p(X,Z), r(Z,Y).

   because, for each of the O(N^2) tuples of p, we look up O(N) tuples of r with the corresponding value of Z.

Definition A *reformulation* of a logic program with a fixed set of queries *is sound and improves asymptotic efficiency of a set of queries* if:

1. the set of answers to each query in the set of queries doesn't change after the reformulation; in addition,
2. there is at least one query in the set of queries whose asymptotic processing time after the reformulation is of lower order than its asymptotic processing time before the reformulation,
3. and the asymptotic storage space of the database after the reformulation is of the same order as before the reformulation, i.e. doesn't increase more than linearly.

The Reformulation Algorithm

Preliminary observations and definitions

A polynomial-time algorithm introduced in this paper reformulates a logic program, with a fixed set of queries, in such a way that the following conditions hold:

- the asymptotic processing time of any query in the set of queries cannot increase (i.e. the reformulation is correct);
- the asymptotic storage space of the database after the reformulation is of the same order as before the reformulation;
- under certain conditions, the output of the algorithm is sound and improves asymptotic efficiency of the given set of queries.

For an extended example which will show all the stages of the algorithm, see next section.

Definition Effective arity:

- the *effective arity of a relation* is the number of its attributes which are not on the right side of any functional dependency for this relation;
- the *effective arity of a database scheme* is the maximal effective arity among the EDB relations;
- the *clausal arity of a literal* in the body of a rule is the minimum of the effective arity of the corresponding relations, on the one hand, and of the number of different variables in this literal, on the other hand.

For the examples, see next section.

Justification

- the effective arity of a relation determines the asymptotic storage space required for the relation;
- the clausal arity of a literal shows the asymptotic storage space required for the materialization of the corresponding relation.

Theorem *Materialization criterion*: the only relations which can be materialized in a reformulation which is sound and improves asymptotic efficiency of a set of queries, are those whose clausal arity doesn't exceed the effective arity of the database scheme.

Input of the algorithm

We apply the algorithm to Datalog databases where the only rules are Horn rules, i.e. there are no negated literals in bodies of rules.

The input to the algorithm consists of:

- the set R of all the relations, together with their (formal) arities; we may also be given (and make good use of) the information about the functional dependencies in the relations in R;
- the database scheme D which is a subset of R and consists of EDB predicates;
- the set P of IDB predicates, P = R - D, and Datalog rules for all the predicates in P;
- the set of queries Q, Q $\subseteq P$.

Output and runtime of the algorithm

If the algorithm is applicable to the given input, then, after it has been run on each input predicate, the algorithm outputs:

1. The set M of new predicates to materialize:
   - the relations underlying these predicates are added to the database scheme;
   - the rules for these predicates are used only during the actual materialization, in the context of a specific database;
2. A rewriting of one or more predicates using the set M;
3. The degree of polynomial reduction of the input's asymptotic runtime complexity.

If the algorithm is not applicable to the given input, it outputs a corresponding message.

Theorem The runtime of the proposed algorithm is polynomial in the number of rules, in the number of literals in bodies of rules, and in the arity of predicates.

Proof By construction of the algorithm.

Stages and steps of the algorithm

The algorithm is run on one predicate at a time and processes each predicate in two stages.

The **first stage of the algorithm** is the reformulation of Datalog rules (see below the phases of the reformulation stage). It consists of finding certain groupings in the body of the predicate which are used to construct Datalog rules for new predicates which are not already in the input (those are called *candidates for materialization*). Then, the clausal arity of each candidate for materialization is calculated, based on these rules.

The candidates for materialization are then used to rewrite the body of the predicate. After the rewriting, the asymptotic processing time for the predicte is calculated again, based on the clausal arity of all the candidates for materialization which are used in the rewriting.

Since the rules for all the candidates for materialization are nonrecursive by construction (as will be shown below), the dependency graphs between them and the input predicates are trees.

A *materialization tree* is a tree such that:

1. all its leaves are input predicates;
2. all its internal nodes are candidates for materialization;
3. there is a "parent-child" link from one node in the tree to another node in the same tree iff the predicate represented by the second node is in the body of the rule for the predicate represented by the first node.

After the first stage of the algorithm, there are as many materialization trees as there are candidates for materialization in the body of the last rewriting of the predicate under consideration.

The **second stage of the algorithm** - the materialization decision stage - consists of examining each materialization tree separately, starting from the root, and of building, from among the predicates represented by the internal nodes of the tree, the minimal set S of internal nodes to be materialized.

First, the recursive materialization-selecting algorithm works, for each materialization tree, to produce the minimal set of candidates for materialization, i.e. such a set that:

1. the reduction in predicate processing complexity is maximized, over all the predicates in this tree;
2. and the materialization of any member of the set doesn't violate the materialization criterion.

Second, for each materialization tree, we compare the asymptotic runtime of the predicate before any rewriting to the asymptotic runtime of the predicate rewritten using the minimal set S. The set S is actually materialized only if the processing complexity of the rewritten predicate is smaller than the original processing complexity of that predicate.

The reformulation stage of the algorithm

In the discussion below, we will need the notions of a head variable and of a nonhead variable of a clause. A head variable of a clause is the variable which belongs to the set of variables of the head predicate of the clause. A nonhead variable of a clause is the variable which belongs to the set of variables in the body of the clause but is not a head variable.

1. The disjunct reformulation phase: The goal of this phase is to discover whether there is a common part in the bodies of some clauses of the predicate and, if this is the case, to reduce the number of the predicate's clauses by creating a new predicate whose body is the disjunction of the remainders of these clauses.

The variable folding phase The variable folding phase of the reformulation algorithm runs on the body of one clause of a predicate at a time. This phase consists of four steps:

- **step 1:** discarding insignificant nonhead variables; this step is applicable if there is a nonhead variable which occurs in only one predicate in the body of the clause;
- **step 2:** merging predicates whose sets of variables are the same;
- **step 3:** predicate subsumption: if all the variables in the set V of variables of a predicate p in the clause body are nonhead variables, and V is a subset of the set of variables of another predicate r in the body of the same clause, and none of the variables in V occur in any other predicate in the clause body, then p may be subsumed by r;
- **step 4:** merging predicate clusters: if the union of the sets of nonhead variables of a group G of predicates in the clause body is disjoint with the union of the sets of nonhead variables of the remaining predicates in the clause body, then G is called a cluster and can be transformed into a single candidate for materialization.

Example

Consider a database scheme D which consists of four stored relations: relation w is unary, relations s and r are binary, and the arity of relation m equals four. The only functional dependency in D is from the first attribute of relation m to the rest of m's attributes. There are three IDB predicates: p, q, and t. Assume that there are no functional dependencies between the attributes of any of the underlying relations. The set of queries consists of only one element q; the rule for q is:

q(A,B,C) :- p(A,G,A), r(G,A), w(G), s(B,D),
m(D,H,C,D).
q(X,Y,Z) :- p(X,T,X), r(T,X), w(T), s(Y,W),
t(Z,W,R).

Thus, the input to the algorithm is:

- the set R of all the relations: R = $\{m, p, q, r, s, t, w\}$; we're also given the information about the functional dependencies in all the relations in R;
- the database scheme D for the subset $\{m, r, s, w\}$ of R;
- the set P of IDB predicates: P = $\{p, q, t\}$ (P = R - D), and Datalog rules for all the predicates in P (the rules for p or t are irrelevant for our purposes);
- the set of queries Q = $\{q\}$; Q is a subset of P.

Preliminary calculations

From the information on predicate arities and on functional dependencies we can conclude that the effective arities of m and w are equal to one, and the effective arities of both s and r are equal to two.

This means that the effective arity of the database scheme D is equal to O(N^2), in terms of the parameter N which is the size of a (hypothetical) database with the scheme D.

In the absence of functional dependencies, the effective arity of each of the IDB predicates (if materialized) would be equal to three: O(N^3).

It can be shown that the asymptotic complexity of processing the query q is equal to O(N^4).

The first (reformulation) stage of the algorithm

Disjunct reformulation: After renaming variables in the clauses of q with the renaming A $\leftarrow X, B \leftarrow Y, C \leftarrow Z, G \leftarrow T, D \leftarrow W, H \leftarrow R$,

we obtain:

q(X,Y,Z) :- p(X,T,X), r(T,X), w(T),
s(Y,W), m(W,R,Z,W).
q(X,Y,Z) :- p(X,T,X), r(T,X), w(T),
s(Y,W), t(Z,W,R).

The common part of the two clauses is p(X,T,X), r(T,X), w(T), s(Y,W); thus, we will introduce a new predicate

mt(Z,W,R) :- m(W,R,Z,W).
mt(Z,W,R) :- t(Z,W,R).

whose effective arity is equal to O(N^3) .

The query q is rewritten as follows:

q_1(X,Y,Z) :- p(X,T,X), r(T,X),
w(T), s(Y,W), mt(Z,W,R).

Variable folding: Step 2: merging predicates (p and r) whose sets of variables are the same:

pr(X,T) :- p(X,T,X), r(T,X).
q_2(X,Y,Z) :- pr(X,T), w(T), s(Y,W), mt(Z,W,R).

Step 3: predicate subsumption (pr and w):

prw(X,T) :- pr(X,T), w(T).
q_3(X,Y,Z) :- prw(X,T),
s(Y,W), mt(Z,W,R).

Step 4: merging predicate clusters:

here, the set $\{T\}$ of nonhead variables of the set $\{prw\}$ is disjoint with the set $\{W, R\}$ of nonhead variables of the remaining predicates $\{s, mt\}$ in the body of the clause. Thus, we can calculate these sets of predicates separately. In particular, we can merge s and mt:

smt(Y,Z,W,R) :- s(Y,W), mt(Z,W,R).
q_4(X,Y,Z) :- prw(X,T), smt(Y,Z,W,R).

Step 1: discarding insignificant nonhead variables:

pw(X) :- prw(X,T).
q_5(X,Y,Z) :- pw(X), smt(Y,Z,W,R).

Step 1: discarding insignificant nonhead variables:

st(Y,Z) :- smt(Y,Z,W,R). q_6(X,Y,Z) :- pw(X), st(Y,Z).

It can be shown that the asymptotic processing time of q has been reduced from O(N^4) to O(N^3).

The second (materialization decision) stage of the algorithm

Among the results of the first stage of the algorithm are two materialization trees, one for each of the candidates for materialization pw and st which are used in the final rewriting q_6 of the query q.

We see that both pw and st can be materialized since their effective arity (O(N) for pw and O(N^2) for st) doesn't exceed the effective arity O(N^2) of the original database scheme.

On the other hand, the materialization of pw and st reduces the asymptotic complexity of processing the query q from O(N^4) to O(N^3).

Thus, the output of the algorithm, which consists of:

- the set P' of new materialized predicates, P' is disjoint with R; P' = $\{pw, st\}$;
- the new database scheme D', D' = D ∪ P';
- a set Q' of rewritings of all the queries in the set Q in terms of predicates in P':
  Q' = $\{q(X,Y,Z) : -pw(X), st(Y,Z)\}$;

is a reformulation of the input which is sound and improves asymptotic efficiency of the given set of queries.

Theoretical Justification

We have proved the following theorems about the proposed algorithm:

Theorem After each stage, phase, and step of the proposed algorithm, the set of answers to each query remains unchanged.

Theorem If the algorithm is applicable to an input, then the output of the algorithm satisfies the definition of a reformulation which is sound and improves asymptotic efficiency of the given set of queries.

Related Work

Selecting relations for materialization is among important research problems in the context of database warehouses (see, e.g., (Widom 95)). This problem is solved in the database community for specific databases, based on quantitative information such as statistics on relative query frequencies.

In this context, (Labio et al 97) present an A* search based solution and heuristics for selecting views and indexes for materialization when the goal is to minimize the warehouse maintenance time; the problem of minimizing query response time is assumed to have been solved by materializing all the queries.

In (Harinarayan et al 96), the problem of selecting views for materialization is solved for databases built in the form of data cubes. This approach is further developed in (Gupta et al 97), which solves the problem of selecting indexes to materialize.

(Gupta 1997) has developed a theoretical framework for the general problem of selecting views to materialize in a fixed data warehouse; he presents polynomial time heuristics for solving the NP-hard probem of minimizing the sum of total query response time and total maintenance costs, for many general query classes.

In the abstraction and reformulation community, the problem of speeding up query processing by means of reformulation has been studied extensively. The research has been conducted mainly in the framework of the theory of abstractions developed in (Giunchiglia and Walsh 92), where abstractions are viewed as syntactic mappings between formulas of formal systems.

In this context, (Ellman 95) has developed a technique called "approximate domain abstraction", where constant symbols of a database are recursively collected into equivalence classes, resulting in a hierarchy of abstracted databases. The hierarchy is used to produce concise answers to queries. The speedup in query processing is achieved by pruning away large regions of the search space while generating query answers at upper levels of the database hierarchy.

(Levy et al 95) consider query rewritings that use materialized views; minimal and complete rewritings are also discussed. A polynomial-time algorithm for finding rewritings is presented; under certain conditions, the algorithm produces minimal rewritings.

(Levy and Sagiv 92) discuss an algorithm for optimizing recursive rules by removing redundant rules. Two types of redundant rules are considered: unreachable and irrelevant rules.

(Levy and Sagiv 95) consider the problem of using qualitative information - integrity constraints - to optimize the evaluation of queries. The paper presents a solution for a large class of queries in presence of recursion.

The paper by (Nayak and Levy 95) generalizes research in query reformulation by providing a semantic framework for abstractions. The authors introduce a new definition of abstraction: an abstraction is a model-level mapping of the original theory in conjunction with simplifying assumptions. A powerful class of (model-increasing) abstractions is studied in detail; the properties of this class help determine the cases when it is possible to build the strongest abstract theory that implements the intended abstraction.

Conclusions and Future Work

We considered the problem of reducing query processing time in databases. We showed that using relational reformulation in order to determine the set of supporting relations to be materialized is a promising approach to solving this problem.

We presented a method of automatically identifying views of the stored relations that are not already defined yet present valuable opportunities for materialization. Reformulations produced by this method preserve answers to all queries and, under certain conditions, asymptotically improve query-processing performance for all the databases representable by the given logic program. Moreover, the reformulation method is efficient to compute.

The research presented here is similar in spirit to the work by (Nayak and Levy 95) (see Related Work); however, we consider a different - iterative rather than strictly sequential - method for constructing model-increasing abstractions. We also address the issue of the computational benefits of using abstractions by trying to understand when model level abstractions lead to computational savings.

We intend to continue working on reformulation for materialization. In particular, we would like to refine the cost model; to consider in detail the problem of merging different materialization trees, with the objective of finding space-optimal materializations; and, finally, to explore the influence of database updates on the sets of answers to reformulated programs.

References

(Ellman 95) Ellman, T. (1995) Approximation and Abstraction Techniques for Generating Concise Answers to Database Queries. AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments. Stanford University, Stanford, California.

(Giunchiglia and Walsh 92) Giunchiglia, F., and Walsh, T. (1992) A theory of abstraction. Artificial Intelligence 57 (2-3), 323-389.

(Gupta 97) Himanshu Gupta (1997) Selection of Views to Materialize in a Data Warehouse. In Proceedings of the International Conference on Database Theory, Delphi, Greece.

(Gupta et al 97) Gupta, H., Harinarayan, V., Rajaraman, A., and Ullman, J. (1997) Index Selection in OLAP. In Proceedings of the International Conference on Data Engineering, Burmingham, U.K.

(Harinarayan et al 96) Harinarayan, V., Rajaraman, A., and Ullman, J. (1996) Implementing Data Cubes Efficiently. In Proceedings of ACM SIGMOD Intl. Conf. on Mngt. of Data, 205-216.

(Hobbs 85) Hobbs, J. (1985) Granularity. In Proceedings of IJCAI-85, Los Angeles, California.

(Labio et al 97) Labio, W.J., Quass, D., and Adelberg, B. (1997) Physical Database Design for Data Warehouses. In Proceedings of the International Conference of Data Engineering.

(Levy and Sagiv 92) Levy, A.Y., and Sagiv, Y. (1992) Constraints and Redundancy in Datalog. In Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS-92, San Diego, CA.

(Levy and Sagiv 95) Levy, A.Y., and Sagiv, Y. (1995) Semantic Query Optimization in Datalog Programs. In Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS-95, San Jose, California.

(Levy et al 95) Levy, A.Y., Mendelzon, A.O., Sagiv, Y., and Srivastava, D (1995) Answering Queries Using Views. In Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS-95, San Jose, California.

(Nayak and Levy 95) Nayak, P.P., and Levy, A.Y. (1995) A Semantic Theory of Abstractions. In Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI-95, Montreal, Canada.

(Subramanian 89) Subramanian, D. (1989) A Theory of Justified Reformulations. Ph.D. dissertation, Computer Science Department, Stanford University.

(Ullman 88) Ullman, J.D. (1988) Principles of Database and Knowledge-Base Systems. Computer Science Press.

(Widom 95) Widom, J. (1995) Research Problems in Data Warehousing. In Proc. 4th Int'l Conference on Information and Knowledge Management, CIKM-95.