Semantic Query Optimization in Object-Oriented Databases

G. N. Paulley and *G. K. Attaluri*

Abstract

In many relational database systems using a relational algebra-based query language, query optimization involves the *syntactic* modification of queries into a "canonical form"[Dat90], and then choosing from possibly several methods of evaluating the query. Semantic query optimization (SQO) is the idea of *semantically* transforming a query using additional schema information, such as integrity constraints. The query is passed through three different phases of logical transformation: *standardization*, *simplification*, and *amelioration*. The result of this process is a query that may, in fact, appear quite different from the original query posed by the user, but is guaranteed to return the same results given the same database instance.

In this paper, we give a brief overview of the concepts of semantic query optimization, and present a survey of the literature. Most of the literature involves the application of SQO to relational systems, or to logic-based models such as *Datalog*[CGT89, Ull88, Ull89]. To show how these techniques may be used with Object-Oriented Database Management Systems (OODBMS), we attempt to apply Chakravarthy et al.'s[CGM90] idea of *semantically constrained axioms* to an extension of Datalog called *Complex-Prolog*[GR89]. Complex-Prolog supports the notions of object identity, classes, and inheritence.

Finally, we present a summary and give some research directions for the application of semantic query optimization to OODBMS.

Contents

1 Motivation for Query Optimization

A major accomplishment of Codd's relational model[Cod90, Dat90] was the idea of *logical data independence*. The *conceptual schema definition* does not involve any considerations of storage structure, or access technique. The conceptual schema is, therefore, defined by information content only. Typically, non-relational database systems such as `IMS` or `IDMS`[Dat90] do not have such a clearly-defined separation between the internal and conceptual schemas. Out of necessity their query languages involve navigational operators, such as the "Get Next Within Parent" function call of `IMS`. Such query languages are termed *procedural*.

Non-procedural database query languages, such as Structured Query Language (`SQL`), have been in existence since the middle 1970s[AC75]. Most of these languages were developed for use with the relational model and are based on a combination of the relational calculus and relational algebra[Dat90, pp. 459]. The operators of the relational algebra and calculus support the data independence supplied by the relational model's conceptual schema. Consequently, users may form database requests without concern for implementation details, and may leave efficiency considerations to the database management system. This means that a *query optimizer* is required to translate the semantic expressiveness of the query into efficient, fundamental operations against the actual storage structures of the relational database. In contrast, procedural queries in non-relational environments require the user to perform the optimization. The user, not the system, is responsible for defining what record-level operations are needed, and in what order they will be performed.

Several advantages can be found for automated query optimization in relational systems[Dat90]:

- The query optimizer typically has access to much more information about the database system than does the user. In particular, statistical information, such as cardinality of domains and relations, is invaluable in finding a query processing strategy.
- Volatile databases may mean a change in strategy is desirable. With automatic query optimization, however, the user's query need not be rewritten to take advantage of different access paths.
- The automated query optimizer can be much more rigorous in deciding the optimality of any one particular query processing strategy, than compared to a user.
- The algorithm used by the optimizer is available to all users of the database. This implies that if the algorithm is robust, queries submitted by a novice will be executed with the same sense of optimality as those of an expert.

Several excellent references regarding the complete framework of query optimization exist in the literature [JK84], [Dat90, Chapter 18],[Ull89, Chapter 11], [Mai83, Chapter 11]. *Conventional* query optimization involves the following steps[JK84, pp. 116]:

1. Find an internal representation into which user queries[1] can be cast. This representation must typically be richer than either the relational calculus or algebra. This results from typical query language extensions such as scalar values and aggregate functions.

[1] In this context, the term "query" not only refers to retrieval operations, but also (and perhaps more importantly) to database updates.

2. Apply logical transformations to the query to *standardize* it, *simplify* it by eliminating redundancy, and improve (*ameliorate*) it if possible. Standardization includes rewriting a query to a common form. The point is that the performance of a query should not depend on how the query was cast originally by the user[Dat90, pp. 459]. Many query languages, including `SQL`, allow queries to be expressed in several alternate, though syntactically different, forms. Amelioration will be discussed in more detail later.

3. Map the transformed query into sequences of possible lower-level operations, and calculate the cost of executing each of these "access plans".

4. Choose the best access plan alternative, and execute it.

In this chapter, we concentrate on *Semantic Query Optimization* (SQO). SQO differs from conventional query optimization in the implementation of Step 2 above. With conventional query optimizers, queries are transformed using only *syntactic* transformations[SO89]. Syntactic transformations include algebraic manipulations and operator re-sequencing. Semantic transformations attempt to exploit any available metadata such as the nature of domains and integrity constraints. In most relational systems today, domain characteristics and integrity constraints are embedded in application programs, and thus are unavailable to the database management system. SQO requires that these constraints be stored with the database system. This scheme delivers several benefits:

- Consistency checks and data element edits, previously coded in (possibly many) application programs, are now under the control of the DBMS. This ensures that *all* application programs adhere to the same integrity constraints.

- As constraints are now enforced by the DBMS, ad-hoc updates are also subject to these constraints.

- Integrity constraint management may be automated. Altering the constraints may require no application changes. The DBMS could automatically determine if any tuple in the database violates a new (or changed) constraint, and inform the database administrator (DBA) appropriately.

- The additional metadata may be used for semantic query optimization.

The most important algorithmic constraint in applying query transformations is to ensure that the original query, and the ameliorated one, are *semantically equivalent*. Two queries are semantically equivalent if their answers are the same for all instances of the database that satisfy a specified set of semantic rules. Note that semantic equivalence does not imply syntactic equivalence, though the latter does imply the former. To illustrate, the following `SQL` queries:

```
SELECT *
FROM EMP
WHERE SALARY > 40K

SELECT *
FROM EMP
```

```
WHERE SALARY > 40K AND JOB = MANAGER
```

are not syntactically equivalent. However, the two queries would be semantically equivalent if the relation EMP was constrained such that all employees earning a salary greater than 40K were employed as managers.

King[Kin84] characterized conventional query optimization as a "hunt for opportunities". Using available metadata as input to semantic query optimization allows "the creation of new search spaces in which to hunt for such opportunities".

This paper is organized as follows. In Section 2, we define the terminology used in SQO, and survey the relevant literature. In Section 3, we define the Horn-clause based query language *Datalog*[Ull88, Ull89, CGT89] and discuss in detail the proposal by Chakravarthy *et al.* for semantic query optimization in relational database systems. In Section 4 we describe an extension to Datalog called *Complex-Prolog*[GR89] that supports an object-oriented data model. In Section 5 we discuss how the approach of Chakravarthy em et al. can be used with Complex-Prolog to support non-procedural query languages for object-oriented database systems. Finally, in Section 6 we present some conclusions and areas for further research.

2 Literature Review

In this section we survey the some of the available literature on semantic query optimization. By far, the literature on SQO references the relational model; the ideas surrounding semantic query optimization are discussed here in this paper. However, it is possible to apply these same ideas to object-oriented database systems, and we attempt to show how this may be done in Section 5.

The literature covers various aspects of semantic query optimization, from a variety of perspectives. These aspects include:

- the type of database system under consideration, and the types of additional semantic knowledge that are exploited.
- how semantic information is maintained in the meta-database. For example, how is a redundant or contradictory integrity constraint detected?
- the manner in which constraint violations are reported to the user. Integrity constraint violations apply not only to queries. Indeed, the database system must detect, whenever an update operation is performed, if any constraints have been violated.
- how the integration of semantic and conventional query optimization is architected. In effect, this means that as alternate queries are generated, a cost function must be applied to determine the profitability of using that *access plan*.
- how semantically equivalent queries are generated.
- how is the *relevant* metadata for a particular query filtered from all the integrity constraints in the metabase.

- how to control the generation process so that only "promising" semantically equivalent queries are considered. This typically involves the use of heuristics in the approach[Kin81, Kin84, CGM90, SO87, SO89, Xu83]. Shekar *et al.* [SSD88] give an analysis of the trade-offs between the costs of query transformation, and the cost of conventional optimization.

2.1 A Taxonomy of Query Transformations

In SQO, predicates may be added or subtracted from a query as long as the query remains semantically equivalent to the original. However, a difficult problem is how to determine if a particular semantically equivalent query is "promising". What we mean by this is how to determine if the transformed query may be executed more efficiently than any of the others that may be generated. A brute force approach would be to generate *all* possible semantically equivalent queries, and evaluate the performance of each using the cost model of the conventional query optimizer. Realistically, however, we must try to reduce the amount of effort required to find a "near-optimal" transformation. This is especially true with ad-hoc queries, where the cost of optimization directly affects the database user[SSD88].

To reduce the search space of possible query transformations, many SQO techniques rely on heuristics. Maintaining our focus on the relational model, these transformations can be categorized as follows:

Restriction Elimination. This heuristic attempts to eliminate strictly redundant predicates to simplify the query. This redundancy is not determined using a *syntactic* transformation. Instead, integrity constraints are used to infer the redundant predicate.

Literal Enhancement. Literal enhancement[JCV84] is another straightforward means to semantically improve a query. The idea is that a query's evaluable predicates may be made more powerful by substituting more restrictive clauses, which may be inferred from the integrity constraints. For example, suppose that a query includes attributes $r_1.a_1$ and $r_1.a_2$, such that $r_1.a_1 > 100$ and $r_1.a_2 = 4$. If the integrity constraints imply that $r_1.a_2 = 4 \rightarrow r_1.a_1 > 400$ then we can replace $r_1.a_1 > 100$ with $r_1.a_1 > 400$. Depending on how the internal structure of the database is organized, and how the conventional query optimizer works, such a transformation may prove more efficient. For example, if $r_1.a_1$ is an indexed attribute, then we may retrieve fewer tuples with $a_1 > 400$ than with $a_1 > 100$. Note the savings is dependent on the distribution of the values of a_1 in relation r_1.

Restriction Introduction. The idea behind this heuristic is to reduce the number of inner scans of a join operation, assuming a nested-loop join strategy. By introducing an additional literal, we may further restrict one or both of the relations involved in the join. This technique is also referred to as *scan reduction*[Kin81, SO89].

Literal Elimination. If an integrity constraint can be used to eliminate a literal clause in the query, we may be able to eliminate a join operation as well. To do so would imply that the relation being dropped from the query does not contribute any attributes in the result. This heuristic is termed *join elimination* by King[Kin81, Kin84], Shenoy and Ozsoyoglu[SO87, SO89], and Xu[Xu83].

Join Introduction. Here, the heuristic attempts to reduce the number of tuples involved overall by introducing another relation into the query. This new relation typically contributes no attributes to the result. However, if the new relation's relative size is substantially smaller than the other relation(s) involved, executing the join may be less costly than proceeding with the original query. The technique is also called *literal introduction*[CGM90] since a literal must be introduced into the query as a join attribute.

Index Introduction. Index introduction[Kin81, Kin84] tries to use an integrity constraint that refers to both a query-restricted attribute, and another attribute in the same relation that is indexed. If this can be achieved, the query cost can be reduced from a possible sequential scan to a series of probes using the index. If the index is *clustered* then the final cost will be further reduced. Note the linking of this heuristic to the physical implementation of the supporting data structure. It is not clear that query transformations can be made entirely independent from the choice of the underlying physical system.

Result by Transformations. This approach by Chakravarthy *et al.* [CGM90] is a hybrid of the heuristics discussed above. The idea is as follows. The set of integrity constraints for the database may include implication constraints, such as "Chicago ships only red parts". A query which asks "What color parts are shipped from Chicago?" may then be answered solely on the knowledge contained within the constraints. In this case, no database access is required. Another situation may involve referential integrity constraints. It may be possible to determine that if the database *contains* a tuple, or set of tuples, meeting certain constraints, then the answer to the query *must* correspond to the existence, or non-existence, of a particular tuple in the database. Although this result will have to be verified by a lookup to the actual database, such a lookup is probably much preferable to executing the original query.

Result by Contradiction. This method is not a heuristic *per se*. During the query transformation stage we may arrive at a contradiction between the integrity constraints of the database and the query predicates. This situation implies a null result, and therefore no database access is required.

Note that the above transformations do not *guarantee* that the modified query will be more optimal in the sense of execution cost. However, many authors [Kin81, SO89, CGM90, JCV84] claim that in many cases such modifications are warranted. What is not clear is how to *prove* that the addition of a particular integrity constraint will improve the efficiency of the query evaluation process without fully costing each access path selection. Moreover, to what *degree* this improvement will take is difficult to evaluate[JCV84, pp. 679].

Use of heuristics in the query amelioration process has led to the application of expert system technology [HZ80, Kin84, Xu83]. Integrity constraints and other metadata may be treated as a rule base, with the heuristics implemented using an expert system.

2.2 Previous Work

The idea of using additional semantic information (metadata) to aid the query optimization process is not new. In 1975, Stonebraker[Sto75] published a series of algorithms to handle a variety of integrity constraints which he termed *assertions*. The assertions are defined using an extension to

QUEL, the relational calculus-based data sublanguage used with INGRES. Several types of integrity constraints which may be stored in the database, including domain constraints and constraints involving aggregates. The integrity constraints are specified using QUEL statements whose predicates result in TRUE if the integrity constraint holds[2]. Stonebraker mentions the need for *transaction timing* (TT) [Cod90, pp. 246-247] of integrity constraints[3]. Enforcing a constraint using query modification is achieved by adding the constraint predicates to QUEL statements which update the database. Various problems with views are discussed, such as being unable to fully realize the base relation tuples that make up the view. Also mentioned briefly is the anomaly of updating a restricted attribute through a view, therefore deleting that tuple from the view definition. Unfortunately, no mention is made of how the applicable assertions are to be found for a given query, how the assertions are maintained, or detecting if two or more assertions are contradictory. The focus of the paper is to enforce integrity constraints, not query optimization through the use of those constraints.

Simon and Valduriez[SV84] implement constraint enforcement for updates in their SABRE system. They separate integrity assertions that may be checked when the update is processed, and those that must be checked when a transaction is completed. To prevent a long-running transaction from being fully backed-out, however, they employ the concept of *integrity checkpoints*. They present a graph-based algorithm to determine where the checkpoints may be made. However, instead of using Stonebraker's query modification technique, their method relies on the use of temporary relations to separate tuples being updated from those that are not. This avoids a subsequent retrieval from the database to ensure that all constraints have been met (in particular, constraints involving aggregates).

PRISM [SK84] is a knowledge-based system that, like SABRE, supports semantic integrity specification and enforcement. An object-oriented approach is used to treat both data and metadata in a uniform way. Constraints are expressed in the PRISM constraint language CL, and stored in a *Constraint Base*. The Constraint Base may be thought of as "resting" on top of a conventional DBMS, specifying semantics for the interpretation of the facts in the database. The major goal of PRISM is to allow the specification and query of constraints in exactly the same manner as conventional database structures.

Hammer and Zdonik[HZ80] pioneered the idea of semantic query optimization. They present an architecture for a knowledge-based solution, which they term a "Knowledge-Based Query Processor" (KBQP). Their motivation for this approach is driven by the unrealistic amount of knowledge a database user must possess to formulate queries that can be executed efficiently. They discuss the issue of determining which integrity constraints are applicable to a given query, conjecturing that an exhaustive search is too expensive. An argument is made for the use of heuristics (such as restriction introduction mentioned above) in controlling the transformation process. Their architecture makes use of multiprocessing, where each task applies a source-level transformation against the original query. A "scoring" system decides the feasibility of a given approach. Constraints are expressed in a modified lambda calculus. The constraint predicates are essentially domain restrictions, as described in more detail in [Sto75, SO89].

King[Kin81, Kin84] developed the idea of SQO at the same time as Hammer and Zdonik. King's

[2]Note the similarity of this approach to the specification of integrity constraints recently adopted as a draft ISO SQL standard[ISO90].

[3]Transaction timing, the capability to defer integrity checking until a commit has been issued by the application, is not currently part of the ISO SQL standard[ISO90]. For further details, see Codd[Cod90].

system is called `QUIST` (Query Improvement Through Semantic Transformation). His work is the first to give a taxonomy of heuristics for query transformation, including detection that a query is unsatisfiable if the query predicates contradict an integrity constraint. There is significant support for domains, including range restrictions. Constraints and queries are expressed in a form of relational calculus. The database is composed of an *extensional database* (EDB) and an *intensional database* (IDB), terms borrowed from work by Minker on deductive databases[GMN84]. Informally, the EDB may be considered the "base" relations in the database, while the IDB may be considered as views on those base relations.

Though King's work is significant in many respects, many possibilities are cited for additional research. Some of these are:

- Queries are presumed to refer to a "virtual relation" encompassing the entire database, with each table joined to the other using a single join attribute.
- King uses a select-project-join (SPJ) subset of relational calculus. Since the join is predefined, queries only allow projection and restriction operations. No mention is made of constructs such as aggregate functions, nor the existence of `NULL`'s.
- Constraints may consist only of conjunctive predicates.
- King's heuristics assume the use of indexing as the sole access mechanism besides sequential scan. Other data structures should also be supported.

Xu[Xu83] considers semantic query optimization for relational databases assuming select-project-join queries only. Three types of integrity constraints are supported:

Domain Rules. Domain rules place a restriction on the domain of an attribute, either with a constant or another attribute. The restriction may be any comparison operator.

Dependency Rules. These rules imply a condition on an attribute iff the conjunction of the conditions on other specified attributes is `TRUE`. All attributes must be contained within the same relation.

Production Rules. Production rules express inter-relational constraints between two relations. A conjunction of conditions on various attributes in either relation,

$$(\forall r \in R)(\forall s \in S)(C(r.A_1) \wedge \ldots \wedge C(r.A_n) \wedge (r.C\theta s.D) \\ \wedge C(s.B_1) \wedge \ldots \wedge C(s.B_m) \rightarrow C(s.B))$$

coupled with a join condition $(r.C\theta s.D)$ involving an attribute from each, then imply a condition on another attribute $C(s.B)$.

Xu discusses aspects of query unsatisfiability and the heuristics of literal elimination, join introduction, and index introduction. The goal of Xu's method is to select the optimal *single* semantically equivalent query, and pass that query to the conventional query optimizer for execution.

A "front-end" semantic query optimizer written in Prolog that performs query transformations for a relational database is the subject of two related approaches[JCV84, Jar86]. The front-end

approach is investigated as it could, in theory, be used with any RDBMS even though the RDBMS does not support semantic query optimization directly. In essence the RDBMS is used only for its conventional optimization methods. The first approach[JCV84] loosely couples an expert system implemented in Prolog to an `SQL`-based relational system, using a meta-language called `DBCL` (Database Call Language) as the interface. The `DBCL` language is manipulated in the same way as tableaus[Ull89], and tableau simplification algorithms provide the optimization mechanism. The types of constraints supported are:

Value bounds. This type of constraint specifies upper and lower bounds for a particular domain.

Functional dependencies. These constraints capture dependencies between two attributes in the same relation.

Referential integrity constraints. These constraints capture existential dependencies between the values that appear in an attribute a_1 in a relation r and the "key" attribute in another relation r_2.

The second approach differs from the first in that a graph-based solution is used to perform the optimization.

Yu and Sun[YS89] discuss an interesting approach to the dynamics of a database system. Their approach to semantic query optimization includes *dynamically* inferring integrity constraints using query results. This acquired knowledge can then be used in the optimization of later queries. In the ubiquitous parts-suppliers example consisting of the relations P (parts), S (suppliers) and SP (parts-supplied-by), suppose we have the following constraints:

- A domain constraint $SP.Pno \subseteq P.Pno$.
- A domain constraint $SP.Sno \subseteq S.Sno$.
- A functional dependency of part name on part color, $P.Name \stackrel{FD}{\rightarrow} P.Color$.

Given two queries Q_1, Q_2 defined as:

Q_1:
```
SELECT P.Name
FROM P
WHERE P.Color = RED
```

Q_2:
```
SELECT P.Name
FROM P
WHERE P.City = CHICAGO
```

we find that the results of the two queries are the same, i.e. $Q_1 = Q_2$. Since the functional dependency $P.Name \stackrel{FD}{\rightarrow} P.Color$ exists, we may obtain the new constraint

$$P.City = chicago \rightarrow P.Color = red.$$

Subsequent queries may then be semantically modified using the inferred integrity constraint.

Graefe and Ward[GW89] describe another interesting approach to handling queries. When `SQL` statements are embedded within application programs, predicates usually refer to program variables. This means that compile-time SQO may lead to suboptimal results, as the values of the variables will not be known until execution time. The optimizer will usually make assumptions about the selectivity of these predicates in this case. Unfortunately, this means that accurate estimates of execution parameters, such as the sizes of intermediate result relations, is impossible. Another problem is that the access path is chosen using the state of the database at compile time, and not when the application is run. The approach in this paper is to monitor the use of access plans and modify them over time as a result of changes to program variables, additional database indexes, and large changes in the cardinality of a relation. Estimates are made using a cost model by Yao[Yao79]. The discussion is based on the relational model, but the authors suggest the methods may be used with other data models (including object-oriented models).

IBM's `Starburst` prototype RDBMS [HFLP89, HCL$^+$90] is an *extensible* system which can support additional internal processing extensions, such as query transformations. Optimization of `Hydrogen`[4] queries is performed using a Query Graph Model (QGM). Query rewrite is implemented using an extensible, forward-chaining rule based approach [HP88]. The focus of the rewrite optimization phase in `Starburst` is the optimization of subqueries and predicate migration (pushing predicates down into lower level operations). Since the system is extensible, however, it is possible for a database administrator to provide additional rules in the form of integrity constraints.

Typically, database systems which detect integrity constraint violation offer a *rollback* mechanism for the transaction in error, or reject database operations which may lead to an inconsistent state. Ceri and Widom[CW90] present an `SQL`-based language which defines a means of *automatically* "repairing" a database using production rules, which are translated from the defined integrity constraints for the database. The relationship of this type of system to SQO is that the production rules themselves are non-procedural updates, and can also benefit from SQO using the defined constraints. An interesting problem is that since the database being corrected is in an inconsistent state, not all integrity constraints may be used in the semantic optimization, since they may no longer hold.

Shenoy and Ozsoyoglu[SO87, SO89] provide a detailed study of two important types of integrity constraints: *subset constraints* and *implication integrity constraints*. The constraints are defined using Horn clause logic as follows:

Subset Constraints. Subset constraints represent a unary inclusion dependency between two attributes. The integrity constraint is composed of two relational predicates and one evaluable (comparison) predicate using either of the operators $\subset$ or $\subseteq$. As an example, the constraint

$$\rightarrow r_2.X_2 \subseteq r_1.X_1$$

means that the values of attribute X_1 in relation r_1 is a superset of the values of attribute X_2 in relation r_2.

Implication Integrity Constraints. These types of constraints define a restriction on the relative domains of two attributes, in the same or different relations. They are defined as clausal

[4] `Hydrogen` is the data sublanguage used in `Starburst`. Essentially `SQL`-like, `Hydrogen` corrects many of the criticisms (such as lack of orthogonality) associated with ANSI `SQL` [Dat86, Chapter 14] [HFLP89].

integrity constraints (Horn clauses with no positive literal, and one or more negative literals) composed of conjunctions of negated predicates. The predicates may consist of evaluable or relational predicates. The relational predicates may be either extensional or intensional. Evaluable predicates may involve θ-comparisons ($\{\theta \in \{=, \neq, >, \geq\}\}$) between:

- a variable and a constant,
- two variables, or
- two variables and an offset (see Rosenkrantz and Hunt[RH80]).

Using our previous example of employees and their salaries, our implication constraint for the rule "only managers earn more than 40K" is

$$Emp(sin, name, job, salary), salary > 40K \rightarrow job = manager.$$

The paper describes in detail a constraint maintenance process to detect contradictory or redundant constraints, using a graph-based approach. Their implementation of semantic query optimization supports four heuristics from King's work, restriction elimination, index introduction, scan reduction, and join elimination. A graph-based approach is employed for defining the transformation algorithms. The significant contribution of the paper are algorithms for determining which integrity constraints are useful to semantically optimize a particular query. Suggestions for further research include handling constraints which may change over time (termed *semantic categorization*) and *partial optimization*, a technique for avoiding complete re-optimization when constraints or database characteristics have changed.

The final selection in our survey is the logic-based approach of Chakravarthy, Grant, and Minker[CGM90, CM86, CFM86, CGM87, KM83, GMN84, NY78, GM81]. [CGM90] is a consolidation of their previous papers, and is the principal reference used here. Their approach is a result of their work with deductive databases, but they describe the applicability of their methods to relational systems. They use a modified form of Datalog[Ull88, Ull89] as their data model; therefore clauses must be Horn, although this restriction is lifted in the section on extensions.

In [CGM90], a database consists of three parts:

- the extensional database (EDB) made up of *facts*.
- the intensional database (IDB), "views" in the relational sense. IDB axioms must be non-recursive.
- integrity constraints (ICs), which are restricted to be non-recursive and may only refer to EDB relations. Their language for integrity constraints handles many of the types of constraints mentioned in the above papers. We categorize these constraints in the following subsection.

A two-phased approach is defined. The first phase consists of compiling the integrity constraints with the IDB axioms resulting in a set of *semantically constrained axioms*. Essentially this means that each IDB axiom has associated with it each applicable integrity constraint. Note that in this stage queries are not considered; since the IDB and IC components are usually stable, the compilation cost may be amortized over all subsequent queries. The compilation step uses a technique

called *partial subsumption*, a form of clausal unification that results in *residues*, portions of ICs that remain after no further unification can take place.

The second phase is the query modification phase: processing queries and transforming them into semantically equivalent queries using the semantically constrained axioms compiled previously. Heuristics are used in choosing appropriate transformations. The query transformations supported include literal elimination, restriction introduction, literal introduction, result by transformations, and unsatisfiable conditions. Proposed extensions to the basic approach include support for non-range-restricted ICs, disjunctive ICs, negated literals in a query, and recursive IDB rules (see [LH88]).

2.3 Categories of Integrity Constraints

Most of the papers described above deal with subsets of the complete range of specifiable constraints. In this subsection we categorize the supported types of integrity constraints, and cross-reference these constraints to their definition and use in the literature. Descriptions of integrity constraints in general, and their properties, may be found in [Ull88, Fag81, CFP82, SU82, MG90]. The papers discussed previously support instances of inclusion and functional dependencies. The differences among the constraint types listed here are small. However, we felt it appropriate to include a categorization of these ICs, if only to clarify the varying terminology among the different authors.

The logic-based approach of Chakravarthy *et al.* is the most general, and encompasses all the constraint types described in other papers. See Table 1 for a cross-reference of dependency types to their sources in the literature.

2.3.1 Inclusion Dependencies

Type 1. An *inclusion dependency* (IND) is an integrity constraint that defines the possible values of an attribute in one relation with the existing values in another relation. Formally, if $R[a_1, \ldots, a_n]$ and $S[b_1, \ldots, b_m]$ are relations (not necessarily distinct), if X is a sequence of k distinct members of R, and Y is a sequence of k distinct members of S then we call

$$R[X] \subseteq S[Y]$$

an inclusion dependency[CFP82]. Inclusion dependencies support existential quantification, and therefore are used to support:

- referential integrity within a database system.
- ISA type hierarchies (eg. every Passenger-plane ISA Plane), a subset of general referential constraints.

INDs correspond to *referential integrity constraints* in [JCV84] which are of the form

$$R(x4, f_1, f_2) \leftarrow S(x_1, x_2, x_3, x_4)$$

where $\{f_1, f_2\}$ are Skolem functions with the variables omitted. *Subset constraints* defined by Shenoy and Ozsoyoglu[SO89] are of the same form[5]. A more general form is used in King[Kin81],

[5]The terminology used in [SO87, SO89] is confusing in that their definition of subset constraints refer to *domains*. Subset constraints, however, are not merely a definition for a domain's bounds, but include an existential qualification in the implication.

Type	[CGM90]	[SO89]	[JCV84]	[Xu83]	[Kin81]
1.	Existential Rules	Subset Constraint	Referential ICs	N/A	Structural Constraint
2.	General Horn Clauses[6]	N/A	Value Bounds	Domain Rule	N/A
3.	General Horn Clauses	Implication Constraint	N/A	Dependency Rule	General Horn Clauses
4.	General Horn Clauses	Implication Constraint	Functional Dependency	Production Rule	General Horn Clauses
5.	General Horn Clauses	Implication Constraint	N/A	Production Rule	General Horn Clauses
6.	General Horn Clauses	Implication Constraint	N/A	N/A	General Horn Clauses

Table 1: Cross-reference of Supported Constraint Types.

that includes additional evaluable predicates:

$$R(x_4, f_1, f_2) \leftarrow S(x_1, x_2, x_3, x_4), (x_3 \theta c)$$

All these different types of inclusion dependencies are supported as *existential rules* in [CGM90]. An IND is indicated as Type 1 in Table 1.

2.3.2 Functional Dependencies

Functional dependencies are a broad class of data dependencies that have been widely studied in the relational database literature[CFP82]. Each of the papers described in Section 2.2 support a subset of the constraints in this class. Formally, functional dependencies (FDs) are defined as follows[CFP82]. If $R[a_1, \ldots, a_n]$ is a relation and X is a sequence of distinct members of R as is Y, then we call the dependency

$$R : X \rightarrow Y$$

a functional dependency of Y on X. That is, R satisfies the functional dependency if whenever tuples $t_1, t_2 \in R$, $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$. The literature extends this definition in two ways. Firstly, the FD may specify a non-equivalence operator θ in the implication $t_1[Y]\ \theta\ t_2[Y]$. Secondly, the dependencies are generalized to more than one relation.

We classify the functional dependencies in Table 1 in terms of the Horn clause notation used in [CGM90]. Both Chakravarthy *et al.* and King support constraint Types 2 through 6; *implication constraints* defined by [SO89] do not support Type 2.

We begin with the simplest forms of allowable constraints.

[6] These Horn clauses are assumed to be *range-restricted*.

Type 2. These ICs restrict the values that an attribute of a relation may have. They are referred to as *domain rules* in [Xu83] and *value bounds* in [JCV84], and are of the form

$$(x_i \theta a) \leftarrow R(x_1 \ldots x_j)$$

This type of dependency is classified by Fagin[Fag81] as a *domain dependency.*

Type 3. To domain dependencies, we add additional comparison predicates that specify a restriction on R, based on other attributes:

$$(x_i \theta c) \leftarrow R(x_1 \ldots x_j), (x_k \theta b)$$

The FD is still defined for a single relation. These are the *dependency rules* in Xu[Xu83].

Type 4. Type 4 ICs represent true functional dependencies, eg.:

$$(x_1 = y_1) \leftarrow R(x_1, x_2, x_3), R(y_1, x_2, y_3)$$

Only equality, as in the formal definition[CFP82], is permitted. These constraints are termed *functional dependencies* in [JCV84]. Note that the constraint applies to only a single relation.

Type 5. Type 4 constraints are extended to support θ-comparisons, although the implication is restricted to a comparison with a constant. The IC may specify more than one relation. Additional qualification predicates may be added to the body of the clause in the form of θ-comparisons between attributes. For example:

$$(x_1 \theta f) \leftarrow R(x_1, x_2, x_3), S(y_1, y_2), (x_3 \theta y_2)$$

These types of ICs are referred to as *production rules* by Xu[Xu83].

Type 6. Type 6 ICs are a combination of types 4 and 5. With this type of constraint, we allow a θ-comparison of any two attributes, and θ-qualification in additional comparison predicates. For example:

$$(x_1 \theta y_1) \leftarrow R(x_1, x_2, x_3, x_4, x5), S(y_1, x_2, y_3), (x_4 \theta y_3)$$

These types of constraints are supported by King[Kin81, Kin84].

3 SQO Using a Logic-Based Approach

In this section we describe the logic-based approach of Chakravarthy, Grant, and Minker[CGM90, CM86, CFM86, CGM87, KM83, GMN84, NY78, GM81]. Our principal reference is [CGM90] which summarizes their earlier work since 1985. Unlike other approaches, such as [SO89] which try to dynamically determine applicable constraints, Chakravarthy *et al.* apply each integrity constraint to each intensional relation in the database (the IDBs) at initialization. They call this step *semantic compilation*; the output of semantic compilation is a set of *semantically constrained axioms* (SCAs). After completion, queries may be semantically transformed using the SCAs during the *semantic query transformation* phase. The output of this phase is an optimal, transformed query, chosen by means of the heuristics discussed in Section 2.1.

3.1 Definitions

The original target for the logic-based approach was *deductive databases*. Much of the terminology used in [CGM90] comes from this subject area. A Prolog-like notation describes not only the schema, but the ICs and queries as well.

A *literal* is an atomic formula, or its negation. A *clause* is a disjunction of literals, written using an implication style as

$$S_1, \ldots, S_m \leftarrow R_1, \ldots, R_n.$$

A literal on the left ($S \leftarrow$) is *positive*. A literal on the right ($\leftarrow R$) is *negative*. Multiple literals appearing on the right of an implication imply *conjunction* between all the literals. Multiple literals appearing on the left imply *disjunction* between the literals. The set of literals on the left are termed the *head* of a clause; on the right, the *body*. A clause is termed *Horn* if at most one clause is in the head ($m = 0, 1$), *definite* if exactly one ($m = 1$), and is termed *disjunctive* otherwise.

Clauses can be further categorized as follows:

- A *goal clause* has a null head.
- A *unit clause* is a definite clause with a null body.
- A *ground unit clause* is a unit clause with only constants occurring as arguments.
- A clause is considered *range-restricted* if every variable in the head also appears in the body as arguments of relational predicates.

Infix operators are used to define *evaluable predicates*.

A database is defined as three components:

1. an extensional component describing database *facts* using function-free unit clauses with no variables, termed the EDB. In [CGM90] a positional notation is used for the variables (attributes) in a relation (both for EDB relations, and IDB relations below).
2. an intensional component (IDB) describing a set of deductive rules, or *axioms*, which in the relational model sense roughly equivalent to views. The IDB is defined using a set of non-recursive range-restricted function-free definite Horn clauses.
3. a set of integrity constraints (ICs), defined using a set of non-recursive range-restricted Horn clauses. Skolem functions are allowed, so that referential constraints may be supported. For example,
$$Empl(x_3, f(x_1, x_2, x_3), g(x_1, x_2, x_3)) \leftarrow Dept(x_1, x_2, x_3)$$
represents the referential constraint "each manager of a department (represented by variable x_3) must exist as an employee". The other attributes of the employee may be any permitted value, represented by the two Skolem functions $f()$ and $g()$.

A restriction of this approach is that it is assumed the database is *structured*, in the sense that every relation is either purely extensional or intensional, and all ICs may only refer to EDB relations.

3.2 Overview

The logic-based approach splits the semantic query optimization process into two distinct phases– the Semantic Compilation Phase and the Query Transformation Phase.

3.2.1 Semantic Compilation Phase

The semantic compilation phase consists of compiling the integrity constraints with the IDB axioms resulting in a set of *semantically constrained axioms*. Several points should be noted here:

- In addition to IDB relations (defined using restrictions, projections, joins, or other relational operators) an IDB relation is defined for each "base" relation of the EDB.
- Semantic compilation is based on the premise that the database schema, and its integrity constraints, do not change often over time.
- It is presumed that some means is available to modify the set of EDBs, IDBs, and ICs and ensure that the modified set of ICs is kept consistent and nonredundant. See [SO89] for a description of their solution.

Essentially the semantic compilation phase means that each IDB axiom has associated with it each *applicable* integrity constraint. Note that in this stage queries are not considered. The compilation cost is amortized over subsequent queries.

The compilation step uses a technique called *partial subsumption*, a form of clausal unification that results in *residues*, portions of ICs that remain after no further unification can take place. Using the notation from [CGM90]:

- A *substitution* σ is a finite set of the form $\{t_1/v_1, \ldots, t_n/v_n\}$ where each v_i is a unique variable and t_i is a term.
- If σ is a substitution and C is a clause, then C_σ is an instance of C where the variables in σ are replaced by their respective terms.
- A clause C (an integrity constraint) *subsumes* a clause D if there is a substitution σ such that C_σ is also in D. Our example below is also from [CGM90]. If we are given clauses C, D

$$\begin{aligned} C &= R(x,b) \leftarrow P(x,y), Q(y,z,b) \\ D &= R(a,b) \leftarrow P(a,z), Q(z,z,b), S(a) \end{aligned} \tag{1}$$

  then C *subsumes* D by $\sigma : \{a/x, z/y\}$.

We illustrate subsumption using 1 above. The steps involved are as follows:

1. D is instantiated as a ground clause, using new constants $k_1, \ldots, k_n$ and a substitution, say θ. In 1 we have $\theta = \{k_1/x\}$, so that

$$D_\theta = R(a,b) \leftarrow P(a,k_1), Q(k_1,k_1,b), S(a)$$

2. D_θ is negated, forming a set of literals

$$\neg D_\theta = \{\leftarrow R(a,b), P(a,k_1) \leftarrow, Q(k_1,k_1,b) \leftarrow, S(a) \leftarrow\}.$$

3. The basic subsumption algorithm tries to determine if a substitution can be found for C using each literal $\in \neg D_\theta$. If a substitution can be found, then C subsumes D. What this really means is that the IDB relation D is *contradictory* as it violates the integrity constraint C[7]. If D is an EDB relation, then it must be empty.

4. *Partial subsumption* occurs when C cannot totally subsume D, which is usually the case. Essentially we are left with a fragment of D that cannot be refuted using the subsumption algorithm. In our SQO approach, $C \in$ IC; so this *residue* of C now becomes part of the semantically constrained axiom for D[8]. An axiom is *semantically constrained* if the residues applicable to that axiom are associated with it.

To illustrate, suppose we have the EDB relation

$$Plane(model, manufacturer)$$

and the IC

$$\leftarrow Plane(model, manufacturer), (model = \text{“Boeing 747”})$$

which means that no 747 aircraft are in the database. Using partial subsumption, we rewrite the IDB axiom as

$$Plane(model, manufacturer) \leftarrow Plane(model, manufacturer)$$
$$\{\leftarrow model = \text{“Boeing 747”}\}.$$

This procedure is done for each combination of ICs and IDB axioms, resulting in a set of SCAs.

As a final comment, note that functional dependencies defined in the set of ICs may allow us to derive other ICs, and thus other residues. These additional residues may prove useful in the query transformation phase, but would be unobtainable from the base set of ICs alone.

3.2.2 Query Transformation Phase

The second phase is the query modification phase, processing queries and transforming them into semantically equivalent queries using the semantically constrained axioms compiled previously. Heuristics are used in choosing appropriate transformations. Examples of query transformations are given in Section 5.

A query is specified as a *goal clause*. Query output variables are denoted by an asterisk (“*”) after the variable name. For a query $\leftarrow Q(x_1^*, \ldots, x_n^*)$, an answer to Q is any tuple $< a_1, \ldots, a_m >$ such that $Q(a_1, \ldots, a_m)$ is provable. For example,

$$\leftarrow Plane(-, model^*, -, -, flight\_crew, -, -), (flight\_crew \geq 4)$$

[7]Note that if a complete substitution θ can be found so that C subsumes D, then we end up with a null clause, which by definition implies a contradiction.

[8]In order for partial subsumption to work, some trivial syntactic modifications may be necessary. These steps are referred to as *expansion* and *reduction* in [CGM90].

finds the model names of all planes requiring a flight crew of at least 4.

The functional components involved in query transformation are as follows:

Query/Residue Modifier. The Modifier uses all applicable SCAs (axioms that refer to relations specified in the query) and transforms the original query into (possibly) a set of equivalent queries[9]. Modification of the queries is performed using either unification, or pattern matching. The residues of the SCAs may also be modified. This is because when a literal is unified with the head of an SCA, the unifier generated is also applied to the *body* of the SCA. This will instantiate the variables in the residues of the SCA with constants from the query.

Reducer. The reduction process removes redundant literals from an SCA, which may occur as a result of the instantiation of residue variables performed by the Modifier.

Filter. The filter eliminates useless residues from the SCA for the given query. This occurs if a literal R (an EDB relation, or an evaluable predicate) is present in a residue body, but x does not appear in the query.

Strategiser. The Strategiser prioritizes the residues that may be used by the Generator (described below). The criteria used for priority determination could be defined by a Database Administrator (DBA), eg. it may be known that the conventional query optimizer that will generate the algebraic access plan may make very efficient use of clustered indexes, whereas its join optimization algorithms may be poor. As previously noted with the Index Introduction heuristic, it is difficult to separate query transformation issues from DBMS implementation characteristics.

Generator. This function uses the heuristics literal elimination, restriction introduction, literal introduction, result by transformations, and unsatisfiable conditions to include or exclude literals which result in a semantically transformed query that (hopefully) can be executed more optimally. The transformed query is then passed to the conventional query optimizer for access plan evaluation, and query execution.

3.2.3 Residue Categories and Query Processing

Residues resulting from semantic compilation are of four types:

1. a null clause,
2. a goal clause,
3. a unit clause,
4. a Horn clause.

During query transformation (the Generator function), we may come across each of these types of residues.

[9]If the transformation results in more than one query, then the answer set of the original query is equal to the union of the answer sets for the generated, semantically equivalent queries.

If a null clause is obtained at any stage, then we have a contradiction, as described above. If it occurs for an IDB relation, then there are no tuples in the view. If it occurs for an EDB relation, then the database is inconsistent.

If a goal clause results, it can be modified to a unit clause by negation (if we have the goal clause $\leftarrow (z < 200)$, we may modify it to be an equivalent unit clause $(z \geq 200) \leftarrow$).

Unit clauses are database assertions. They can be used for literal elimination, if another clause can be resolved using the assertion, and hence eliminate a join. Other possibilities for their use are join introduction, literal enhancement, and literal introduction. For literal elimination, we must ensure that eliminating a join does not eliminate any output variables for the query.

Horn clauses contain a non-empty head, and a non-empty body. Horn clause residues can be used in two situations:

- The residue body subsumes the query. If C is the head of the residue and θ was the substitution, then C_θ must be true for the query, and can be used as an assertion like a unit clause.

- The residue can be used to limit a search in evaluating the query (this corresponds to the heuristic "Result by Transformations"). In particular, functional dependencies of the form $R : x \rightarrow y$ for relation $R(x, y, z)$ yields a semantically constrained axiom of the form

$$R(x, y, z) \leftarrow R(x, y, z)\{y = y' \leftarrow R(x, y', z')\}.$$

  The query $\leftarrow R(a, y^*, z)$ would then be interpreted as saying that a second y' value must be identical to any y value already in the result set; meaning that the output of the query has *at most* one result tuple. We would still have to access the database, however, to ensure that the tuple $\{a, y, z\} \in R$. An existential quantification would be needed to remove the lookup requirement.

4 Complex-Prolog

The relational model has dominated the database area well over a decade. However, the semantic simplicity of the relational model has motivated researchers to look for semantically rich *object-oriented* data models. Moreover, deductive languages based on first-order logic are being suggested as a more expressive alternative to relational languages. Datalog[Ull88, Ull89] is a well-known deductive logic language.

Researchers have recently attempted to combine object-oriented data models with logic programming. Recent proposals in this direction include [KL89, Mai86]. Though the resulting proposals are promising, they lack efficient implementation strategies. There have been enormous efforts to make non-procedural relational languages, such as `SQL`[AC75], and logic languages such as Datalog, efficient through query optimization. As discussed previously, attempts have been made to exploit additional semantic information about the database schema for query optimization[Kin81, Kin84, CGM90, JCV84, Xu83, SO89]. This form of query optimization is termed *semantic query optimization*.

In this section we describe an existing complex-object logic language called *Complex-Prolog* [GR89], with some minor modifications. These modifications allow the addition of *integrity constraints* (ICs) to the language. With the addition of ICs, we can apply the techniques of semantic query optimization to the language using the logic-based approach from Chakravarthy *et al.*[CGM90].

Complex-Prolog combines object-oriented semantics (generalization, inheritance, and complex objects) with logic. The relational data model with Datalog is used as the underlying formalism to implement a program in this logic. The resulting system has the richness of object-oriented models and the efficiency of a relational database system.

4.1 Complex-Prolog Definitions

The alphabet U of Complex-Prolog consists of the following sets:

I - the set of countably infinite integers.

W - the set of infinite character strings.

N - the set of *object identifiers*, including *nil*.

C - the set of class (extensional) predicate symbols, characterized by arity.

P - the set of intensional predicates, characterized by arity and argument types.

D - the set of comparison predicates, such as $=$, $<$ etc.

R - the set of record symbols, characterized by arity.

A - the set of attribute names, including the special name *self*.

V - the set of variables.

The set F of function symbols is defined as $\{C \cup R\}$.

4.2 Schema

The set N is divided into non-disjoint subsets according to their class affiliation; so each class symbol c in C has a set $ext(c)$ of objects belonging to c. This set is known as the *extension* of c. Members of $ext(c)$ are known as *instances*.

A (ground) *term* is an element of I, W, N, or a function application. If f is a function symbol and $t_1, .., t_n$ are (ground) terms, then $f(t_1, .., t_n)$ is a (ground) term.

A *type* is a set $X \cup \{nil\}$, where X is a subset of all terms. Since *nil* is a member of all types, we will characterize types by members of the set X.

Types in Complex-Prolog are defined as follows:

1. The set I of integers is a type.
2. All character strings $w \in W$ of size n, where n is a natural number, constitute a type denoted by $char[n]$.

3. Any subrange of I is a type.
4. Any class $c \in C$ is a type. Members of c are given by $ext(c)$.
5. If $t_1, .., t_n$ are types and $f \in R$, $f(t_1, .., t_n)$ is a (record) type. Members of this set are all terms of the form $f(v_1, .., v_n)$, where $v_1, .., v_n$ are members of types $t_1, .., t_n$, respectively.

An attribute a is a pair $< s, t >$, where $s \in A$ and t is a type. In general we refer to an attribute by its name.

An attribute is syntactically represented in one of the following ways:

- $s : t$, where $s \in A$ and t is a type specified by the first three categories. Such an attribute is called an *atomic attribute*.
- $s : t$, where $s \in A$ and $t \in C$. Such attributes are called *reference attributes*.
- $s : f(a_1, .., a_n)$, where $f \in R$, $s \in A$ and $a_1, .., a_n$ are attributes. These are called *record attributes*.

A class description is a triple $< c, p_c, a_c >$, denoted by $cd(c)$, where $c \in C$, p_c is a (possibly null) list of $c_i \in C$, and a_c is a list of attributes. The first attribute a_c is always *self:c*. We refer to class description by its class name, if there is no ambiguity.

The list p_c is known as the *parent classes* of c, and indicates that a member of $ext(c)$ is also in $ext(d)$, and attributes of d are added to a_c, for each $d \in p_c$. This concept is known as *generalization* and c is said to inherit the attributes of each class in p_c.

A *schema* is a set S of class descriptions. Multiple inheritance is allowed in S, subject to the usual restrictions that a class may not inherit the same attributes twice, and that attribute names are unique.

Let *ancestors(c)* denote the (ordered) list of all r in S such that $r \in p_c$. This exact order has no bearing on our discussion, but all classes in the schema should have ancestors ordered in some manner that is permanent.

The *closure* of a class c, $closure(c)$, is a list of attributes as defined below:

- $<a_c>$ if $ancestors(c)$ is empty.
- $<a_c, closure(p_1), .., closure(p_n) >$[10], where $p_1, .., p_n \in ancestors(c)$ in the same order.

The arity of a class symbol c in a class scheme S is the number of attributes in $closure(c)$.

Given a class description c, with

$$closure(c) =< a_1, .., a_n > \text{ where } \forall a_i : \ a_i =< s_i, t_i >$$

a (simple) instance of c is a ground term $c(s_1 : v_1, .., s_n : v_n)$ [11] such that $\forall v_i : 1 \leq i \leq n, v_i \in t_i$. The term v_1 is called the *object identifier* of the instance and cannot be *nil*. The above instance of c is often denoted by

$$c(s_1 : v_1.., s_p : v_p, T_{r_1}, .., T_{r_k})^{12}$$

[10] Here, we use a flattened list rather than a nested list.

[11] Conflicts with duplicate attribute names $closure(c)$ are resolved by the order of the attributes. We can represent this instance in positional notation as $c(v_1, .., v_n)$.

[12] Positional notation can be used here as well.

where $v_1, .., v_k$ are the terms corresponding to a_c, and $T_{r_1}, .., T_{r_k}$ are tuples of terms such that $r_1(T_{r_1}), .., r_k(T_{r_k})$ are instances of $r_1, .., r_k$, respectively, for every $r_i \in ancestors(c)$.

Example 4.1 *Consider the class* Airport *described below:*

$$Airport(name : char[25], address : Address(city : char[25], \\ country : char[25]), alternative : Airport).$$

Here, attribute *address* is of the record type $Address(city : char[25], country : char[25])$, *Address* being the record symbol; *name* is a of string of 25 characters.

The attribute *alternative* is a referential attribute whose value must be in *ext(Airport)*. An instance of *Airport*, written in non-positional notation, is

$$Airport(self : o1, name : \text{“Pearson”}, \\ address : Address(city : \text{“Toronto”}, country : \text{“Canada”}), \\ alternative : o2).$$

We can define a subclass of *Airport*, *Civilian_airport*:

$$Civilian\_airport(earliest\_time : int, latest\_time : int) \ isa \ Airport.$$

The closure of *Civilian_airport*, written *closure(Civilian_airport)* is

$$< self : Airport, earliest\_time : int, latest\_time : int, name : char[25], \\ address : Address(city : char[25], country : char[25]), \\ alternative : Airport > .$$

We have not taken advantage of the complex structure of objects; for example, we have no way to represent the name of the alternative Airport (with object identifier $o2$) in the *Airport* instance $o1$ above. We define *extended types and instances* of a class to describe the substructure of an instance.

Given a schema S and an attribute $a =< s, t >$ in S, the *extended type* e of a is defined as follows:

1. If a is either an atomic or *self* attribute, $e = t$.
2. If a is a reference attribute of the form $s : c$, where c is a class description in S with closure $< a_1, .., a_n >$, then $e = t \cup \overline{e}$, where $\overline{e}$ is the type[13] $c < e_1, .., e_n >$ such that $\forall i, 1 \leq i \leq n, e_i$ is the extended type of a_i.
3. If a is a record attribute of the form $s : f(a_1, .., a_n)$, then $e = f(e_1, .., e_n)$, such that $\forall i, 1 \leq i \leq n, e_i$ is the extended type of a_i.

[13]Notice that t is the type defined by $ext(c)$, the extension of class $c \in C$.

The second case warrants careful examination. If a class description contains a reference attribute with that class name as its type, the extended type of that attribute will have countably infinite members. This is required since any degree of extension of the object represented by such an attribute is valid. We are safe as long as we do not try to evaluate these infinite values.

An *extended instance* of a class c, with closure $< a_1, .., a_n >$ is a term $c(s_1 : v_1, .., s_n : v_n)$ such that $\forall i, 1 \leq i \leq n, v_i$ belongs to the extended type e_i of a_i. Every simple instance of c is also its extended instance but the converse is not always true.

Example 4.2 *An extended instance of* Airport *is:*

$$\begin{aligned} &Airport(self : o1, name : \text{ ``Regina International Airport''},\\ &\quad address : Address(city : \text{ ``Regina''}, country : \text{ ``Canada''}),\\ &\quad alternative : Airport(self : o2, name : \text{ ``CFB Moose Jaw''},\\ &\qquad\qquad address : Address(city : \text{ ``Moose Jaw''},\\ &\qquad\qquad country : \text{ ``Canada''}),\\ &\qquad\qquad alternative : o1)) \end{aligned}$$

4.3 Language

In preparation for converting Complex-Prolog to Datalog, we can re-categorize the alphabet U of the Complex-Prolog language as follows[14]:

1. the set of constants, N, I, and W.
2. the set of function symbols, F.
3. the set of predicate symbols consisting of
   - class symbols C appearing in schema S. These symbols are called *class* or *extensional* predicates.
   - comparison predicates D, including operators such as $<, =, \neq$.
   - the set of *intensional* predicates, P.
4. the set of variables, V, the symbol '-' being used to denote an anonymous variable.
5. the set of Boolean operators and constants ($\{\leftarrow, \vee, \neg, \wedge, TRUE, FALSE\}$).

A *term* is defined as a

- constant, or
- variable, or
- record application $f(s_1 : v_1, .., s_n : v_n)$, where $f \in R$ (such a term is called a *record term*), or
- class application $g(s_1 : v_1, .., s_n : v_n)$, where $g \in C$. These terms are called *class terms*.

[14]Note that these sets were introduced in Subsection 4.1.

A ground term is a term without variables.

Given terms $t_1, .., t_n$, we define a predicate of Complex-Prolog as follows:

- If p is a class symbol and $p(s_1 : t_1, .., s_n : t_n)$ unifies with a simple instance, $p(name_1 : t_1, .., name_n : t_n)$ is called a *simple class predicate*. If $p(name_1 : v_1, .., name_n : v_n)$ unifies with an extended instance, it is called an *extended class predicate*.
- If p is an intensional predicate symbol, $p(t_1, .., t_n)$ is called an *intensional predicate*.
- If p is a comparison predicate symbol, $p(t_i, t_j)$ is called a *comparison predicate*.

A *predicate* is a simple (or extended) class, intensional, or comparison predicate.

Example 4.3 *The following are class predicates associated with Example 4.1:*

$Airport(self : x, name : y)$

$Civilian\_airport(self : x, name : y, earliest\_time : p, latest\_time : q)$

From the definition of class predicate, we can also say:

- The arity of a class predicate is the size of its closure.
- Class predicates are typed. The type an argument of a class predicate is the type of corresponding attribute in the class description.

We require that intensional predicates be typed. This means that every intensional predicate has a unique parity, and its arguments are typed. The type of intensional predicates can be explicitly specified or implicitly derived from the logic program. We assume that intensional predicate types are implicitly defined in the program.

A *rule* is a Boolean formula of the form

$$H \leftarrow B_1, .., B_n$$

where H (head) is a class or intensional predicate, and B_i (body predicate) is any predicate. Body predicates can be positive or negative (which means the rule can be non-Horn). A rule with empty body is called a *fact* and a rule without a head is called a *query*. The head predicate of a rule r is said to be defined by r. Only (simple) class predicates are defined through facts. Thus simple instances of classes will be specified as facts. Intensional predicates are defined by rules that are not facts.

Consider a rule r in the language. We say that a variable x appearing in r is *limited* if it appears in

- a non-negated non-comparison predicate in the body of r, or
- an equality predicate whose second argument is either a constant or a limited variable.

A rule r is *safe* if every variable occurring in r is limited.

A logic program is a set of rules. A logic program is *safe* iff its rules are safe. Henceforth we consider only safe programs.

We assume that a logic program is stratifiable, and we may use stratification to specify the semantics of a Complex-Prolog program.

4.4 Databases, Semantics, and Queries

A Complex-Prolog database is a tuple $< S, L >$, where S is a schema and L is a logic program (set of rules). A database is *consistent* iff

1. each instance[15]of a class in S has the same value for every *self* attribute in its closure and no two distinct instances of the same class have the same value for the $self$ attributes (object identity).

2. for each object identifier $o \neq nil$ occurring in an instance of a class $c \in S$, there is an instance[16] in some class of S with o as its identifier (termed a *key dependency* [Fag81]).

Since only simple instances of classes can be specified as facts, we need to define a way to satisfy extended class predicates. Let P be an extended class predicate and let $q(a_1 : y_1, .., a_m : y_m)$ be the first class term occurring in P. We define the *hidden rule* of P as follows:

$$P \leftarrow \overline{P}, q(a_1 : y_1, .., a_m : y_m),$$

where $\overline{P}$ is obtained from P by replacing the class term $q(a_1 : y_1, .., a_1 : y_m)$ by y_1.

Example 4.4 *Let $p(a_1 : x, a_2 : q(y, z))$ be an extended class predicate. The corresponding hidden rule is*

$$p(a_1 : x, a_2 : q(b_1 : y, b_2 : z)) \leftarrow p(a_1 : x, a_2 : y), q(b_1 : y, b_2 : z)$$

The *overall definition* σ of a database is the union of its program and hidden rules for extended predicates of all classes.

The semantics of a Complex-Prolog program is similar to that of typical logic programs. In particular, the set of all terms involving R, C, I and W is the Herbrand universe for any Complex-Prolog logic program. The comparison predicates are assumed to be defined by a possibly infinite set of facts. Similarly, it is assumed that the types of constants are specified through a possibly infinite set of facts. Since a program is assumed to be stratified, the overall definition σ of a program is also stratified and the minimal model can be computed through a fixpoint operation that computes facts for each strata, starting from the bottom. The set of facts inferred from the database are those in the minimal model.

Let Q be a query with free variables $x_1, .., x_n$. The answer to Q is the set of tuples of terms $< v_1, .., v_n >$ such that, when we substitute v_i for $x_i, 1 \leq i \leq n$, all the predicates in Q are inferred from the overall definition of the database.

Example 4.5 *The following query retrieves the airports in Canada:*

$$\leftarrow Airport(self : x, name : y^*, address : Address(country : \text{“Canada”}))$$

[15]Recall that instances are specified as facts in the logic program.

[16]Only a $referential$ attribute value can be an identifier. The class of each referential attribute must have an instance to account for this.

4.5 Integrity Constraints

An integrity constraint (*IC)* is an assertion about the database. ICs do not affect the semantics of a database and hence we decided to treat them separately.

Integrity constraints are defined using a Horn-clause rule of the form:

$$H \leftarrow B_1, .., B_n$$

where $H, B_1, .., B_n$ are class, intensional, or comparison predicates. Because of the difficulties involved with recursive predicates, we permit only non-recursive intensional predicates in ICs. But non-recursive intensional predicates can be replaced by equivalent class and comparison predicates, leaving ICs with class and comparison predicates. Henceforth we consider ICs that do not contain intensional predicates, the same restriction imposed for Datalog ICs in [CGM90].

An IC of the form above asserts that H is true whenever $B_1, .., B_n$ are true. Similar to rules in Complex-Prolog programs, IC variables are also universally quantified. Furthermore, IC rules must be safe (every variable must be limited).

Example 4.6 *The integrity constraint that no flights may arrive or depart from Pearson Airport after 2300 hrs. is expressed by:*

$$(y \leq 2300) \leftarrow Civilian\_airport(self : -, name : x, latest\_time : y), \\ (x = \text{“Pearson”})$$

Example 4.7 *If the village of St. Jacobs has no* Airport*, we may say:*

$$\leftarrow Airport(self : x, address : Address(city : \text{“St. Jacobs”}))$$

We allow existential quantification of variables in the head of an IC. If a variable occurs only in the head, it is implicitly assumed to be existentially quantified[17].

Example 4.8 *Consider the IC:*

$$P(attr_1 : x, attr_2 : y) \leftarrow R(attr_3 : x, attr_4 : z), (x = 300)$$

The variable y in the IC is existentially quantified and $y = f(x, z)$, where f is some Skolem function.

Since we use non-positional notation, existentially quantified variables and corresponding attributes can be left out of integrity constraints. So we may write the above rule as:

$$P(attr_1 : x) \leftarrow R(attr_3 : x, attr_4 : z), (x = 300)$$

Example 4.9 *Consider the IC asserting that every Airport has an alternative Airport near-by:*

$$Airport(self : x) \leftarrow Airport(self : -, alternative : x)$$

This is an example of asserting referential integrity. Notice that the 'missing' attributes of *Airport* in the head of the IC are indeed existentially quantified variables (Skolem function applications).

[17] Note that otherwise the IC is no longer safe.

4.6 Translation of a Complex-Prolog Database to a Relational Database using Datalog

We now describe the translation of a Complex-Prolog database to a relational Datalog program.

4.6.1 Translation of a Complex-Prolog Schema

Let c be a class in the schema S with

$$closure(c) =< a_1.., a_q, closure(r_1), .., closure(r_k) >$$

and let

$$I = \ c(a_1 : v_1, .., a_q : v_q, T_{r_1}, .., T_{r_k})$$

be a simple instance in the database. Assume that there exists a mapping M to convert object identifiers into integers. The flattening function μ that transforms the terms $(v_1, .., v_n)$ into atomic values is defined by:

- $\mu(v_i) = M(v_i)$ if v_i is an object identifier.
- $\mu(v_i) = v_i$ if v_i is a constant or variable.
- $\mu(v_i) =< b_1 : \mu(c_1), .., b_m : \mu(c_m) >$ if $v_i = \ g(b_1 : c_1, .., b_m : c_m)$ is a record term.

The tuple t associated with I is given by:

$$t =< a_1 : \mu(v_i), .., a_q : \mu(v_q) >$$

where the inherited attributes and their values have been left out.

The relation R associated with the class c is the set of tuples corresponding to the instances of $ext(c)$ in the database. In fact, a relation will just contain the 'facts' [18] in the Datalog program for the corresponding class. The definition of this relation is constructed by applying a new flattening function α to the non-inherited attributes $(a_1, .., a_q)$ of the class c. This function is defined as follows:

- $\alpha(a_i) = s_i : int$ if a_i is either a reference or a subrange attribute with name s_i.
- $\alpha(a_i) = a_i$ if a_i is a string or an integer attribute.
- $\alpha(a_i) =< v(c_i), .., v(c_n) >$ if $a_i =< s_i, g(c_1, .., c_n) >$ is a record attribute.

The relation R corresponding to c is $c(v(a_1), .., v(a_q))$.

The underlying relational database is the set of relations corresponding to the classes in the schema. The facts of the Datalog program become tuples in the relations of their corresponding classes.

Example 4.10 *The relational schema corresponding to the Airport class of Example 4.1 is:*

$$Airport(self : int, name : char[25], city : char[25], country : char[25], \\ alternative : int)$$

[18] How to generate those Datalog facts for a class is explained later.

4.6.2 Complex-Prolog Rules to Datalog Rules

A Datalog predicate is either a database (extensional) predicate, if it corresponds to a stored relation (facts), or a derived (intensional) predicate, if it is defined by rules. A predicate may also be a comparison predicate. No function symbols are allowed. A Datalog rule is of the form

$$A \leftarrow L_1, .., L_n$$

where $L_1, L_2, ..L_n$ are literals. A is a class predicate if the body is empty (a fact) and a derived predicate otherwise. All facts are ground. Since the arity of Datalog extensional predicates is small, positional notation is used.

We now describe how Complex-Prolog rules and facts can be translated to Datalog rules.

Let $P = \ p(s_1 : v_1, s_2 : v_2, .., s_n : v_n)$ be a simple predicate of arity n. Let the equivalent positional form of P be $p(v_1, .., v_n)$. Assume that

$$closure(p) =< a_1, .., a_q, closure(r_1), .., closure(r_k) > .$$

The *isa* rule for P is defined as:

$$P(X, Y_1, .., Y_k) \leftarrow p(X), r_1(Y_1), .., r_k(Y_k), y_{1_1} = x_1, .., y_{k_1} = x_1$$

where X is a list of p's own attribute arguments, $Y_1, .., Y_k$ are the attribute arguments of $r_1, .., r_k$ respectively, and $r_i(Y_i)$ is a simple class predicate, one for each class $r_i \in ancestors(p)$. y_{i_1} is the first (self) attribute argument in the list Y_i, x_1 is the argument corresponding to the self attribute in X.

The above rule is safe since all the argument variables in the head occur in the body as well. Notice that the rule requires the simple predicate p to be completely specified, that is, contain arguments for all attributes in $closure(p)$.

Example 4.11 *The* isa *rule associated with the simple predicate*

$$Civilian\_airport(self : s_1, name : n, address : a, alternative : t,$$
$$self : s_2{}^{19}, earliest\_time : e, latest\_time : l)$$

becomes:

$$Civilian\_airport(s, n, a, t, e, l) \leftarrow Civilian\_airport(s_1, e, l),$$
$$Airport(s_2, n, a, t), s_1 = s_2$$

We need a function λ to flatten record terms in a class predicate and replace object identifiers with integers. Let x be a term; then:

- $\lambda(x) = M(x)$ if x is an object identifier.
- $\lambda(x) = x$, if the corresponding attribute a is either an atomic or reference attribute.

[19] $self : s_2$ is an attribute inherited from the Airport class.

- $\lambda(x) =< \lambda(y_1), .., \lambda(y_n) >$, where $y_1, .., y_n$ are new variables, if the corresponding attribute $a =< s, g(c_1, .., c_n) >$ is a record attribute and x is a variable.
- $\lambda(x) =< \lambda(z_1), .., \lambda(z_n) >$ if $x = g(s_1 : z_1, .., s_n : z_n)$ is a record term.

Given a simple Complex-Prolog predicate $p(s_1 : x_1, .., s_q : x_q)$ without class terms, where $s_1, .., s_q$ are all p's own attributes, the corresponding Datalog predicate is

$$p(\lambda(x_1), .., \lambda(x_q))$$

Since every intensional predicate symbol in Complex-Prolog is implicitly typed, we can extend the definition of λ to intensional predicates similarly.

Notice that both *isa* rules and the λ function require a class predicate to be completely specified. However, a class predicate in a rule (IC) written in non-positional notation may not have all its arguments specified. Let $p(s_1 : v_1, .., s_q : v_q)$ be an *incompletely* specified class predicate in some rule r. Assume that n is the arity of p and $s_{q+1}, .., s_n$ are the unspecified attributes. Then r cannot be a fact with p as its head, since p will then not define a valid simple instance of the class p. Since class predicates can occur only in the body of rules, r must be of the form:

$$s(v_1, .., v_k) \leftarrow B_1, .., B_m, p(s_1 : v_1, .., s_q : v_q)$$

where $B_1, .., B_m$ are other predicates in the body. We introduce new variables $v_{q+1}, .., v_n$ for the unspecified attributes of p. All these variables are universally quantified as required. The resulting rule will be:

$$s(v_1, .., v_k) \leftarrow B_1, .., B_m, p(s_1 : v_1, .., s_q : v_q, s_{q+1} : v_{q+1}, .., s_n : v_n)$$

We can apply a similar modification to any incompletely specified class predicate in the body of integrity constraints. However, an IC can have a class predicate in its head and this predicate can be incompletely specified. Consider an integrity constraint $\mathbf{IC}_k$ with all class predicates in the body completely specified. Let $\mathbf{IC}_k$ be:

$$\mathbf{IC}_k \ = p(s_1 : v_1, .., s_q : v_q) \leftarrow B_1, .., B_q$$

where p is the class predicate described above and $B_1, .., B_q$ are any valid predicates.

We can introduce new variables for the unspecified attributes as done earlier. But, these variables must be existentially quantified to correctly represent the meaning of the IC (and leave it safe). We therefore introduce new Skolem functions $f_{q+1}, ..f_n$ of arity m, where m is the number of variables $u_1, .., u_m$ (which must be universally quantified) in the body of $\mathbf{IC}_k$. We rewrite the IC as:

$$p(s_1 : v_1, .., s_q : v_q, s_{q+1} : f_{q+1}(u_1, .., u_m), .., s_n : f_n(u_1, .., u_m)) \leftarrow B_1, .., B_q$$

The algorithm to transform a Complex-Prolog rule into (possibly several) Datalog rules is described below. Our algorithm is from [GR89], modified to support the addition of ICs into the language.

Algorithm 1 *Complex Prolog rules to Datalog rules.*

Input: Any Complex-Prolog rule r.
Output: One or more equivalent Datalog rules.
Method:

- If r is a fact involving predicate p, use the *isa* rule to obtain a set of facts[20] for p and each class $s \in ancestors(p)$.
- If r is a rule, modify it such that class predicates are completely specified. While there is a class or intensional predicate p in the rule do the following:
  - If p is an extended class predicate, obtain the conjunction of equivalent simple predicates using the hidden rules repeatedly. If p is positive in r, replace p by the conjunction of predicates obtained. If p is negated in r, generate the rule for a new intensional predicate q as the conjunction of predicates obtained due to p. Replace p by $\neg(q)$.

    If p is a simple predicate, replace p in r by the body of predicates obtained using the *isa* rule. Handle negation as done above.
  - For each class or intensional predicate p in the rule:
    - Flatten each argument term x using the λ function defined above.
    - Replace every occurrence of x in the rule by $\lambda(x)$. Replace a comparison predicate c with tuple arguments by multiple c predicates comparing the components, which should be atomic.

The resulting Datalog rules are equivalent to r.

4.6.3 Translating Integrity Constraints

Complex-Prolog allows the specification of domain bounds for attributes in the definition of a class predicate. Domain bounds correspond directly to two Datalog ICs, one for the lower bound and one for the upper bound. For generalization and the inheritance of attributes, we must define a referential constraint between a class and its subclass, such that each object identifier (the *self* attribute) present in the relation representing the subclass also exists in the superclass. Other superclass variables are defined in the IC by Skolem functions.

Example 4.12 *To show that a Military_airport* isa *Airport, we define the IC:*

$$Airport(x, f_1(x,y), f_2(x,y), f_3(x,y), f_4(x,y), f_5(x,y), f_6(x,y))$$
$$\leftarrow Military\_airport(x,y)$$

[20] Recall that these facts form tuples for the relations that correspond to its class in the Complex-Prolog model.

Algorithm 2 *Complex Prolog ICs to Datalog ICs.*

Input: Integrity constraints defined in the Complex-Prolog schema.
Output: One or more equivalent Datalog ICs.
Method: Let $\mathbf{IC}_k$ be an integrity constraint including the ones generated by the conversion of value bounds and *isa* rules. For each of the ICs, we now:

- Modify $\mathbf{IC}_k$ such that class predicates are completely specified. $\mathbf{IC}_k$ may now contain Skolem functions.
- If the head is a class predicate p:
  - If p is an extended class predicate, replace it by the equivalent conjunction of simple predicates, using the hidden rules. If p is equivalent to a conjunction of predicates, generate multiple ICs with each of them as a head predicate. Apply the following steps for each of the resulting ICs:
    - If the head predicate p is simple, replace it by the body of predicates obtained using the *isa* rule. Since we restrict an IC to have one head predicate, multiple ICs will result if p is replaced by multiple predicates. Process each of the ICs using the following steps:
    - For each class predicate q in the body of IC:
      - Replace the class predicate by equivalent predicates as specified in the above algorithm.
    - Apply the flattening function λ to the terms of all predicates and replace terms by the results of the λ function applications.
    - Generate multiple comparison predicates if the arguments are tuples.

We notice that both the rule transformation algorithm and the integrity constraint transformation algorithm terminate. An extended class predicate occurring in a program has a finitely nested structure, hence it takes finite time to generate equivalent simple class predicates. Because we restrict subclasses in the Complex-Prolog schema to be acyclic, application of *isa* takes only finite time.

4.6.4 Answering Queries

The steps described above will be used to convert a Complex-Prolog query into an equivalent Datalog query. Since no variables are eliminated during the translation, we can apply the reverse transformation to the results of the Datalog query to get the results of the original Complex-Prolog query.

5 Applying SQO to Complex-Prolog

In this section we describe how semantic query optimization techniques from [CGM90] may be applied to an object-oriented model, defined using Complex-Prolog.

In the previous section we discussed an algorithm from Greco and Rullo that converts a Complex-Prolog into an equivalent Datalog program. Our addition of ICs to the Complex-Prolog language allows us to take advantage of SQO techniques to perform query processing. Recall from Section 3 that the heuristics used in the query transformations as described in [CGM90] are applicable to the relational model. We assume in the query transformation process for a Complex-Prolog query that these same transformation techniques can be used with an object-oriented model.

5.1 Sample Complex-Prolog Database

Our example database models various aspects of the air transportation business; airlines, aeroplanes, flights, and airports.

1. $Plane(model : char[15], manufacturer : char[20], runway\_length : int,$
   $flight\_crew : 1..5, age : 0..50, lifetime : 5..60)$
2. $Cargo\_plane(item\_maxheight : int, item\_maxlength : int,$
   $item\_maxweight : int, item\_maxwidth : int)$ *isa* $Plane$
3. $Passenger\_plane(maximum\_passengers : 1..500, cabin\_crew : 1..25,$
   $cargo\_capacity : int)$ *isa* $Plane$
4. $Airline(name : char[25], home\_airport : Airport)$
5. $Airport(name : char[25],$
   $address : Address(city : char[25], country : char[25]),$
   $geographical\_postion : Geopos(latitude : int, longitude : int),$
   $max\_runway\_length : int)$
6. $Military\_airport(armed\_forces\_dept : char[25])$ *isa* $Airport$
7. $Civilian\_airport(earliest\_time : int, latest\_time : int)$ *isa* $Airport$
8. $Flight(plane : Plane, departure\_time : int, duration : int)$
9. $Civilian\_flight(airline : Airline, origin : Civilian\_airport,$
   $destination : Civilian\_airport)$ *isa* $Flight$
10. $Military\_flight(origin : Airport, destination : Airport)$ *isa* $Flight$
11. $Stops(flight : Flight, airport : Airport, arrival\_time : int,$
    $departure\_time : int)$

5.2 Equivalent Datalog Schema

The Datalog schema for the Air Transportation database is determined by using the conversion algorithm described in the previous section. Recall that domain specifications in the extensional predicates, and *isa* hierarchies, are converted into Datalog integrity constraints. Our predicates are presented in positional notation, to match the notation in [CGM90].

1. $Plane(self : int, model : char[15], manufacturer : char[20],$
   $runway\_length : int, flight\_crew : int, age : int, lifetime : int)$
2. $Cargo\_plane(self : int, item\_maxheight : int, item\_maxlength : int,$
   $item\_maxweight : int, item\_maxwidth : int)$
3. $Passenger\_plane(self : int, maximum\_passengers : int,$
   $cabin\_crew : int, cargo\_capacity : int)$
4. $Airline(self : int, name : char[25], home\_airport : int)$
5. $Airport(self : int, name : char[25], city : char[25], country : char[25],$
   $latitude : int, longitude : int, max\_runway\_length : int)$
6. $Military\_airport(self : int, armed\_forces\_dept : char[25])$
7. $Civilian\_airport(self : int, earliest\_time : int, latest\_time : int)$
8. $Flight(self : int, plane : int, departure\_time : int, duration : int)$
9. $Civilian\_flight(self : int, airline : int, origin : int, destination : int)$
10. $Military\_flight(self : int, origin : int, destination : int)$
11. $Stops(self : int, flight : int, airport : int, arrival\_time : int,$
    $departure\_time : int)$

Our referential integrity constraints for this Datalog representation are as follows:

IC$_1$ *Cargo_plane* isa *Plane*.

$$Plane(x_1, f_1(x_1, x_2, x_3, x_4, x_5), f_2(x_1, x_2, x_3, x_4, x_5),$$
$$f_3(x_1, x_2, x_3, x_4, x_5), f_4(x_1, x_2, x_3, x_4, x_5), f_5(x_1, x_2, x_3, x_4, x_5),$$
$$f_6(x_1, x_2, x_3, x_4, x_5))$$
$$\leftarrow Cargo\_plane(x_1, x_2, x_3, x_4, x_5)$$

IC$_2$ *Passenger_plane* isa *Plane*.

$$Plane(x_1, f_1(x_1, x_2, x_3, x_4), f_2(x_1, x_2, x_3, x_4), f_3(x_1, x_2, x_3, x_4),$$
$$f_4(x_1, x_2, x_3, x_4), f_5(x_1, x_2, x_3, x_4), f_6(x_1, x_2, x_3, x_4))$$
$$\leftarrow Passenger\_plane(x_1, x_2, x_3, x_4)$$

IC$_3$ *Military_airport* isa *Airport*.

$$Airport(x_1, f_1(x_1, x_2), f_2(x_1, x_2), f_3(x_1, x_2), f_4(x_1, x_2),$$
$$f_5(x_1, x_2), f_6(x_1, x_2))$$
$$\leftarrow Military\_airport(x_1, x_2)$$

IC$_4$ *Civilian_airport* isa *Airport*.

$$Airport(x_1, f_1(x_1, x_2, x_3), f_2(x_1, x_2, x_3), f_3(x_1, x_2, x_3), f_4(x_1, x_2, x_3), \\ f_5(x_1, x_2, x_3), f_6(x_1, x_2, x_3)) \\ \leftarrow Civilian\_airport(x_1, x_2, x_3)$$

IC$_5$ *Civilian_flight* isa *Flight*.

$$Flight(x_1, f_1(x_1, x_2, x_3, x_4), f_2(x_1, x_2, x_3, x_4)) \\ \leftarrow Civilian\_flight(x_1, x_2, x_3, x_4)$$

IC$_6$ *Military_flight* isa *Flight*.

$$Flight(x_1, f_1(x_1, x_2, x_3), f_2(x_1, x_2, x_3)) \\ \leftarrow Military\_flight(x_1, x_2, x_3)$$

In addition, value bounds on the attributes *flight_crew, age, lifetime, maximum_passengers,* and *cabin_crew* are defined as follows:

IC$_7$, **IC**$_8$ Value bounds on attribute *Plane.flight_crew*:

$$(x_5 \geq 1) \leftarrow Plane(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$$

$$(x_5 \leq 5) \leftarrow Plane(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$$

IC$_9$, **IC**$_{10}$ Value bounds on attribute *Plane.age*:

$$(x_6 \geq 0) \leftarrow Plane(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$$

$$(x_6 \leq 50) \leftarrow Plane(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$$

IC$_{11}$, **IC**$_{12}$ Value bounds on attribute *Plane.lifetime*:

$$(x_7 \geq 5) \leftarrow Plane(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$$

$$(x_7 \leq 60) \leftarrow Plane(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$$

IC$_{13}$, **IC**$_{14}$ Value bounds on attribute *Passenger_plane.maximum_passengers*:

$$(x_2 \geq 1) \leftarrow Passenger\_plane(x_1, x_2, x_3, x_4)$$

$$(x_2 \leq 500) \leftarrow Passenger\_plane(x_1, x_2, x_3, x_4)$$

IC$_{15}$, **IC**$_{16}$ Value bounds on attribute *Passenger_plane.cabin_crew*:

$$(x_3 \geq 1) \leftarrow Passenger\_plane(x_1, x_2, x_3, x_4)$$

$$(x_3 \leq 25) \leftarrow Passenger\_plane(x_1, x_2, x_3, x_4)$$

Although we have defined integrity constraints for subclasses, a pair of classes related by an *isa* relationship are defined in our Datalog model as *distinct* relations. For convenience, we can define a *view* [21] (an intensional predicate) for each subclass and its ancestors. The join attributes between each relation are simply the object identifiers; they are guaranteed to exist via the integrity constraints defined above. Defining such a view is not *necessary*, since *isa* relationships are converted to ICs and we can make use of these ICs when evaluating queries.

5.3 Semantic Compilation and Query Transformation

In this Subsection, we give two examples of utilizing semantic query optimization heuristics on our converted Complex-Prolog schema. To illustrate the use of these techniques, we define some further ICs, first using the Complex-Prolog language, and then its equivalent Datalog form:

IC$_{17}$ *The database does not have any over-aged planes.*

$$(p < q) \leftarrow Plane(self : -, age : p, lifetime : q)$$

Recall that Complex-Prolog constraint need not be completely specified. The equivalent Datalog constraint is as follows:

$$(x_6 \leq x_7) \leftarrow Plane(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$$

IC$_{18}$ *Passenger planes that require a cabin crew of 10 or more carry at least 350 passengers.*

$$(z \geq 350) \leftarrow Passenger\_plane(self : x, cabin\_crew : y, max\_passengers : z), (y \geq 10)$$

The equivalent Datalog constraint is as follows (Skolem functions are shown without arguments):

$$(x_2 \geq 350) \leftarrow Passenger\_plane(x_1, x_2, x_3, x_4), Plane(x_1, f_1, f_2, f_3, f_4, f_5, f_6), (x_3 \geq 10)$$

[21] Defining a view, in fact, may not be so "convenient" if our underlying relational system does not support update operations through a view; many systems, in fact, do not [Cod90]. For our purposes, view update support for PK-FK and PK-PK join views is sufficient.

Note that in this case the *Plane* predicate is redundant to $\mathbf{IC}_2$ already established.

In addition to the specific constraints $\mathbf{IC}_{17}$ and $\mathbf{IC}_{18}$, we could add referential constraints and functional dependencies to our air transport schema, as described in [JCV84]. The referential constraints ensure that each object reference in a extensional predicate refers to an existing tuple in the associated relation. From functional dependency constraints we can derive further residues, as explained in [CGM90]. However, to help simplify the examples we will omit these constraints. Our Datalog schema above was defined using positional notation, but we included the attribute names and types to better illustrate the transformation from Complex-Prolog. From now on, we will use the strictly positional notation of [CGM90].

5.3.1 Semantically Constrained Axioms

At the end of the Semantic Compilation process, our cross-product of EDB predicates (mapped directly into IDB axioms) and the 18 integrity constraints results in the following semantically constrained axioms (SCAs). The notation used to describe these axioms is taken from [CGM90]; conditional predicates which must be true for a particular axiom are listed in braces. To conserve space, Skolem functions are shown with no arguments.

$\mathbf{SCA}_1$ $Plane(x_1, x_2, x_3, x_4, x_5, x_6, x_7) \leftarrow Plane(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$
$\{(x_5 \geq 1); (x_5 \leq 5); (x_6 \geq 0); (x_6 \leq 50); (x_7 \geq 5); (x_7 \leq 60); (x_6 \leq x_7)\}$

$\mathbf{SCA}_2$ $Cargo\_plane(x_1, x_2, x_3, x_4, x_5) \leftarrow Cargo\_plane(x_1, x_2, x_3, x_4, x_5)$
$\{Plane(x_1, f_1, f_2, f_3, f_4, f_5, f_6) \leftarrow; (y_5 \geq 1); (y_5 \leq 5); (y_6 \geq 0); (y_6 \leq 50);$
$(y_7 \geq 5); (y_7 \leq 60); (y_6 \leq y_7)\}$

$\mathbf{SCA}_3$ $Passenger\_plane(x_1, x_2, x_3, x_4) \leftarrow Passenger\_plane(x_1, x_2, x_3, x_4)$
$\{Plane(x_1, f_1, f_2, f_3, f_4, f_5, f_6) \leftarrow; (y_5 \geq 1); (y_5 \leq 5); (y_6 \geq 0);$
$(y_6 \leq 50); (y_7 \geq 5); (y_7 \leq 60); (y_6 \leq y_7); (x_2 \geq 350) \leftarrow (x_3 \geq 10)\}$

$\mathbf{SCA}_4$ $Airline(x_1, x_2, x_3) \leftarrow Airline(x_1, x_2, x_3)\{\}$

$\mathbf{SCA}_5$ $Airport(x_1, x_2, x_3, x_4, x_5, x_6, x_7) \leftarrow Airport(x_1, x_2, x_3, x_4, x_5, x_6, x_7)\{\}$

$\mathbf{SCA}_6$ $Military\_airport(x_1, x_2) \leftarrow Military\_airport(x_1, x_2)$
$\{Airport(x_1, f_1, f_2, f_3, f_4, f_5, f_6) \leftarrow\}$

$\mathbf{SCA}_7$ $Civilian\_airport(x_1, x_2, x_3) \leftarrow Civilian\_airport(x_1, x_2, x3)$
$\{Airport(x_1, f_1, f_2, f_3, f_4, f_5, f_6) \leftarrow\}$

$\mathbf{SCA}_8$ $Flight(x_1, x_2, x_3, x_4) \leftarrow Flight(x_1, x_2, x_3, x_4)\{\}$

$\mathbf{SCA}_9$ $Civilian\_flight(x_1, x_2, x_3, x_4) \leftarrow Civilian\_flight(x_1, x_2, x_3, x_4)$
$\{Flight(x_1, f_1, f_2, f_3) \leftarrow\}$

$\mathbf{SCA}_{10}$ $Military\_flight(x_1, x_2, x_3) \leftarrow Military\_flight(x_1, x_2, x_3)$
$\{Flight(x_1, f_1, f_2, f_3) \leftarrow\}$

$\mathbf{SCA}_{11}$ $Stops(x_1, x_2, x_3, x_4, x_5) \leftarrow Stops(x_1, x_2, x_3, x_4, x_5)\{\}$

5.3.2 Unsatisfiable Query

Our first example shows how the integrity constraints can be used to determine that a `null` result can be the only possible answer for a particular query.

Example 5.1 *Query: Find manufacturers of passenger planes outliving their lifetime.*

$$\leftarrow Passenger\_plane(self : x, manufacturer : z^*, age : p, lifetime : q),$$
$$(p \geq q)$$

Recall that Complex-Prolog queries that pertain to subclasses in the schema must be expanded to include inherited attributes; in the relational sense, this means implying a join (perhaps several) between the hierarchically related extensional predicates. Thus, our equivalent Datalog query is:

$$\leftarrow Passenger\_Plane(x_1, x_2, x_3, x_4), Plane(x_1, y_2, y_3^*, y_4, y_5, y_6, y_7),$$
$$(y_6 > y_7)$$

It is clear from the Datalog form of the query that it contradicts $\mathbf{SCA}_3$. Therefore the query answer is $\emptyset$.

5.3.3 Index Introduction

Our second example shows how we can introduce an indexed attribute into the query so that the execution plan is more efficient.

Example 5.2 *Query: Find the manufacturers of passenger planes that require a cabin crew of at least 10.*

$$\leftarrow Passenger\_plane(self : x, manufacturer : y^*, cabin\_crew : z), (z \geq 10)$$

Our equivalent Datalog query is:

$$\leftarrow Passenger\_Plane(x_1, x_2, x_3, x_4), Plane(x_1, y_2, y_3^*, y_4, y_5, y_6, y_7),$$
$$(x_3 \geq 10)$$

Again, from $\mathbf{SCA}_3$ we can infer that $(x_2 \geq 350)$. If an index exists on the *maximum_passengers* attribute, but not on the *cabin_crew* attribute, then we can introduce the literal $(x_2 \geq 350)$ into the query using the *index introduction* heuristic. Our transformed query then becomes:

$$\leftarrow Passenger\_Plane(x_1, x_2, x_3, x_4), Plane(x_1, y_2, y_3^*, y_4, y_5, y_6, y_7),$$
$$(x_3 \geq 10), (x_2 \geq 350)$$

Many additional examples can be generated from the above schema, especially if we include referential constraints and functional dependencies. Although we have skipped many details, it appears that the logic-based approach by Chakravarthy *et al.* can be used on an object-oriented data model, by means of translating the schema and the database rules into their equivalent Datalog form.

6 Conclusion

Capturing more meaning in a database system by managing integrity constraints can lead to a number of possible benefits:

- Since the database manager now enforces integrity constraints instead of being encoded within application programs, we can ensure that all application programs adhere to the same set of consistency rules.
- *Ad-hoc* updates to the database may now be controlled.
- Altering constraints may involve no changes to application programs.
- The additional metadata may be used to semantically optimize queries.

By extending Datalog to support complex objects, Greco and Rullo[GR89] derived a database programming language called Complex-Prolog. Complex-Prolog schemas and queries support the notions of generalization, inheritance, and object identity.

In this paper, we showed that by adding integrity constraints (ICs) to Complex-Prolog we could apply the semantic query optimization techniques from Chakravarthy *et al.* [CGM90] defined for a relational model to support a complex-object model. The technique is applicable to both data models because Complex-Prolog may be converted to an equivalent Datalog (relational) representation.

There remains to be done significant work in both the areas of SQO and complex-object data models. Some of the issues remaining in semantic query optimization include:

- We can attempt to relax our restrictions on the specification of IDB rules and ICs, such as permitting an IC to include an IDB predicate in its head[LH88].
- Little work appears to have been done to determine a better way of reporting database integrity violations to the user. With semantic query optimization, we may infer additional constraints from the given set of ICs. It may not be obvious to a user that their query or update cannot be executed due to an integrity violation (or contradiction) when the violated constraint contains a predicate that does not appear in the query. Furthermore, the set of integrity constraints used for the inference may be quite large.
- A similar problem to the above exists for database administrators (DBAs) who wish to add or update ICs to the metabase. It may be difficult for the DBA to determine why a new constraint contradicts the existing base of ICs, given a large IC set and the inferences performed.
- The heuristics used for SQO, as defined originally by King[Kin81, Kin84], were identified as appropriate for the relational model, using (possibly) clustered indices on attributes. Further work in semantic query optimization has been based on these heuristics, and their underlying data structure assumptions. Indeed, our examples of the previous section assume that the relational heuristics are valid with our (translated) Complex-Prolog schema.

  Additional research needs to address different physical data structure implementations, such as hashed organizations, or derived relations[BCL86, LY87]. Further, indexing issues for

complex-object data models are still being actively studied, and much further work in this area is required[Kim90, pp. 336]. SQO heuristics must be adapted to support the new types of indexing schemes required with object-oriented database systems.

- Further extension of SQO techniques can be applied to distributed databases as well. Using SQO techniques with a distributed database is both more rewarding, and more challenging. SQO is more rewarding with distributed systems since there may be a tremendous improvement in query processing time if, for example, a relation stored at another site can be eliminated from a query. SQO is more challenging because integrity constraints from different individual databases may be necessary to optimally transform the query. Furthermore, the integrity constraints present at each site may be incompatible in a "global" view; see [TK81].

  Using SQO for homogeneous relational database systems in a distributed fashion is a complex issue. If we allow heterogeneous database systems to form the distributed system (sometimes termed a *federated database* [LMR90, SL90]), the complexities become more severe.

7 Acknowledgements

Glenn Paulley would like to acknowledge the financial support of the Information Technology Research Centre (ITRC), NSERC, and the Great-West Life Assurance Company.

Gopi Krishna Attaluri would like to acknowledge the financial support of ITRC.

The authors would like to thank Andrej Brodnik, Hemin Xiao, and Grant Weddell for their comments on an earlier version of this paper.

References

[AC75] M. M. Astrahan and D. D. Chamberlin. Implementation of a structured English query language. *Communications of the ACM*, 18(10):580–587, October 1975.

[BCL86] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *Proceedings of the 12th International Conference on Very Large Data Bases*, New York, New York, August 1986. IEEE Computer Society Press.

[CFM86] U. S. Chakravarthy, D. H. Fishman, and J. Minker. Semantic query optimization in expert systems and database systems. In L. Kerschberg, editor, *Expert Database Systems: Proceedings of the First International Workshop (1984)*, pages 659–674. Benjamin/Cummings, 1986.

[CFP82] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion dependencies and their interaction with functional dependenices. In *Proceedings, First ACM Symposium on the Principles of Database Systems*, Los Angeles, California, March 1982. Association of Computing Machinery.

[CGM87] U. S. Chakravarthy, John Grant, and Jack Minker. Foundations of semantic query optimization for deductive databases. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan Kaufmann, Los Altos, California, 1987.

[CGM90] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, June 1990.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[CM86] U. S. Chakravarthy and Jack Minker. Multiple query processing in deductive databases using query graphs. In *Proceedings of the 12th International Conference on Very Large Data Bases*, Kyoto, Japan, August 1986. IEEE Computer Society Press.

[Cod90] E. F. Codd. *The Relational Model for Database Management, Version 2*. Addison-Wesley, Reading, Massachusetts, 1990.

[CW90] S. Ceri and Jennifer Widom. Deriving production rules for constraint maintenance. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990. IEEE Computer Society Press.

[Dat86] C. J. Date. *Relational Database - Selected Writings*. Addison-Wesley, Reading, Massachusetts, 1986.

[Dat90] C. J. Date. *An Introduction to Database Systems*, volume one. Addison-Wesley, Reading, Massachusetts, fifth edition, 1990.

[Fag81] Ronald Fagin. A normal form for relational databases that is based on domains and keys. *ACM Transactions on Database Systems*, 6(3):387–415, September 1981.

[GM81] John Grant and Jack Minker. Optimization in deductive and conventional relational database systems. In Hervé Gallaire, Jack Minker, and Jean Nicolas, editors, *Advances in Database Theory*, volume 1, pages 195–234. Plenum Press, New York, New York, 1981.

[GMN84] Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[GR89] Sergio Greco and Pasquale Rullo. Complex-Prolog: A logic database language for handling complex objects. *Information Systems*, 14(1):79–87, 1989.

[GW89] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In *ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, June 1989.

[HCL+90] Laura M. Haas, Walter Chang, Guy M. Lohman, et al. Starbust mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–159, March 1990.

[HFLP89] Laura M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, June 1989.

[HP88] Waqar Hasan and Hamid Pirahesh. Query rewrite optimization in Starburst. Research Report RJ6367, IBM Corporation, Research Division, San Jose, California, August 1988.

[HZ80] Michael Hammer and Stanley B. Zdonik, Jr. Knowledge-based query processing. In *Proceedings of the 6th International Conference on Very Large Data Bases*, Montreal, Quebec, October 1980. IEEE Computer Society Press.

[ISO90] ISO. *Information Technology – Database Language SQL 2 Draft Report*. International Standards Organization ISO/IEC JTC 1/SC 21, December 1990.

[Jar86] Matthias Jarke. External semantic query simplification: A graph-theoretic approach and its implementation in PROLOG. In L. Kerschberg, editor, *Expert Database Systems: Proceedings of the First International Workshop (1984)*, pages 675–691. Benjamin/Cummings, 1986.

[JCV84] Matthias Jarke, Jim Clifford, and Yannis Vassiliou. An optimizing Prolog front-end to a relational query system. *ACM SIGMOD Record*, 14(2):296–306, June 1984.

[JK84] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[Kim90] Won Kim. Object-oriented databases: Definition and research directions. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):327–341, September 1990.

[Kin81] Jonathan J. King. QUIST - A system for semantic query optimization in relational databases. In *Proceedings of the 7th International Conference on Very Large Data Bases*, Cannes, France, September 1981. IEEE Computer Society Press.

[Kin84] Jonathan J. King. *Query Optimization by Semantic Reasoning*. UMI Research Press, Ann Arbor, Michigan, 1984.

[KL89] Michael Kifer and George Lausen. F-logic: A higher-order language for reasoning about objects, inheritance and schema. In *ACM SIGMOD International Conference on Management of Data*, pages 134–146, 1989.

[KM83] Madhur Kohli and Jack Minker. Intelligent control using integrity constraints. In *Proceedings of the National Conference of Artificial Intelligence*, pages 202–205, 1983.

[LH88] Sanggoo Lee and Jiawei Han. Semantic query optimization in recursive databases. In *IEEE Fourth Conference on Data Engineering*. IEEE, 1988.

[LMR90] Witold Litwin, Leo Mark, and Nick Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[LY87] Per-Åke Larson and H. Z. Yang. Computing queries from derived relations: Theoretical foundation. Research Report 87-35, University of Waterloo, Waterloo, Ontario, Canada, August 1987.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland, 1983.

[Mai86] David Maier. A logic for objects. In *Workshop on Foundations of Deductive databases and Logic Programming*, pages 6–26. Washington, D.C., 1986.

[MG90] Roika Missaoui and Robert Godin. The implication problem for inclusion dependencies: A graph approach. *ACM SIGMOD Record*, 19(1):36–40, March 1990.

[NY78] J. M. Nicolas and K. Yazdanian. Integrity checking in deductive data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 325–344. Plenum Press, New York, New York, 1978.

[RH80] Daniel J. Rosenkrantz and Harry B. Hunt, III. Processing conjunctive predicates and queries. In *Proceedings of the 6th International Conference on Very Large Data Bases*, Montreal, Quebec, October 1980. IEEE Computer Society Press.

[SK84] Allan Shepherd and Larry Kerschberg. PRISM: A knowledge based system for semantic integrity specification and enforcement in database systems. In *ACM SIGMOD International Conference on Management of Data*, Boston, Massachusetts, June 1984.

[SKY89] Philip C. Sheu, R. L. Kashyap, and S. Yoo. Query optimization in object-oriented knowledge bases. *Data and Knowledge Engineering*, 3(4):285–302, February 1989.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[SO87] Sreekumar T. Shenoy and Z. Meral Ozsoyoglu. A system for semantic query optimization. In *ACM SIGMOD International Conference on Management of Data*, San Francisco, California, May 1987.

[SO89] Sreekumar T. Shenoy and Z. Meral Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):344–361, September 1989.

[SSD88] Shashi Shekhar, Jiadeep Srivastava, and Soumitra Dutta. A formal model of trade-off between optimization and execution costs in semantic query optmization. In *Proceedings of the 14th International Conference on Very Large Data Bases*, New York, New York, August 1988. IEEE Computer Society Press.

[Sto75] Michael Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the 1st International Conference on Very Large Data Bases*, San Jose, California, May 1975. IEEE Computer Society Press.

[SU82] Fereidoon Sadri and Jeffrey D. Ullman. A complete axiomatization for a large class of dependencies in relational databases. In *Proceedings, 12th ACM Symposium on the Theory of Computing*, Los Angeles, California, April 1982. Association of Computing Machinery.

[SV84] Eric Simon and Patrick Valduriez. Design and implementation of an extendible integrity subsystem. In *ACM SIGMOD International Conference on Management of Data*, Boston, Massachusetts, June 1984.

[TK81] K. Tanaka and Y. Kambayashi. Logical integration of locally independent relational databases into a distributed database. In *Proceedings of the 7th International Conference on Very Large Data Bases*, Cannes, France, September 1981. IEEE Computer Society Press.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1.* Computer Science Press, Rockville, Maryland, 1988.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 2.* Computer Science Press, Rockville, Maryland, 1989.

[Xu83] G. Ding Xu. Search control in semantic query optimization. Research Report TR-83-09, University of Massachusetts, Amherst, Massachusetts, 1983.

[Yao79] S. Bing Yao. Optimization of query evaluation algorithms. *ACM Transactions on Database Systems*, 4(2):133–155, June 1979.

[YS89] Clement T. Yu and Wei Sun. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):362–375, September 1989.